Q compare waterfall model & spiral model of software development.

ans

Waterfall Model	Spiral Model
① It is simple & easy	① Complex.
② It follows linear sequential method.	② Follows evolutionary method.
③ Applicable for small projects.	③ Used for large projects
④ There is high amount of risk in waterfall model.	④ There is low amount of risk in spiral model.
⑤ Requires least maintenance.	⑤ Requires typical managment.
⑥ Reusability is extremely unlikely.	⑥ To certain extent, reusability is possible.

→ waterfall model

Advantages ① simple to understand & use and it is a better choice for software development projects.

② It functions well for smaller tasks & projects with well-defined requirements.

③ It is dependable & predictable technique for developing software.

④ It is sequential, linear strategy that makes it simpler to estimate time & resources needed for every project phase.

⑤ It offers clear picture of end products appearance & functionality.

Disadvantages ① doesn't enable end user feedback.

② New phase starts only after prior phase has been finished. Can't be kept in real world projects.

Date

Phases may overlap to increase efficiency & decrease costs.

③ Unsuitable for complicated projects because its linear & sequential nature complicates handling various dependencies & interrelated components.

④ Testing is usually done towards end of development process.

Spiral Model

Advantages: ① Includes risk analysis & management at every stage. Increasing security

② suggested for large & complicated projects.

③ It offers an incremental & iterative approach

④ Offers regular evaluations & reviews.

⑤ enables numerous iterations.

Disadvantages: ① costly model than others, unsuitable for small projects.

② can be complex because it contains numerous iterations of software development process.

③ Can be time consuming because it necessitates numerous assessments & reviews.

④ More complex than other models

Inference

Waterfall & spiral models are 2 main software models in SDLC. spiral model differs from waterfall as it includes definite identification of risk, which minimizes chances of project failure.

Waterfall model is risky because it has no iterations & everything should be done at once for a product, requirement must be collected, planned, designed, coded & integrated all at once.

Date

Q2. What is role of user participation in selection of life cycle model?

ans: User participation plays important role in selection of life cycle model. User participation helps ensure that the selected model aligns with user needs & project goals.

① Requirements Elicitation. - different life cycle models are better suited to diff. types of requirements & user input helps identify most appropriate model

② Feedback & Reviews provided by user are important.

③ Risk assessment - users often have insights into potential risk associated with project.

④ Resource allocation - can help determine availability of user resources.

⑤ Project Goals & Objectives - user participation helps align the selected model with project's specific objectives

⑥ Acceptance & ownership.

⑦ Communication & Collaboration. ⑧ Customization.

Q3 What are advantages of developing a prototype of a system.

ans: ① It make easy to detect errors.

② We can find missing functionality easily.

③ There is scope of refinement, new requirements can be easily accommodated.

④ Ensures greater level of customer satisfaction & comfort.

⑤ It helps both developers & users understand the system better.

⑥ It can actively involve users in development phase

⑦ Ideal for online system.

Date

Q4 Describe characteristics of software contrasting it with those of Hardware

Software	Hardware
- set of instructions that tells comp exactly what to do	- Physical part of computer that causes processing of data.
- developed & manufactured	- It is manufactured.
- It can't be executed without hardware	- can't perform any task without software.
- Intangible	- Tangible.
- It doesn't wear out over time.	- It wears out over time.

Q5. Describe role of management in software development with help of examples.

Ans: Management of software development is dependent on four factors: People, product, Process & project.

People: Requires good managers, who can understand the psychology of people & provide good leadership. He can increase probability of success.

Product: Objectives & scope of work should be defined clearly to understand requirements. Alternatives should be discussed. It may help managers select best approach within constraints imposed by delivery deadlines, budgetary etc.

Process - Provides framework from which comprehensive plan can be established. If process is weak end product will suffer

Project - Proper planning is required, understand do's & don'ts. Define requirement & freeze concrete reqt.

- software development life cycle.
- COCOMO. DFD, UML, class diagram. coupling

Software Engineering

systematic, disciplined, cost effective technique for software development.
Engineering approach to develop a software.

1945 - 65 : origin 1965 - 85 - crisis only 2% software were used.

1990 - 2000 : Internet.

SDLC (software development life cycle)

Customer
Service Provider

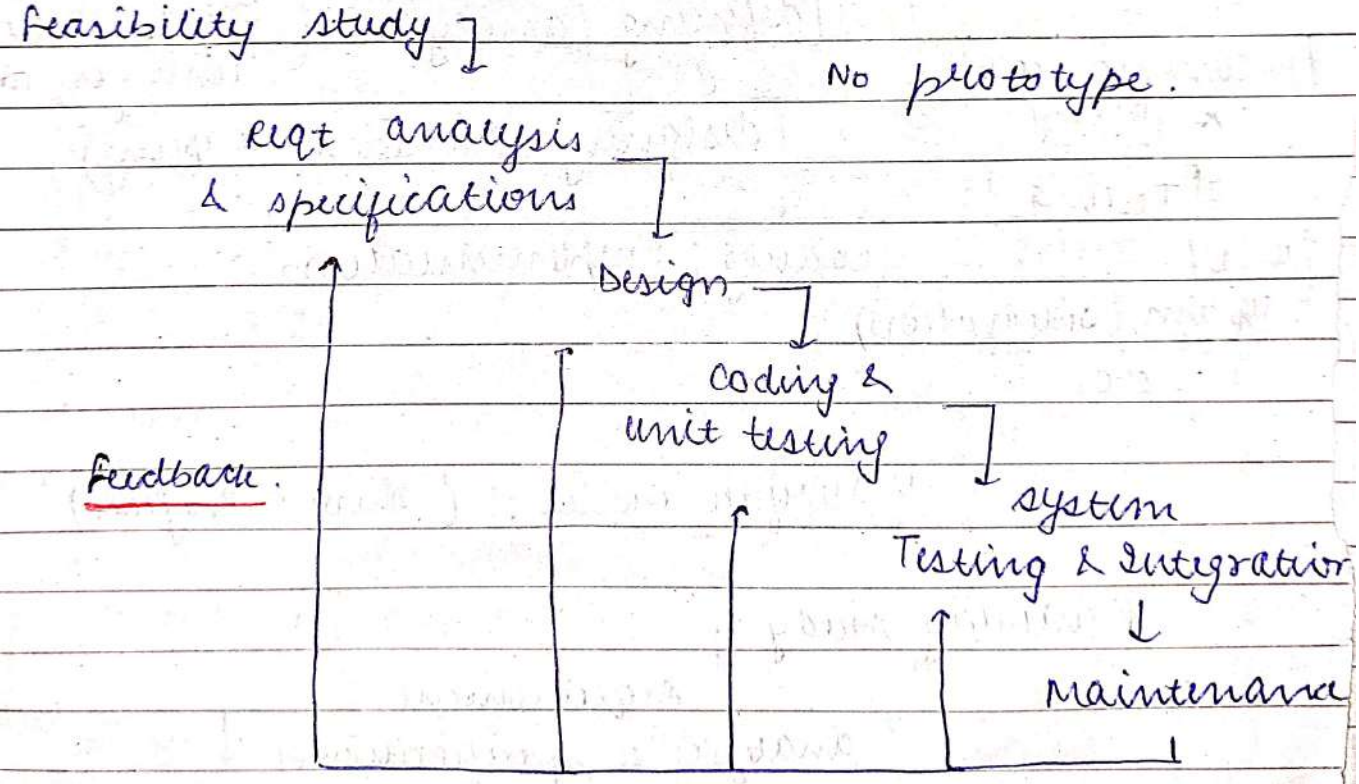
Planning → defining / analysis (SRS document) (cost estimation, Time, constraints etc.)
↓
designing (cohesion, coupling)
↓
coding / implementation
↓
Testing (unit / system / integration) etc
↓
Deploy
↓
Maintenance

Waterfall model. (small Projects)

Feasibility study
↓
requirement analysis & specifications (SRS doc)
↓
design
↓
Coding & unit testing
↓
system testing & integration (α, β)
↓
Maintenance (60% effort) (Bugs & problems)

Adv	DisAdv.
- Base model	- No feedback
- simple & easy	- No experiment.
- small Projects	- No parallelism
- Customer knows his requirements clearly.	- High Risk.
	- 60% efforts maintenance
	- Rigid model.

Iterative Waterfall model.

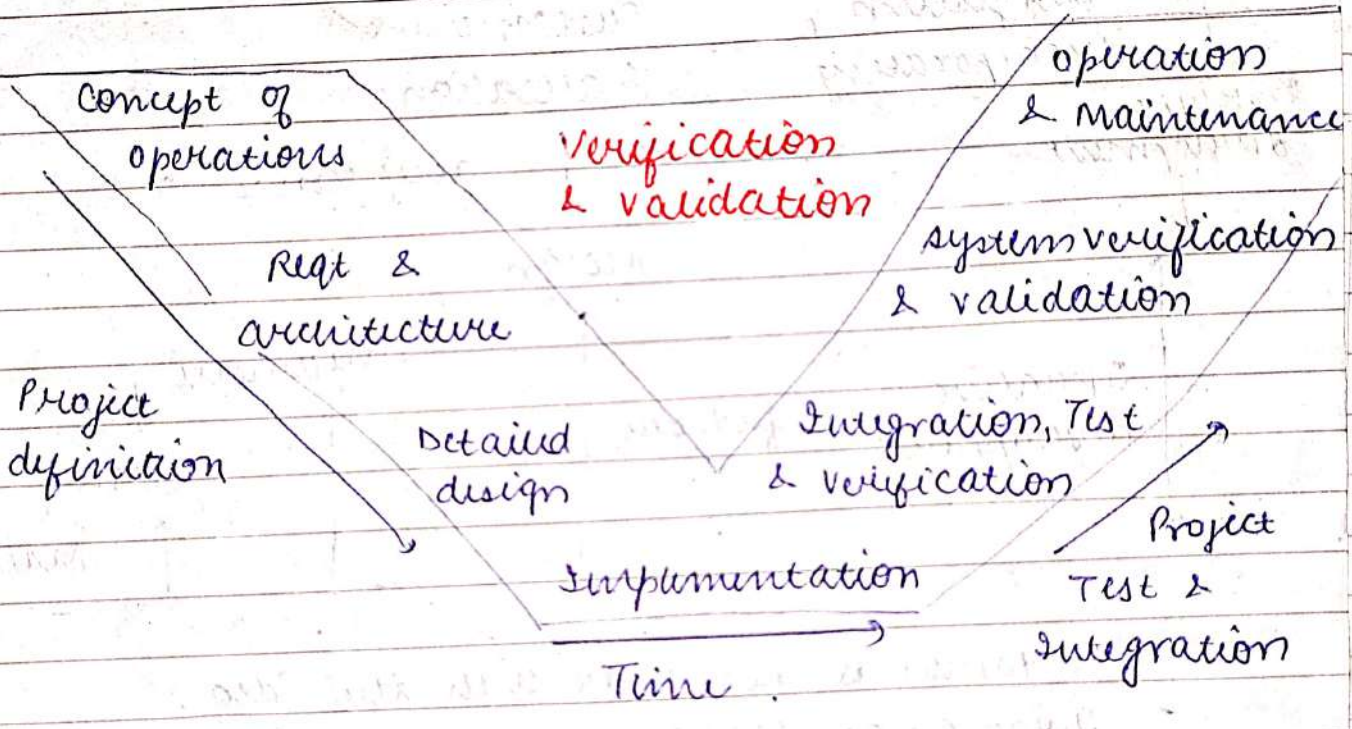


- Disadvantages - No phase overlapping
  - No intermediate delivery
  - Rigid (no changes)
  - less customer interaction

(V) → Most emphasis on testing

V-shaped Model. step by step we check & verify

- also known as verification & validation Model.
- Extension of waterfall Model.
- Testing is associated with every phase of lifecycle
- Verification Phase (requirement analysis, system design, architecture design, Module design)
- validation Phase (unit Testing, Integration system, acceptance testing)



Adv

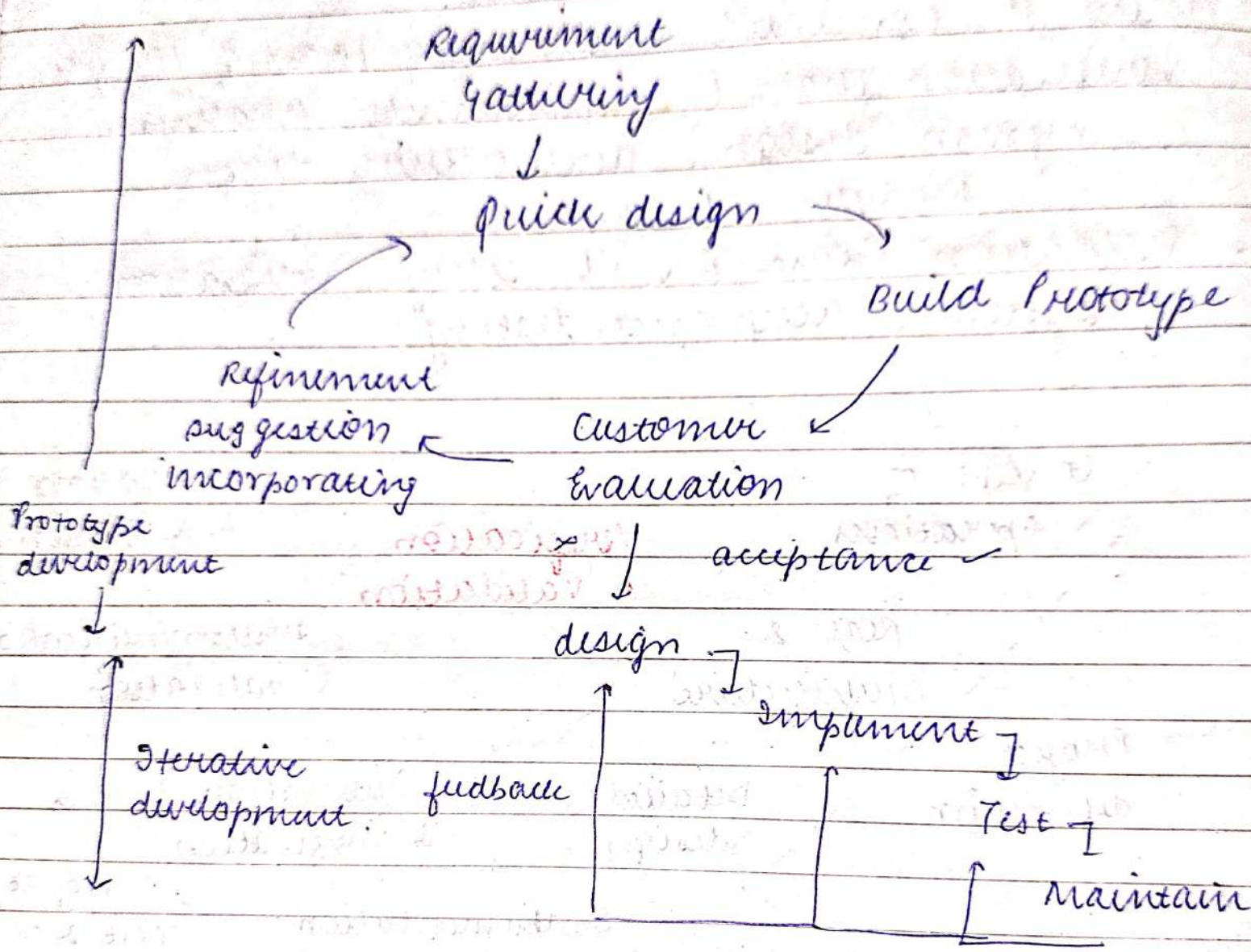
- Time saving
- Good understanding of project in beginning
- every component must be testable.
- Progress can be tracked easily
- Proactive defect Tracking

Disadv.

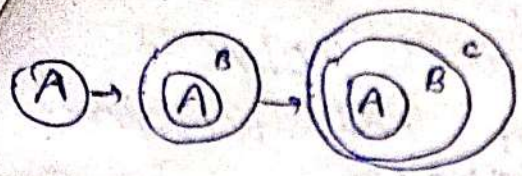
- No feedback so less scope of changes.
- Risk analysis not done.
- Not good for big / Object oriented projects.

dummy/sample.

Prototyping Model

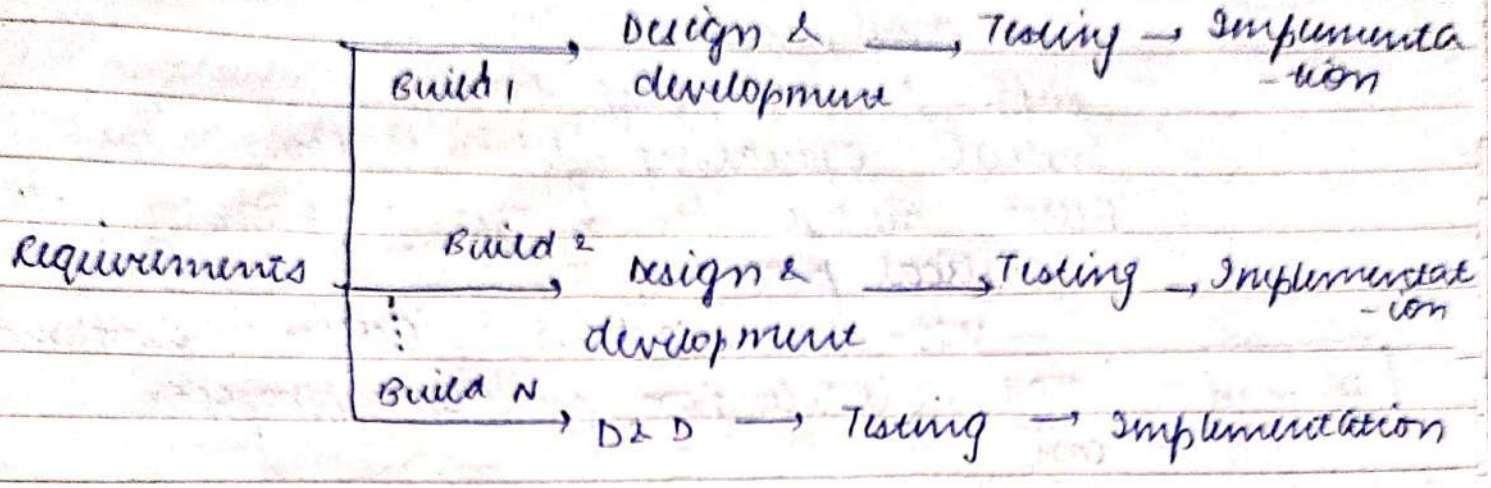


- Customer is not clear with the idea.
- Throw away Model
- Good for technical & requirement risks
- Increase in cost of development.



Initially all requirements r fixed

Incremental model.



- Module by module working
- Customer Interaction maximum
- Large projects
  - Early release Product Demand
  - Flexible to changes
  - Error chances r last.

Evolutionary model.

- Combination of iterative & incremental model of software development life cycle.
- Incremental model first implement a few basic features & deliver to customer. Then build next part & deliver again & repeat until desired system is fully realized.
- No long term plans r made
- Iterative model → main advantage is its feedback process in every phase
- also known as design a little, build a little, test & deploy a little model.
- suitable for projects with evolving & changing requirements
- allows for early delivery of basic working system

Baseline system : minimal system is developed with most critical features
- Iterative enhancement (add features & functionality). feedback

Adv -
- ~~customer requirements & clearly specifie~~
- Risk analysis is better
- supports changing envi.
- Initial operating time is less
- Better suited for large mission critical projects.
- User satisfaction & Better understan
- User satisfaction & Better understan

Disadv -
- not suitable for small projects.
- cost
- Highly skilled resources r required
- complex

Rough Requirements specification
↓
Identify core & other parts to be developed incrementally.
↓
Develop core part using iterative waterfall model
↓
Collect customer feedback & modify requirements.
↓
develop next identified features using iterative waterfall model
↓ all features complete
Maintenance

delivery of new version to customer

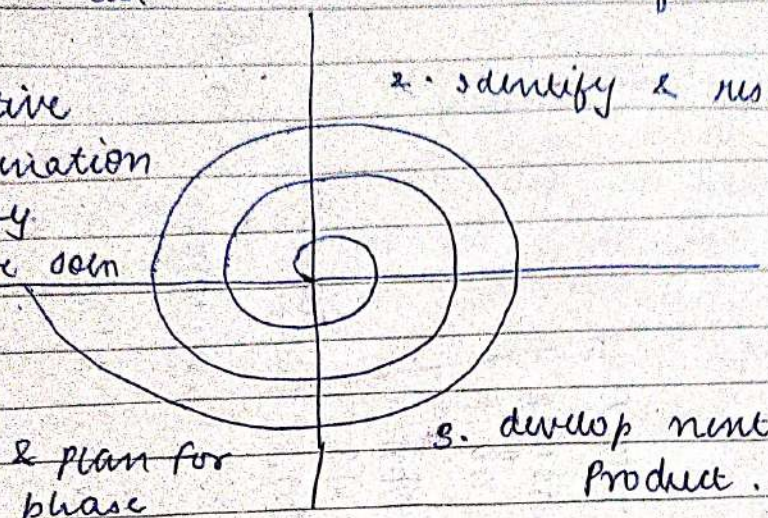
characteristics → Incremental, feedback driven adaptable, early delivery, complex Projects

Focus on Risk

Not for small Projects with well defined requirements.

Each cycle represents a phase in software dev

Spiral Model



- Risk Handling
  - Radius of spiral = cost.
  - angular dimension = Progress
  - Meta model (uses adv of all other models)

Adv - Risk handling
- flexible
- Large projects & complex
- Customer satisfaction
- Continuous Improvement

Disadv - Complex
- Expensive
- Too much risk analysis
- Time
- Alot of expertise is required

→ Iterative & Incremental.

→ doesn't follow linear path but repeats a series of cycle

- Prototyping is incorporated to manage uncertainties & gather user feedback

- Flexibility in accomodating changes & adjustments
- Continuous customer involvement is encouraged through development process

Classical Waterfall	Iterative Waterfall	Prototype Model	Incremental Model	Evolutionary Model	RAD Model	Spiral Model	Agile
Basic rigid inflexible not for real projects	Basic Problem is well understood + feedback	- user req not clear - costly - No early lock on requirements - High user involvement - Reusability	- Module by module delivery - easy to test & debug	- large Projects	Time & cost constraint - user at all levels - Reusability	Risks, not adv. for small Projects - No early lock on requirements - less experience can work Process divided into sprints	Flexible

Software Requirement.

- description of features & functionalities of the target system.
- description of what system should do.
- (RE) requirement engineering refers to the process of defining, documenting & maintaining requirements in the engineering design process.

4 step process, that includes -

1. Feasibility study
2. Requirement Gathering / elicitation
3. software requirement specification.
4. software Requirement validation

Tool support for Requirement engineering

- Observation reports (user Obs)
- Questionnaires (interviews, surveys & Polls)
- use cases - user stories
- Requirement workshops - mind mapping
- Prototyping - Role Playing

- Functional v/s Non Functional Requirements.

Requirements which r related to functional / working aspect of software.

Non Functional - expected characteristics of target software. (security, storage, config^n, Performance, cost, interoperability, flexibility, disaster recovery, accessibility

(*) Software Requirements specification (SRS)
- description of a software system to be developed

- lays out functional & non-functional requirements of software

- may include set of use cases that describe user interactions that software must provide to user for perfect interaction.

SRS structure

① Purpose - Intended audience - scope
- definitions - References.

User interfaces, system interfaces (servers, protocols), constraints, assumptions & dependencies, user characteristics.
(Pages, navigation)

System features & Requirements.
- Functional requirements. - Use cases
- External interface requirements. (3rd Party tie up)
- logical database requirement.
- Non functional requirements.
(safety, availability, reliability)

- Deliver for approval.

→

User Requirements.

1. easy & simple to operate. (eg Windows v/s Linux)
2. Quick response. (eg Paytm, Google Pay)
3. efficiently handling operational work.
4. Customer support.

User Reqt specification.
(URD) / USD is doc used in SE that specifies what user expects the software to be able to do.

- Contractual agreement.

(*) Data Flow diagram DFD.

- For analyzing existing as well as proposed systems.
- Focus on movement of data btw external entities & processes & btw processes & data stores.
- simple to learn & use
- Also called Bubble chart.

Y DFD? Provides an overview of

- what data system processes
- what Transformations r done
- what data is stored.
- Results Produced.

Graphical nature → Good comm. tool btw -

1. user & analyst.
2. analyst & system designer

DFD elements.

- source / sink (external entities) ☐
- Data flows (→) (←)
- Processes (○)
- Data stores. (═) (database) ↓ repository of data

External entities ☐
- either supply / receive data.
- don't process data.

Source → entity that supplies data to system
sink → receives data from system

↓	↑
Data stores	Data stores
Writing	Reading

External entity can't read / write to data store

Rules of Data flow.

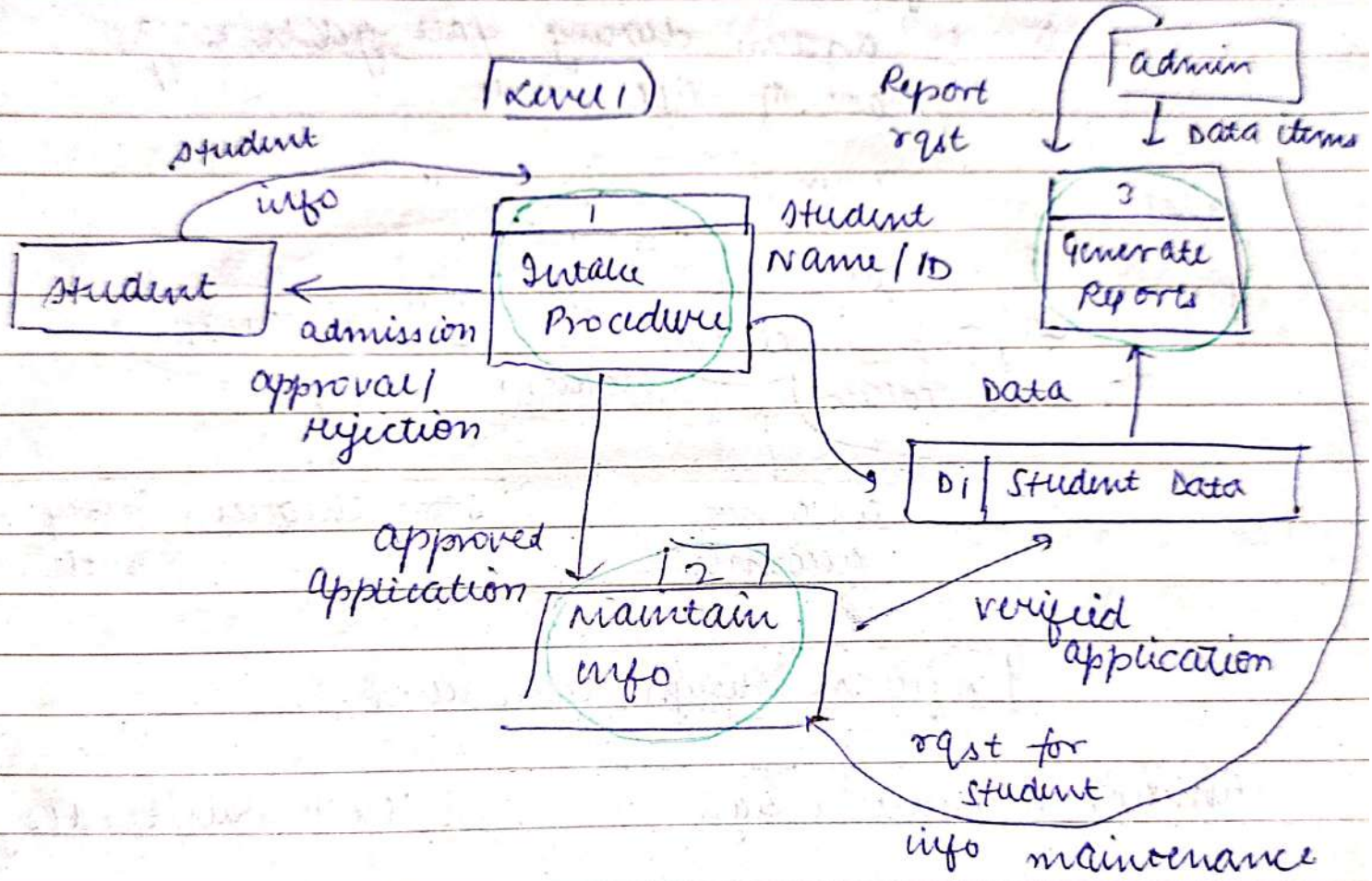
Data can flow from
- external entity to process ☐ ⇄ ○ ✓
- Process to external entity ○ → ☐
- Process to store & back
- Process to Process

Data can't flow from.
- External entity to external entity ☐ → ☐ ✗
- External entity to store ☐ → ═ ✗
- Store to external entity ═ → ☐ ✗
- Store to store ═ ⇄ ═ ✗

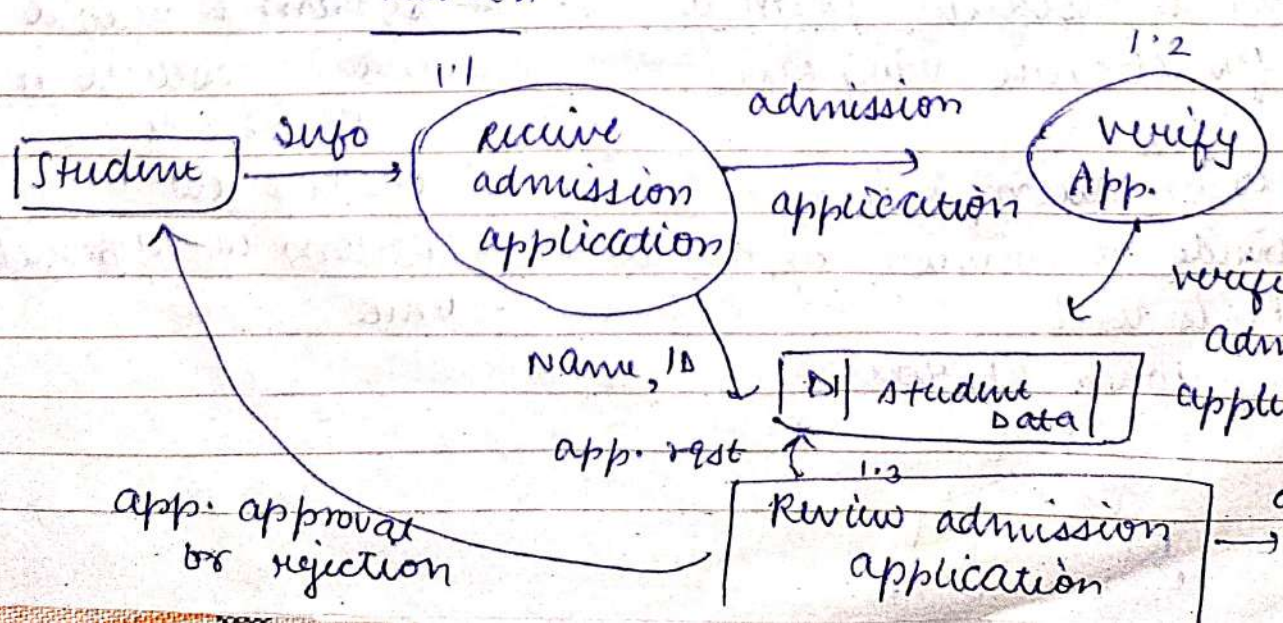
Levels of DFD 0 1 2

Level 0

Level 1



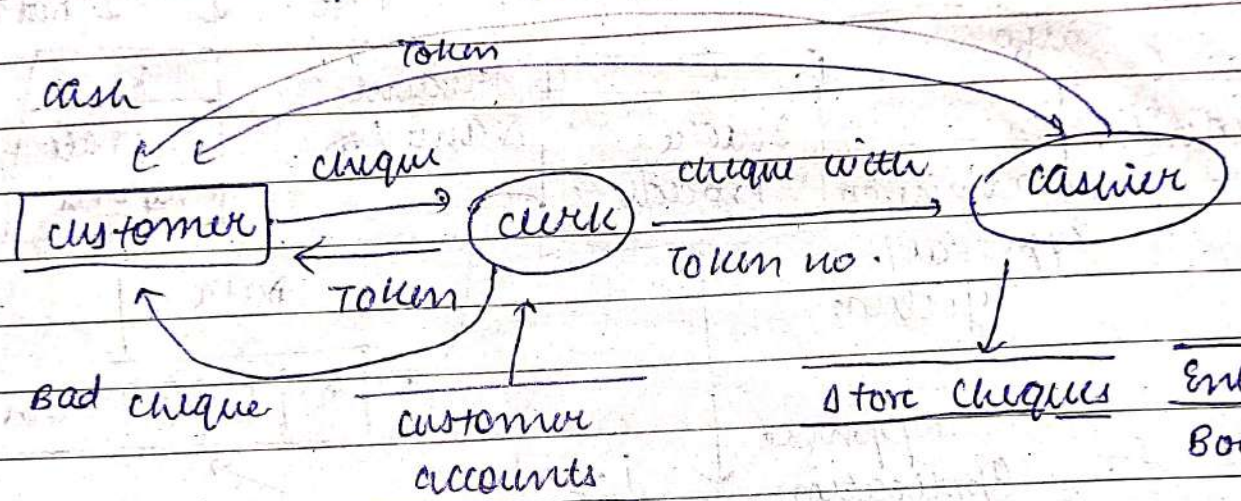
Level 2



Logical & Physical DFD

- Physical DFD - similar to document flow diagram specifies who does the operations ~~specify~~ specified by logical DFD

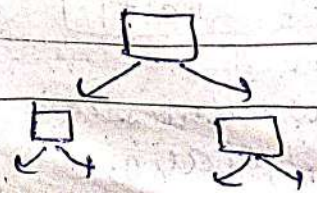
- Physical DFD may depict physical movements of goods.
- can be drawn during fact gathering phase of life cycle



software design approaches

Function oriented design	Object oriented design
- system is designed from a functional viewpoint.	- system is viewed as a collection of objects (ie entities)
- Top down decomposition	- Bottom up approach
- Divide & conquer approach	- UML
- DFD is used	

Top down approach



SPM (software Project Management)

- Main goal is to enable grp of developers to work effectively towards successful completion of project.

- Identify factors that make job complex
  changeability, complexity, uniqueness, Possibility of multiple solutions.

change in technology

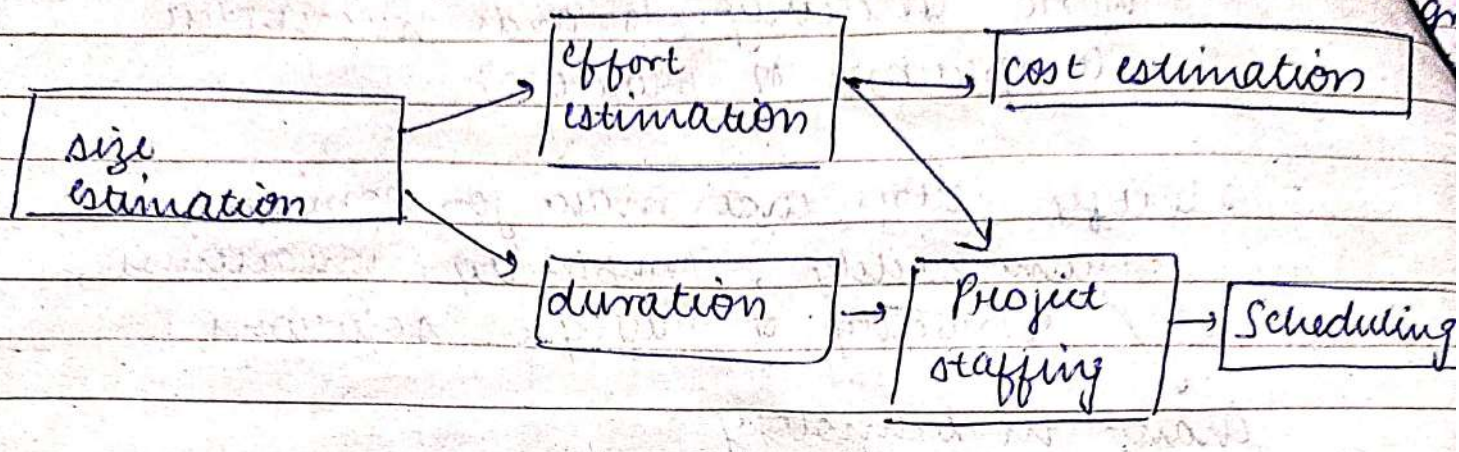
Job responsibilities of Project Management
- Planning - organizing - staffing - directing
- monitoring - controlling
- Innovating - Representing

Project Manager - managerial skills, Technical skills
Problem solving skills, coping, conceptual
leadership, communication

(*) Project Planning - Estimation (cost, duration, effort)
staffing (staff org., Plans)
scheduling manpower & other Resources
Risk Management
Miscellaneous - quality assurance plan
Configuration & Installation plans

effort → Person month.
duration → month.

Precedence ordering



Risk Management

Risk → uncertain event / condition, if occurs has a positive / negative effect on project's objectives

Risk Management plan – doc that a project manager prepares to foresee risks, estimate impacts & define responses to risks

Reactive v/s Proactive Risk Management strategies

– Reactive : tries to reduce damage of potential threats & speed an orgnz recovery from them but assumes that those threats will happen eventually

eg Insurance →
even after using helmet, seat belt / regulation

RM [access control → identify, analyse, prioritize

Proactive → identifies threats & aims to prevent the events from ever happening in first place

Types of Risks

① Business — Building product that no one wants / loosing budgetary commitments.

② Technical risk — concerned with quality, design, implementation, interface maintenance problems.

③ Project → Concerned with schedule, cost, resources, customer related issues

Risk assessment.

- Process to rank risks in terms of their damage.

- Determine avg probability of occurrence value for each risk.

- determine impact for each component based on impact assessment matrix (severity value)

- Risk assessment values r determined by multiplying scores for probability & severity values together

RA = P X S.

p(risk occurrence

Risk Control

specific actions to reduce risk event's probability of happening.

eg correct inspection & maintenance of car reduce likelihood of mechanical failures.

Risk Mitigation

set of actions to reduce impact of risk event.

eg airbags.

Risk avoidance, transfer, reduction, (Insurance / 3rd Party)

Risk Monitoring

Project estimation Techniques

estimation of various project parameters → Project planning activity

Parameters of Project to be estimated

- Project size
- effort needed
- duration
- cost.

Accurate estimation is imp. for resource planning & scheduling

Estimation Techniques

- Empirical - Heuristic - Analytical

expert judgment (1 expert) | delphi cost (team of experts)

(Empirical → makes guess based on common sense)

$a_2 * e^{d_1}$

COCOMO
Constructive cost model.

size → LOC line of code. (2000 to 100k
Function point.

Organic — 2 - 50KLOC small (Familiar)
Semi - detached. — 50 - 300KLOC medium
embedded — over 300KLOC Large (complex)

effort → Person Month.

1 Person Month → effort put by a person in a month.

Basic
Intermediate
complex

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

Organic $T_{dev} = 2.5(\text{effort})^{0.38}$ Months.
semi-detached $T_{dev} = 2.5(\text{effort})^{0.35}$ "
embedded $T_{dev} = 2.5(\text{effort})^{0.32}$ "

Q. Software project using semi detached mode LOC = 30k

effort $E = a_1(KLOC)\exp(a_2)$ PM.
$E = 3.0(30)\exp(1.12)$ LOC = 30k
$E = 135$

Duration $= 2.5(135)$

Productivity $= \frac{KLOC}{\text{effort}}$.

$= \frac{\text{effort}}{\text{duration}}$

- Good for quick, rough & early estimate of software costs.

- not account for differences in hardware constraints, personnel quality & experience, use of modern tools & techniques & so on.

- accuracy is limited

Intermediate → Basic + 15 cost drivers

Q Reliability = High KLOC = 30k. Embedded software

$E = a_1 (KLOC)^{a_2} \times (EAF)$

$a_1 = 2.8$ $b_1 = 1.20$.

$E = 2.8 (30)^{1.20} \times (1.15)$

↳ High value from table

$= 191$ PM.

$D = b_1 (E)^{b_2}$

$= 2.5 (191)^{0.32} = 13$ months approx.

$N = E/D = 191/13 = 15$ persons approx.

Merits

- can be applied to almost to entire software product for easy and rough cost estimation during early stage
- can be applied on software product component level → accurate estimation.

Demerits

- effort multipliers r not dependent on phases
- Product with many component is diff. to estimate.

Both modes consider a software product as single Homogeneous entity

complete →
Incorporate all characteristics of intermediate version with assessment of cost drivers impact on each step of SE process

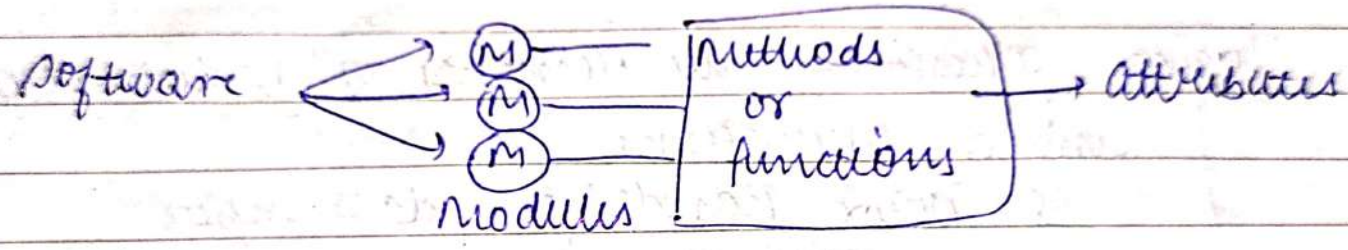
Verification	validation
1) are u building it right? (In every Phase)	1) Have u build the right thing? (when software is developed)
2) check whether an artifact conforms this previous artifact.	2) check final product against specification
3) Done by developer	3) Done by Testor
4) Concerned with phase containment of errors	4) aim to make final product error free
5) Methods Involves review, Inspection, unit testing & Integration testing	5) Involves system Testing
6) static & dynamic activities → only review.	6) only dynamic

③

Design phase transforms SRS document into a form easily implementable in some programming language.

SRS doc $\xrightarrow[\text{activities}]{\text{design}}$ Design docs

module consists of several fns & associated data structures.



Modularity is a fundamental attribute of any good design.

Decomposition of problem clearly into modules:

① modules r almost independent of each other

② divide & conquer principal

If modules r independent → modules can be understood separately.

– reduces complexity greatly.

Modules should display – high cohesion
low coupling

→

Cohesion within module.
↳ togetherness

Cohesion → measure of functional strength of a module.
→ cohesive module performs a single task / fn.

Coupling btw 2 modules → a measure of deg of interdependence / interaction b/w 2 modules.

logical cohesion → all elements of module perform similar operations
eg error handling, data input, data output.
↳ logically categorized elements r put together in a module

Coupling → How closely 2 modules interact / How interdependent they r

deg of coupling btw 2 modules depends on their interface complexity.

classes of coupling
- data
- stamp
- control
- common
- content

↓ degree of coupling

Fan-in : How many modules r coming in.
↳ Indegree

Depth : no. of levels of control
width : overall span of control.

UML
Unified modelling language

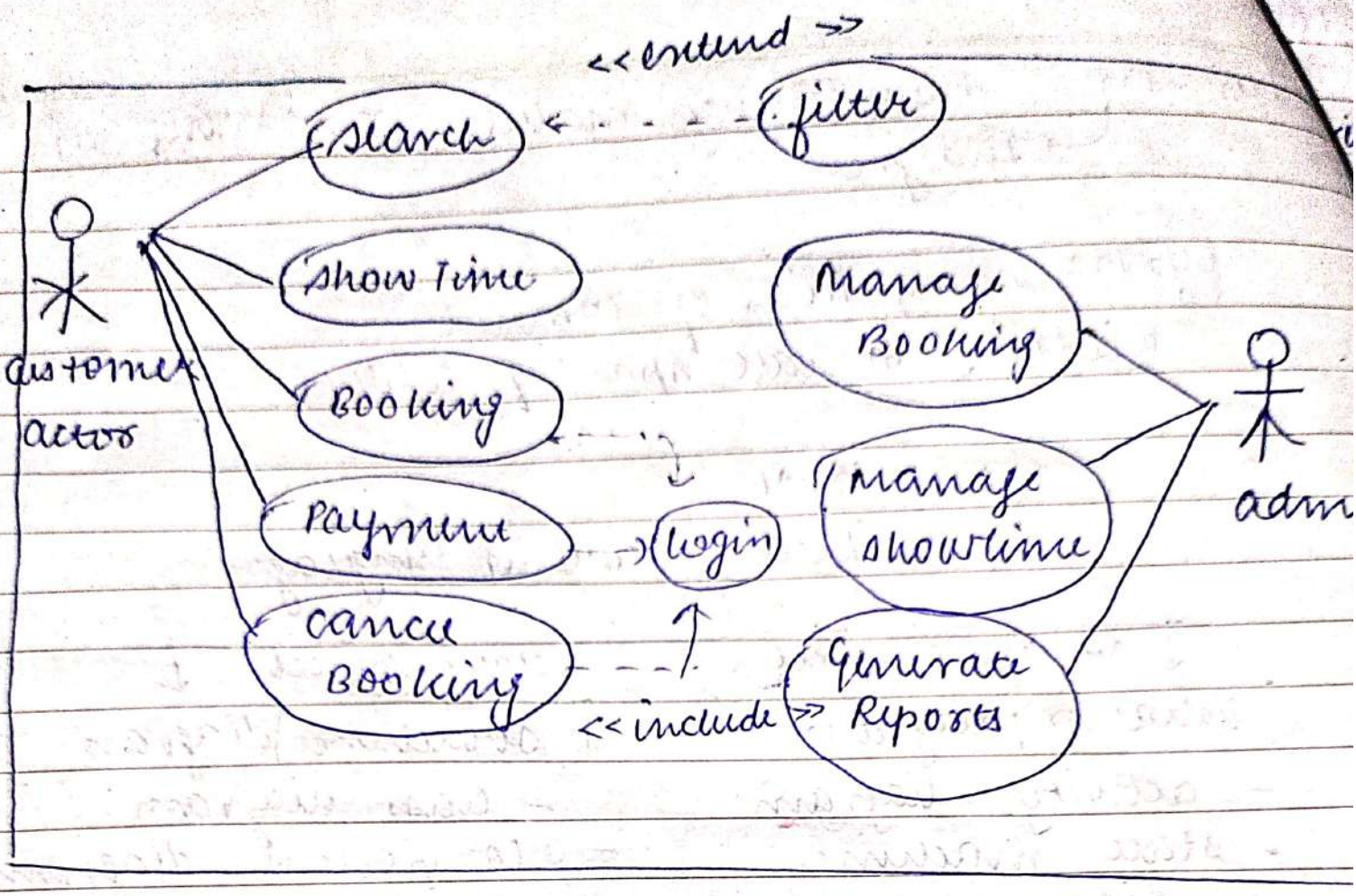
dynamic aspect	static aspect
Behavior diagram	structure diagram
- activity diagram	- class diagram
- state machine diagram	- component diagram
- use case diagram	- deployment diagram
	- object diagram

interactions
functionalities
state Transitions

CAR → Relation
component → attributes

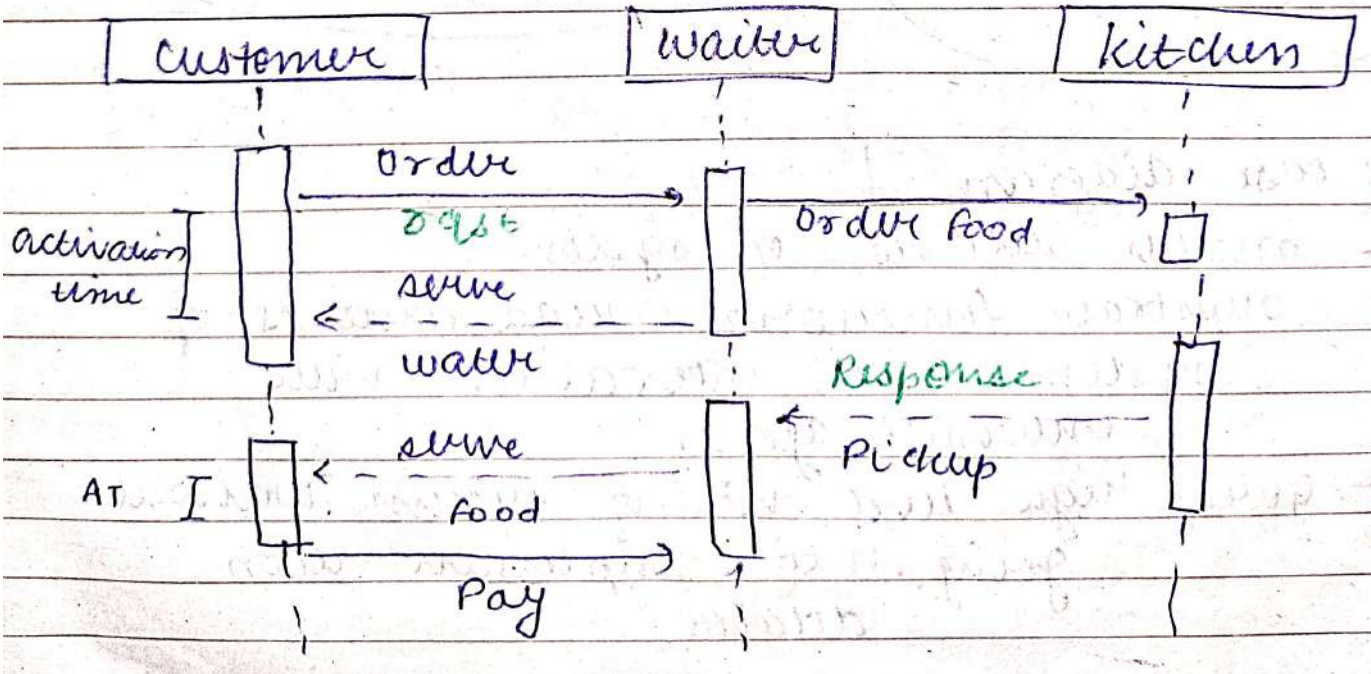
use case diagram

- models behavior of system.
- Illustrate functional requirements of system & its interaction with external agents.
- Gives high level view of system without going into implementation details.



Sequence diagram

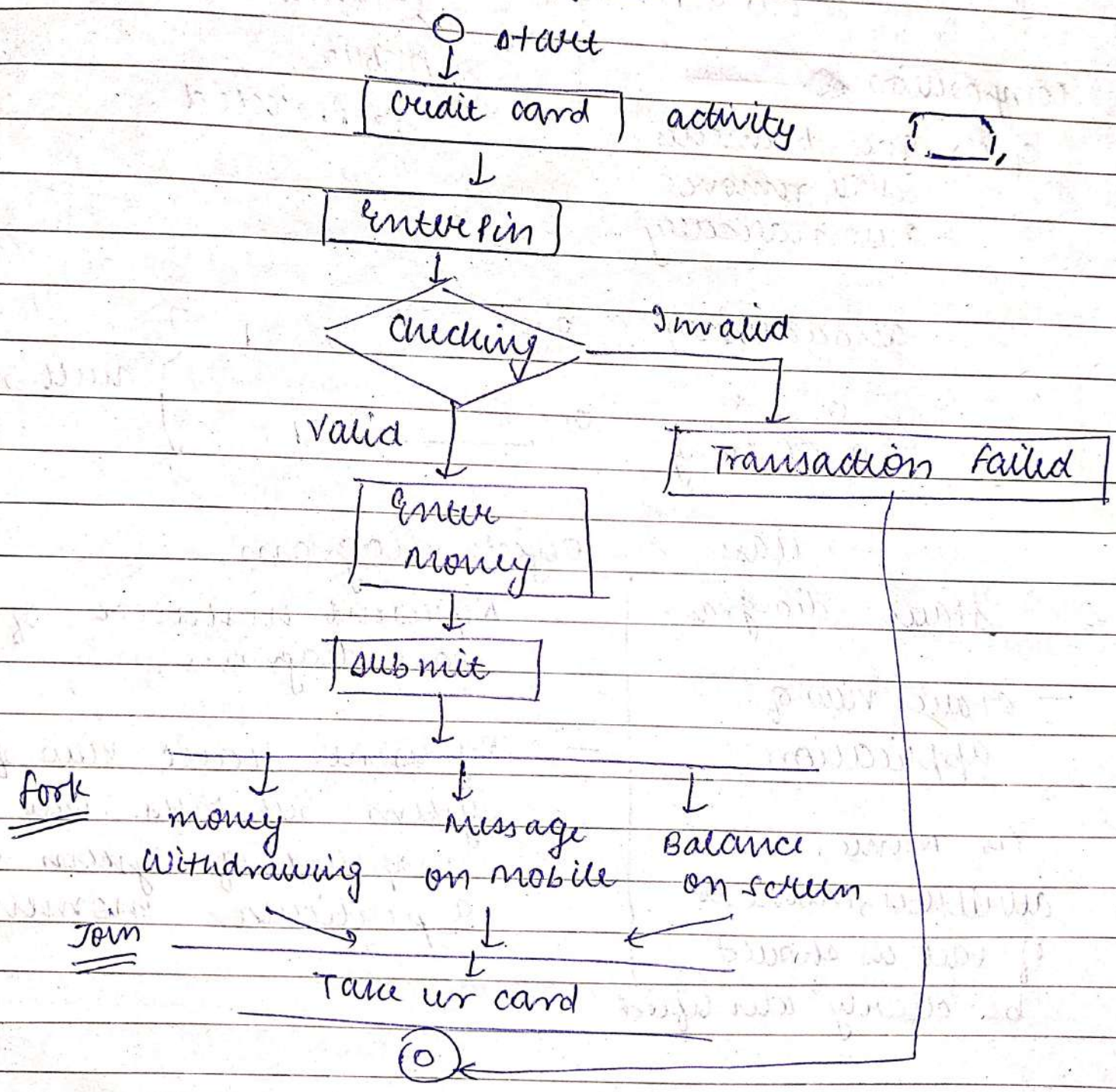
multiple objects. Events occur in timely ~~order~~ manner



Respons (- - -)
Request (——)

activation Time : Time taken after sending rqst to get response

Activity diagram



class diagram : each component is represented as cls.

account
⊖ acc_no - balance
+ deposit () + withdraw() + addloan ()
⊕ removeloan ()

Private

public

Protected.

association ———

aggregation ◇———
has a relation

composition ◆———
If cls goes branches will remove automatically

associativity 0...* ——— 1

0...* 0 or many

0...1 ——— 1

multiplicity

class & object diagram

static diagram	Represent instance of class diagram.
— Static view of application	— Represent static view of system but this view is snapshot of system at a particular moment.
- cls name, attributes, methods of each cls should be clearly identified	

Customer
- customer ID : string - name : string - mail : string
Placeorder() view Details ()

OTT_subscription
subscription ID: str Date : Date.
subscribe () unsubscribe ()

Measure of software characteristics that r measurable & countable
↑
Software Metrics.

Provides insights about quality, size, complexity & efficiency
size of software, cost to develop, bugs, when to stop testing

Types of metrics ① Product : characteristics of product such as size, complexity, design features, performance, efficiency, reliability & portability

② Process : describes effectiveness & quality of processes that produce software product
eg – effort reqd ② time to produce product
③ effectiveness of defect removal
④ No. of defect found during testing

③ Project : describes project characteristics & execution
→ No. of developers – cost & schedule
– Productivity.
↳ To determine size & complexity of code by TC

Token count : Symbol / keyword specifying algorathmic action is considered operator.
symbol representing data → operand.
Punctuation marks – operators.
var, const, labels → operands.
Operators → while, for, printf, {}, (), fn names & eof

(iii) Halstead software science Metrics.
$\eta = \eta_1 + \eta_2$ η = vocabulary.
η_1 → unique operators
η_2 → unique operands.

→ total occurrence of operators

$N = N_1 + N_2$ N = Program length

↳ total occurrence of operands

$N = 2 * LOC$ If program line has 1 operator & 1 operand

$V = N * \log_2 \eta$ (volume)

Program level (L) = $\frac{V^*}{V}$

value of L ranges btw 0 and 1.

L = 1 Program is written in highest possible level

Difficulty (D) = 1/level → (Total operators/2) * (operands / Total unique operators)

as volume ↑ Program level ↓ Difficulty ↑

$$E = \frac{V}{L} = D * V$$

Program vocabulary.

η = Total distinct operators & operands

Estimated Program length

$$N = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

eg :

Operators	Occ.	Operands	occurrence
main()	1	—	—
—	1	Pattern	1
for	2	line	3
==	3	found	2
!=	4	s	3
getchar	1	lin	1
()	1	c	5
&&	3	i	4

--	1	s	2
return	4	t	3
++	6	i	5
printf	1	j	3
>=	1	k	6
strinden	1	1	1
y	3	Manine	1
getline	1	\0	8
while	2	\n	4
{	5	strings "oud"	2
=	10	—	—
[ ]	9	—	—
,	6	—	—
;	14	—	—
EOF	1	—	—
$\eta_1 = 24$	84	$\eta_2 = 18$	$N_2 = 55$

Program vocabulary = $\eta = 42$

Program length $N = N_1 + N_2$
$= 84 + 55 = 139$

estimated length:
$N = 24 \log_2 24 + 18 \log_2 18 = 185.115$
% error $= 24.91$

Program volume $\Rightarrow V = 749.605$ bits
$V^* = 20.4417$.

estimated program level $= \frac{2}{\eta_1} \times \frac{\eta_2}{N_2}$
$= 0.02727$.

$$\text{Effort} = V/L = \frac{749.605}{0.02727}$$

$$= 27488.33$$

$$\text{Time } T = \frac{E}{\beta} = \frac{27488.33}{18} = 25.452 \text{ minutes}$$

Data structure Metrics

Gauge complexity & efficiency of DS used in a software

	Input	Internal
Software Planner	Program size, No. of developers in team	Model Parameters Constants Coefficients

Output
Est. Project effort Est. Project duration

Live variables → Var is live from beginning of a procedure to end of procedure.

→ live at particular statement only if it is referenced certain no. of statements before & after

$$\text{avg } \overline{LV} = \frac{\text{sum of count of live variables}}{\text{count of executable statements}}$$

$$\text{Module weakness} = WM = \overline{LV} \times \gamma$$

$$\text{avg life of var } \gamma = \frac{\text{sum of count of live variables}}{\text{No. of unique variables}}$$

evaluating flow of info / data through software system.

↑

Information flow Metrics

Analyze how data moves & interacts within system

If component has to do numerous discrete tasks it is said to lack cohesion.

Component : any element identified by decomposing a system into its constituent parts.

Cohesion : deg. to which a component performs a single function

Coupling : deg. of linkage btw one component to other in same system.

Basic IF Model:

① Fan in → count of no. of other components that can call / pass / control component A.

② Fanout → No. of components that r called by component A.

$$IF(A) = [Fan\,in(A) * FAN\,OUT(A)]^2$$

① Note level of each component in system design.

② For each component count no. of calls to that component (Fanin)

③ Count Fanout for each component. No. of calls from the component.

④ calc IF value

⑤ Calc level sum : sum of IF values for all components

system sum = sum IF values for total system design.

High fan in → Components lack cohesion.
High fan out → lack of cohesion
High IF → highly coupled components.

Sophisticated Model

diff. → defn of fan in & fan out.

a = no. of components that call A
b = no. of parameters passed to A from component higher in hierarchy
c = no. of parameters passed to A from components lower in hierarchy
d = no. of data elements read by A

fan IN = a + b + c + d

e = no. of components called by A
f = no. of parameters passed from A to components higher in hierarchy

g = no. of parameters passed from A to comp lower in hierarchy
h = no. of data elements written to by A

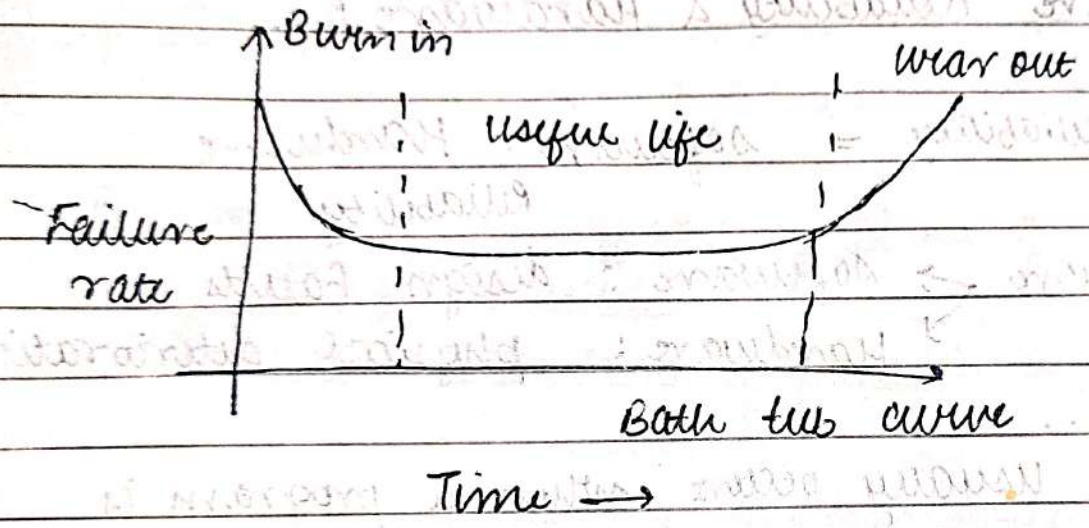
fan out = e + f + g + h

Probability of software functioning without failure under specified conditions for a period.

Software Reliability

Software becomes more reliable over time instead of wearing out.

It either works in given cnt. / not.

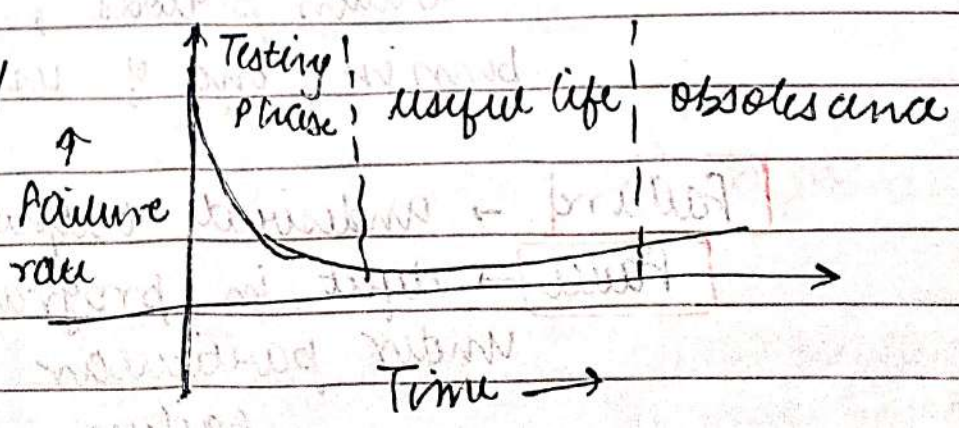


Burn-in : Failure rate is quite high initially & starts ↓ gradually as time progresses

Useful life Period : Failure rate is approximately constant

Failure rate ↑ in wear out phase due to wearing out / aging of components.

Software Reliability curve



Factors affecting ① Fault tolerance.
② error handling & recovery
③ Testing & validation ④ Redundancy

Software Reliability → Operational reliability
ability of a system / component to perform its required functions under stated conditions for a specific period of time

Software Reliability & Hardware R —

System Reliability = software + Hardware Reliability

src of failure → software : design faults
→ Hardware : physical deterioration

Failure usually occurs when a program is exposed to an environment that it was not developed / tested for.

Design reliability concept hasn't been applied to Hardware

software reliability tends to change continually during test periods.
Hardware reliability may change during certain periods, such as initial burn in / end of useful life.

Failure → undesired output
Fault → defect in program, when executed under particular conditions causes failure.

② **execution time** → time spent by processor in executing instructions of that program.

Calendar time → familiar time

Clock time → time elapsed from start to end of the program execution of running comp.

Failure behavior is affected by ① no. of faults in software being executed.
② execution envt. / operational profile of execution

Software Reliability Models.

consider factors ① fault ② fault Removal ③ Envt.

Reliability: probability that program won't fail during certain period under stated conditions

Basic Execution Time Model

To predict & estimate reliability of system / comp over time
assumed that failure may occur acc. to a non Homogeneous poisson process

Poisson Process (eg) : no. of calls expected in given time.

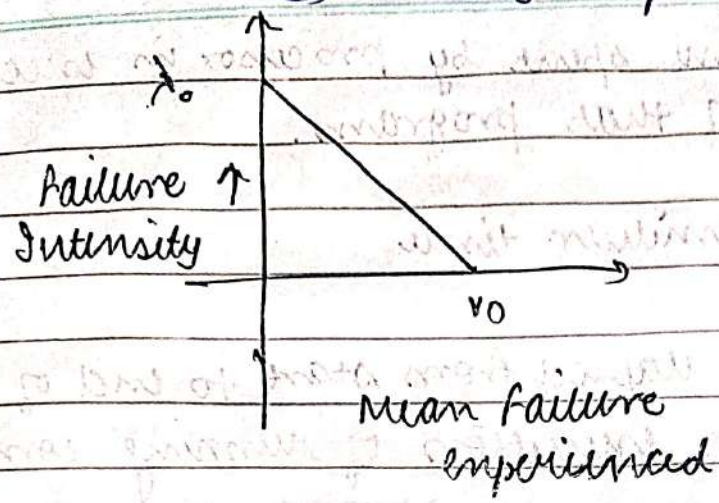
$$\lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{v_0}\right)$$

λ_0 → initial failure experienced

$\left(1 - \frac{\mu}{v_0}\right)$ → no. of failure experienced at given time

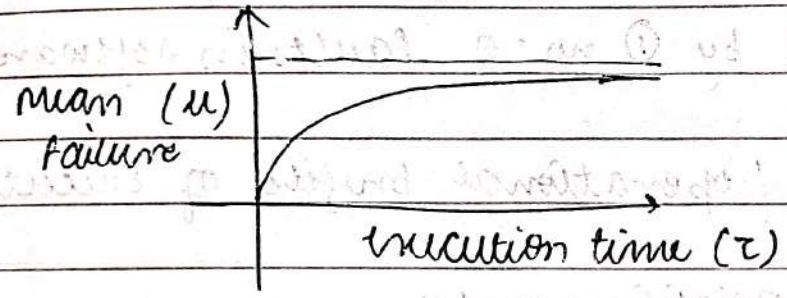
μ → expected no. of failure

assumptions ① constant failure rate.
② exponential distribution.
③ memoryless property ④ Homogenity



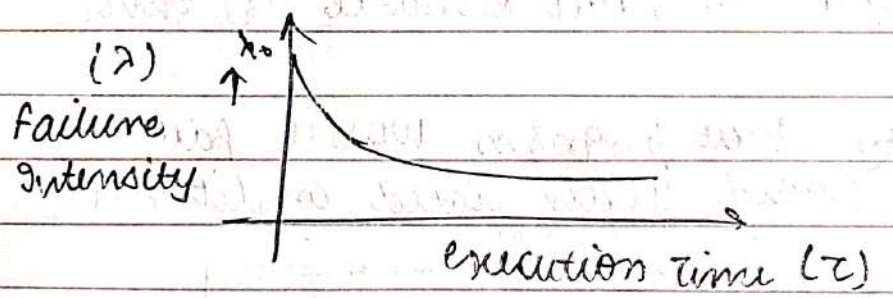
$$\frac{d\lambda}{d\mu} = -\frac{\lambda_0}{v_0}$$

-ve sign shows that negative slope.



$$\frac{d\mu(\tau)}{d\tau} = \lambda_0\left(1 - \frac{\mu(\tau)}{v_0}\right)$$

$$\mu(\tau) = v_0\left(1 - \exp\left(\frac{-\lambda_0\tau}{v_0}\right)\right)$$



$$\lambda(\tau) = \lambda_0 \exp\left(-\frac{\lambda_0\tau}{v_0}\right)$$

~~Failure~~

$$\Delta\mu = \frac{v_0}{\lambda_0}(\lambda_p - \lambda_f)$$

λ_p → Present failure intensity

λ_f → failure intensity objective

$\Delta\mu$ → expected no. of additional failures to be experienced to reach failure intensity objective

λ_0 → Initial failure intensity

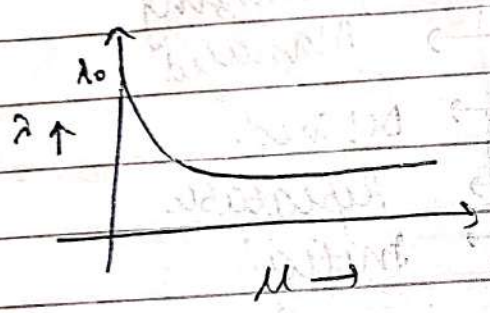
additional execution time

$$\Delta\tau = \frac{\nu_0}{\lambda_0} \ln\left(\frac{\lambda_P}{\lambda_F}\right)$$

Logarithmic Poisson Execution Time Model

Failure intensity fn : $\lambda(\mu) = \lambda_0 \exp(-\theta\mu)$

$\theta \rightarrow$ Failure intensity decay Parameter



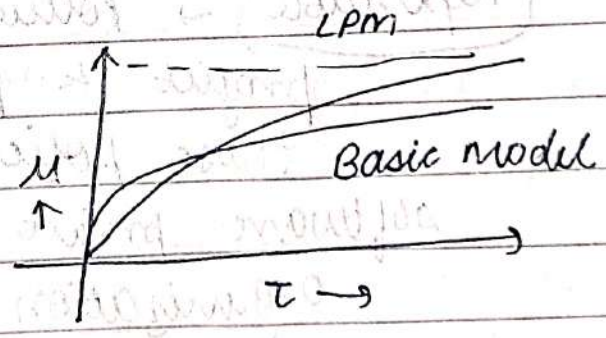
Accounts for change in failure rate over time & incorporates aging process

Logarithmic Poisson Execution Time Model

$$\lambda(\mu) = \lambda_0 \exp(-\theta\mu)$$

θ = Failure intensity decay Parameter

$$\text{Slope} \Rightarrow \frac{d\lambda}{d\mu} = -\theta\lambda$$



Calendar Time Component

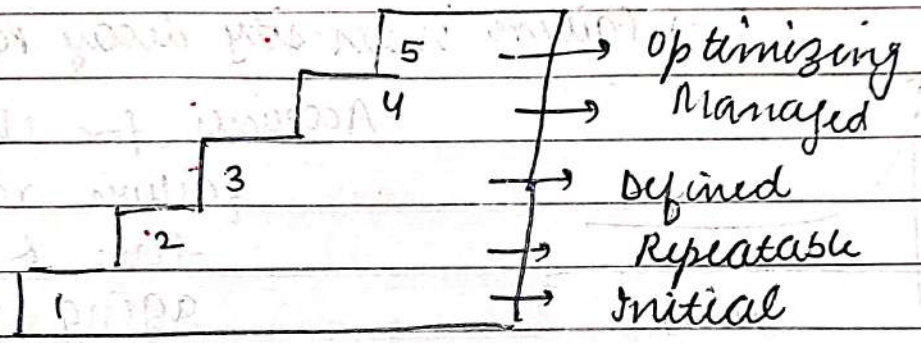
Relates execution time & calendar Time by determining calendar time to execution time ratio at any given point in time

Provides framework for accessing & improving org. soft. development processes.

(CMM) Capability Maturity Model.

Not a software life cycle Model.
Strategy for improving software process

To judge maturity of software process of an organization and to identify key practices reqd to increase maturity of these processes.



→ inconsistent & chaotic

(Initial) → lowest level
software process is unpredictable because it depends totally on current staff
- Std Practices & Processes r abandoned during crisis

(Repeatable) → policies for managing software project & procedures to implement those policies r established.
software project standards r defined & the organization ensure they r faithfully followed.
Has consistent project management process to track cost, schedule, functionality

(Defined) → standard process for developing & maintaining software across org is documented including both software engineering & management processes.

all projects across org use approved & tailored version of org standard process for developing, testing & maintainability.

Managed → Org. sets quantitative quality goals for both software products & processes. Productivity & quality r measured for imp. software process activities.

Optimizing → Improving performance through incremental & innovative methods. entire org. is focused on continuous process improvement.

organizations have means to identify weaknesses & strengthen process proactively with goal of preventing occurrence of defects.

ISO 9000 → Not industry specific
↳ Globally recognized quality management standard
↳ expressed in general terms
↳ series of standards is a set of documents dealing with quality systems used for quality assurance purposes.

series of 5 related standards that r applicable to wide variety of industrial activities.

Mapping ISO to CMM.

① Management Responsibility → ISO 9001 requires that quality policy be defined, documented understood & maintained

In CMM, management responsibility is primarily addressed in software quality assurance

② Quality system (including procedures & instructions)

③ Contract Review → to determine whether the requirements r adequately defined, agreed with bid & can be implemented.

④ Design Control → to control & verify the design established
Planning design activities, identifying input, output and verifying design

⑤ Document control

⑥ Purchasing : Purchased products conform to their reqt.

⑦ Purchaser supplied Product

⑧ Product identification & traceability

⑨ Process control

⑩ Inspection & Testing

Software Maintenance

Fixing things that break / wear out.

In software, nothing wears out, it is either wrong from beginning / we decide later that we want to do something diff.

Error correction, enhancement of capabilities
deletion of obsolete capabilities &
optimization.

→

Categories of Maintenance

① Corrective Maintenance

Refers to modifications initiated by defects in software.

defect can result from design errors, logic errors & coding errors

defects ← design, logic, coding

addressing & fixing issues identified post deployment

② Adaptive Maintenance

includes modifying software to match changes in ever changing envr.

Includes any work initiated as a consequence of moving software to a diff. hardware operating platform.

eg OS/Hardware updates.

③ Perfective Maintenance

improving processing efficiency / performance or restructuring software to improve changeability.

Refers to enhancements.

↳ Based on user feedback / new requirement

Modify

Problems → Program is changed by someone who didn't understand it clearly.

- High staff turnover within IT industry. Many systems r maintained by persons who r not original authors

④ Preventive Maintenance

Proactively making changes to prevent future issues / improve maintainability

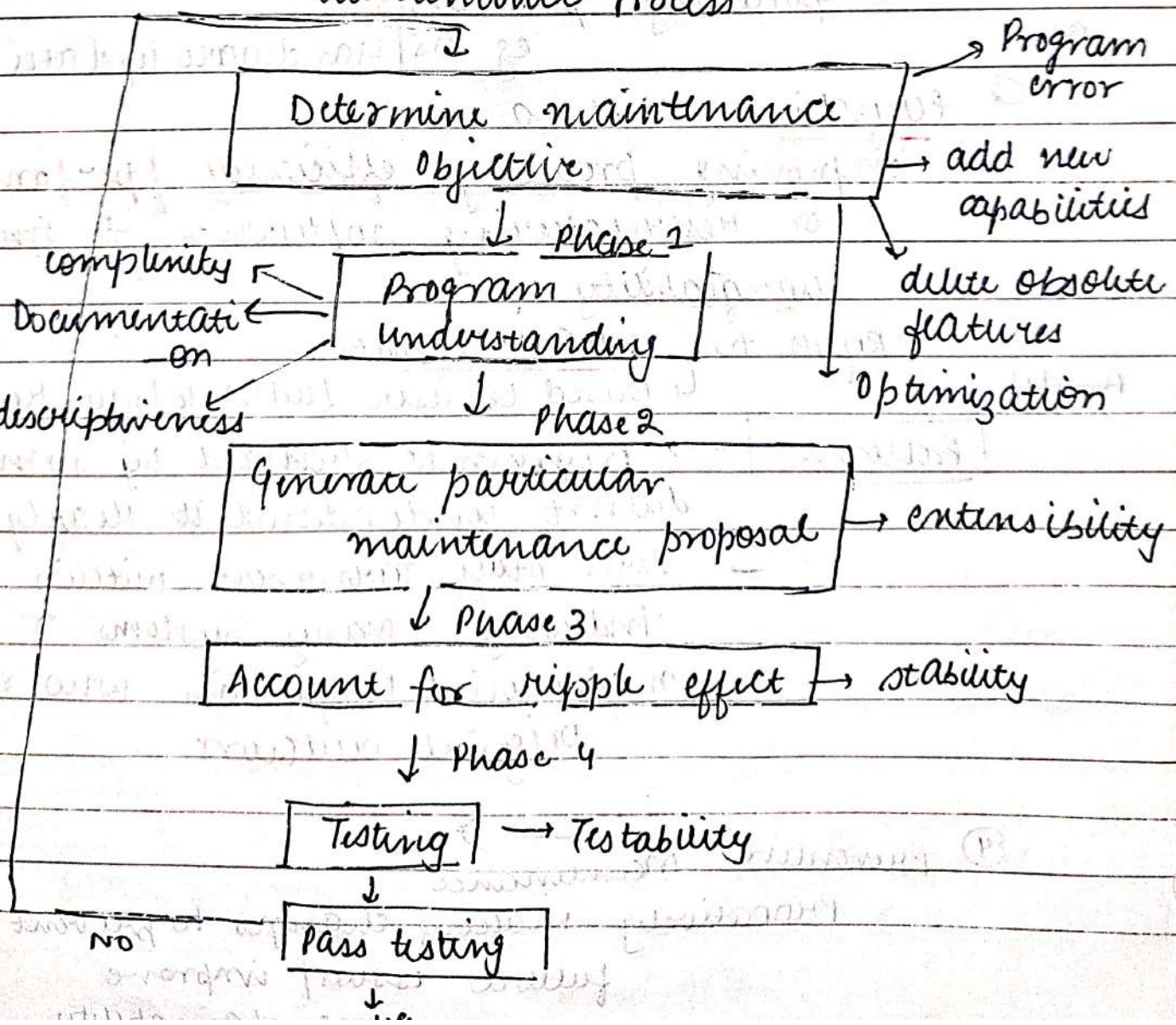
Soln to maintenance problems.

① Budget & effort allocation.
more time & resources should be invested in development - specification & design of more maintainable system rather than allocating resources to develop unmaintainable or diff. to maintain system.

② Complete Replacement of system

③ Maintenance of existing system

Maintenance Process



Program understanding : ① analyzing ② complexity ③ Documentation ④ self descriptiveness of program contribute to understanding

Generating Particular maintenance proposal
clear understanding of maintenance objective & program to be modified.

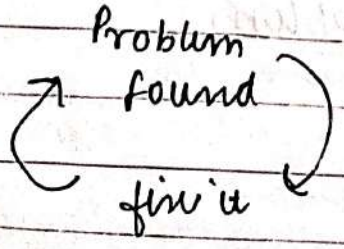
→ Ripple effect : effect of modification may not be local to modification but may also affect other portions of program.

→ Modified Program Testing : to ensure modified program has atleast same reliability level as before

→ Maintainability → More diff. to under → More diff. to maintain. ease with which software can be understood, adapted and/or enhanced

Maintenance modes.

Quick fix mode. → adhoc approach to maintaining a model. fire fighting approach, waiting for problem to occur & then trying to fix it ASAP.



Fixes would be done without detailed analysis of long term effects.

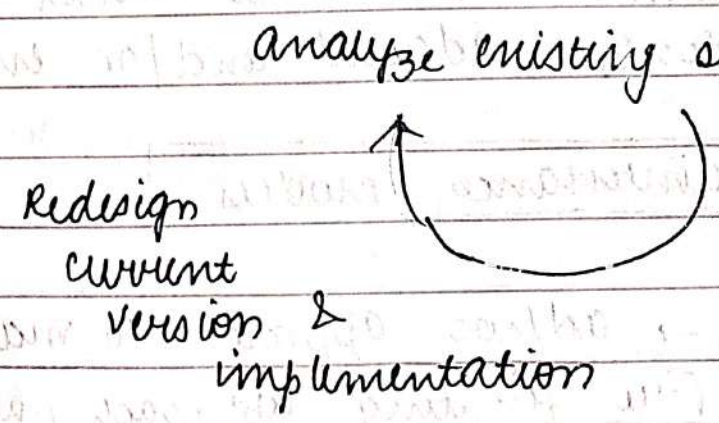
eg system is developed & maintained by a single person, he/she can come to learn system well enough to be able to manage without detailed documentation to be able to make ~~too~~ instinctive judgements about how & How not to implement change.

Iterative enhancement model.

changes in software system throughout its lifetime r an iterative process.

Invt. where reqt was not understood & a full system could not be built.

3 stage cycle
- analysis
- characterization of modifications
- Redesign & Implementation



In first iteration problem areas r identified, next iteration would specifically address problem.

→

Reuse Oriented Model

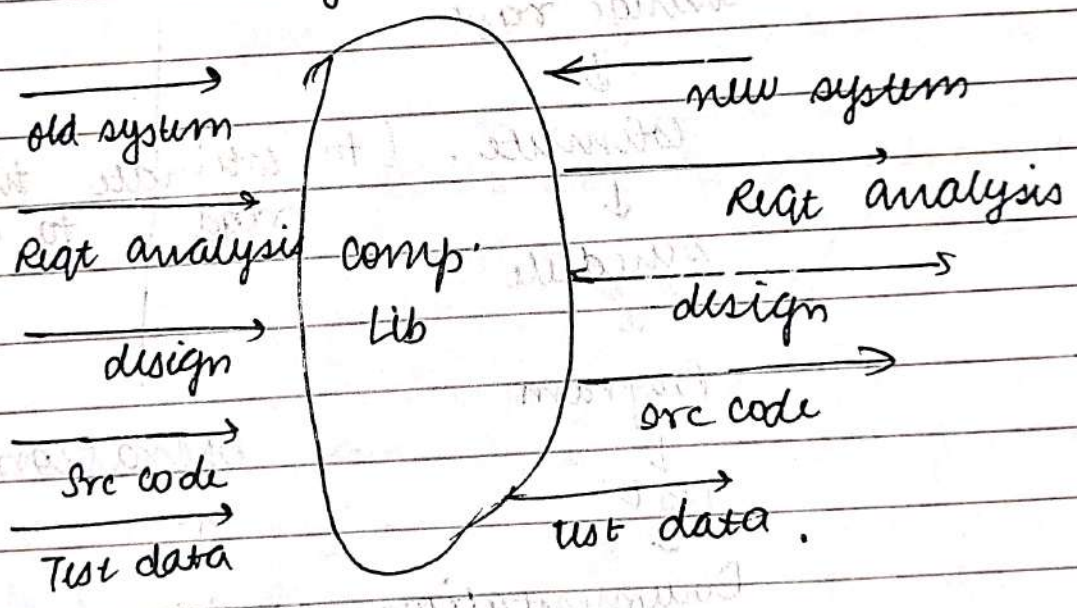
Principle : maintenance could be viewed as an activity involving reuse of existing program components.

4 steps ① Identification of parts of old system that r candidates for reuse

② understanding these system parts.

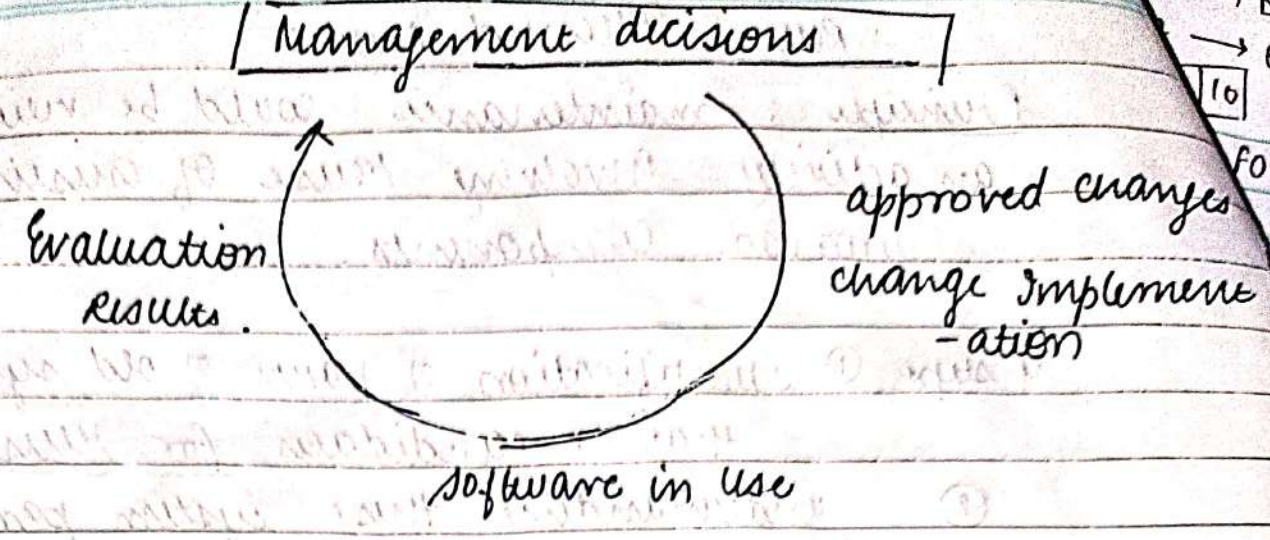
③ Modification of old system parts to new requirements.

④ Integration of modified parts into new system

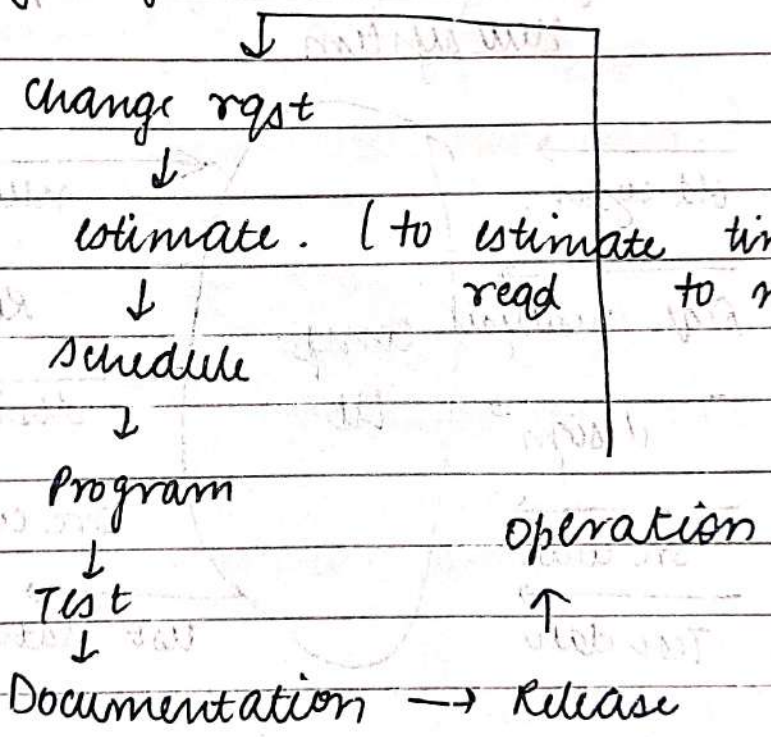


Boehm's Model : Model for maintenance process based upon economic models & principles.

Boehm's thesis → economic models & Principles could not only improve productivity in maintenance but also help understand process.



Taute maintenance Model.
very easy to understand & implement.



Regression Testing :

When we modify software we retest it.
This is called regression testing.
Retesting modified parts & ensuring that no new errors have been introduced into previously tested code.

selective retest technique → reruns all tests in existing test suite. Economical

Serves purposes such as

① ↑ confidence in correctness of the modified program.

② locate errors

③ preserve quality & reliability of software

④ ensure software's continued operation.

Development	v/s	Regression testing
① create test suites & Plans		① make use of existing suites & Plans.
② test all software components.		② retest affected components that have been modified.
③ Budget gives time for testing		③ doesn't give time
④ Perform testing once on a software Product.		④ Many times
⑤ under pressure of release date of software		⑤ crisis situations under greater time constraints

selective retest technique → Partition an existing test suite into subsets containing reusable, retestable & obsolete test cases.

① coverage techniques
② minimization technique
③ safe techniques [Inclusiveness, Precision]
[efficiency, generality]

Reverse Engineering

Process followed in order to find diff., unknown hidden info about software system

areas it is applicable to ① Program comprehension ② Re-documentation
③ Recovery of design approach & design details
④ Identifying reusable components.
⑤ Identifying components that need restructuring.

Software Re-engineering → taking ~~the~~ existing legacy systems & re-implementing, modifying / restructuring & enhancing them to make them more maintainable, ↑ Performance

system documentation → contains all facets of system including analysis, specification, design, implementation, security, error, Recovery

Reverse engineering
analyzing system's design, structure / code to understand its functionality, logic, architecture without having access to its original documentation or source code

uses — legacy system understanding
interoperatability

↑ (Puppet, Chef)

Configuration Management → identifying, organizing & controlling software & hardware components & changes to these throughout SDLC.

— Version control, change management, Baseline Management
(Git)

Benefits — ① Consistency, Traceability, Risk Mitigation.

Issue Tracking system → Jira, Redmine

Info flow metrics — Info flow complexity
Info flow rate.
Data dependency metrics

Significance — Security & Privacy analysis
Performance optimization
Understanding system interaction

significance of Halstead metrics

① Quantitative assessment
Provides quantifiable measures to access software complexity & size

② Predictive analysis
Estimation of
- Potential issues
- Effort
- Time reqd

③ Decision support.
- code optimizn
- modification
- refactoring based on metrics

④ Comparison

⑤ software Maintenance

Types Data structure Metrics — depth of Inheritance tree
High DIT indicates ↑ complexity
- Coupling Metrics
evaluate interdependence b/w modules / components
Fan-in, Fan-out.
- Cohesion Metrics
- Size & Complexity of DS

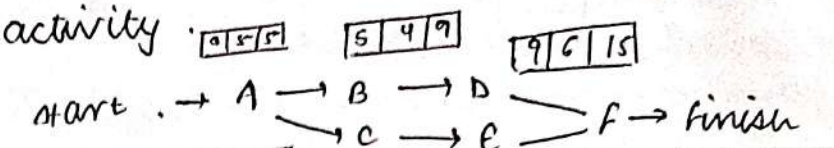
Imp
- Quality assessment
- Performance optimizn
- Maintainability analysis

~~Cast~~ Critical Path Method. (optimization) [PERT/CPM]

Q. Do CPM analysis & identify critical path of project.

start with independent activity

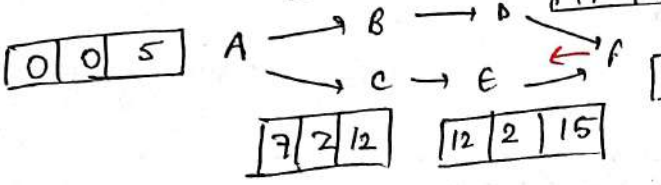
activity	Predecessor	Duration (in months)
A	–	5
B	A	4
C	A	5
D	B	6
E	C	3
F	D,E	4



ES	D	EF
A		
LS	SL	LF

[Forward Pass].

earliest start.
latest start.
slack time LF − EF
LS = LF − ~~Delay~~ duration



← [Backward Pass]

Critical Path: where nodes have slack time 0.

Q In PERT/CPM, merge event represents the completion of 2/+ events.

Program evaluation Revert technique

→ o → compute / Burst event → beginning

Verification

1. are you building it right? (In Process)
2. check whether an artifact conforms its previous artifact.
3. Done by developer
4. concerned with phase containment of errors.
5. Method involves review, Inspection, unit testing and integration testing.
6. static & dynamic activities

Validation.

1. Have you build the right thing? (After completion)
2. check final product against specification
3. Done by Tester
4. Aim is to make final product error free.
5. Involves system testing.
6. only dynamic

Testing tools
- Test Management (HP, ALM, Test Rail)
- Automation Testing (Selenium, JUnit)
- Performance Testing (JMeter, Load Runner)
- Security Testing tools (Owasp, Burp Suite)

Software Testing

- **Unit Testing (Coding)**
- **Integration Testing**
  - → Bang Bang
  - → Top down
  - → Bottom up
  - → Mixed.
- **System Testing**
  - Based on who is doing Testing
    - alpha (team)
    - beta (customers)
    - acceptance
  - Performance/ Non functional
    - volume
    - load
    - stress
    - security
    - configuration
    - compatibility
    - Recovery
    - Install
- **Regression Testing**
  - Done in maintenance Phase.

Q. A software was tested using error seeding strategy in which 20 errors: seeded when tested, 16 of seeded errors were detected

[S = Total errors seeded in the code.
s = Total seeded errors detected during testing.
N = total errors n = Total non seeded error.

Q) What is estimated no. of undetected errors?

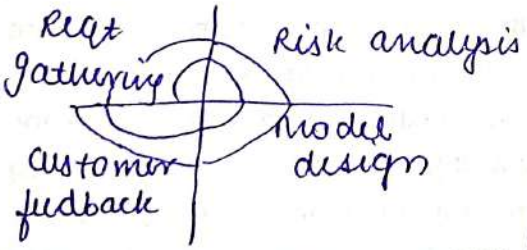
$\frac{n}{N} = \frac{s}{S}$ $\boxed{N = \frac{n \times S}{s}}$ undetected errors $= \frac{n \times S - (s)}{s}$

$= \frac{200 \times (20-16)}{16} = \frac{200 \times 4}{16}$

$= 50.$

Total errors $N = \frac{\cancel{200}^{50} \times \cancel{20}^{5}}{\cancel{16}\ \cancel{4}} = 250.$

Q. Project risk factor is considered in : spiral Model.



Q. No. of function points of a proposed system is calculated as 500. suppose the system is planned to develop in Java and LOC/FP ratio of Java is 50. Estimate the effort (E) required to complete the project using effort formula of basic COCOMO given below

$\boxed{E = a(KLOC)^b}$ assume $a = 2.5$ $b = 1.0$

size estimation → LOC / FP $\frac{LOC}{FP} = 50.$ $\frac{LOC}{500} = 50$ $LOC = 25000$

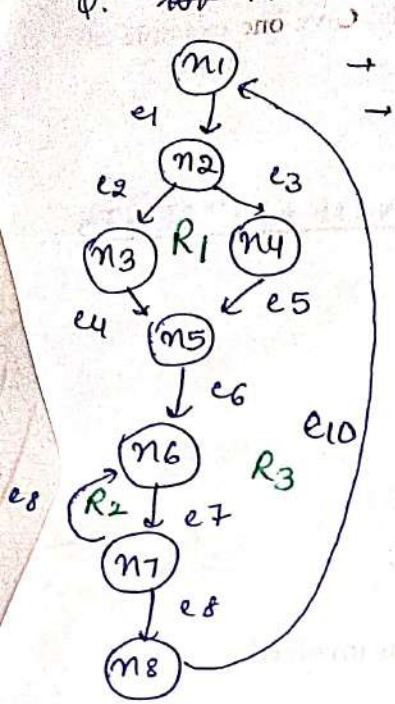
↓

effort → cost, schedule time.

- simple (organic) upto 50k
- semi detatched 51k - 300k
- Embedded Over 300k

Cyclomatic complexity

Q. ~~wt~~ Find the cyclomatic complexity of following software code.



→ no. of bounded regions +1 R+1

→ no. of edges − no. of vertex + 2. E−V+2

→ Predicate nodes +1.

3 + 1 = 4.
↳ Bounded regions.

- $n_1\ n_2\ n_3\ n_5\ n_6\ n_7\ n_8\ n_1$
- $n_1\ n_2\ n_4\ n_5\ n_6\ n_7\ n_8\ n_1$
- $n_1\ n_2\ n_3\ n_5\ n_6\ n_7\ n_6\ n_7\ n_8$

Q2. Which is white box testing technique? Tester should have knowledge of source code. ∴ indepent paths r found using cyclomatic complex. check all test cases.

- Path coverage] white box.
- equivalence class testing → divide test data into groups
- cause effect graphing → Relation btw cause & effect. eg username & Password.
- state based testing
  ↓
  state Transition
  Multiple inputs r provided (positive & -ve) and output is checked.

} Black box testing.

Design phase transforms SRS document into a form easily implementable in some programming language

SRS —design activities→ Design DOC

Module → fns + data structures associate

Modularity − fundamental attribute of of any good design.

Decomposition of a problem clearly into modules: ① modules r almost independent of each other. (Least dependency)

② divide & conquer principle.

modules should display − high cohesion (within module)
− low coupling (btw 2/ more modules)

Cohesion: measure of functional strength of a module.
- performs a single task/ function. (Intra Module)

Coupling between 2 modules: Measure of degree of interdependence or interaction b/w 2 modules.

Logical cohesion → all elements of module perform similar operations. eg: error handling, data input, data output.
when logically categorized elements r put together into a module.

Procedural → same algo. sequential → O → O → O
sequentially related elements r grouped

Classes of coupling:

data
stamp
control
common
content.

degree of coupling ↓

Fan-in: indicates how many modules directly invoke a module.
↳ Indegree

Fan-out: outdegree.

Unit Testing - Level 1 testing White box testing.
↳ checking small pieces of code to ensure that the individual parts of program work properly on their own.
→ unit tests are used to test individual blocks (units) of functionality.
- done by developers. eg car → engine, fuel, ignition, body etc.
- detects defects early in development cycle

Integration Testing: to evaluate compliance of a system/component with specified functional requirements.
- occurs after unit testing and before system testing.

Types: - Big bang - Mixed (sandwich) - Top down - Bottom up
- Testing while integrating.
- White box testing

↓ Highest priority module or unit is tested first.

System Testing: validates complete & fully integrated software product. To evaluate end-to-end system specifications.
Black box testing. Matching with SRS document.

categories
- α (who has made the Product)
- β (friendly customers)
- acceptance (client)

→ Functional
→ Non Functional req. (stress, load, scalability)

Types of system Testing : smallest component is tested by development team. (Unit testing)
Conducted on a complete integrated system to evaluate compliance

Non functional

① **Performance testing** : carried out to test the speed, scalability stability and reliability of software product/application

② **Load Testing** : to determine behavior of system / software product under extreme load.

③ **Stress Testing** : to check robustness of system under varying loads.

④ **Scalability Testing** : to check performance in terms of its capability to scale up / down the no. of user request loads.

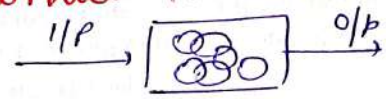
⑤ **Security Testing** : (SQL injection, active passive attack, denial of service attack) response ?

White box Testing → also called clear box testing, structural, glass box testing.

↳ can be applied at unit, integration & system levels of software testing process.

Tools used - Veracode, Nunit, CPPunit, RCUNIT.

Tests the internal structure & working of an application.



White box test design techniques : ① Control flow testing ② Data flow testing ③ Branch testing ④ statement coverage ⑤ Decision coverage ⑥ Path testing.

White box testing	Black Box testing
① done by developers	① Test engineers
2) what software is supposed to do, also aware of how it does	2) what but not how.
3) understanding of programming language is must.	3) No need.
4) look src code & test logic of the code.	4) ~~No need to~~ verify functionality based on req specification.
5) should know internal design of code.	5) No need to know internal design of code.

6) Test design Techniques :
control flow Testing, Data flow, Branch, statement coverage, decision coverage, Path testing

7) applied mainly at unit testing level but now in integration, system level also

decision table testing, all pairs testing, equivalence partitioning, Boundary value analysis, cause effect graph.

7) can be applied virtually to every level of software testing : unit, integration, system & acceptance.

Statement coverage Technique. – used to design white box test cases.

- Involves execution of all statements of source code atleast once.
- used to calculate total no. of executed statements in the source code out of total statements present in source code
- Covers dead code, unused code & branches.
- white box testing.

eg largest of 3 no.'s
- o/p kitni statements ko execute karne ke baad ai h.
- Max lines should be covered.

Condition coverage Techniques. – used to cover all the conditions
- also called Predicate coverage in which each one of the boolean expression have been evaluated to both True and False

eg
```
read a, b, c;
if (a == 0 || b == 0){
    print 1;
else if (c == 0 && d == 0)
    print 2;
}
```

Test case 1: a = 0 b = 0 c = 0 d = 0

print statements executes → $\frac{1}{4}$ .25% cover

eg a = 1 b = 0 c = 0 d = 0.

F OR T = T (1)

coverage = $\frac{2}{4}$ = 50%.

eg. a = 1 b = 1 c = 0 d = 0.

F || F = F.

T && T = T (2)

coverage = $\frac{4}{4}$ = 100%.

Data flow Testing – white box testing.

Focuses on – in which statement variables r defined
– in which statement variables r used.

- designs test cases that cover control flow paths around variable definitions & their uses in the modules.

```
1. read a,b,c;
2. if (a>b)
3.     x = a+1
4.     print x;
   else
5.     x = b-1
6.     print z;
```

Variables	defined	used in statement.
a	1	2, 3
b	1	2, 5
c	1	NA
x	3, 5	4
z	NA	6.

advantages – variable that is declared but never used within the program. (c)

- variable that is used but never declared. (z)
- variable that is defined multiple times before it is used. (x)
- Deallocating a variable before it is used.

Boundary value Testing : Black box testing techniques r helpful for detecting any errors / threats that happened at the boundary values of valid / invalid partitions rather than focusing on the center of input data.

- To design test cases. (min max).

efficient minimal test cases

Test case 1 : assume speed of car is taken from 40 to 80 to get the best fuel efficiency

Invalid case	Valid output.	Invalid.
(Min val − 1) = 39	(Min, +Min, Max, −Max) 40, 41, 79, 80	(Max value + 1) 81.

Software Maintenance – to modify & update software applications after delivery to correct errors and improve performance

Inclusive activity that includes: error corrections, deletion of obsolete capabilities, enhancement of capabilities, optimization

Types : ① Corrective 2) Adaptive 3) Preventive 4) Perfective.

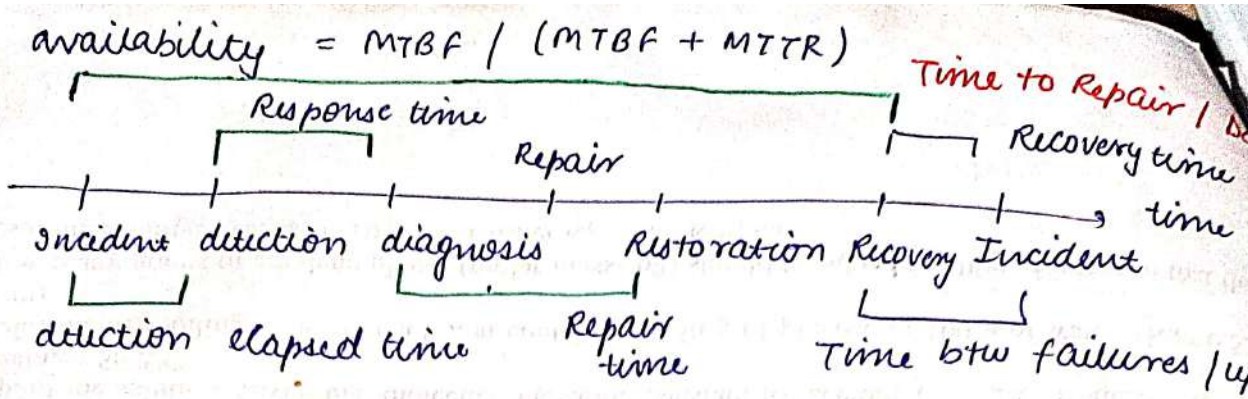
Maintenance Metrics : ① Mean time btw Failure (MTBF) : avg time available for a system / component to perform its normal operations btw failures. (UPTIME)

* MTBF = (sum of operational time / total no. of failures)

② Mean time to Repair (MTTR) – avg time reqd to repair a failed component DOWNTIME

* MTTR = (sum of downtime periods / total no. of failures

availability = MTBF / (MTBF + MTTR)



Q A machine that runs for 24 hrs. During that time, it failed twice & each time it took an hour to get it back up & running. FIND MTBF and MTTR.

ans: $2 \times 1 = 2$ hrs downtime. MTBF = $(24-2)/2 = 11$

MTTR = $2/2 = 1$ hrs. (↓ Operational hrs)

To Repair.

$A = \frac{11}{11+1} = 91.6\%$

Reverse engineering → helps to improve understanding of the underlying source code for maintenance & improvement of software. — Redocumentation of legacy systems.

code → module specification → design → Requirement specification.

- Used by malware developers to find vulnerabilities in OS to build virus that can exploit system vulnerabilities
- Cryptanalysis to find vulnerabilities in substitution cipher, symmetric key algo.

Computer aided software engineering (CASE) — software tools used to design & implement applications.

- For developing high quality, defect free & maintanable software. To make qualitative ~~too~~ softwares

Planning & req : Flow chart, UML, eclipse, cucumber SoapTest, Bugzilla

Performance Testing. Non Functional testing. can determine how a system performs in terms of responsiveness & stability under particular workload.

measures - stability, speed, scalability & throughput of your product.

① Load Testing : evaluates product's ability to perform under increased load.

eg Maximum capacity of web server

stress testing : used to understand upper limits of capacity within the system. eg : capacity of website during peak traffic hours.

Volume testing → ability of dbms to handle a large amount of data.

endurance : evaluates system's performance over long time under normal / expected load conditions.

Regression Testing : verifies that changes made to system, such as bug fixes / new features don't impact previously working functionality.

→ Full Testing entire application.
Partial : testing only parts that were affected.

Software Testing

Process of testing software product. satisfied users, lower maintenance cost, more accurate & reliable results.
→ demonstrating that errors r not present.

Error → mistake, syntax error / misunderstanding specifications, logical errors.

error may lead to one/more faults. Representation of error is fault. Failure occurs when a fault executes.

Test case : describes input description & expected output description.

Verification : Process of evaluating a system / component to determine whether the products of a given development phase satisfy the conditions imposed at start of that phase.

Validation : evaluating system / component during / at end of development process to determine whether it satisfies the specified requirements.

Acceptance Testing : When software is developed for specific customer. Tests r conducted to enable customer validate all the requirements. Conducted by end user / customer

α tests → done ~~on~~ at developers side by customer

β → conducted in real environment.

Functional testing : testing is based on functionality of program
↳ involves only observation of output for certain input values
↳ ignore internal structure of code.
↳ Black box testing.

Boundary value analysis : Input maybe on boundary, below/above boundary. eg 1–100 input, boundary values: 1, 2, 99 and 100

Robustness Testing extension of boundary value analysis.
If ~~input~~ extreme values r exceeded with a value slightly greater than maximum & a value slightly less than minimum.

Worst case Testing → divide input data into grps/classes to ↓ no. of test cases while ensuring coverage

Equivalence class Testing Input domain is partitioned into finite no. of equivalence classes such that one can assume that test of a representative value of each class is equivalent to test of any other value.

If one test case detects an error all other in class would be expected to find same error.

eg : determination of roots of eqn
- Not quad eqn
- Real roots $b^2-4ac>0$
- Imaginary roots $b^2-4ac<0$
- equal roots $b^2-4ac=0$

Δ
- equilateral
- Isosceles
- scalene
- Not a Δ

↓ redundancy in test cases
ensure cases r representative of entire class of i/ps

$\langle x, y, z \rangle$ $x=y=z$

$\langle x, y, z \rangle$ $x=y, x \neq z$

Input domain based classes : $\{x : x<1\}$ $\{x : x>100\}$ $\{x : 1 \le x \le 100\}$ $\{z : z<1\}$

Decision Table Based Testing : For describing situations in which a no. of combinations of actions r taken under varying set of conditions.

Condition stub & action stub.

systematic representation
compact test cases

Decision table in which all entries r binary r called limited entry decision tables.

c_1: x, y, z are sides of Δ	N	Y							
c_2: x = y		Y				N			
c_3: x = z		Y		N		Y		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	X			

Cause effect Graphing : one weakness of boundary value analysis & equivalence partitioning is that these don't explore combinations of input circumstances. These consider only single input ~~characteristics~~ conditions.

→ aids in selecting in a systematic way, high yield set of test cases. represents logical relation b/w inputs & outputs to generate efficient test cases

Cause: side x < sum of sides y + z

c_2 : $y < x+z$

c_3 : $z < x+y$

c_4 : $x = y$

c_5 : $x = z$

c_6 : $y = z$

effects: Not a Δ

e_2 : scalene

e_3 : isosceles

e_4 : equilateral

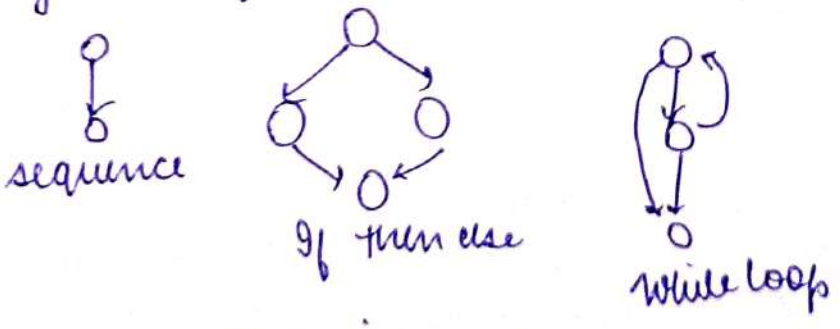
e_5 : Impossible

Combinations of conditions.

Path
Data-flow
Mutation

Structural Testing → white box testing. Examines internal structure of program to ensure adequate test coverage.

Flow graph – control flow of a program can be analysed using a graphical representation. directed graph. nodes r either statements / fragments of statement. edges represent flow of control.



DD path graph is used to find independent paths.
↳ decision to decision path graph

Independent paths r used to ensure that ① every statement in program has been executed atleast once
② every branch has exercised for true & false conditions

Cyclomatic complexity → also known as structural complexity.
→ gives internal view of code.
→ used to find no. of independent paths through a program

An independent path → through program that introduces ~~tt~~ at least one new set of processing statements / a new condition

Data flow testing : Focus points : ① where variables receive values
② where these values r used / referenced.

Mutation Testing - Fault based technique. similar to fault seeding mutations to program statements r made to determine properties about test cases. Multiple copies of program r made & each copy is altered (mutant). Mutants r executed with test data to check whether they r capable of detecting change

Software Testing → evaluating & validating software applications / systems to ensure they meet specified requirements & perform as expected
- Helps to identify errors, bugs, defects, thereby ↑ software quality & reliability.

Imp - Bug detection, Improving quality, customer satisfaction, cost efficiency

Methods - Manual, automated, Black box, white box

Process phases - requirement analysis, Test planning, design, Cnvt setup, execution, defect reporting, Tracking, management, regression testing, Test closure

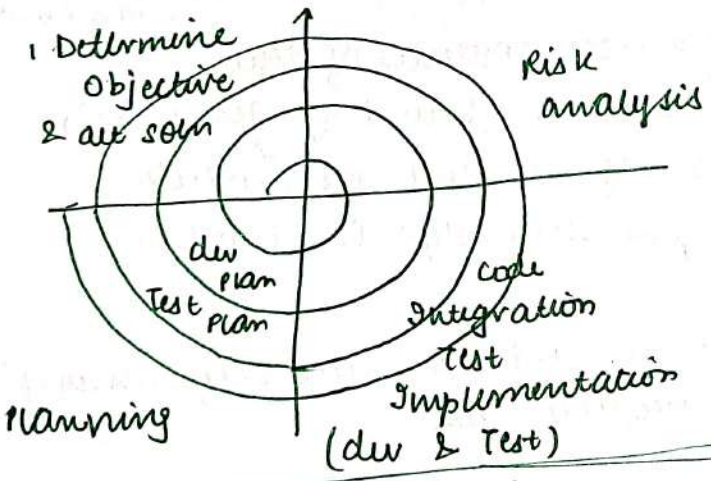
Functional Testing → verifying that application / system performs its functions as expected. Testing functionality against requirements.

Function oriented design	Object oriented
① Basic abstractions r real world functions	- data abstraction where ~~into~~ real world entities r represented.
② Functions r grouped by which higher level fn is obtained.	- On the basis of type of data they operate
③ state info is represented in ~~centralized~~ shared memory	- centralized memory.
④ for computation sensitive application.	- evolving system which mimics a business process / business case
⑤ Top down approach	- Bottom up approach.

Software dev PROCESS

① Reqt gathering
↓
Design
↓
Implementation, Test
maintain

Spiral model



Good SRS characteristics

① complete : complete reqt specification & capability to respond.

② consistent ③ correct

④ Modifiable : Related corners should be grouped ⑤ Ranked

⑥ Testable ⑦ Traceable

⑧ unambiguous ⑨ valid

⑩ verifiable

cohesion →

① coincidental – parts of module r grouped arbitrarily.

② logical : grp because they r logically categorized

③ Procedural : follow a sequence of execution

④ communicational info : Operate on same data

⑤ sequential : final output from 1 part is input to another part.

⑥ functional : contribute to single well defined task of module.

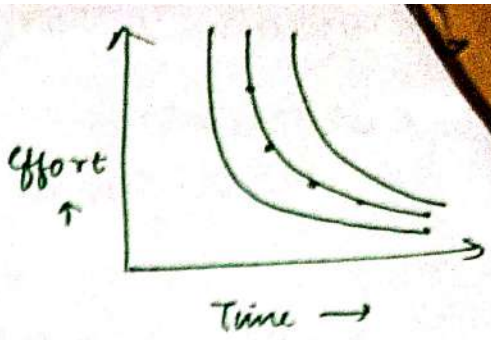
coupling → strength of interconn. btw program units

⑦ temporal – all statements activated at same time

⑧

Putnam Resource allocation Model

describes time & effort to finish a project of specified size. Earliest developed & most widely used



$$\frac{(B)^{1/3} \cdot size}{Productivity} = (Effort)^{1/3} \cdot (Time)^{4/3}$$

simple Model

B → scaling factor

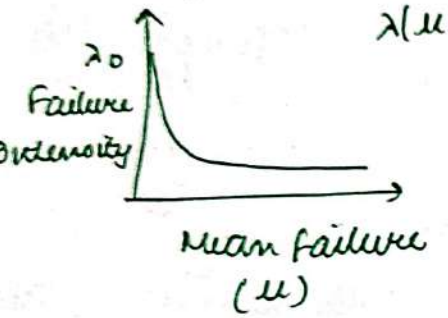
$$Effort = \left[\frac{size}{Productivity \cdot Time^{4/3}}\right]^3 \cdot B$$

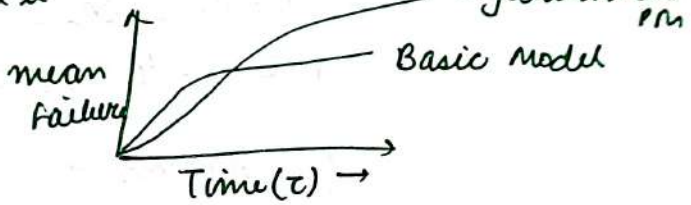
Logarithmic Poisson Model

regression analysis used to model count data & contingency tables.

$$\lambda(\mu) = \lambda_0 \exp(-\theta \mu)$$

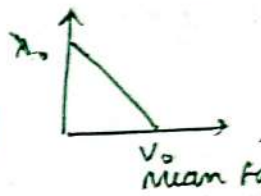
$$\frac{d\lambda}{d\mu} = -\lambda_0 \exp(-\mu\theta)$$





Basic execution Time Model

$$\lambda(\mu) = \lambda_0 \left[1 - \frac{\mu}{\nu_0}\right]$$



Content coupling → worst → if one directly references contents of other

Common coupling → access to global data (bound by global ds)

Control coupling → passing control flags so that one module controls sequence of processing steps in another module

Stamp coupling – global variables r shared selectively among routines that require data

Data coupling

Error discrepancy btw computed, observed / measured value / cond n and true, specified / theoretically correct value / condition.
can be misunderstanding – internal state of software
– oversight of memory management etc

Failure – inability of system / component to perform reqd fns within specified performance requirements. eg Bug, crash, fault, exception

Bug fault in program that causes program to perform in unintended / unanticipated manner

ISO 9000	CMM
- any industry	- developed for software industry.
- addresses corporated business process	- software eng. activities
Specifies min reqt	- gets to technical aspect of software eng.
- restricts to what is reqd	- How to fulfill reqt.
- No levels	- 5 levels.
- No sequence of steps to follow	- ✓
- Contract management	- Project Tracking
- Packaging, delivery & installation management, contract management.	- org level processes, Process development, integrated management.

Fast : Facilitated application Technique.
- reqt elicitation technique
- close gap btw developers intend & what users expect.
- team oriented approach
- Rules for preparation & participation r established
- agenda is suggested
- facilitator is assigned to control meeting.
- defn mech. - Reqt doc from both side
- each attendee is asked to make 3 objects / instances
  - Objects of inputs
  - Obj of output
  - Obj that will be part of env that surrounds system.

Rev engineering : commonly used. Discovers how something works.
Re- seeks to improve curr design. Reduced risk, reduced cost.
steps : source code Translation → Rev Eng → Program structure improvement → Program modularisation → Data Re-Eng.

Rev : takes finished product with aim of discovering how it works by testing it. done by companies to copy & understand parts of competitor product.

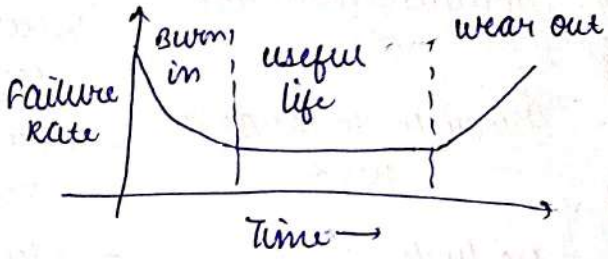
QFD → Quality function deployment → user reqt & requests into product design. goal → build product that does exactly what customer wants instead of delivering product that emphasizes expertise the builder already has.
classifies customer desires, identifies importance, and eng. char that maybe relevant. can be applied at any system composition level. system parameters r not correlated to stakeholder char

Fault → incorrect step, process / data defn which causes prog to perform in unintended manner.

ISO 9000	v/s	CMM
- International std org		- capability maturity model
- specifies min. requirement	ISO	- applies to any type of industry.
- restricts itself to what is reqd. suggest how to fulfill req		- specifically developed for software industry.
- Provides pass / fail criteria		- 5 levels.
- No levels.		- goes into technical aspect of SE.
- Provides grade for process maturity		

Debugging → locating & fixing / bypassing bugs in program code / Hardware device.

Bath tub curve



Cohesion	Coupling
- Relation within module	- Relation btw modules
- shows module's relative functional strength.	- Shows the relative independence among modules
- ↑ cohesion ideal	- low coupling ideal

Big Bang Integration Testing → all components / modules r integrated simultaneously after which everything is tested as a whole

- everything is finished before integration testing starts. - Time consuming
- difficult to trace cause of failure.

Software Crisis : period in early history of SD when industry faced significant challenges & difficulties in producing software that met desired quality, cost, delivery targets.

* Waterfall Model : linear & traditional software development life cycle model. classic approach : Project is divided into phases, each phase must be completed before next begins.

① Req gathering ② design ③ implementation, ④ Deployment (Testing) ⑤ Maintenance.

Sequential, Rigid, less adaptable to changing requirements, well documented, suited for projects with well understood & stable requirements. Progress is monitored at each stage. Risky if initial req r not well understood, costly defects.

* Prototype Model : provides tangible representation of software funct
- Quality & features before final system. Throwaway : to explore specific design ideas / req. Once validated, prototype is discarded. Evolutionary : evolve them into final system.
- Iterative & feedback drives process. - Risk management
- early user involvement - adaptable to changing req. ↓ risk
- Time consuming - Prototypes may not always accurately reflect final product.

* Evolutionary Model

Iterative + Incremental Model. Projects with evolving & changing req. found in complex & dynamic environments. early delivery of basic system then iteratively enhances.

Minimal system → adding more features, functionality & improvements → feedback & refinement.

Incremental, adaptable, early delivery, effective for large, complex projects.
↓ Risk

* Spiral Model : divides process into several phases, doesn't follow a linear path → risk analysis (technical, schedule, cost), flexibility in accomodating adjustments & changes. Continuous customer involvement. effective for large projects, adaptability, expertise is req. high development cost

Software Requirement : gathering, documenting & understanding needs & constraints of stakeholders (what & how system should achieve) ① elicitation : Interacting with stakeholders

Techniques → survey, interviews, workshops.

Documentation (desc, diagram, use cases) → Analysis (clear, complete, consistent, feasible) → ambiguities r addressed → validation: reqt should align with project goals (within budget & Time constraint) → verification (free from error)

- RE : document, validate, analyze, maintain software reqt.
  elicitation → documentation → analysis → specification → validation & verification
  foundation of SE, alignment with users needs, risk management, Quality assurance
- Functional Reqt : fns, features, capability & interactions that software system must provide. define behavior of system under various conditions outlines actions software should do.
  What system does eg: user Registration, calc total order cost
- Non functional : Quality attributes / constraints. How system should perform. Performance, reliability, security, scalability, Response time, cross cutting concerns. eg RT < 2sec
- System Reqt : Tech Details (Hardware, Software), lang, framework, db, Performance, compatability, constraints
- (FAST) function analysis System Technique : structured Method. Decompose complex system into smaller managable parts. Identify primary fns, decompose fns into sub-fns, hierarchial diagram showing relation btw fns. → linkage & constraints
  used in: engineering, design, systems analysis
- (QFD) Quality function deployment : Translate customer needs & reqt. into specific product / system features. Gather customer needs → correlate needs with product features → technical Reqt → Prioritization → deployment Matrix
  used in Product development where customer satisfaction is key
- Use Case Approach : Interactions btw user & software. Provides a clear understanding of system functionality from user's perspective. User centric view of system functionality.
- Reqt Documentation : overview → scope → functional & Non functional reqt → UR → SR → constraints → Data Models

SRS : Comprehensive doc foundation of SD Project. Outlines FL NF reqts. Clear & unambiguous description of what system should do & how. – No room for ambiguity. Covers all aspects. Focuses on meeting needs & expectations of end users & stakeholders. Traceability ~~with~~ btw req & source. Structured. Examines whether project can be ~~feasible~~ realistically completed within time & budget. Dependencies & Interactions. End user involvement (feedback) alignment with Project goals, Quality.

IEEE std – common framework for high quality SRS, format & structure is provided. Content guidelines. Clear & unambi-guous lang

Project Planning : scope, objective, approach, timelines, budget & resource allocation.

- **Project size estimation**: ① Expert judgement (when limited info) analogous estimation (based on similarities of curr & prev Project) Bottom up estimation. – Lines of code – No. of entities in ER – Function Points.

LOC KLOC – Thousand lines of code. – universally accepted – Close estimation. – easily quantified – simple – diff lang diff LOC – No proper ~~measuring~~ std. – diff. to estimate size in early stages.

Function Point Analysis – no. of fns of each type, Unadjusted FP, Total deg of influence, Value adjustment factor, Function Point count.

Types – External i/p, o/p, enquiries, Internal files, external interface files.

	simple	avg	complex
Ext I/p	3	4	6
Ext o/p	4	5	7
Enquiry	3	4	6
Logical Files (I)	7	10	15
Ext. LF	5	7	10

range of TDI → 0 to 70

Value adjustment Factor

$= [(TDI) * 0.01] + 0.65$

Find Function Point Count.

FP Pt Cnt = UFP * VAF.

– easily used in early stages of Project planning – Not good for real time systems.

COCOMO (constructive cost model)

→ Basic : simple, effort estimated on basis of LOC & project

3 Modes
- organic : small & straight forward. Experienced or
- semi detached : moderate complexity. some Innovation & complexity.
- Embedded : large & complex Projects
  Real time / mission critical systems.

→ Intermediate : framework for estimating effort, project duration & cost based on range of cost drivers & factors

(E) Effort = a * (KLOC)^b * EAF

↓ Person Months ← const. ↑ ↳ effort adjustment factor

Tailored estimation.

Factors : development flexibility. modern tools & techniques.

→ Detailed : software reuse, documentation, quality control.

very large & complex Projects. - structured & systematic
- tailoring estimation to project specific char. - relies on LOC as primary i/p
- estimates based on historical data.
- accuracy depends on relevance of data

Putnam Model : Relation btw project size, effort & no. of resources

Key factors : Size, effort per size (E/S), Productivity $(P) = \left(\frac{1}{E/S}\right)$

Resource constraints (availability [illegible] $\left(\frac{S}{E}\right)$

Effort (E) = $\frac{S}{P}$ $\frac{\text{Project Size}}{\text{Productivity}}$ schedule [illegible] $\frac{\text{Effort}}{\text{Rsrc Constraint}}$

simple & Intuitive. Large & complex Projects.

Risk Management : Identify (bugs, changes in req, resource const) evaluate potential impact & probability of risk. Prioritize risks based on severity, code reviews, testing, planning, Monitoring & Tracking, Response Planning.

Cohesion : deg. to which comp. r focused on doing a single, well defined task. How closely r modules related? ① Coincidental (lowest) elements r not related, elements of appear to be put together coincidentally.

② Logical : share logical relation. ③ Temporal Cohesion : used at same time. May do diff. fns.

④ Procedural : Part of common Procedure / Algo

⑤ communication → operation on same data / exchange info

Sequential → executed in specific order Functional: highest level
Perform well defined fn / task. easier to understand, maintain & reuse

Coupling : deg. of interconnectedness btw modules / components
How one module relies on functionality of another. Low coupling is preferred → more flexible & maintainable
- Tight - modules r highly dependent on each other
- loose - Modules r less dependent on each other. changes in one module r less likely to affect others.

UID - attractive, simple, Responsive, clear, consistent

$FP = CAF \times UFP$.

0 1 2 3 4 5 → essential
No incidental influence | avg significant → Moderate

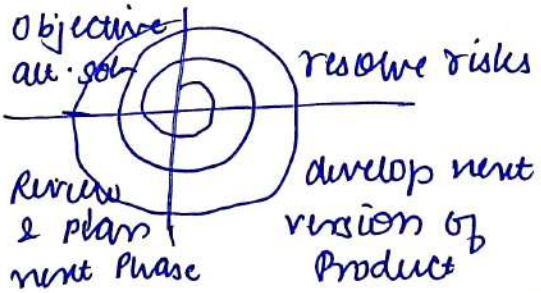
COCOMO. Effort $= a_b(KLOC)$
development Time $= c_b(E)$
$= 2.5(E)^{0.38}$ Months.

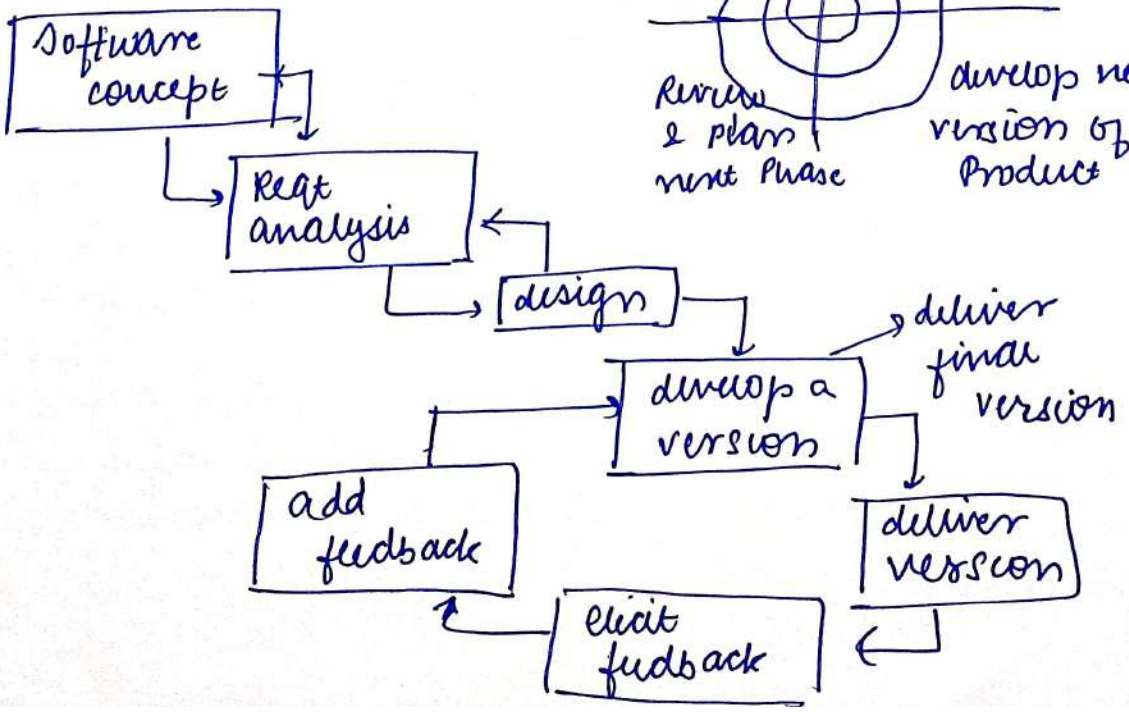
Productivity $= \frac{KLOC}{effort}$.

Effort $= a_i * (KLOC)^{b_i} \times EAF$

Prototype → Req gathering → Design → Build Prototype → Customer evaluation → Refining → End Product

Evolutionary model





$E = a(KLOC)^b$.
time $= c(Effort)^d$
Person $= \frac{Effort}{time}$