VI sem

Software Engineering

Syllabus

Unit-I :- Introduction

Software crisis, Software processes, Software life cycle models: Waterfall, Prototype, evolutionary & Spiral models
Overview of Quality Standards like ISO 9001, SEI-CMM.
Software Metrics:
Size Metrics like LOC, Token Count, Function Count, Design Metrics, Data Structure Metrics, Information Flow Metrics

Unit-II Software Project Planning

Cost estimation, static, single & multivariate models, cocomo model, Putnam Resource Allocation Model, Risk M.
Software requirement Analysis & specifications:
Problem Analysis, Data flow Diagrams, Data Dictionaries, Entity Relationship diagrams, Software Requirement & Specifications, Behavioural and non-behavioural requirements, Software Prototyping.

UNIT-I

* **Software** :- A set of instructions, data or a program used to operate computers and execute specific tasks. Software is a generic term used to refer to application scripts (ex: code content of any software) and programs that run on a device.

Objective: To improve the software development process in order to deliver good quality, maintainable software in time & within budget.

(to avoid problems like: → bug fixing → hardware compatibility → requirement of other software to run.

As per a report by IBM:
"31% of projects get cancelled before completion, 53% exceed their cost estimates."

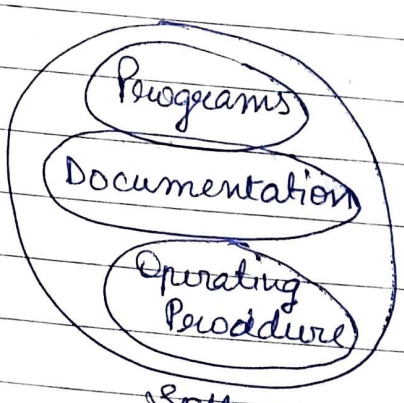
Article from Charles C. Mann's article.
"Microsoft released Windows XP on October 25, 2001. On the same day, the company posted 18 megabytes of patches (kind of "bug fixes") on its website for bug fixes, compatibility, updates, and enhancements. Two patches were intended to fix important security holes. Microsoft advised users to back up critical files before installing patches".

Definition of Software Engineering

By Fritz Bauer: The establishment and use of sound engg. principles in order to obtain economically developed software that is reliable & efficient.

By Stephan Schach: A discipline whose aim is the production of quality software, a software that is delivered on time, within budget and that satisfies all requirements

* Program vs Software (technically different)



Operating procedures: Instructions to set up & use reaction to system failure, operating manual

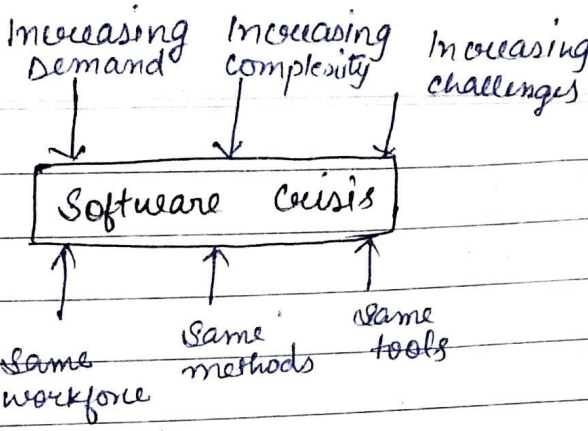
Documentation: Testing, Analysis, Design, Implementation

* Software Crisis

Software crisis is the failure of software development that leads to incomplete & degreading performance of software products.

Causes:
1. Projects running over-budget.
2. Project running over-time.
3. Not enough resources.
4. Software cannot handle complex instructions.

ex: OS 360 launched by IBM → but not reliable & efficient so it was avoided by people.



Software engineering
① More test cases
② Experienced team Members

★ Software Processes
A way in which we produce software

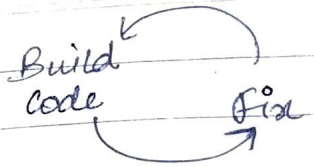
1. Software specification : The functionality of the software and constraints on its operations must be defined.
2. Software development: Software to meet the requirement must be produced.
3. Software Validation : Software must be tested to ensure it does what customer wants.
4. Software evolution : Software must evolve

Q: ~~why it is difficult to organize/improve software processes ?~~
~~→ Not enough time.~~
~~→ Lack of knowledge.~~
→ Insufficient commitment.
If process is weak, the end product will suffer.

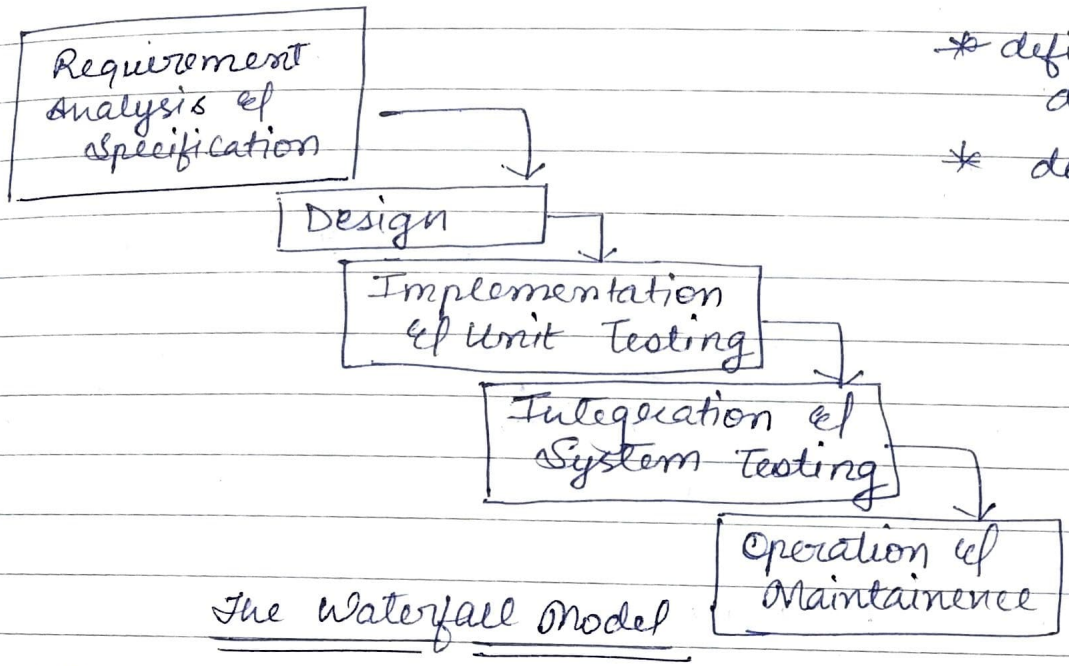
★ Software Life Cycle Model
In the IEEE standard glossary of software engineering terminology, the software life cycle is :
" The period of time that starts when a software product is conceived and ends when the product is no longer available for use.
Life cycle includes : requirement phase, design phase, implementation phase, test phase, installation & check-out phase, operation & maintainence phase, and sometimes retirement phase".

SDLC = Software Development Life Cycle.

I. BUILD AND FIX MODEL



II. WATERFALL MODEL



* define before design
* design before code

The Waterfall Model

→ Phases always occur in order & don't overlap.
→ Developer must complete each phase before starting next
→ It resembles a cascade of waterfalls.

1. Requirement Analysis & Specification : To understand exact requirements of customers and to document them properly. Describing "what" not "how".
The resultant document is SRS document.
SRS = Software requirement specification.

2. Design Phase : To transform SRS into a structure that is suitable for implementation in some programming language. The resultant is software design description (SDD) document and it contains overall software architecture.

3. Implementation & Unit Testing : Design is implemented & coding phase proceeds. During Testing major activities are performed to examine & modify code.

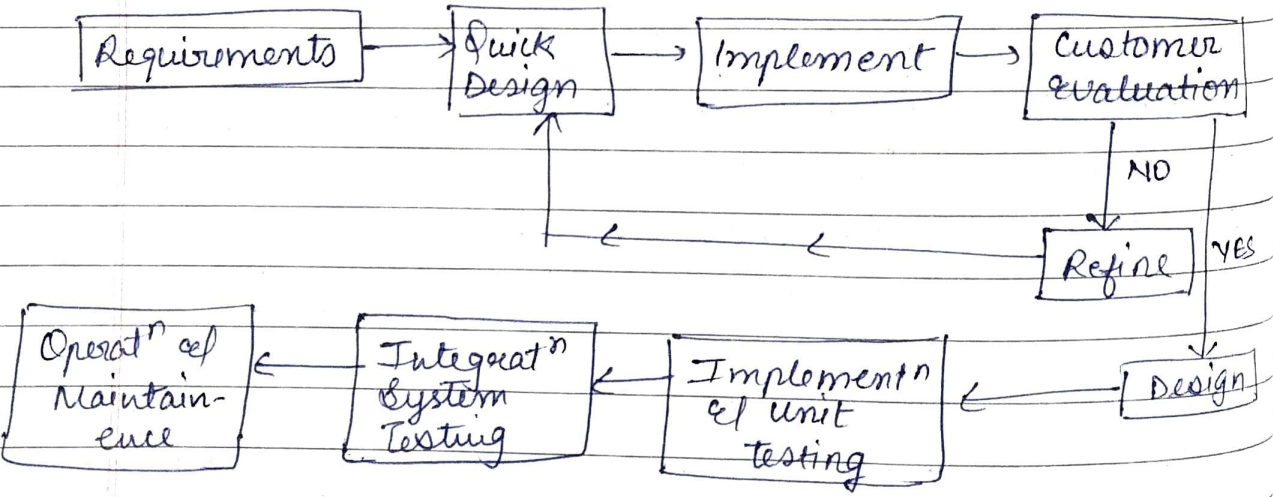
4. Integration & System Testing: Testing of entire system is done after linking each module.
5. Operation & Maintainence: It includes error correction enhancement, removing obsolete functions, optimization

* Problems of Waterfall Model
1. Difficult to define all requirements in start
2. Working version is seen very late.
3. Real projects are rarely sequential.
4. Not suitable for accomodating any change.

* If an organization has experience in building accounting system then building a new accounting system can be easily managed with waterfall model.

III PROTOTYPE MODEL

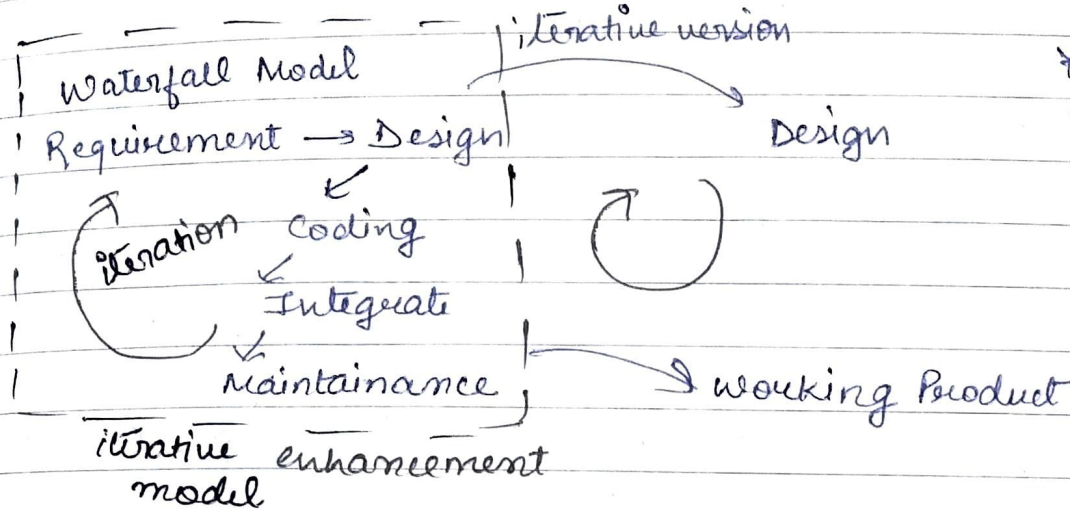
First a working prototype of software instead of developing actual software.
The prototype is evaluated by costumer and feedback is given to refine the final software.



Problems:
1. Customers can be lazy.
2. Prototype must be delivered quickly.

IV EVOLUTIONARY PROCESS MODEL

→ It resembles iterative enhancement model but differs in a way that a usable product is not needed at end of each cycle.



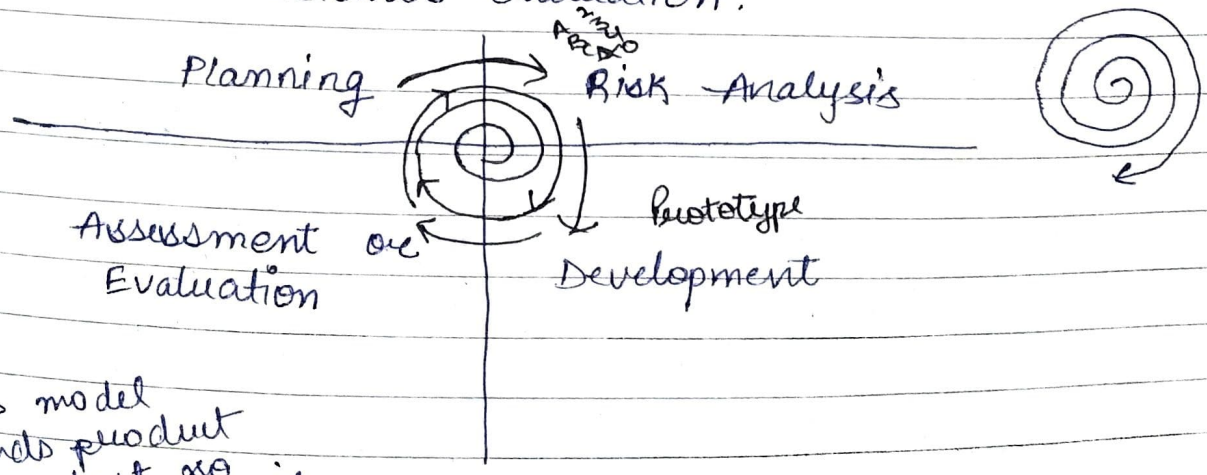
→ It should be used when it is not necessary to provide a minimal version of the system quickly.
→ It is useful when requirements are unstable or not well understood in the beginning.

V SPIRAL MODEL

→ Barry Boehm recognized lack of "project risk" factor into a life cycle model and tried to correct this using Spiral model.

Four phases:

1. Planning: Objectives, alternatives, constraints.
2. Risk Analysis: Analyze alternatives and to identify risks involved.
3. Development: Product development and testing.
4. Assessment: Customer evaluation.



This model depends product to product so this is a basic model.

Requirement	W	P	S	E
1. Easily defined	✓	✗	✗	✗
2. Change often	✗	✓	✓	✗
3. Defined early	✓	✗	✗	✗/✓
4. Requirements indicating complex system	✗ (Disadv. of water.)	✓	✓	✓

evolutionary doesn't give end product at every change.

Quality Standards

* ISO 9001 (International Organization for Standards)
  ↳ mostly valid in Soft. Dev.

→ Internationally recognized Quality Management System (QMS)
→ It helps to continually monitor and manage quality across your business so you can identify areas for improvement.
→ Internal audits (checkings event) are conducted.
→ Applies to organizations engaged in: design, development, production and servicing, testing and installation.

features for software development organization:

1. Documentation should be utilized and maintained properly.
2. Testing of product should be done.
3. Organizational aspects should be addressed.

SEI - CMM

* Software engineering Institution - Capability Maturity Model.
* It is used to design of improve software development.
* It is based on organizational maturity that determines effectiveness in delivery quality software.

5 level of maturity:

1. Regressive: chaos, Panic, Heroic efforts.
2. Repeatable: Process documented, lack of implementation.

3. Consistent : Processes integrated, training programs.
4. Quantitative : Metrics : track productivity, processes, products.
5. Optimizing : Well documented, post-implementation reviews, always improving and optimizing.

* Software Metrics

A unit of measurement of a software product or software related process.

OR

The continuous application of measurement based techniques to the software development process & its products to supply meaningful and timely management information.

What are we measuring?

- → Execution Speed
- → no. of errors
- → lines of code

} Direct Measures

- → efficiency
- → reliability
- → portability

} Indirect Measures

Objective: 1. Using numbers to improve the software or the process of developing software.
2. To improve management of process.

Applications:

1. To estimate size and cost.
2. Controlling whole project.
3. Prediction of quality levels for software.

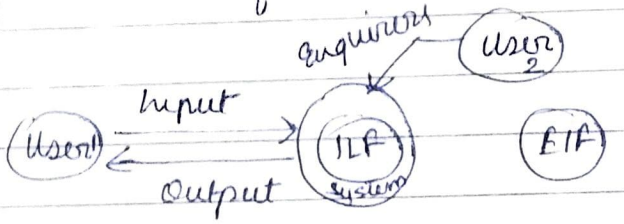
1. * Lines of code (LOC)

→ Basic idea is to estimate size.
→ No general agreement about what is line of code.
Some include data declarations, comments or unexecutable statements, others exclude them.
→ Productivity = LOC / (Person-Month)

→ Measuring by lines of code is like measuring a building by no. of bricks used.
→ Lines of code can predict programming time, but fail to tell anything about efficiency, reliability etc.

2. * Function Count
→ Counting functions instead of components.
→ Measuring functionality from user's point of view
→ Measurement is independent of technology used.

Inputs - information entering system
Outputs - Information leaving system
Enquiries - request to instantly access information
Internal logic files - information held within system
External ~~logic~~ interface files - information held outside system



Features and Advantages:
1. Independent of language, tools or methodologies used.
2. Function point can be estimated from requirement and design.
3. It is a relevant approach than LOC.

	Low	Average	High ← Complexity
External Inputs	3	4	6
External Output	4	5	7
External Enquirers	3	4	6
Internal Logic files	7	10	15
External Interface files	5	7	10

* These all values are different for different product.

UFP ≡ Unadjusted Function Point
CAF ≡ Complexity Adjustment ≡ $0.65 + 0.01 \Sigma F$
$\Sigma F = [I \times (\bullet 14)]$

influence : 0, 1, 2, 3, 4, 5 (given in ques)

FP : UFP × CAF

Ques No. of inputs = 50 ; # outputs = 40
enquiries = 35 ; # user files = 06 ; # external interfaces = 04
CAF and weighting factors are average.
Complete function point.

I : 0, 1, 2, 3, 4, 5 ⇒ average value = 3

UFP → from table (take average coz :)

4, 5, 4, 10, 7

UFP = 50×4 + 40×5 + 35×4 + 6×10 + 7×4

= 200 + 200 + 140 + 60 + 28 = 628

CAF = 0.65 + 0.01 (3×14)

= 0.65 + 0.01 (42)

= 0.65 + 0.42 = 1.07

FP = UFP × CAF = 628 × 1.07 = 671.96

3.* **TOKEN COUNT** (Halstead's Software Metrics)

Tokens (considered programs as token)

P = operators + operands

$n = n_1 + n_2$

n = vocabulary of a program or no of unique tokens

n_1 = no. of unique operators (+, −, /, *, while, for, printf, ;, (), fun())

n_2 = no. of unique operands (a, 5, 4, bc, de)

$N = N_1 + N_2$

N_1 = program length

N_1 = total occurence of operators

N_2 = total occurence of operands

avg. prog

$$\hat{N} = n_1 \log_2 n + n_2 \log_2 n_2$$ + -- most generalised formula

there are lots of formula's used for diff prog but not suits every program.

N = 2 * LOC (if each line has 1 operator & 1 operand)

Program Volume : $V = N * \log_2 n$

Volume is proportional to program size represents the size (in bits) of space necessary for storing program.

Potential Volume V^* ≡ represents program having minimal size

$$V^* = (2 + n_2^*) \log_2 (2 + n_2^*)$$

n_2 ≡ no. of operands

n_2^* ≡ no. of potential Operands

Program Level :- The higher level of language, the less effort it takes to develop a program using that language.

$$\boxed{L = V^*/V}$$

$L \in [0,1]$

If $L = 1$ → Program is having min. size.

else if $L = 0$ (approx.) → program is having max. size.

Difficulty ≡ $D = 1/L$

⇒ If the program level decreases, difficulty to handle increases.

Programming Effort : Amount of effort needed to translate an algorithm into implementation in specified programming language.

$$E = V/L = D * V$$

Language Level = $\lambda = L * V^* = L^2 * V$.

Programming Time ≡ shows time (in minutes) needed to translate algorithm into implementation in specified program language.

$$T = E/(\beta * S)$$

$\beta = 60$

$S = 18$ moments/second

Counting Rules for C.

1. Comments ignore.
2. Identifier and function declarations are ignored.
3. Global variables are counted as multiple occurences of some variables.
4. Variables with same name in different functions are counted as unique.
5. function calls are considered operators.
6. do, while, for, if, else etc are operators.

7. Switch, case are operators.
8. Return, default, continue, break, Size of operators.
9. (), ,, ; are operators.
10. GOTO is operator but label is operand.
11. a*b vs *ptr / && vs &.
12. "~~arrangements~~" "array" name and index are operands, [] is operator.
13. Hash derivatives are ignored.

Ques
```
int sort (int x[], int n)
{ int i, j, save, iml;
  /* This function sorts an array */
  if (n<2) return 1;
  for ( c=2; i<=n; i++ )
  { iml = c-1;
    for (j=1; j<=iml; j++)
      if (x[i] < y[j])
      { save = x[i];
        x[i] = x[j];
        x[j] = save;
      }}
  return 0; }
```

Find n, N, V, E, λ

Operators	Occurences
int	4
()	5
,	4
[]	7
if	2
<	2
;	11
for	2
=	6
-	1
<=	2
++	2
return	2
{}	3
	53

Operands	Occurences
SORT	1
x	7
n	3
i	8
j	7
save	3
iml	3
2	2
1	3
0	1
	38
	10

$n = n_1 + n_2$

$= 14 + 10 = 24$

$N = 53 + 38 = 91$

$V = N \log_2 n$

$\boxed{V = 91 \log_2 24}$

$\hat{N} = 14 \log_2 14 + 10 \log_2 10$

$= 14. \_ \quad + 10. \_$

* Data Structure Metrics

→ Software Planner:

Input	Internal Data	Data Output
1. Size	1. Rate per developer	1. Project efforts
2. No. of developers	2. Other factors	2. Duration
3 Duration		3. Est. cost

ex: Necessity?

A ← 30, 20, 10 B ← 15, 20, 10

chances of Error A(↑) coz of ↑ input

VARS ≡ count of variables

n_2 = VARS + unique constraints + labels

$N_2 = n_1V_1 + n_2V_2 + n_3V_3 + \text{-----}$

n_2 and N_2 are robust and variations in algorithm computation schemes for computing them do not seem to affect inordinately other measures based on them.

- A count of the amount of data input to, processed in, and output from software is called data structure metrics
- Some data metrics concentrate on variables and constants within each module and ignore input/output dependencies.
- Others concern themselfves with input/output situation.
- We will discuss about measuring amount of data, the usage of data within modules, and degree to which data is shared among modules.

* Usage of Data within a Module

What is live variable?

1. A variable is live from beginning of a procedure

to the end of a procedure. ✗ Not appropriate

2. A variable is live at a particular statement only if it is referenced certain no. of statements before or after that statement. ✗ Not appropriate

3. A variable is live from its first to its last reference within a procedure.

"A variable is live at same point if it holds a value that may be needed in future, or equivalently if its value may be read before the next time the variable a written to".

ex: int a;
a=5; } → live
a=a+5; → value change

ex: { int a, b;
a=9, b=10; — 2
printf ... 1 var. live
a=a+10; 1
} point 1

point 2
printf ... 1;

$$\overline{LV} = \frac{\text{Sum of count of live variables}}{\text{count of executable statement}}$$

(Average no. of live var.)

eg:
```
# include <stdio.h>                    0
void swap (int x[], int k)             0
{ int t;  → declare 0   ↳ declare
  t = x[k];
  x[k] = x[k+1];   t is also live
  x[k+1] = t;
}                                      0
```

No. of statement	Live Vari
1	0
2	0
3	0
4	t, x, k
5	t, x, k
6	t, x, k
7	0

* Variable Spans

```
01   scanf ("%d %d", &a, &b)      a, b
32   x = a;                       a, b
45   y = a - b;                   (b=3)
53   z = a;                       (a=5)
60   printf ("%d %d", a, b);
```

A metric that captures the essence of how often a variable is used in a program is called the Span (SP). Size of span indicates no. of statements that pass b/w successive uses of variables.

n statement → n-1 spans

a = 4, b = 2

⇒ a is used more

Span { a = 10, 12, 7, 6 = 35/4
b = 23, 14 = 18.7

* Applying Knowledge to Program

$$\overline{LV}_{program} = \frac{\sum_{i=1}^{m} \overline{LV_i}}{m}$$

$$\overline{SP}_{prog.} = \frac{\sum_{i=1}^{n} \overline{SP_i}}{n}$$

Program Weakness (WP)

$WM = \overline{LV} * r$ (for module)

r = average life of variables.

$$WP = \frac{\sum_{i=1}^{m} WM_i}{m}$$

* Information Flow Metrics

→ Every system has components, work and structure of these component determine complexity of function.

→ Cohesion: If a component has to do numerous descrete task it is said to lack cohesion.
Cohesion can be defined as the degree to which a component performs a single functions.

→ Coupling: Degree of linkage b/w one component & others.

* components having lack of cohesion and high linkage (highly coupled) are less reliable and difficult to maintain.

* Information Flow Model

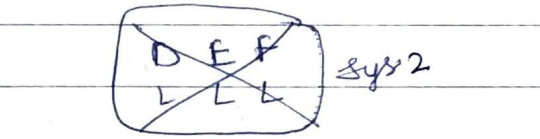
If metrics are applied to components (every small)

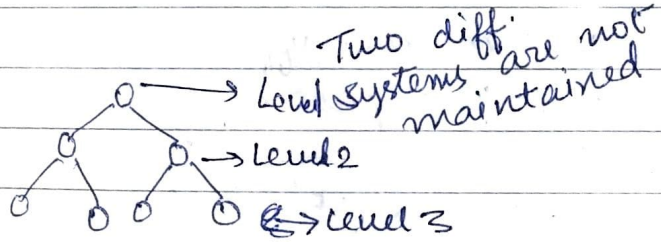
Three Measures:

1. FAN IN : No. of components that can call A.
2. FAN OUT : No. of components that are called by A.

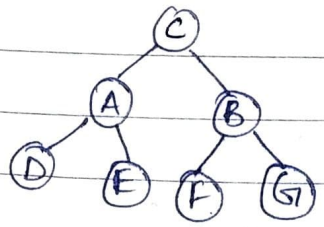
$$IF(A) = [FANIN(A) \times FANOUT(A)]^2$$

* It can be done early
* can be updated easily
* Automation might be possible
* everything or every component is relative.
* High FANIN means lack of cohesion.
* High FAN OUT means lack of cohesion or no abstraction
* High IF indicates highly coupled components.
* Individual components → LEVEL SUM → SYSTEM SUM





System Sum $= IF(L_1 + L_2 + L_3)$



$= IF(A) + \cdots + IF(G)$

* Identify nightmare (IF↑) components.
* 25% of components with highest scores for FAN values should be investigated.
* Sometimes things can't be improved. So, let's reduce the failure rate.
* Sudden increases in IF values across levels indicate missed level of abstraction. (Abstraction improves it)
* Final System SUM value gives overall complexity rating and helps to compare alternative designs.

* Software Project Planning

I. Cost Estimation

* How much will it cost to develop?
  - → cost ∝ Development time
  - → how to determine?
    - → Past experience
    - → Using Models we are going to discuss.
  - → Race Against Time

* For cost estimation, Models we use :-
  - → Static Vs Dynamic
  - → Predictors are variables used as input to predict output
    - → Static (Single variable static)
    - → Multivariable Static

Static Single Variable Model

* Output can be something other than cost (time or effort)

$$C = aL^b$$

L ≡ Lines of code or sometimes diff.

a & b ≡ constants derived from past exp.

$$E = 1.4L^{0.93}$$
$$DOC = 30.4L^{0.90}$$
$$D = 4.6L^{0.26}$$

⇒ SEL Model

Static Multivariable Models

→ multi inputs

→ Input are like (user participation, memory constraints etc)

$$E = 5.2L^{0.91}$$
$$D = 4.1L^{0.36}$$
$$I = \sum_{i=1}^{29} w_i x_i$$

currently not in use (used by IBM) ⇒ Walton-Felix Model

$x_i = \{-1, 0, 1\}$

w_i = weight

I ≡ productivity index

29 ≡ variables correlated to productivity

* The Constructive Cost Model (COCOMO)

→ Hierarchy of Software cost estimation models.
→ Divided into basic, intermediate and detailed sub models
→ 3 classes applicable on all 3 above models :

1. Organic Projects : 2-50 KLOC, experienced Team, In-house
   eg. payroll project
2. Semi-detached : 50-300 KLOC, Low experience, Variable
   e.g. Database System
3. Embedded : 300 KLOC or more, Little or no. Experience
   Complex Hardware
   e.g. Air Traffic Control

* Basic Model

Project	a_B	b_B	c_B	d_B
organic	2.4	1.05	2.5	0.38
embedded	3.0	1.12	2.5	0.35
semi-detached	3.6	1.20	2.5	0.32

$$E = a_B (KLOC)^{b_B}$$

$$D = c_B (E)^{d_B}$$

$$SS = E/D$$

$$P = \frac{KLOC}{E}$$

E = effort applied in person/months
D = development time in months
SS = staff size
P = Productivity

* It estimates very quickly and roughly.
* KLOC are given in question.
* less accuracy.

* Intermediate Model

→ More accuracy as compared to basic model.
→ cost is predicted according to actual project environment

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

where $EAF \in [0.9, 1.4]$

$$D = c_i (E)^{d_i}$$

Project	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

New Attributes :-

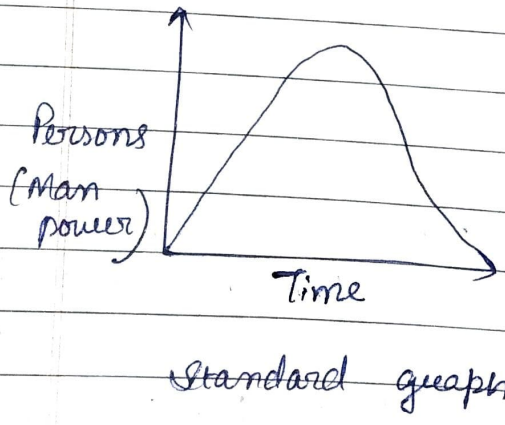
→ Required Software Reliability
→ Database size
→ Main storage constraints
→ Programmer capability
→ A total of 15 new attributes is there as compared to Basic Model.

* Detailed Model

→ Refined version of Intermediate Model.
→ Cost is calculated phase by phase.
1. Plan/requirements
2. Product Design
3. Programming
4. Integration / Test

* The Putnam Resource Allocation Model

→ It is base on Rayleigh curve. Norden observed this for Hardware Developed Projects. Then Putnam applied it for Software.



Standard graph

It can be :-

are also possible

⇒ The equation of above curve represents manpower, measured in persons per unit time as a function of time. ((P-Y)/Y)

imp $m(t) = \frac{dy}{dt} = \boxed{2kate^{-at^2}}$

cummulative man power → $\boxed{y(t) = k[1 - e^{-at^2}]}$ imp

k = constant

$y(\infty) = k$ (end)

$y(0) = 0$ (start)

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

$\Rightarrow \boxed{t_d^2 = \frac{1}{2a}}$

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

Estimate of development time

$E = 0.3935k$

$\boxed{M_0 = \frac{k}{t_d\sqrt{e}}}$

Average rate of software team build-up: $\boxed{\frac{M_0}{t_d}}$

* dt time is calculated in years.

16 months = $1 + \frac{4}{12}$

$= 1 + \frac{1}{3}$ year

dy/dt : manpower utilization rate per unit time

t : elapsed time

a : parameter ($1/t^2$)

(k) : area under curve $[0, \infty]$

constant for a particular curve

$m(t) = \frac{dy}{dt}$

t_d = time where max.m effort rate occurs. (peak development time or peak time) and also total project development time

k = total project cost/effort

M_0 = peak manning/no. of person employed

$\sqrt{e} = 1.648$

t_d = delivery time

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

$k = 95$ PY $\quad t_d = 1 + \frac{9}{12} = 1.75$ years

$M_0 = \frac{k}{t_d\sqrt{e}} = \frac{95}{1.75 \times 1.648} = 32.94$

Average rate of software team build up $= \frac{M_0}{t_d} = \frac{32.94}{1.75} = 18.82$

* Software Risk Management
* Software surprises can occur any time. Let's reduce their chance of survival.

What is Risk?
Possibility of an unpleasant happening. Risk can threaten the success of project.

Impacts of Risk
1. cost
2. Schedule
3. Success
4. Quality
5. Morale of team

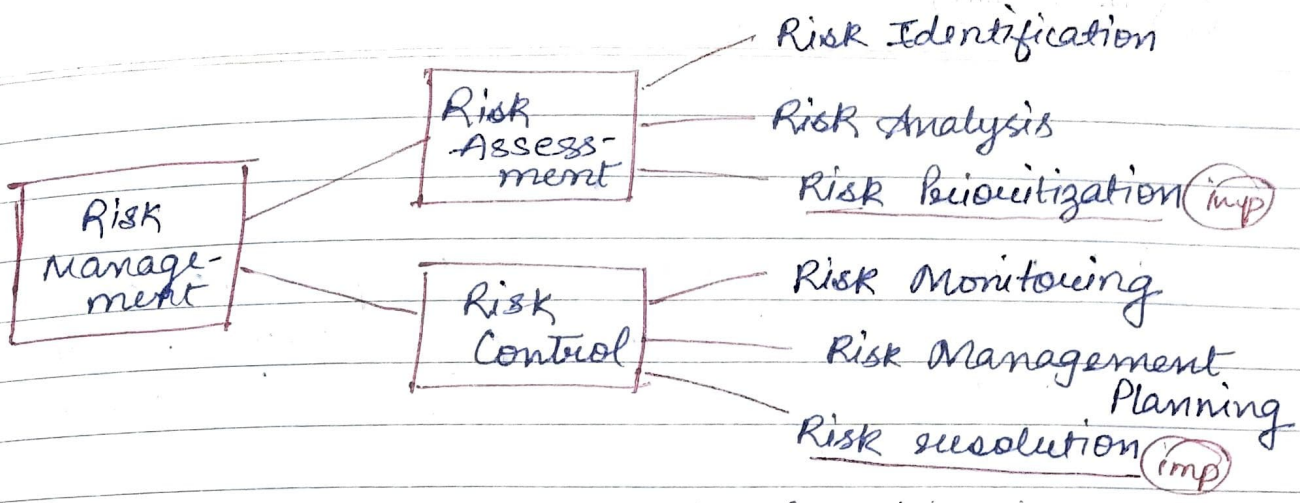
Types of Risks
1. Due to Dependency
   1. Outsourcing project
   2. Inter-group Dependency
   3. Wrong input / data items by customer.

2. Due to requirement issues
   1. Lack of clear product vision
   2. Lack of agreement on product requirement.
   3. Rapidly changing requirement.

3. Management is poor.
   → No decision making
   → Poor communication
   → Personality conflicts
   → Unavailability of testing facilities.

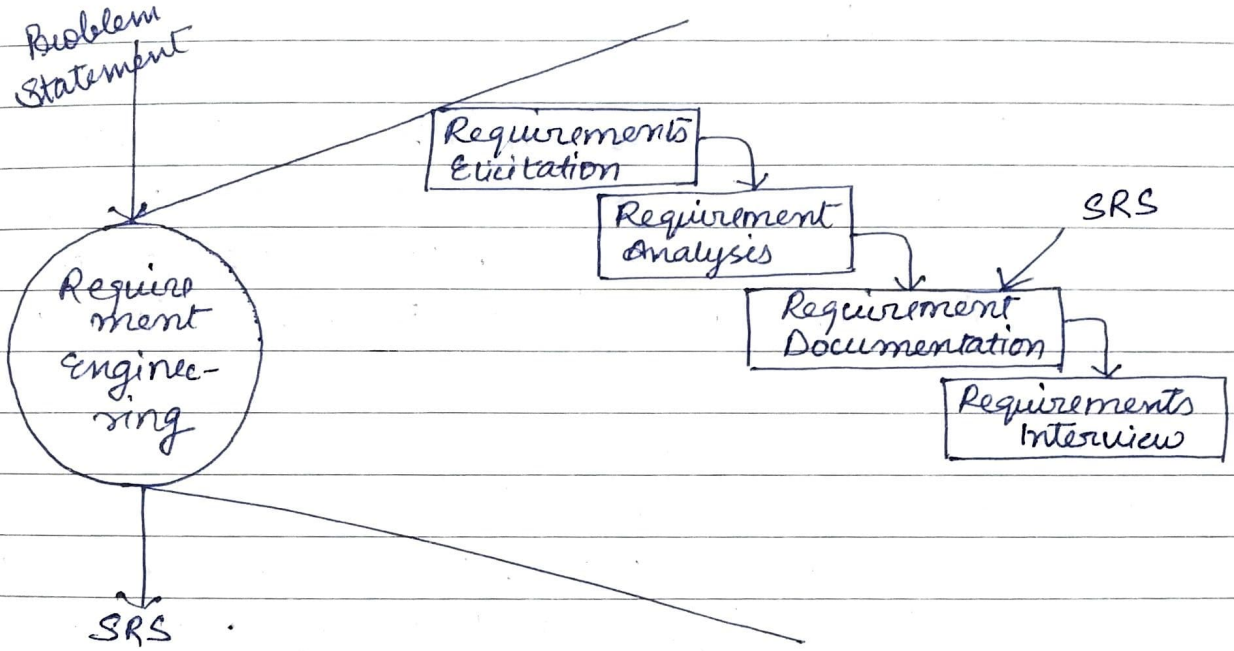
4. Lack of Training or Documentation
   → Poor documentation
   → No experience (team for low cost)
   → No. or inadequate training.

Risk Management



* Software Requirements : Analysis & specifications

Requirements: Describes "what" of a system, not "how".



Types of Requirements

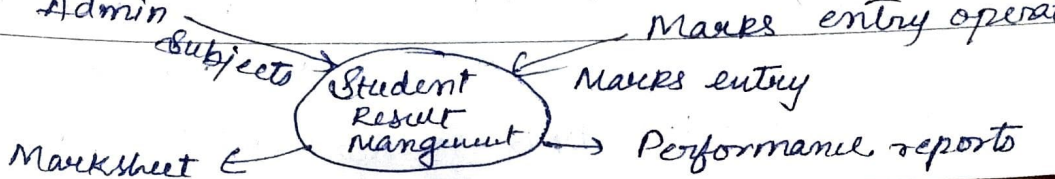
1. Known Requirements
2. Unknown Requirements
3. Undreamt Requirement
4. Functional Requirement
5. Quality Requirement

ex: A → B → C (unknown)

* Requirements Analysis

Requirements are analyzed and then refined by scrutinizing them in order to remove any ambiguity. Steps are:

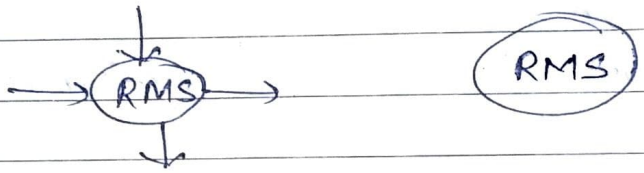
(i) Draw the context Diagram:



(II) Developement of prototype :
→ It is optional and can be a bit time consuming
→ It should be built on low cost.

* Backend should be focussed.

(III) Model the requirements : Represent all the f" and entities in graphical format. It includes data flow diagrams, entity relationships diagrams, data dictionaries etc.



(IV) Finalize the requirements : The inconsistences in requirements have been corrected and flow of data has been analyzed. Now the requirements are finalized and documented in a prescribed format.

* Data Flow Diagrams (DFD)

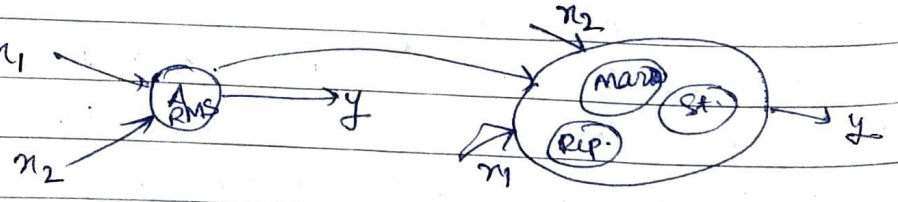
1. All names should be unique.
2. It is not a flow chart. It contains arrows to represent flowing data, but any order is not represented by them.
3. → Data flow

○ Process

□ source or sink

Data store : A repository of data

ex :



Levelling : Level-0- DFD ≡ fundamental system model or context diagram.

Level-1-DFD → more detailed repres.

* Data Dictionary

They are repositories to store information about all data items defined in DFS.

What type of information is stored ?

1. Name of data items
2. Other names
3. Purpose
4. Related data items
5. Range of values
6. Data structure or Data flow

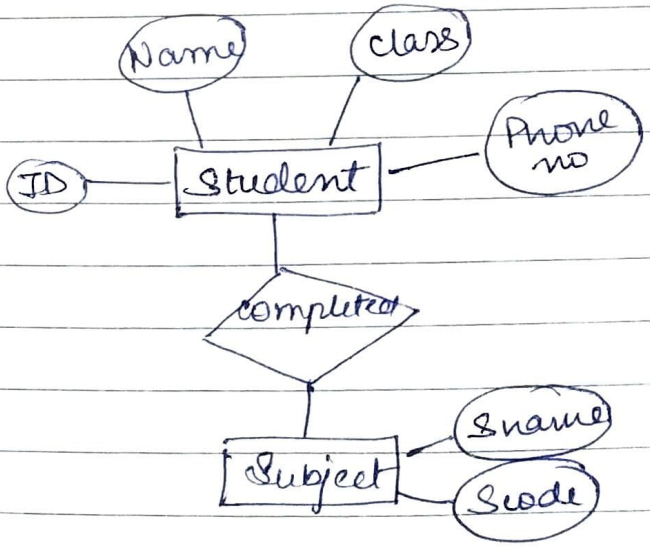
Uses of data dictionaries :-

Create an ordered listing of all data items.
Create an ordered listing of subset data items.
Finally, design software and test cases.

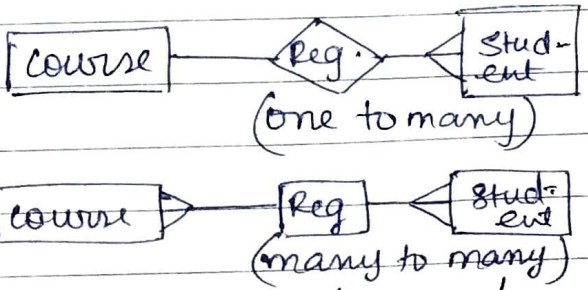
* ER Diagram (Entity-Relation Diagram)

It is detailed and logical representation of the data for an organization.

It consists of data entities, relationships, attributes



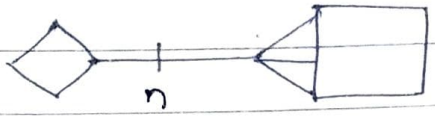
Degree(completes) = 2



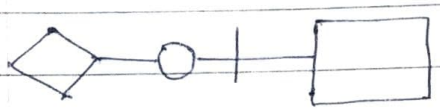
Cardinality = No. of instances of entity B that can be associated with each instance of entity A.

Diff. types of cardinalities :-

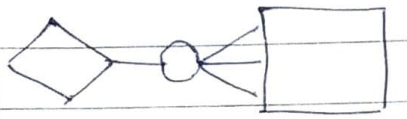
1 cardinality (mandatory)
{1}



M cardinality (Mandatory)
{1, 2, 3, ...}



Optional 0 or 1 cardinality {0,1}



M cardinality (optional)
{0, 1, 2, ...}

Candidate Key
→ can uniquely identify any entity.

* DFD, Dict., ER Diag. → helps in Analysis of requirement

Maintainance – Once software passes all stages without issue, it will be maintained and upgraded time to time to adapt changes.

---

* Software Prototyping
→ It is partial implementation of a system.
→ It basically allows ~~access~~ users to explore and criticize proposed systems before undergoing the cost of full-scale development.

Different type of approaches:
(i) Throw-away approach: Prototype is constructed in order to learn and improve and then discarded
(ii) Evolutionary approach: Instead of discarding prototype, it is re-adapted. (further correction)

Benefits:
→ Misunderstanding b/w customer's and developers is reduced.
→ Missing functionalities are detected.
→ Custom and difficult requirements are refined.

Approach	Throwaway	Evolutionary
(1) Development	→ Quick, Rough, Untidy	→ Tidy, No. sloppiness
(2) what to build (diff. to build or understable)	→ Difficult Non-understable	→ Easy
3) Goal	→ Prototype must be discarded.	→ Prototype is required.

Behavioral requirements: Input, Output, I-SO
Non-behavioral requirements: Reliability, Efficiency etc.

* Software Design:

UNIT - III

Cohesion & Coupling classification of cohesiveness & Coupling, Function Oriented Design, Object Oriented Design, User Interface Design.

Software Reliability:
Failure and faults, Reliability Models: Basic Models, Logarithmic Poisson Model, Calender Time Component, Reliability Allocation.

* Software Design: what how

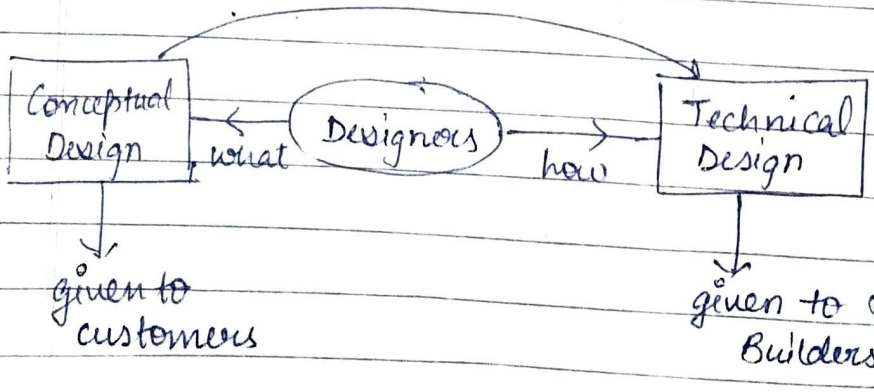
what is design? (SRS → SDD) → coding

Design is the highly significant phase in the software development where the designer plans "how" a software system should be produced to make it functional, reliable and reasonably easy to understand, modify and maintain.

The main purpose is to produce a solution to problem given in SRS.

* Good Design is always system dependent.
* Design is highly important to decrease the gap b/w specifications and coding.

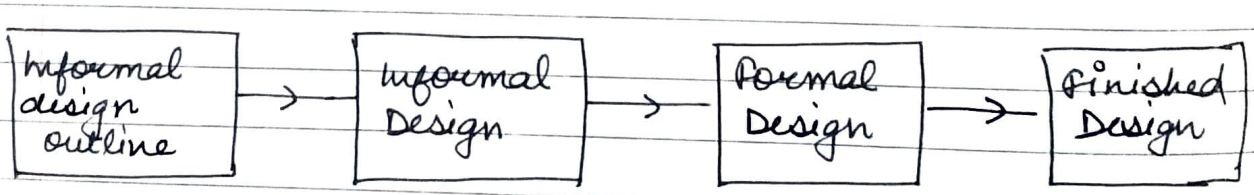
Basically two types → Conceptual.
→ Technical.



* Characteristics of Good Design

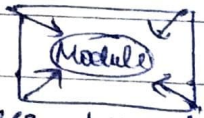
(i) Correct and complete
(ii) Understandable
(iii) Maintainable and to facilitate maintainance of the produced code.
(iv) Durability
(v) Utility
(vi) Directly Codeable (Difficult to achieve)

⇒ Finished design document is not achieved immediately, instead the design document is enhanced iteratively through a number of different phases.
⇒ The starting point is an informal design which is refined by adding information.
⇒ The design process includes adding details as the design is developed with constant backtracking to correct earlier designs. (examine mistakes)



Cohesion

Cohesion is a measure of the degree to which the elements of a module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Cohesion may be viewed as a glue that keeps the module together.



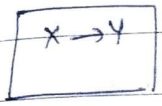
Modularization :- Process of dividing a software system into multiple modules where each module works independently.

Cohesion = Strength of relations within modules.

* A good software design will have high cohesion.

Types / classification of cohesion

Functional Cohesion : X & Y are present within a module and perform same functional task. (Highest desirable)



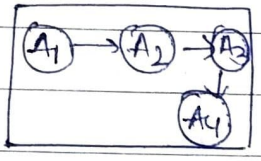
Sequential Cohesion : X's output → Y's input where X, Y ∈ same module.

Communicational Cohesion : X & Y both operate on same input data or contribute towards same output data.

* both elements affects each other

Procedural Cohesion : A1 → A2 → A3 ----->

* These modules are difficult to maintain.

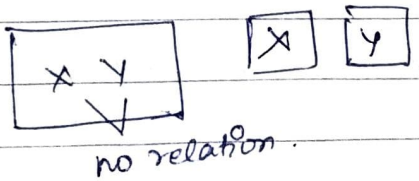


Temporal Cohesion : X and Y must perform around same time. ex: boot { } / shutdown { }.

Logical Cohesion : X and Y perform logically similar operations.

ex: X (percentage calc.) & Y (percentile calc.)

Coincidental Cohesion : X and Y have no conceptual relationship.

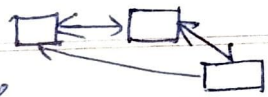


FS CP TL C : For solving Conceptual Problems take laboratory classes.

Best ↑ worst ↑

Order of desirability of cohesion ↓

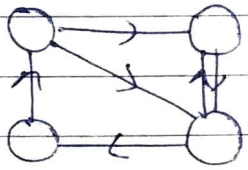
✱ Coupling



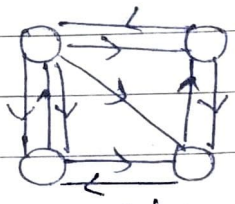
low coupling is preferable.

It is a measure of the degree of the interdependence between modules.

✱ Two modules with high coupling are strongly interconnected



low coupling



High coupling

loose coupling can be achievable by:

1. Avoid passing undesirable data calling module.
2. Controlling the no. of parameters passed amongst modules

✱ Types of Coupling

- Data coupling: If A and B communicate by only passing of data. (minimized data) (best)
- Stamp coupling: When complete data structure is passed from A to B. Tramp (Entire data) is passed.
- Control Coupling: If A and B communicate by passing of control information.
- External coupling: A module has dependency on other module. It includes communication to external tools and devices.
- Common Coupling: A and B have shared data. It can be difficult to determine which module is responsible for having set a variable to a particular value. (data is basically exposed)

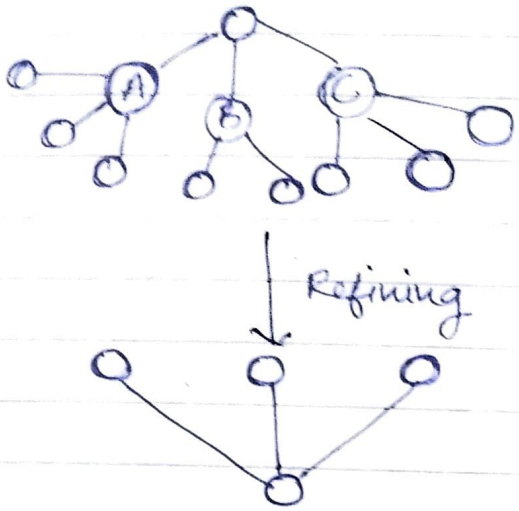
We can't determine which module changed the data.

- Content Coupling: One module can modify the data (least preferable) of another module or control flow is passed from one module to other module.

* Function Oriented Design

The design is decomposed into a set of interacting units where each unit has a clearly defined function. Basically, the system is designed from a functional viewpoint

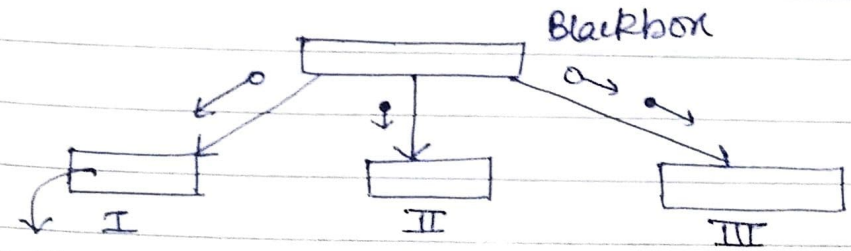
This type of design generally follows top-down :



{ Reusability of code will ↑ }

Strategies to implement function Oriented Design :

1. Data Flow Diagram (DFD) : It maps out the flow of information for a system
2. Data Dictionaries : They are repositories to store information about all data items in DFDs. They include Name of the items, Aliases, Description, Related data items, Range of values.
3. Structure charts :



Internal operations are not defined in Structure charts.

4. Pseudocode : Short, concise, English language phrases. They help to reduce external documentation.

* The whole concept of function oriented design relies on identifying functions which transforms inputs into outputs.

* Object Oriented Design

Object Oriented Design relies on focusing attention on the data instead of the function.

Objects: Object is an entity able to save a state or information and which offers a number of operations to either examine or affect this state.
The term entity/identity means that objects are distinguished by their inherent existence and not by descriptive properties.
Objects communicate by passing messages to each other.

Class : Set of objects that share common characteristics
Attribute : Data value held by the object in a class.
Operations : Functions or Transformation that may be applied to objects or by objects.

Inheritance, Polymorphism, Abstraction, Encapsulation.
Hierarchy : It arises due to inheritance.

Advantages :

1. Programs are easy to understand and maintain.
2. Programs are easy to test and debug.
3. Code Reusability.
4. Models are the real world.

Disadvantages : Proper knowledge is needed to implement. Design becomes tricky sometimes.

* User Interface Design

It is the front-end application view of System.
Features : Attractive, Simple to use, Interactive, Responsive.

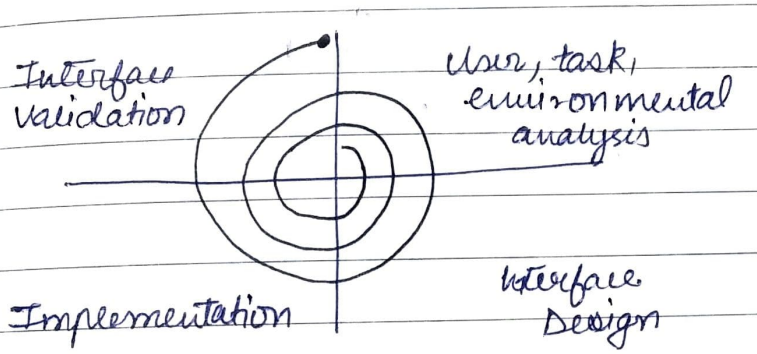
Types: Command line Interface (CLI)

→ Shows or operates interface with the help of commands.

Graphical User Interface (GUI)

→ Operated only graphical interface

Spiral Model for User Interface



I User, task, environmental analysis and modeling
Skill, knowledge and types of user, space, light or noise in user environment, other factors.

II Interface Design : Input, Output, tasks performed, response time, error handling

III Implementation : Creating prototype, Menues, error, error messages, commands and other interative components.

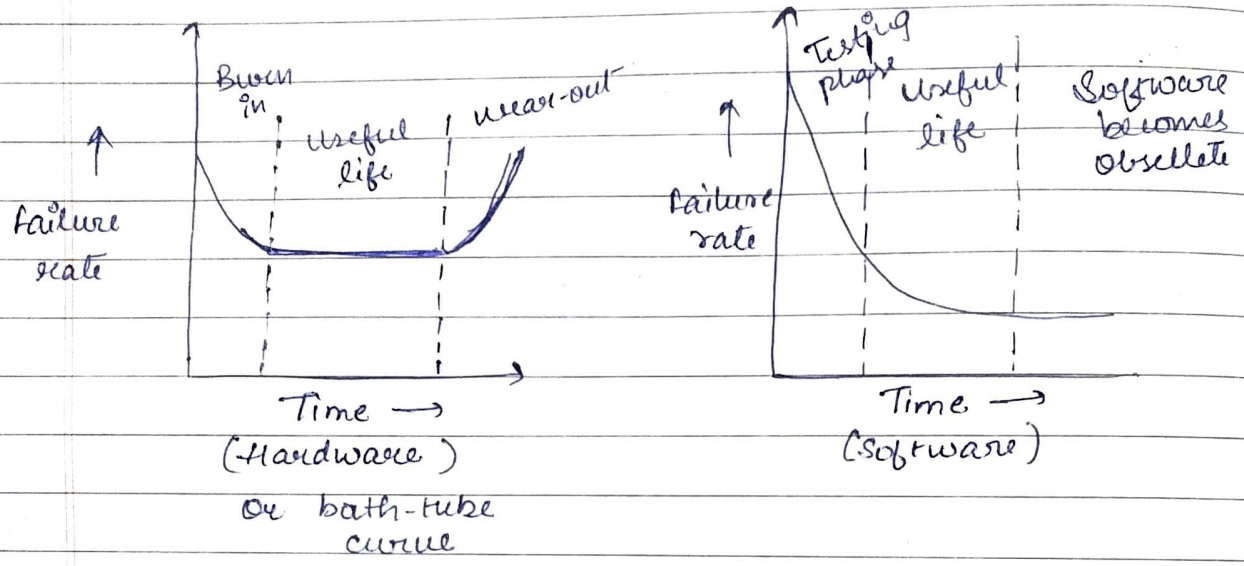
IV Validation : Testing the interface, all requirements of user must be met, interface must work properly, tasks must be performed correctly.

Characteristics of Good User Interface

1. Attractive, simple to use, interactive, responsive.
2. All interaction mechanisms must be supported
3. Hide technical internals from casual users.
4. Direct interaction with objects on screen.

5. Shortcuts must be defined.
6. Meaningful default settings
7. Consistency across a family of application.
8. Place the user in control.

Software Reliability



→ Why software becomes obsolete?

1. Change in environment
2. Change in Technology
3. Faster Alternatives
4. Poor GUI
5. Security issues

Q What is Software Reliability?

It is the probability of a failure free operation of a program for a specified time in a specified environment

If a system is employed by average user for a time say 10 hr and have a reliability of 0.95. This means the system would operate without failure for 95 of these periods out of 100.

* The source of failures in software is design faults while in hardware, failures are due to physical deterioration.

* Software reliability tends to change continually during test periods. (New problems + Repair action + New code)

* Failures & Faults

A fault is the defect in the program that when executed under particular conditions, causes a failure.

* The program must execute for a failure to occur.
* A fault can be source of more than one failure to occur.
* Fault is the property of the program rather than property of its execution.
* Fault is mostly created due to error by programmers.

Failure is the inability of a system or a component to perform required function according to its specification.

* The no. of faults is difference b/w introduced and removed.
* Faults might be introduced while modifying an existing code to remove faults.

How to find faults? Compiler diagnostics, code

Ways to characterize failure occurences in time:

1. time of failure
2. time interval between failures.
3. failures experienced in a time interval.

* All these quantities are random variables.

Failure behaviour is affected by:

1. The no. of faults in the software being executed
2. The execution environment.

Objective of Reliability Testing:

1. To discover the main course of failure.
2. To find no. of failures occuring in specific period of time.
3. To find perpetual structure of repeating failure.

Software Reliability Models

In reliability models, our emphasis is on failures rather than faults.

Software reliability models specifies the general form of the dependence of the failure process on the factors mentioned.

$\tau \equiv$ execution time (only time taken in execution) $t \equiv$ calender time

* The reliability of a program increases through fault correction and hence the failure intensity decreases

* Basic Execution Time Model / Basic Model

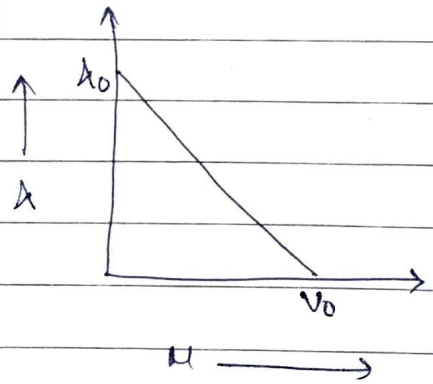
$$\lambda(\mu) = \lambda_0 \left[1 - \frac{\mu}{V_0}\right] \quad \text{—(1)}$$

λ : failure intensity

λ_0 = initial failure intensity at the start of execution

V_0 = Number of failures experienced, if program is run for infinite time period.

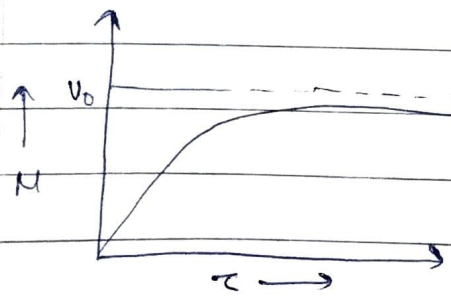
μ = Average or expected number of failures experienced at a given point of time.



$$\frac{d\lambda}{d\mu} = \frac{(-)\lambda_0}{V_0} \quad \text{—(2)}$$

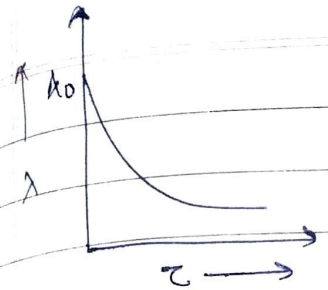
failure intensity always reduces

Negative sign signifies a decreasing trend in a failure intensity.



$$\mu(\tau) = V_0 \left[1 - \exp\left(\frac{-\lambda_0 \tau}{V_0}\right)\right] \quad \text{—(3)}$$

(failure experienced)



$$\lambda(\tau) = \lambda_0 \exp\left(\frac{-\lambda_0 \tau}{\nu_0}\right) \quad -(4)$$

(failure intensity)

$$\Delta u = \frac{\nu_0}{\lambda_0}(\lambda_p - \lambda_f) \quad -(5) \qquad \Delta\tau = \frac{\nu_0}{\lambda_0} \ln\left(\frac{\lambda_p}{\lambda_f}\right) \quad -(6)$$

λ_p = Present Failure intensity — Δu = additional failures

λ_f = Failure intensity objective — $\Delta\tau$ = additional time

Q Assume that a program will experience 200 failures in infinite time. It has now experienced 100. The initial failure intensity was 20 failures/CPU hr.

(i) Determine current failure intensity

(ii) Find the decreament of failure intensity

(iii) Calculate the failures experienced and failure intensity after 20 and 100 CPU hrs of execution.

(iv) Compute additional failures and additional execution time required to reach the failure intensity objective of 5 failures/CPU hr.

Use the basic execution time model for above calculations.

(i) $\nu_0 = 200 \quad M = 100 \quad \lambda_0 = 20$ failures/CPU hr

$$\lambda = \lambda_0\left[1 - \frac{M}{\nu_0}\right]$$

$$= 20\left[1 - \frac{100}{200}\right] = \frac{20 \times 1}{2} = 10$$

$$\boxed{\lambda = 10}$$

(ii) decreament in failure intensity (change in failure intensity)

$$\frac{d\lambda}{dM} = -\frac{\lambda_0}{\nu_0}$$

$$= -\frac{(\lambda - \lambda_0)}{\nu_0} = -\frac{(20-10)}{200} = \frac{-10}{200}$$

$$= -\frac{1}{20}$$

(iii) $M(\tau) = v_0 \left[1 - \exp\left(\frac{-\lambda_0 \tau}{v_0}\right)\right]$ $\tau = 20$

$= 200 \left[1 - \exp\left(\frac{-20 \times 20}{200}\right)\right]$

$= 200 \left[1 - \exp(-2)\right]$

$= 200 \left[1 - 0.14\right]$

$= 200 \left[0.86\right] = 172$ $\boxed{M(\tau) = 172}$

Similar for 100.

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

$= 20 \exp\left(\frac{-20 \times 20}{200}\right)$ $= 20 \times 0.14$

$\boxed{= 28}$

Ly, for 100

(iv) additional failure.

$\Delta u = \frac{v_0}{\lambda_0}\left(\lambda_p - \lambda_f\right)$

$\Delta u = \frac{200}{20}\left(10 - 5\right) = 10(5) = 50$

additional time

$\Delta \tau = \frac{v_0}{\lambda_0} \ln\left(\frac{\lambda_p}{\lambda_f}\right)$

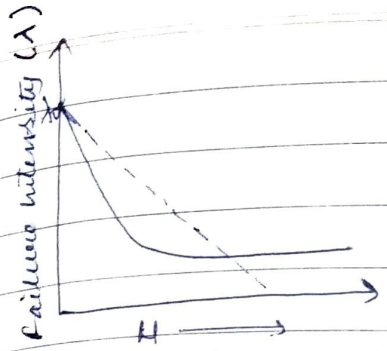
$= \frac{200}{20} \ln\left(\frac{10}{5}\right) = 10 \ln(2) = 10 \times 0.693$

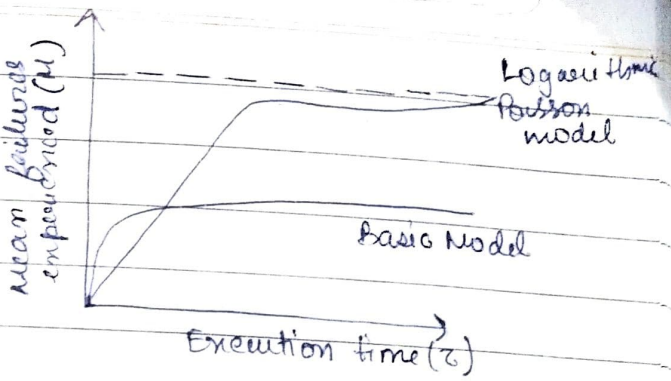
$= 6.93$

* Logarithmic Poisson Executive Time Model

$\boxed{\lambda(u) = \lambda_0 \exp(-\theta u)}$

θ = Failure intensity decay parameter
= It also represents the relative change of failure intensity per failure experienced.





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

$$\frac{d\lambda}{dH} = -\theta\lambda$$

*At larger values of execution time, the logarithmic model will have larger values of failure intensity.

$$\mu(\tau) = \frac{1}{\theta} \ln(\lambda_0 \theta \tau + 1)$$

$$\lambda(\tau) = \frac{\lambda_0}{\lambda_0 \theta \tau + 1}$$

$$\Delta\mu = \frac{1}{\theta} \ln\left(\frac{\lambda_P}{\lambda_F}\right)$$

$$\Delta\tau = \frac{1}{\theta}\left[\frac{1}{\lambda_F} - \frac{1}{\lambda_P}\right]$$

Ques Assume that the initial failure intensity is 20 failures/CPU hr. The failure intensity decay parameter is 0.02/failures. We have experienced 100 failures up to this time.

(i) Find current failure intensity.

(ii) Calculate the decreament of failure intensity per failure.

(iii) Find the failures experienced and failure intensity after 20 and 100 CPU hours of execution.

(iv) Compute the additional failures and additional execution time required to reach the failure intensity objective of 2 failures/CPU hour.

Use logarithmic poisson execution time model.

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

$\theta = 0.02$

$\lambda_0 = 20$

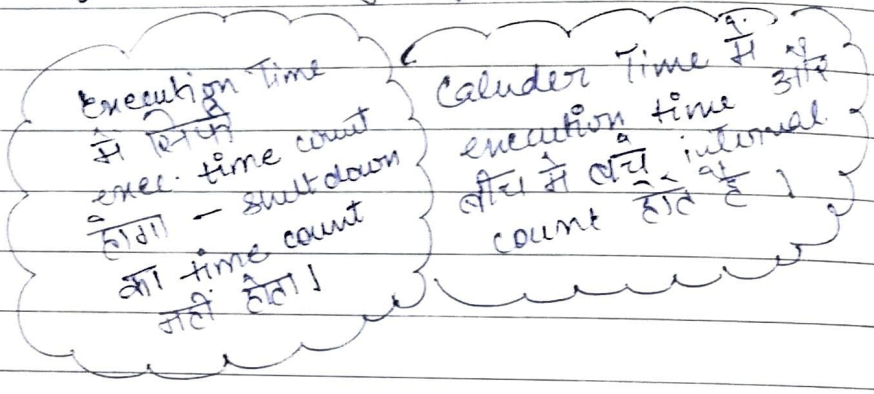
$M = 100$

$\lambda = 20 \exp(-0.02 \times 100)$

$= 20 \exp(-2) = 20 \times 0.14$

* Calender Time Component

* It is used to determine calender time to execution time ratio at any given point on time.
* The calender time component is of greatest significance during phases where the software is being tested and repaired. During this period one can predict the dates at which various failure intensity objectives will be met.



Resource Usage

$$x_r = \theta_r \tau + M_r M$$

θ_r = resource usage per CPU hr

M_r = resource usage per failure

x_r = Usage of resource

Usage parameters

Resources	CPU hr	failure
failure Ident	θ_x	M_I
failure Correct	0	M_F
~~Failure~~ Time	θ_c	M_C
~~Execution~~ Computer		

$x_c = M_C \Delta M + \theta_c \Delta \tau$

$x_F = M_F \Delta M$

$x_I = M_I \Delta M + \theta_I \Delta \tau$

→ use them in Question

$$\frac{dx_r}{d\tau} = \theta_r + M_r \lambda$$

where λ = failure intensity

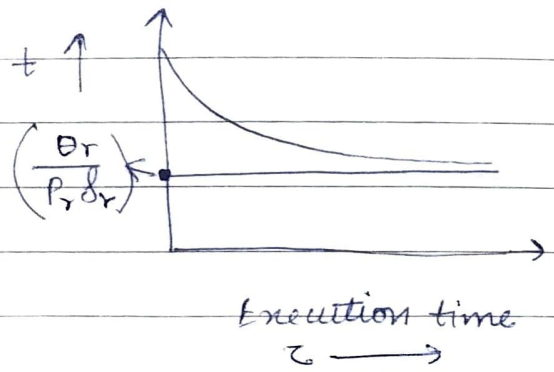
* Failure intensity decreases with execution time (testing)
* Resources quantities are assumed to be constant for period over which the model is being applied

Instantaneous ratio of calender time to execution time

$$\frac{dt}{dr} = \frac{dx_r}{dr}\left(\frac{1}{P_r \delta_r}\right) \cdot (\theta_r + \mu_r \lambda)\left(\frac{1}{P_r \delta_r}\right)$$

P_r = resource available

δ_r = utilization



* Software Testing

Processes:

Planning + Designing of Test Cases + Execution of program with test cases + interpretation of outcome + collection and management of data

Testing is the process of executing a program with the intent of finding errors. Software testing is the process of testing the software product.

Advantages: Higher quality software product, more satisfied users, lower maintainance cost.

Characteristics:

1. It is very expensive process.
2. Good testing involves much more than just running the program a few times.
3. It helps to raise quality and reliability of software.

↑ We have to demonstrate the program has errors instead of demonstrating that the program has no errors.

? Why should we Test?

The earlier the errors are discovered and removed, the lower is the cost of their removal.

* Who should Test?
Testing Team (seperately).

Errors (bugs) → Fault → Failure

* Set of test cases is called test suite.

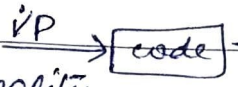
Testing : Verification + Validation
↓
difference.

Acceptance Testing : Software is developed for a specific customer. Customer validate all requirement.

Alpha Testing : Software is not for specific customer Potential customers are identical to conduct test at developer's site.

Beta Testing : Beta version is released.
(Ubuntu Studio → Beta testing)
→ low cost
→ customer कुछ बातें बता देते हैं problems
→ Most popular

* Functional Testing (Blackbox testing)
In this ~~process~~ approach, testing is based on the functionality of the program and is known as functional testing. It involves only observation of the output for certain input values ⇒ There is no attempt to analyze the code. Internal structure of code is ignored.
{ contents of black box are ignored }



* Various strategies are there to design test cases.

* Boundary Value Analysis

It is purely based on the concept that test cases that are close to boundary conditions have a higher chance of detecting an error. eg. $a \le x \le b$, $c \le y \le d$

$x = a, a+1, b, b-1$ → $4n+1$ possibilities

Robustness testing (testing in electrical circuit)

→ $6n+1$ testing at $(a-1, b+1)(c-1, d+1)$

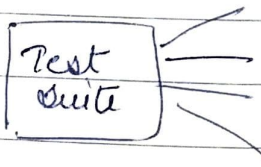
- $(a, -)(-, d)$
- $(a+1, -)(-, d-1)$
- $(b, -)(-, c)$
- $(b-1, -)(-, c+1)$
- $\left(\frac{a+b}{2}, -\right)\left(-, \frac{c+d}{2}\right)$

worst case testing:

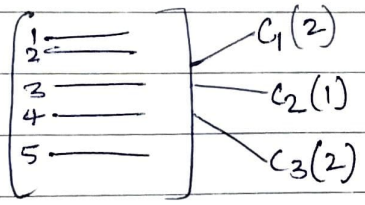
→ both values x, y will disturb
eg: $(a-1, c-1)$

* Boundary Value Analysis is not valid for Boolean Variables coz (it has 2 fix values T or F)

* Equivalence class Testing



divided into classes



Only 3 test cases will be checked instead of 5.

* The basic idea is to choose one test only one element for each equivalent class.

eg: Accept any no. b/w 1 & 99.

Equivalent test classes are :-

(i) Any no. b/w 1 & 99 $\{1, 5, 7, \dots\}$

(ii) Any no. less than 1 $\{-1, -2, 0, \dots\}$

(iii) Any no. >99

(iv) Every other character which is not a no. $\{\$, !, a\}$

* Equivalence classes should also be specified for input domain → Yes, for output domain.

* Decision Table Based Testing

ex: x, y, z sides to make Δ.

C_1: x, y, z are sides of Δ	N	Y							
C_2: x = y	–	Y				N			
C_3: x = z	–	Y		Z		Y		N	
C_4: y = z	–	Y	N	Y	N	Y	N	Y	N
a_1: Not a Δ	✓								
a_2: scalene									✓
a_3: Isosceles				✓		✓	✓		
a_4: Equilateral		✓							
a_5: Impossible			✓	✓	✓				

Limited entry Decision Table

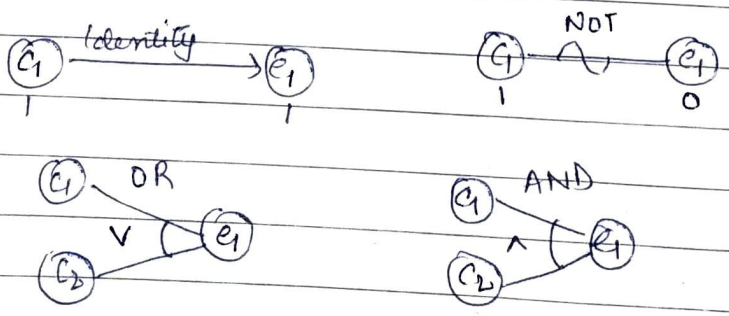
Extended Decision Table

~~x < y+z~~
x < y+z

* Cause Effect Graphing

One weakness of boundary value analysis and equivalence partitioning is that these do not explore combinations of input circumstances. They basically consider only simple input conditions.

* It is a method of generating test cases representing combinations of conditions

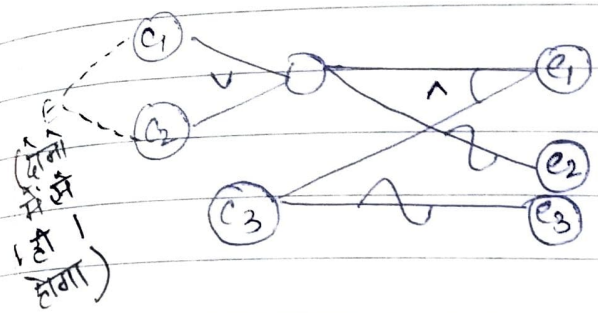


Basic cause effect graph symbol

e.g The characters in column 1 must be an A or B.
The characters in column 2 must be a digit.
If both above statement are true, file update is made

Causes: c_1: character in column 1 is A
c_2: character in column 1 is B
c_3: character in column 2 is a digit

Effects: e_1: update made (जब x और y में हो)
e_2: message x
e_3: message y



E: exclusive (atmost 1)
I: Inclusive (atleast 1 one होगा)
O: One and only one
R: For c_1 to be one,
R (c_1 1
c_2 1

Quest^n ex:

* Special Value Theory

Tester uses his or her domain Knowledge, experience with similar programs and information about 'soft spots'.
Also called as adhoc testing.
Heavily dependent on abilities of testing person.

* Structural Testing (white Box)

Unlike functional testing, internal structure is examined.
We don't pay attention to specifications, instead focus is on examination of program's logic.
It may find those errors, which may have missed by functional testing. It is also called dynamic white box testing. If we test without running the program, then it is called static white box testing.

I/P → [code] → O/P

Path Testing

It is completely based on selecting a set of test paths. We have to pick enough paths to assure that every source statement is executed atleast once

It is most applicable to units and modules and requires deep knowledge of program's structure.

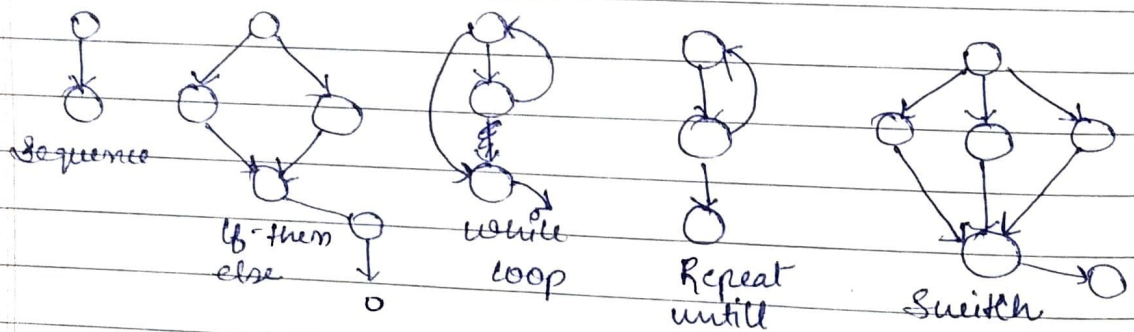
It involves

1. Generating set of paths that will cover every branch in the program.
2. Finding a set of test cases that will execute every path in this set of program paths.

(Applicable for short programs)

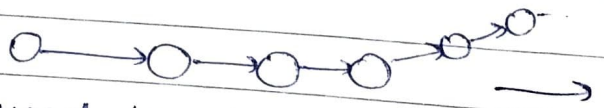
Path Testing

Flow Graph



DD path graph (Decision to Decision path graph)

We concentrate only on decision nodes.



It is used to find independent paths. We should execute all independent paths atleast once during path testing.

Independent Path: Any path through DD path graph that introduces atleast one new set of processing statements or new conditions ⇒ It must move along at least one edge that has not been traversed before.

* Good Path testing ensures:

(i) Every statement in the program has been executed atleast once.

(ii) Every branch has been exercised for true and false condition.

** Standard tools are there to make path graph.

Numerical of path graph :-

* Data Flow Testing

Another form of structural testing.

It concentrates on usage of variables.

(i) Statements where variables recieve values.

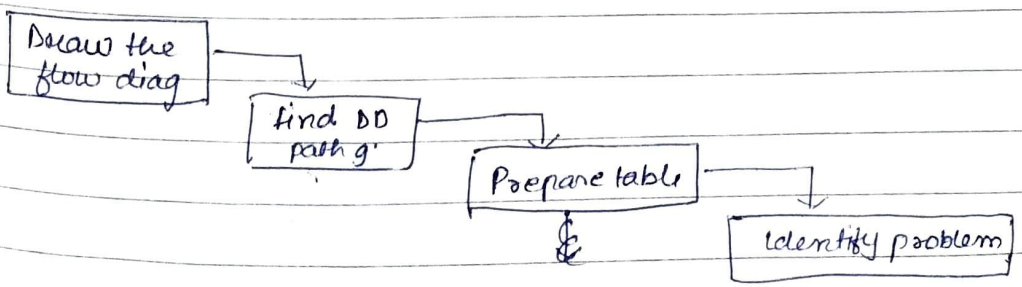
(ii) Statements where these values are used or recognized

What is Detect ?

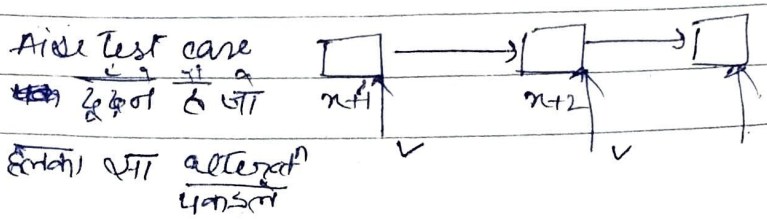
(i) Variable is defined by not recognised / used.

(ii) Variable used but never defined.

(iii) Variable defined more than once.



Mutation Testing



Mutations to program statements are made in order to determine properties about test cases.
Multiple copies of a program are made, and each copy is altered, this altered copy is called mutant
Mutants are then executed with test data to determine whether the test data are capable of detection the change between the original program and the mutated.
A mutat that is detected by a test cases is termed "killed" original → I order → II order → ...

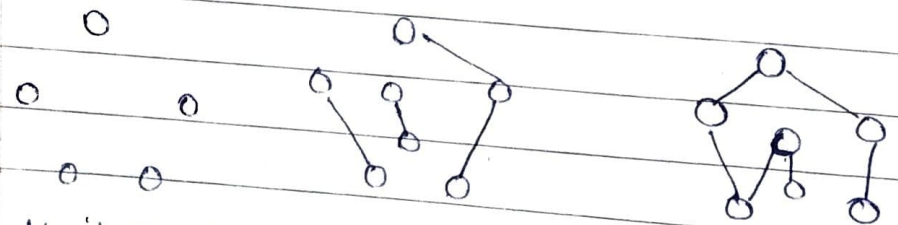
For specific test suite, mutants can be either killed, live or equivalent.

$$\text{Mutat}^n \text{ killed for a test suite} = \frac{\text{Killed}}{\text{total - equival}} \times 100$$

* Level of Testing

1. Unit Testing
2. Integration Testing
3. System Testing

* The Testing methods discussed in previous slides were applicable to unit testing.
* System Testing is functional rather than structural.



Unit Testing Integrat^n testing System Testing

Activities carried after or along testing :—
1. documentation, produced modification, discussion with customer to schedule installation, training material.

* Unit Testing (→ Integ. T → System)

Taking a module and running tests on it in isolation by using prepared test cases and interpreting the results.

Advantages:

1. Error location is easy
2. Prevents multiple error interaction

Problems: No one to call, No one to be called by, how to output?

Solution: Stubs: They are used to replace the modules which (scaffording) are called by the module to be tested.

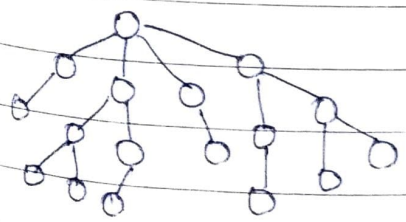
They do minimum data manipulation.

* Validation Testing is used in Unit Testing.
* Multiple units can be Tested parallely.
* Unit Testing can be left out sometimes also.

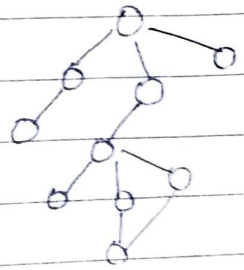
(Testing के लिए test case का Duplicate बनाने के लिए Testing हो जाती)

* Integration Testing

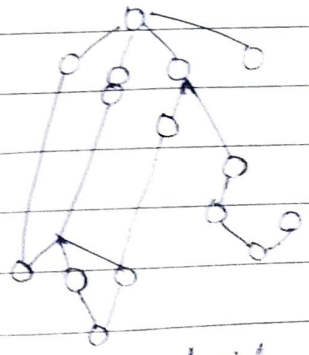
Main purpose is to check correct implementation of independent modules.



Top-Down Integration



Bottom-up Integration



Sandwich Integration

choose approach in such a way min^m no of non-tested cases will come across

* which one to apply?

* Not all modules are ready for integration
Top-down integration is an exercise in faith that every thing will work out well. while bottom-up integration shows that things really do work.

* we are moving away from structural testing and towards functional testing.

(focussing on functional testing instead of Structure)

* System Testing

* Testing System's capabilities is more important than Testing its component

→ We should Test old capabilities first rather than new ones

priority ↓

Usable	convenience, clarity
Secure	Sensitive Data
Compatible	Portability and Flexibility
Dependable	Methods for recovery of backup
Documented	User Manuals

Validation :- check complete software with customer's perspective.

Check all entries of SRS

* Debugging

Identifying the cause of error. It is the activity of locating and correcting errors.

Characteristics of Bugs:

1. System may lie in one part, cause might be in other part.
2. Symptom may disappear when other error is corrected

3. System may be intermittent
4. System might be due to human error and very difficult to trace.

Approaches:
1. Print Statement
2. Core Dumps
3. Trial & error
4. Backtracking
5. commenting or Removing
6. Induction & deduction

* can use print stat to check caught value

* Induction Approach

1. Keep the data handy: All available data, systems about the problem, different test cases which make the error & symptoms disappear.

2. Organize the data: Structure the data to observe patterns Search for specialized observations.
e.g. errors only occurs when no stock is left or stock = 0

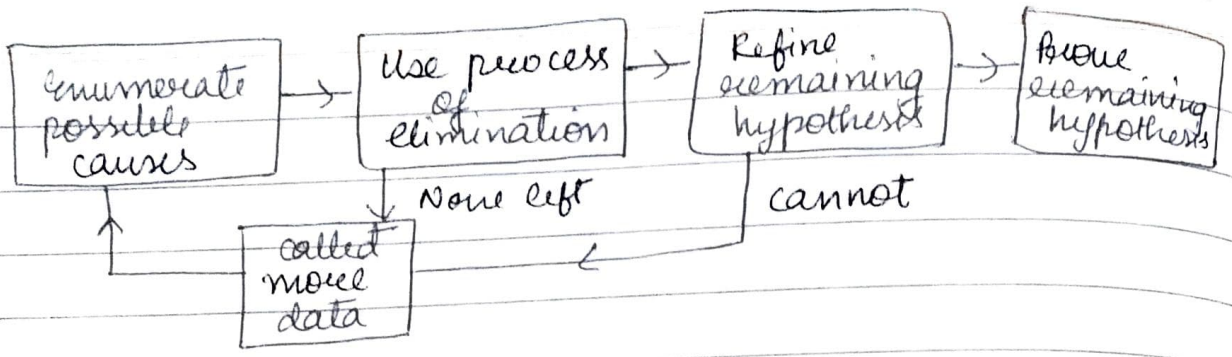
3. Devise a hypothesis: We have to theorize using all the data ~~collected~~. It will result in modification in facts or events instead of theory implementation

4. Prove the hypothesis: Don't skip this step ~~to~~ straitaway for fixing the problem. Failure to do this might reduce only a portion of problem.

* Deduction Approach

Data → Theory → hyp.
Theory → Data → Ded.

1. Enumerate the possible causes: Develop a list of all theories.
2. Use the data to eliminate them: If all are diminated, additional data is needed to devise new theories.
3. Refine the remaining hypothesis.
4. Prove the remaining hypothesis.



Debugging tools : Debugging compilers, automatic test cases generation, memory dumps, compiler diagnosis, compiler with meaningful error message and error detection features.

* Testing Tools (Static & Dynamic)
  - → Test-file generator (D) → code inspector (S)
  - → Test-data generation (D) → Standard enforcer (S)
  - → Output comparator (D) → Test Archiving system (D)

  (Static → general synt. error) (Dynamic → list all errors)

* Testing Standards (ISO/IEC 29119)

ISO/IEC	29119 -1 : Concepts and definitions of Software
ISO/IEC	29119 -2 : Test Processes
ISO/IEC	29119 -3 : Test Documentation
ISO/IEC	29119 - 4 : Testing Techniques
ISO/IEC	29119 -5 : Keyboard based software Testing
ISO/IEC	9126 : functionality, Reliability, efficiency, maintainability

Others :- IEEE 829, IEEE1061, 1059, 1008 - - -

* Software Maintainance

Error corrections + Enhancement of capabilities + deletion of obselette capabilities + optimization

Any work done to change the software after it is in operation is considered to be maintainence work.

Objectives / Advantages :

1. Preserves the value of software over time.
2. Expands the customer base.

3. Becoming easier to use for user.
4. Meeting additional requirement
5. Increases overall life span of software.

I Corrective Maintenance: It corrects design errors, logic errors and coding errors.
Emergency fixes → Solution → Patching
Problem → Increased program complexity and ripple effects.
(Prog. 1 Part में बदलाव
दूसरे Part पे effect आता)

II Adaptive Maintainance: Modifying the software to adopt to new environment. (Business rules + Government policies + work patterns + operating plateforms).

III Perfective Maintenance: Improving processing efficiency, or performance, adding more functionalities.
Results in a better, faster, smaller, better documented, software with these more functionality

IV Miscellaneous:
Management of Maintenence
** Survey of Lientz and Swanson

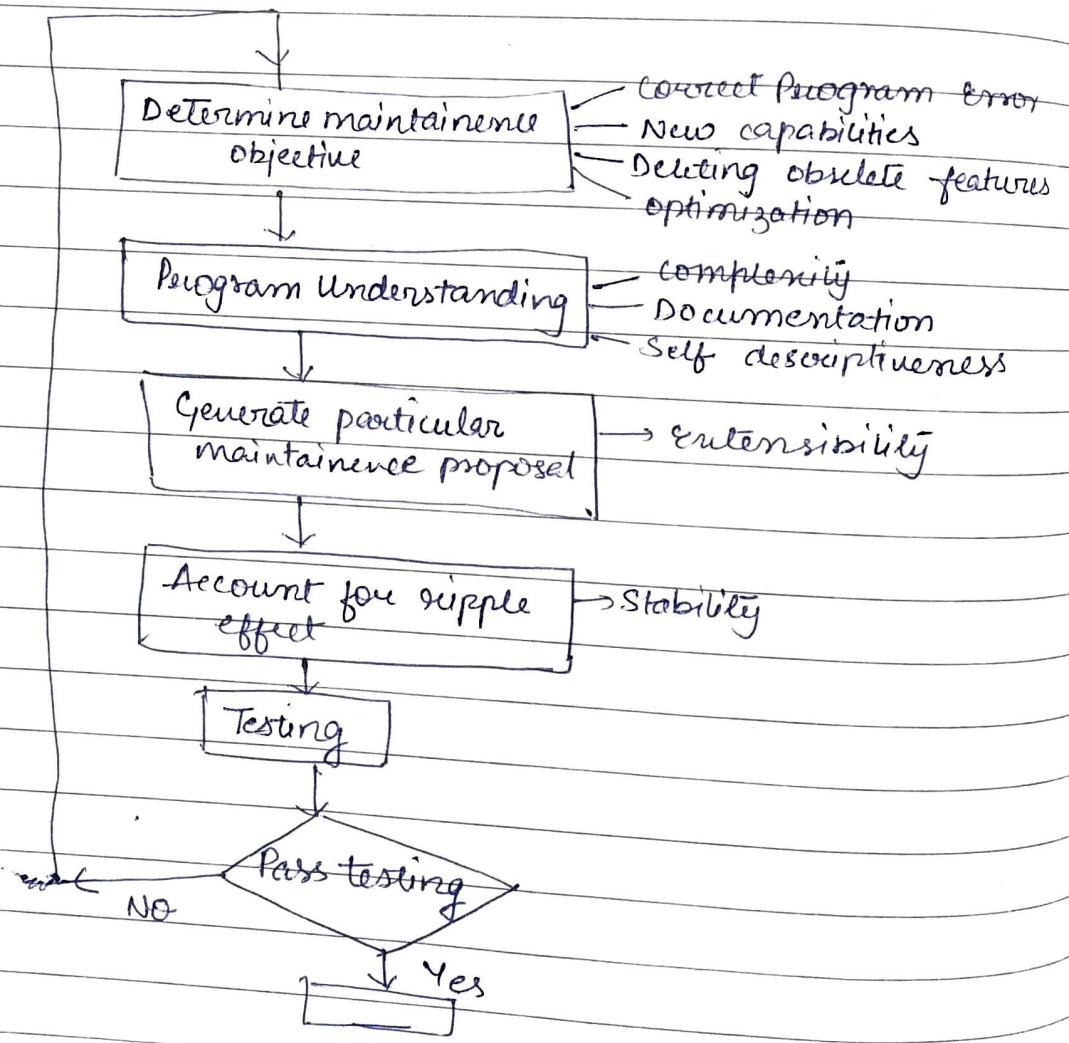
Emergency Debugging	12.4%	Distribution of effort in maintainence
Routine "	9.3%	
Data Environment Adaptat^n	17.3%	
Changes in OS & Hardware.	6.2%	
Enhancement for Users	41.8%	
Doumedat^n Improvement	5.5%	
Code Efficiency "	4.0%	
Others	3.5%	

All can be managed except (emergency debug)

✱ Problem during Maintainance

1. Program written by other person or group of person.
2. Program changed or altered by someone who didn't understand it clearly.
3. The ripple effect.
4. Absence of documentation.
5. Information gap or communication gap.
6. Complex Design of systems is hard to change.

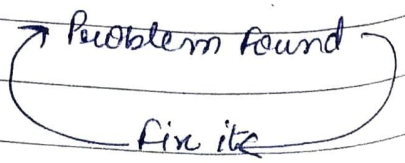
✱ Maintainance Process



✱ Maintainence Models

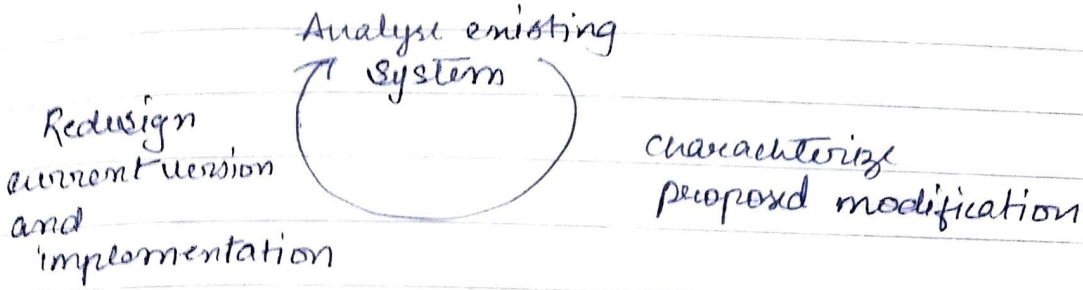
I Quick Fix Model

* Beneficial when system is developed and managed by single person.
* Quick & cheap.



* But it gives rise to long term problems.

II Iterative Enhancement Model



* No availability of full documentation
* Maintainence team can't analyse the existing product program completely

III Reuse Oriented Model

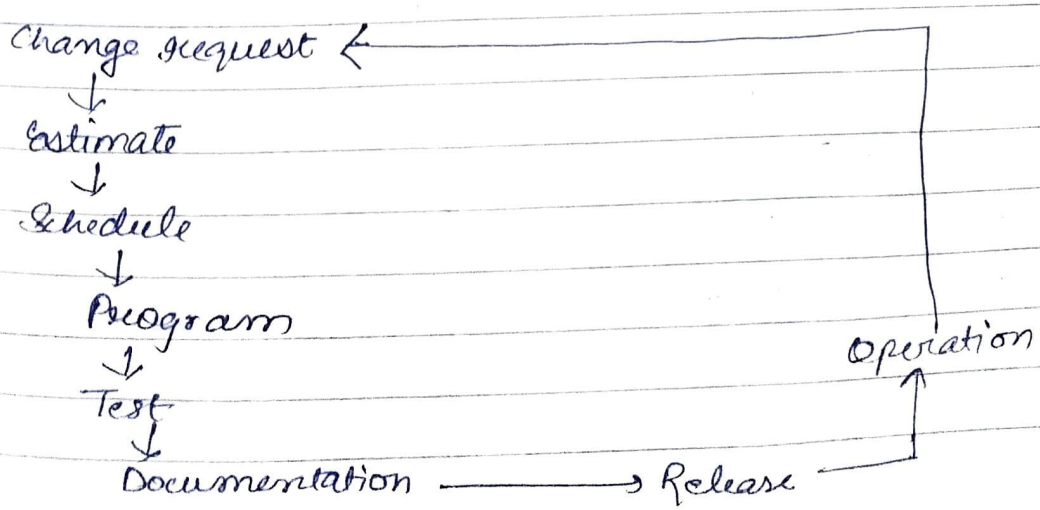
* Maintainance could be done by reusing the existing program components.

Steps:

1. Identification of the parts of the old system that are candidate for reuse.
2. Understanding these system part.
3. Modification of old system parts.
4. Integration of modified parts into the new system.

* It could be seen as a model which keeps on developing prototypes after prototypes.

IV Taute Maintainance Model



✱ Reverse Engineering

It is a process to recover information from the existing code or any other intermediate document.

Software Reverse Engineering is a process of recovering the design, requirement, specification, functions of a product its from analysis of code.

Implementation → Design → specification

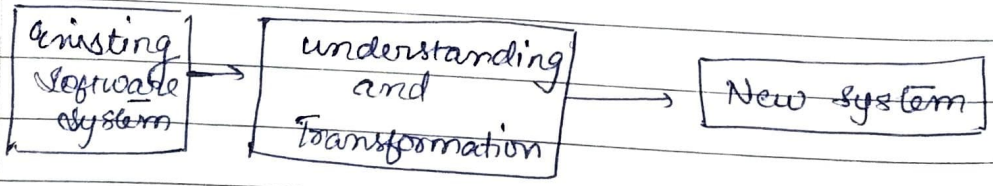
Steps:

(1) Collecting information
(2) Examining "
(3) Extracting the structure
(4) Recording the functionality
(5) Recording Data flow
(6) Recording Control flow
(7) Review Extracted Design
(8) Generate Documentation

✱ Software Re-engineering

Making improved and more maintainable systems from existing ones. It includes: Translating source code, restructure old code, migrate to new platform, re-documentation.

✓ Docum. → Reeng: D
✗ ReverseEng → New D



Cost factors: Quality of Software, Tools available, extent of conversion, availability of staff

* Configuration Model

Development of procedures and standards for cost effective managing and controlling changes in an evolving software system.

Procedure:

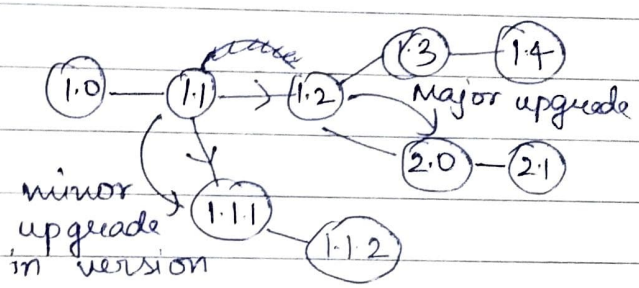
1. The identification of the components and changes.
2. The control of the way by which the changes are made.
3. Auditing the changes.
4. Status accounting-recording and documenting everything.

Document required: Project plan, Source code listing, SRS, testcases / test plan, SDD, User Manuals

* Version Control System

Different version upgrades are updated for any software time-to-time.

ex:



* Documentation

It is a written record of facts about a software system recorded with the intent to convey purpose, content and clarity.

Types: User documentation, System Documentation etc.

User Documentation: (Non-Technical)

contains description of all functionalities.

Contents:

1. System Overview
2. Installation Guide
3. Beginner's Guide
4. Reference Guide
5. Enhancement
6. Quick reference card
7. Networking & security

* System Documentation (Technical)

Analysis, specification, design, implementation, testing security error diagnosis and recovery.

Contents:

1. Objective of Entire System.
2. SRS
3. Specifications / Design information
4. Implementation
5. System Test Plan
6. Acceptance Test Plan
7. Data Dictionaries
8. Other types of Documentation: User manual, operator manual maintainance manual etc.