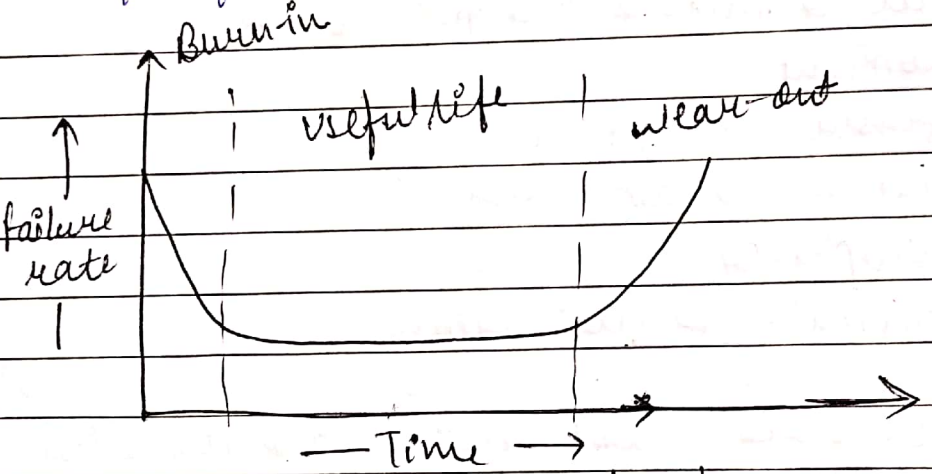
# Module-4

## Software Reliability, Testing and Maintenance

Software reliability is the probability of failure-free software operation for a specified period of time in a specified environment. Software reliability is also an important factor affecting system reliability. It differs from hardware reliability in that it reflects the design perfection, rather than manufacturing perfection. The high complexity of software is the major contributing factor of software reliability problems.

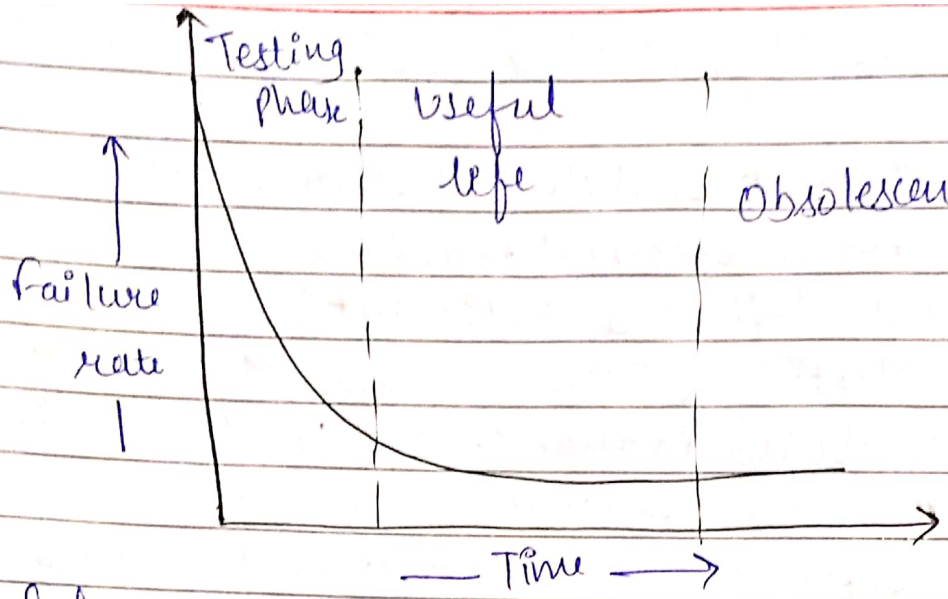



Bath tub curve for hardware reliability.

There are three phases in the life of any hardware component i.e. burn-in, useful life and wear out. In burn-in phase, failure rate is quite high initially, and it starts decreasing gradually as time progresses. It may be due to initial testing in the premises of the organisation.

During useful life period, failure rate is approximately constant. Failure rate increases in wear-out phase due to wearing out of components. The best period is useful life period.

We do not have wear out phase in software.




Software may be retired only if it becomes obsolete. Some of contributing factors are given below:

→ change in environment
→ change in infrastructure/technology
→ major change in requirements.
→ increase in complexity
→ extremely difficult to maintain.

According to IEEE standard, "Software reliability is defined as the ability of a system or component to perform its required functions under stated conditions for a specified period of time".

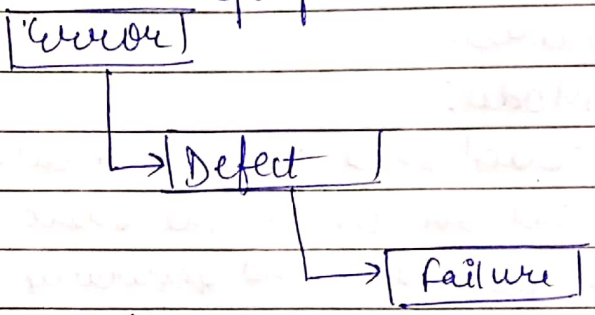
Software reliability is also defined as the probability that a software system fulfills its assigned task in a given environment for a predefined number of i/p cases, assuming that the hardware and the i/ps are free of error.

# Failures and Faults:

Fault:- Software fault is also known as defect, arises when the expected result don't match with the actual results. It can also be error, flaw, failure or fault

in a computer program. Not every fault causes a failure. In other words, a fault is a static software characteristics which causes a failure to occur.

Failure:- A failure corresponds to unexpected run-time behavior observed by a user of the software. It is defined as the deviation of the delivered service from compliance with the specification.



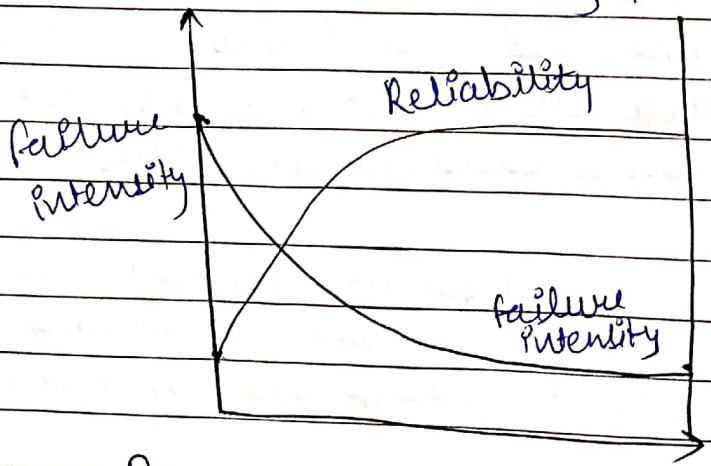

Reasons for failure:

↳ Environmental conditions, which might cause hardware failures or change in any of the environmental variables.

↳ Human error while interacting with the software by keying in wrong inputs.

↳ Failures may occur if the user tries to perform some operation with intention of breaking the system.

Failure behavior is affected by two principal factors:-




① The no. of faults in the software being

Reliability

## Software Reliability Models:-

A software reliability model indicates the form of random process that defines the software failure to

time. Software reliability models have appeared as people try to understand the features of how and why software fails, and attempt to quantify software reliability.

A reliability growth model is a numerical model of software reliability, which predicts how software reliability should improve over time as errors are discovered and repaired.

# Basic Execution Model:

This model was established by J.D. Musa in 1979 and it is based on execution time. The basic execution model is the most popular and generally used reliability growth model, mainly because:

1. It is practical, simple and easy to understand.
2. Its parameters clearly relate to the physical world.
3. It can be used for accurate reliability prediction.

The basic execution model determines failure behavior initially using execution time. Execution time may later be converted on calendar time.

The failure behavior is a nonhomogeneous Poisson process, which means the associated probability distribution is a Poisson process whose characteristic vary in time.

The mean value function is based on exponential distribution.

Variable involved:-

1. Failure intensity ($\lambda$): number of failures per unit time
2. Execution time ($\tau$): time since the program is running
3. Mean failures experienced ($\mu$): mean failures experienced in a time interval.

In a basic execution model, the mean failures experienced $\mu$ is expressed in terms of execution time $\tau$ as

$$\mu(\tau) = v_0 \times \left(1 - e^{-\lambda_0/v_0 \tau}\right)$$

where,

$\lambda_0$ → Initial failure intensity at the start of execution.

$v_0$ → total no. of failures occurring over an infinite time period; expected no. of failures to be observed eventually.

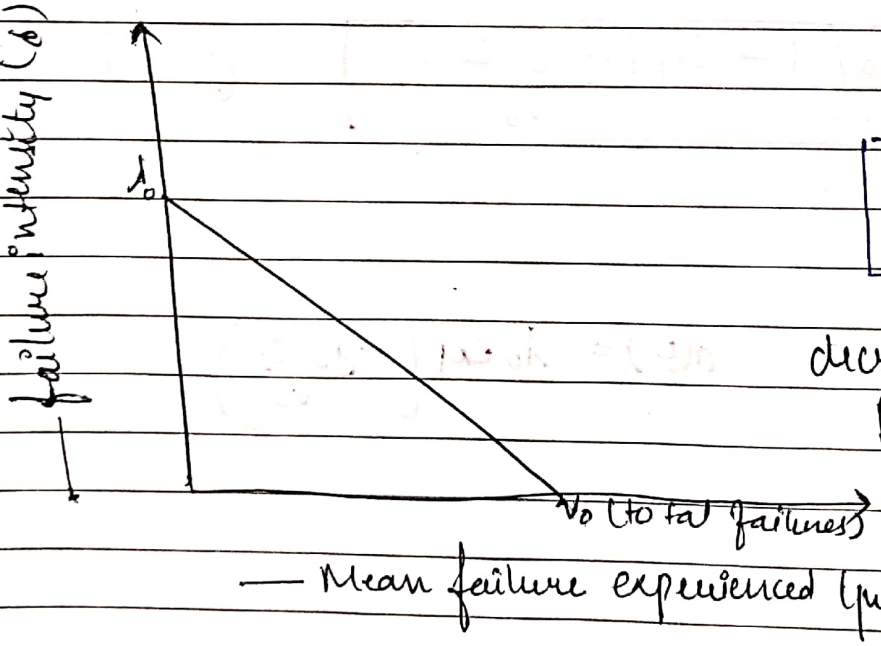
The failure intensity($\lambda$) expressed as a function of the execution time($\tau$) is given by

$$\lambda(t) = \lambda_0 \times e^{-\lambda_0/v_0 \tau}$$

The failure intensity $\lambda$ is expressed in terms of $\mu$ as

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

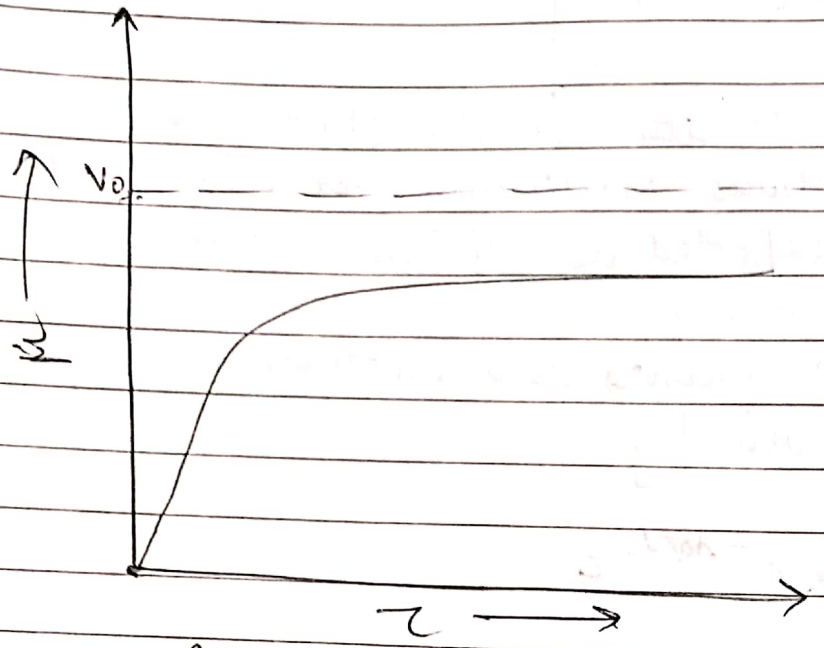
→ Current failure intensity.




$$\frac{d\lambda}{d\mu} = \frac{-\lambda_0}{v_0} \quad \text{——(2)}$$

↓

decrement of failure intensity per failure.



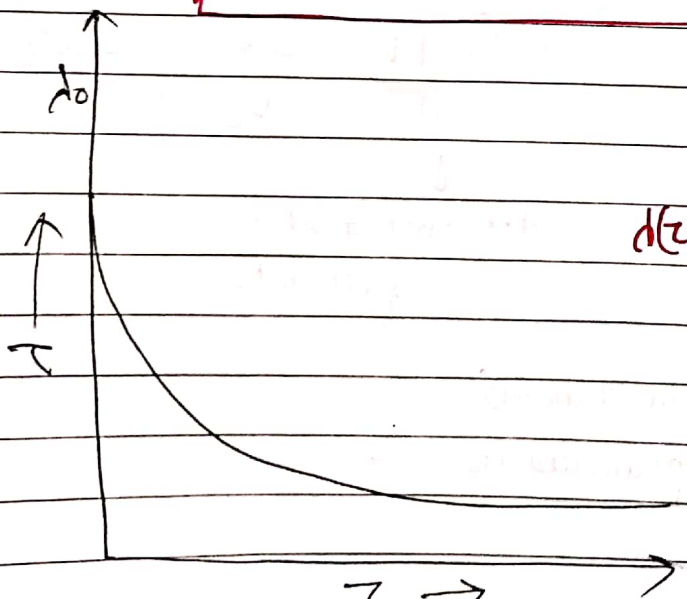

Relationship b/w $\tau$ and $\mu$.

eq$^n$. (1) can be written as

$$\frac{d\mu(\tau)}{d\tau} = \lambda_0\left(1 - \frac{\mu(\tau)}{V_0}\right)$$

When solved.

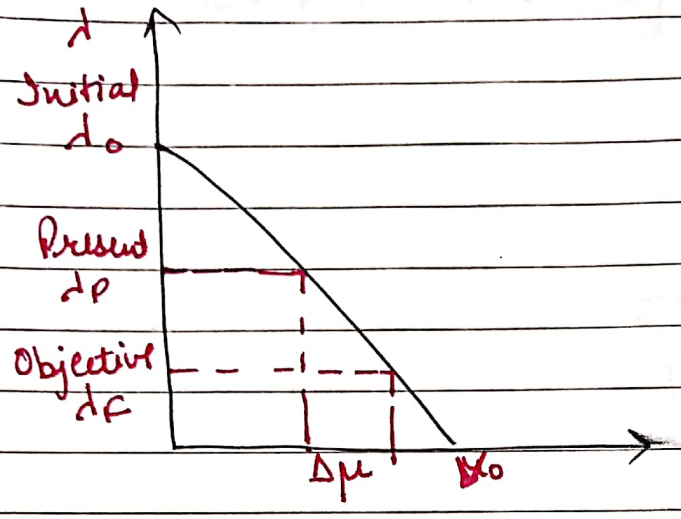
$$\mu(\tau) = V_0\left(1 - \exp\left(-\frac{\lambda_0 \tau}{V_0}\right)\right) \quad \text{---(3)}$$




$$\lambda(\tau) = \lambda_0 \exp\left(-\frac{\lambda_0 \tau}{V_0}\right)$$

Failure intensity vs execution time.

expected no of failures → $\Delta\mu$.
additional execution time → $\Delta\tau$.



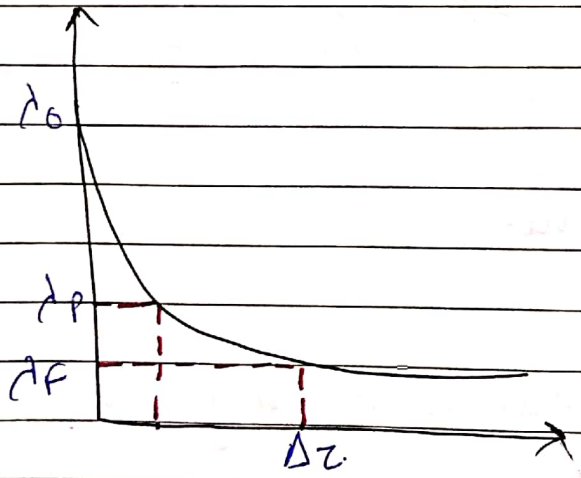

$$\Delta\mu = \frac{v_0}{\lambda_0}(\lambda_p - \lambda_F)$$

$\lambda_0$ → initial failure intensity.

$\lambda_p$ → Present failure intensity

$\lambda_F$ → Failure intensity objective.

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




Additional time required.

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

Example:- 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 the current failure intensity.

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

(iii) Find the decrement of failure intensity per failure.

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

Soln. $V_0 = 200$

$\mu = 100$

$\lambda_0 = 20$.

(i) current failure intensity.

$$\lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{V_0}\right)$$

$$= 20\left(1 - \frac{100}{200}\right)$$

$$= 20\left(1 - 0.5\right) = 20(0.5) = 10 \text{ failures/CPU hr.}$$

(ii) decrement of failure intensity per failure

$$\frac{d\lambda}{d\mu} = \frac{-\lambda_0}{V_0} = \frac{-20}{200}$$

$$\frac{d\lambda}{d\mu} = -0.1 \text{ /CPU hr.}$$

(iii) (a) failures experienced and failure intensity after 20 CPU hr.

$\tau = 20$

$$\mu(\tau) = V_0\left(1 - \exp\left(\frac{-\lambda_0 \tau}{V_0}\right)\right)$$

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

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

$= 200\,(1 - 0.135335)$

$\mu(\tau) = 173$ failures

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

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

$= 20 \exp(-2)$

$= 20(0.1353)$

$\lambda(\tau) = 2.71 \frac{\text{failures}}{\text{CPU hr}}$

(b) Failures experienced and failure intensity after 100 CPU hr.

$\tau = 100$ CPU hr.

Failure experienced $\mu(\tau) = \nu_0 \left(1 - \exp\left(\frac{-\lambda_0 \tau}{\nu_0}\right)\right)$

$= 20 \left(1 - \exp\left(\frac{-20 \times 100}{200}\right)\right)$

$= 20 \left(1 - \exp(-100)\right)$

$= 20$ failures (almost)

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

$= 20 \exp\left(\frac{-20 \times 100}{200}\right)$

$= 0.000908 \frac{\text{failures hr.}}{\text{CPU}}$

(4) Additional failures.

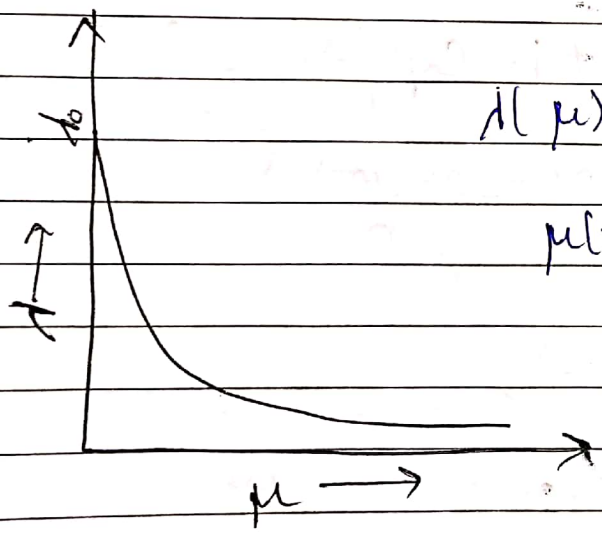
$$\Delta\mu = \frac{v_0}{\lambda_0}(\lambda_p - \lambda_f)$$

$$= \frac{200}{20}(10-5) = 50 \text{ failures}$$

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

$$= \frac{200}{20}\ln\left(\frac{10}{5}\right) = 6.93 \text{ CPUm.}$$

# Logarithmic Poisson Execution Time Model:-




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

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

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

$$\Delta\mu = \frac{1}{\theta}\ln\left(\frac{\lambda_p}{\lambda_f}\right)$$

$$\Delta\tau = \frac{1}{\theta}\left[\frac{1}{\lambda_f} - \frac{1}{\lambda_p}\right]$$

$\theta$ → failure intensity decay parameter

# Software Testing:-

Testing is a process of executing a program with the aim of finding error. To make our software perform well it should be error free. If testing is done succesfully it will remove all the errors from the software! In other words software testing is a verification and validation process.

| Verification | Validation |
| --- | --- |
| → Verification is to check whether the software confirms to specifications | → Validation is to check whether software meet the customer requirement. |
| → It is the process to make sure the product satisfies the cond". imposed at the start of development phase. | → Process to make sure the product satisfies the specified requirements at the end of development phase. |
| → It does not involve executing the code | → It involves executing the code. |

→ It is human based checking of documentation and files.
→ done by development team to provide that the S/W is as the specifications.

→ It is computer based execution of program.
→ carried out with the involvement of client and testing team.

Principles of Testing :-

- All the test should meet the customer requirements.
- To make our software, testing should be performed by third party.
- All the tests to be conducted should be planned before implementing it.
- It follows pareto rule (80/20 rule) which states that 80% of errors comes from 20% of program components.
- Start testing with small parts and extend it to large parts.

Test, Test case and Test suite:

Test and Test case terms are used interchangeably. Test case describes an i/p description and an expected o/p description. Inputs are of two types:

1) Preconditions → circumstances that hold prior to test case execution.
2) actual i/ps → identified by some testing methods.

Expected o/ps are also of two types, post conditions and actual outputs. Every test case will have an identification.

During testing, we set necessary preconditions, give required i/ps to program, and compare the observed o/p with expected o/p. If both are different, then, there is a failure and it must be recorded properly in order to identify the cause of failure. If both are same, then

there is no failure and program behaved in an expected manner.

A good test case has a high probability of finding an error.

| Test case ID: | |
|---|---|
| Section - I (Before execution) | Section - II (After execution) |
| Purpose: | Execution History: |
| Precondition: (If any) | Result: |
| I/Ps: | If fails, possible reason: |
| expected O/Ps: | Any other Observation? |
| Post conditions: | Any suggestion: |
| Written by: | Run by: |
| Date: | Date: |

Test case template

The set of test cases is called a test suite. Any combination of test cases may generate a test suite.

## Functional Testing:-

Functional testing refers to testing, which involves only observation of the output for certain input values. There is not attempt to analyse the code, which produces the output. The internal structure of the code is ignored. Therefore, the functional testing is also referred to as black box testing in which the contents of the black box are not known. Functionality of the black box is understood completely in terms of its inputs and outputs.

Here, functionality is focused rather than the internal structure of the code. There are a number of strategies or techniques that can be used to design test cases

which are succesful in detecting errors.

▷ Boundary value Analysis:-

Testcases that are close to boundary conditions have a higher chances of detecting an error. Boundary conditions means, an i/p value may be on the boundary, just below the boundary (upper side) or just above the boundary (lower side).

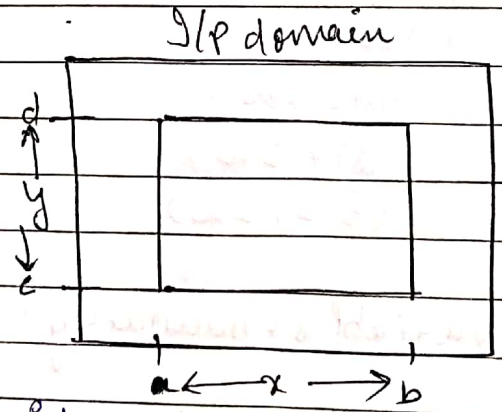
Suppose we have i/p variable x with a range from 1 to 100. The boundary values are 1, 2, 99, 100.

Consider a program with two input variables x and y. These i/p variables have specified boundaries as:

$$a \leq x \leq b$$

$$c \leq y \leq d$$

Both the i/ps are bounded by two intervals [a, b] and [c, d] resp. for i/p x, testcases are designed with values a and b, just above a and just below b. Similarly for y, testcases with values c and d, just above c and just below d. These test cases will have more chances to detect an error.




The basic idea of boundary value analysis is to use input variable values at their minimum, just above minimum, a nominal value, just below their max. and at their maximum. Here, reliability theory known as "single fault" assumption is assumed. This says that

failures are rarely the result of the simultaneous occurrence of two or more faults.
Thus, boundary value analysis test cases are obtained by holding the values of all but one variable at their nominal values and letting that variable assume its extreme values.

The boundary value analysis test cases for program
$100 \leq x \leq 300$
$100 \leq y \leq 300$
Nominal value = 200 (100+300/2).
Minimal value = 100
Maximum value = 300
value just above minimum = 101
" " below maximum = 299.
Possible test cases:- (200, 100)
(200, 101)
(200, 299)
(200, 300)
(200, 200)
(100, 200)
(101, 200)
(299, 200)
(300, 200)

For a program of n variables, boundary value analysis yields $4n+1$ test cases.

Example:- Consider a program for the determination of the nature of roots of quadratic equation. Its i/p is a triple of positive integers (say a, b, c) and values may be from interval [0, 100]. The program o/p may have one

of the following words:
[Not a quadratic eq$^n$., Real roots; Imaginary roots; Equal roots]
Design the boundary value test cases.

Sol$^n$. Quadratic eq$^n$. will be of type:

$ax^2+bx+c=0$

Roots are real if $(b^2-4ac)>0$
Roots are imaginary if $(b^2-4ac)<0$
Roots are equal if $(b^2-4ac)=0$
eq$^n$. is not quadratic if $a=0$

Max. value = 100
Min. value = 0
Total no. of test cases = 4(3)+1 = 13
Nominal value = 50
Just above min. = 1
Just below max = 99.

| Test case | a | b | c | Expected O/Ps |
|---|---|---|---|---|
| 1 | 0 | 50 | 50 | Not quadratic |
| 2 | 1 | 50 | 50 | Real |
| 3 | 50 | 50 | 50 | Imaginary |
| 4 | 99 | 50 | 50 | Imaginary |
| 5 | 100 | 50 | 50 | Imaginary |
| 6 | 50 | 0 | 50 | 〃 |
| 7 | 50 | 1 | 50 | 〃 |
| | ~~50~~ | ~~50~~ | | |
| 8 | 50 | 99 | 50 | 〃 |
| 9 | 50 | 100 | 50 | Equal. |
| 10 | 50 | 50 | 0 | Real |
| 11 | 50 | 50 | 1 | Real |
| 12 | 50 | 50 | 99 | Imaginary |
| 13 | 50 | 50 | 100 | Imaginary. |

Example:- Consider a program for determining the previous

date. Its i/p is a triple of day, month and year with values in the range

$1 \leq$ month $\leq 12$

$1 \leq$ day $\leq 31$

$1900 \leq$ year $\leq 2025$

The possible outputs would be previous date or invalid i/p date. Design the boundary value test cases.

| Sr | Month | Day | Year |
|---|---|---|---|
| Min | 1 | Min = 1 | Min = 1900 |
| Max. | 12 | Max = 31 | Max = 2025 |
| Nom. | 6 | Nom = 15 | Nom = 1962 |
| upper | 2 | upper = 2 | upper = 1901 |
| lower | 11 | lower = 30 | lower = 2024 |

Possible No. of test case = $4(3)+1 = 13$.

| Testcase. | Month | Day | Year. | Expected o/p |
|---|---|---|---|---|
| 1 | 6 | 15 | 1900 | 14 June, 1900 |
| 2 | 6 | 15 | 1901 | 14 June, 1901 |
| 3 | 6 | 15 | 1962 | 14 June, 1962 |
| 4 | 6 | 15 | 2024 | 14 June, 2024 |
| 5 | 6 | 15 | 2025 | 14 June, 2025 |
| 6 | 6 | 1 | 1962 | 31 May, 1962 |
| 7 | 6 | 2 | 1962 | 1 June, 1962 |
| 8 | 6 | 30 | 1962 | 29 June, 1962 |
| 9 | 6 | 31 | 1962 | 30 June, 1962 |
| 10 | 1 | 15 | 1962 | 14 January, 1962 |
| 11 | 2 | 15 | 1962 | 14 February, 1962 |
| 12 | 11 | 15 | 1962 | 14 Nov, 1962 |
| 13 | 12 | 15 | 1962 | 14 Dec, 1962. |

2.) Equivalence class Testing

Equivalence class Testing, also known as Equivalence class Partitioning (ECP) and Equivalence partitioning is a black box testing technique that applies to all levels of testing. It is used by the team of testers for grouping and partitioning of the test i/p data, which is then used for testing the software product into a number of different classes.

These classes resemble the specified requirements and common behavior or attributes of the aggregate i/ps.

The test cases are designed and created based on each class attribute and one i/p is used from each class for the test execution to validate the software functioning, and simultaneously validates the similar working of the software product for all other i/ps present in their respective classes.

It improves the quality of test cases, by removing the vast amount of redundancy and gaps that appear in the boundary value testing.

The I/p and the O/p domain is partioned into mutually exclusive parts.

Example:- If a variable x lies b/w 1 and 100 then different classes can be

(1) $1 \leq x \leq 100$

(2) $x < 1$

(3) $x > 100$

Example:- Previous date program with three i/ps, day, month, year.

$1 \leq month \leq 12$

$1 \leq day \leq 31$

$1900 \leq year \leq 2025$

Sol^n: outputs

$O_1$ → All valid i/ps

$O_2$ : Invalid date

$I_1$ : $1 \le m \le 12$

$I_2$ : $m < 1$

$I_3$ : $m > 12$

$I_4$ : $1 \le d \le 31$

$I_5$ : $d < 1$

$I_6$ : $d > 31$

$I_7$ : $1900 \le Y \le 2025$

$I_8$ : $Y > 2025$

$I_9$ : $Y < 1900$

} Equivalence classes

| Testcase | Month | day | year |
|---|---|---|---|
| 1 | 6 | 15 | 1962 |
| 2 | 0 | 15 | 1962 |
| 3 | 13 | 15 | 1962 |
| 4 | 6 | 0 | 1962 |
| 5 | 6 | 32 | 1962 |
| 6 | 6 | 15 | 2026 |
| 7 | 6 | 15 | 1889 |
| 8 | 2 | 31 | 1962 |

Qus) - Triangle problem ( $1 \le$ side $\le 100$)

Sol^n: $O_1$ : Equilateral Δ

$O_2$ : Isosceles Δ

$O_3$ : Scalene Δ

$O_4$ : Not a Δ

$I_1$ : $x < 1$

$I_2$ : $x > 100$

$I_3$ : $1 \le x \le 100$

$I_4$ : $y < 1$

$I_5$ : $y > 100$

$I_6$ : $1 \le y \le 100$

$I_7$ : $z < 1$

$I_8$ : $z > 100$

$I_9$ : $1 \le z \le 100$

$I_{10}$ : $x = y = z$.

$I_{11}$: $x = y$; $x \neq z$

$I_{12}$: $y = z$; $x \neq y$

$I_{13}$: $x \neq y$; $x \neq z$; $y \neq z$

$I_{15}$: $x = y + z$

$I_{16}$: $x > y + z$

$I_{17}$: $y = x + z$

$I_{19}$: $y > x + z$

$I_{20}$: $z = x + y$

$I_{21}$: $z > x + y$

| Testcase | x | y | z | o/p |
|---|---|---|---|---|
| 1 | 0 | 50 | 50 | Invalid i/p |
| 2 | 101 | 50 | 50 | Invalid i/p |
| 3 | 50 | 50 | 50 | Equilateral |
| 4 | 50 | 0 | 50 | Invalid i/p |
| 5 | 50 | 101 | 50 | Invalid i/p |
| 6 | 50 | 50 | 0 | Invalid i/p |
| 7 | 50 | 50 | 101 | Invalid i/p |
| 8 | 50 | 50 | 60 | Isoceles Δ. |
| 9 | 50 | 60 | 50 | " |
| 10 | 60 | 50 | 50 | " |
| 11 | 50 | 60 | 40 | Not a Δ. |
| 12 | 50 | 40 | 100 | " |
| 13 | 100 | 50 | 50 | " |
| 14 | 50 | 100 | 50 | " |
| 15 | 50 | 30 | 90 | " |

3. Decision Table Based Testing:-

The decision table is a software testing technique which is used for testing the system behavior for diff. i/p combinations. This is a systematic approach where the different i/p combinations and their corresponding

system behavior are captured in tabular form. It is known as the cause effect table because of an associated logical diagramming technique.

- Decision tables are very much helpful in test design technique.
- It helps tester to search the effects of combinations of different i/ps and other software states that must invoke various business rule.

There are four portions of a decision table name, condition stub, action stub, condition entries, action entries. When conditions $c_1$, $c_2$ and $c_3$ are all true actions $a_1$ and $a_2$ occur.

For a single problem

| Conditions | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $c_1$: x<y+z? | F | T | T | T | T | T | T | T | T | T | T |
| $c_2$: y<x+z? | — | F | T | T | T | T | T | T | T | T | T |
| $c_3$: z<x+y? | — | — | F | T | T | T | T | T | T | T | T |
| $c_4$: x=y? | — | — | — | T | T | T | T | F | F | F | F |
| $c_5$: x=z? | — | — | — | T | T | F | F | T | T | F | F |
| $c_6$: y=z? | — | — | — | T | F | F | F | F | F | T | F |
| $a_1$: Not a Δ | X | X | X | | | | | | | | |
| $a_2$: Scalene | | | | | | | | | | X | |
| $a_3$: Isoceles | | | | | | | X | | X | X | |
| $a_4$: equilateral | | | | X | | | | | | | |
| $a_5$: Impossible | | | | | X | X | | X | | | |

Example:

| Condition | | | | |
|---|---|---|---|---|
| c1: username | F | F | T | T |
| c2: Password | F | T | F | T |
| output | F | F | F | T |

Example:- Consider a program for the determination of previous date.

$1 \le m \le 12$

$1 \le day \le 31$

$1900 \le year \le 2025$

Sol:- The i/p domain can be divided into following domains

$I_1$: [$M_1$: month has 30 days]

$I_2$: [$M_2$: month has 31 days except March, August, January]

$I_3$: [$M_3$: Month is march]

$I_4$: [$M_4$: Month is August]

$I_5$: [$M_5$: Month is January]

$I_7$: [$D_1$: day = 1]

$I_8$: [$D_2$: $2 \le day \le 28$]

$I_9$: [$D_3$: day = 29]

$I_{10}$: [$D_4$: day = 30]

$I_{11}$: [$D_5$: day = 31]

$I_{12}$: [$Y_1$: year is a leap year]

$I_{13}$: [$Y_2$: year is a common year]

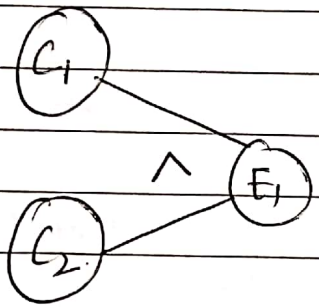
4.) Cause effect Graphing Technique:

Cause effect graphing technique is a technique that aids in selecting, in a systematic way, a high yield set of test cases. It has a beneficial effect in pointing out incompleteness and ambiguities in the specifications ~~that follow~~

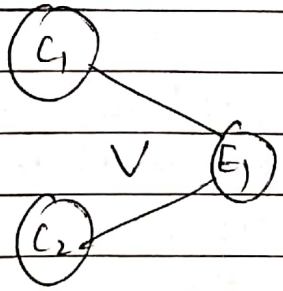
It comes under the black box testing technique which underlines the relationship between a given result and all the factors affecting the result. It is used to write dynamic test cases.

The dynamic test cases are used when code works dynamically based on user i/p. In this technique, the i/p

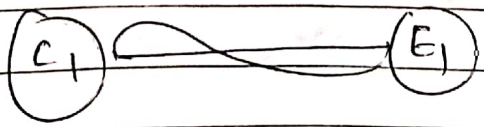
conditions are assigned with causes and the result of these i/p conditions with effects.
It is based on a collection of requirements and used to determine minimum possible test cases which can cover a maximum test area of software.
This technique converts the requirements specification into a logical relationship b/w the i/p and o/p conditions by using logical operators like AND, OR and NOT.
Notations used:-
① AND:- $E_1$ is an effect and C1 and C2 are the causes. If both C1 and C2 are true, then effect E1 will be true.



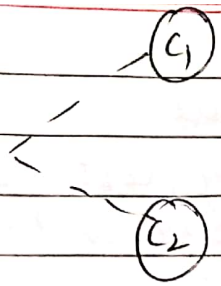

② OR:- If any cause from C1 and C2 is true, then effect E1 will be true.




③ NOT:- If cause C1 is false, then effect E1 will be true.




④ Mutually exclusive:- When only one cause is true.

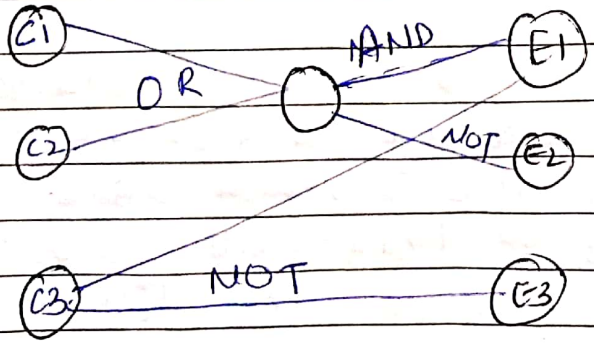



Example:- The character in column 1 should be either A or B and in the column 2 should be digit. If both columns contain appropriate values then update is made. If the i/p of column 1 is incorrect, i.e. neither A nor B, then X will be displayed. If the i/p in column 2 is incorrect i.e. i/p is not a digit, then message Y will be displayed.

- A file must be updated, if the character in the first column is either "A" or "B" and in the second colum it should be a digit.
- If the value in the first column is incorrect, then message X
- If value in the second column is incorrect, then message Y.

Sol^n, Causes: C1:- Character in column 1 is A
C2: Character in column 1 is B
C3: Character in column 2 is digit

Effects: E1 - update made (C1 OR C2) AND C3
E2 - X NOT C1 AND NOT C2.
E3 - Y (NOT C3)



## Structural (White Box) Testing:-

This type of testing permits us to examine the internal structure of the program. In this strategy, test cases are derived from an examination of program logic.

The knowledge of the internal structure of the code can be used to find the number of test cases required to guarantee a given level of test coverage. Simple objectives of structural testing are harder to achieve.

In this technique, all specifications are being checked against the implementation. It is necessary because there might be parts of code, which are not fully exercised by functional tests. It may find those errors, which have been missed by functional testing

### ⇒ Path Testing:-

Path Testing is the name given to a group of test techniques based on judiciously selecting a set of test paths through the program. If the set of paths is properly chosen, then it means that some measure of test thoroughness is achieved. It requires complete knowledge of the program structure and used by developers to test their own code. This type of testing involves:

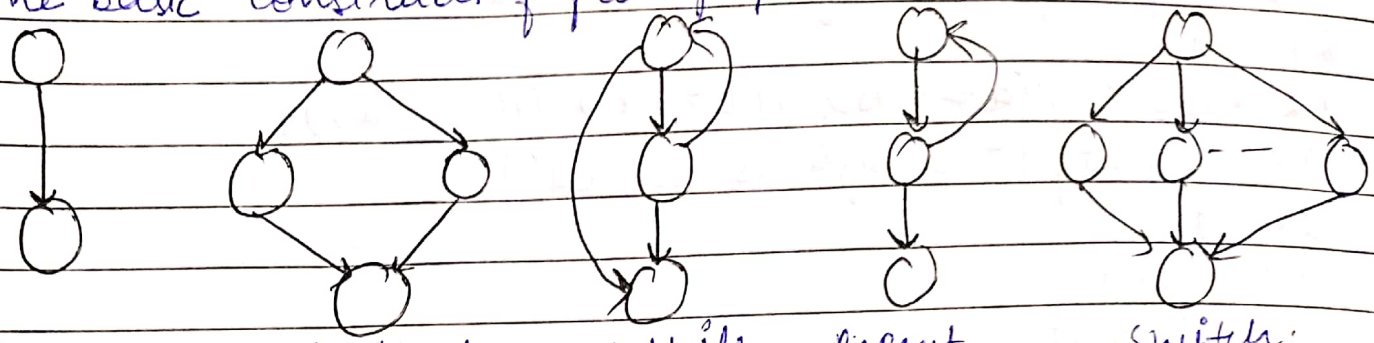
i) generating a set of paths that will cover every branch in the program.

ii) finding a set of test cases that will execute every path in this set of program paths.

### ↳ Flow Graph:-

The control flow of a program can be analysed using a graphical representation known as flow graph. The flow graph is a directed graph in which nodes are either entire statements or fragments of a statements and edges represents flow of control. The flow graph

can easily be generated from the code of any problem.
The basic constructs of flow graph are:

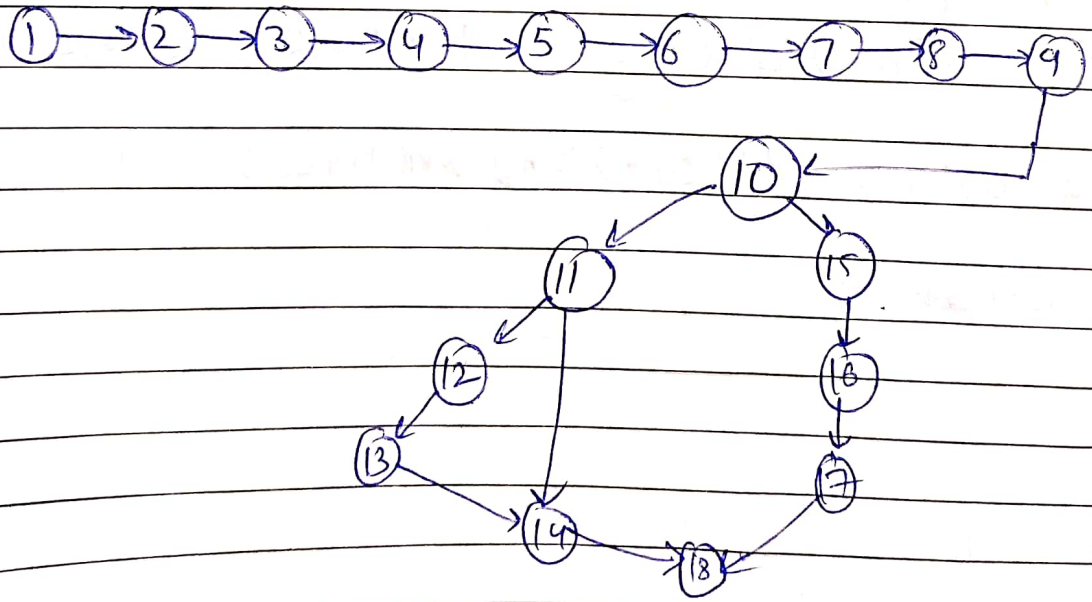


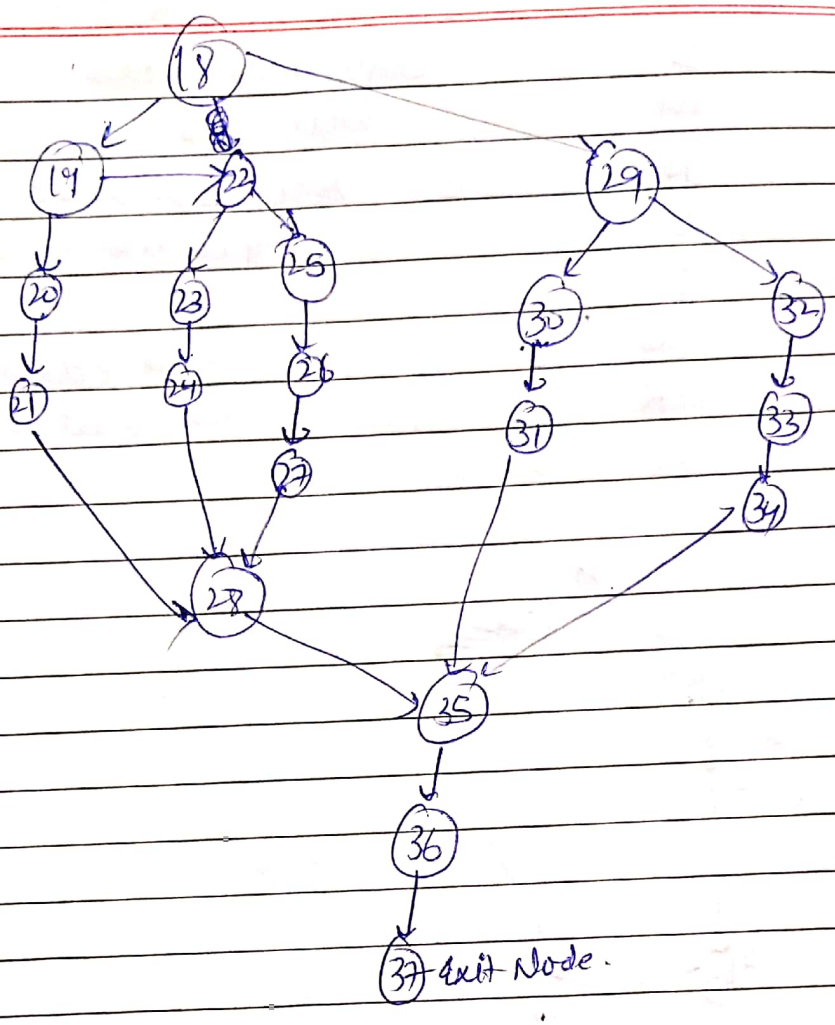
Sequence | If-then-else | While | Repeat until loop | Switch

Example:-

```
#include<stdio.h>
1. int main()
2 {
3   int a, b, c, boolean=0;
4   printf("Enter side-a");
5   scanf("%d", &a);
6   printf("Enter side-b");
7   scanf("%d", &b);
8   printf("Enter side-c");
9   scanf("%d", &c);
10  if((a>0)&&(a<=100)&&(b>0)&&(b<=100)&&(c>0)&&(c<=100))
    {
11      if((a+b)>c)&&((c+a)>b)&&((b+c)>a))
        {
12          boolean=1;
13      }
14  }
15  else {
16      boolean=-1;
17  }
18  if(boolean==1){
```

```
19  if((a==b)&&(b==c)){
20      printf("Triangle is equilateral");
21  }
22  else if((a==b)||(b==c)||(c==a)){
23      printf("Triangle is isoceles");
24  }
25  else{
26      printf("The triangle isscalene");
27  }
28  }
29  else if(boolean==0){
30      printf("Not a triangle");
31  }
32  else{
33      printf("The i/p belongs toinvalid range");
34  }
35  getch();
36  return 1;
37  }
```

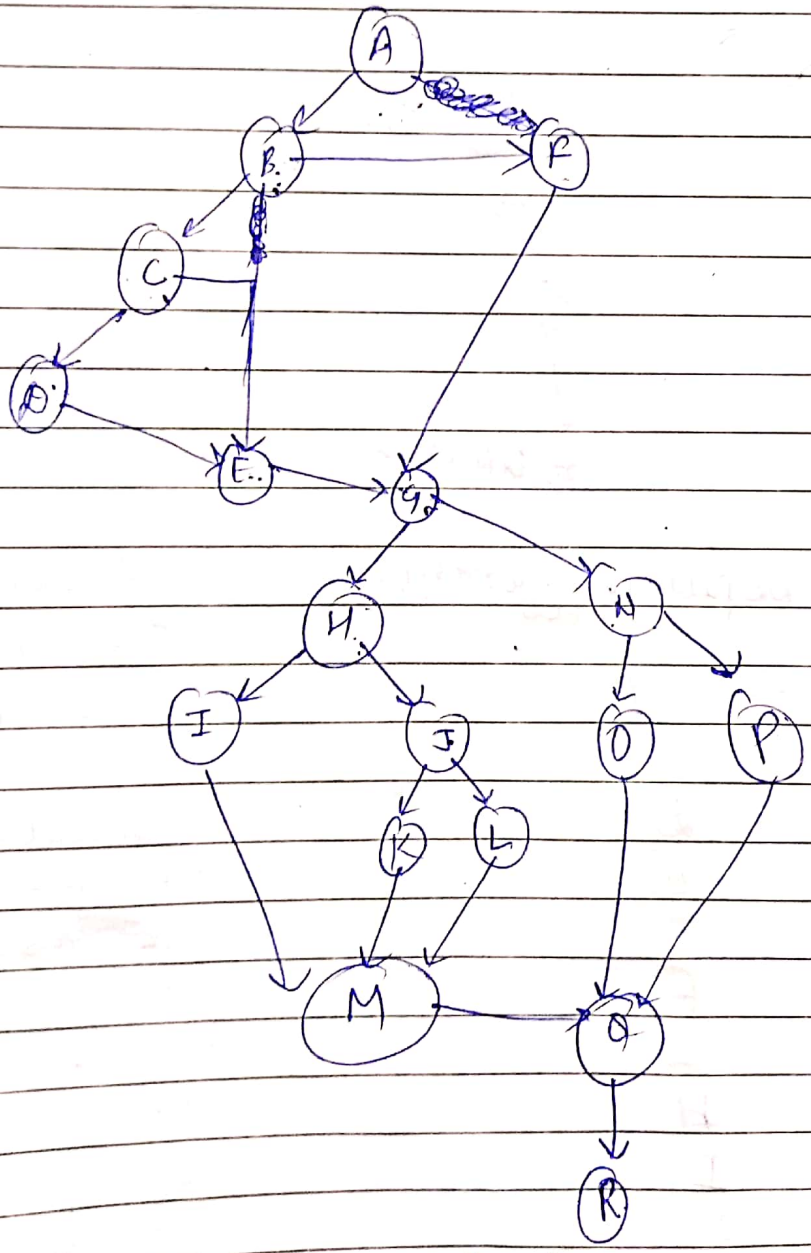





DD graph:-

| Flow graph nodes | DD path graph corresponding Nodes | Remarks |
|---|---|---|
| 1 to 9 | A | Sequential |
| 10 | B | Decisi° on node |
| 11 | C. | Decision node |
| 12, 13 | D | Sequential |
| 14 | E | Two edges are joined here. |
| 15 | ~~F~~ | ~~Decision~~ |
| 16, 17 | F | Sequential |
| 18 | ~~G~~ | Decision + 2 edge join |
| 19 | ~~H~~ | Decision |
| 20, 21 | I | Sequential |
| 22 | J | Decision Node |
| 23, 24 | K | Sequential node |

| | | |
|---|---|---|
| 25, 26, 27 | L | Sequential nodes |
| ~~26~~ 28 | M | Three edges are combined here |
| ~~28~~ 29 | N | Decision node. |
| 30, 31 | O | Sequential nodes |
| 32, 33, 34 | P | " |
| 35 | Q | Three edges are combined |
| 36, 37 | R | Sequential nodes |




2D Graph.

Independent paths

(i) A B F G N P Q R
(ii) A B F G H I M Q R
(iii) A B F G N O Q R
(iv) A B C E G N P Q R
(v) A B C D E G N O Q R
(vi) A B F G H J K M Q R
(vii) A B F G H J L M Q R

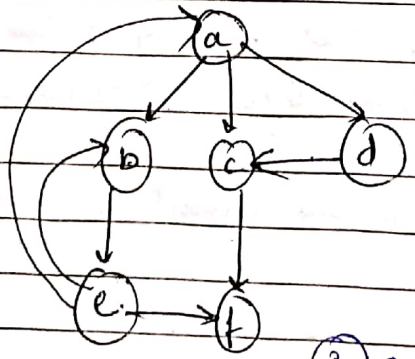
Cyclomatic Complexity:-

The cyclomatic complexity is also known as structural complexity because it gives internal view of the code. This approach is used to find the number of independent paths through a program. This provides the upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once and every condition has been executed on its true or false side.

The complexity measure is defined in terms of independent paths. An Independent path is any path through the program that introduces at least one new set of processing statements or a new condition.

Cyclomatic metric V(G) of graph G with n vertices, e edges, and P connected components is $V(G) = e - n + 2P$

e.g.




$n = 6$

$e = 9$

$P = 1$

$V(G) = 9 - 6 + 2$

$= 5$

(i) a c f (iii) a d f (v) a b e b e f.
(ii) a b e f (iv) a b e a c f

Several properties of cyclomatic complexity

(1) $V(G) \geq 1$

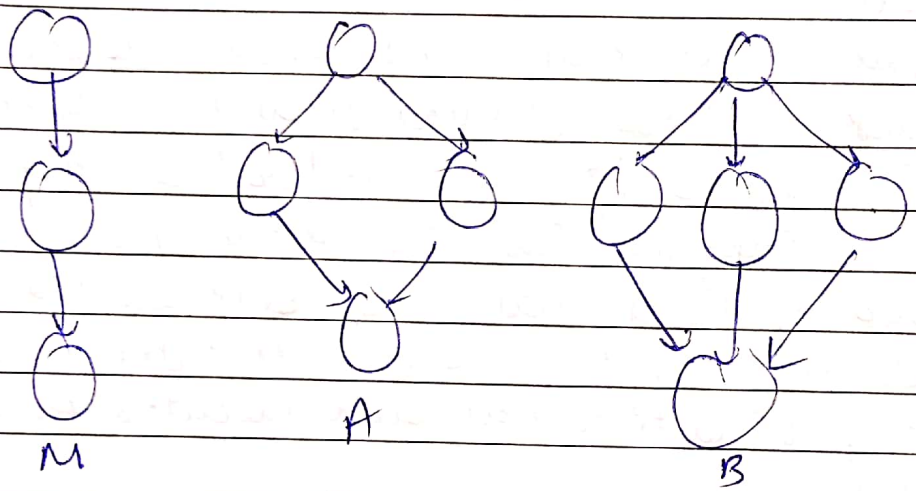
(2) $V(G)$ is max. no of independent paths in Graph.

(3) Inserting & deleting functional statements to G does not affect $V(G)$.

(4) G has only one path if and only if $V(G)=1$.

(5) Inserting a new row in G increase $V(G)$ by unity.

(6) $V(G)$ depends only on the structure of G.




In (M U A U B) , P=3

Two alternate methods for the complexity calculation

(1) Cyclomatic complexity $V(G)$ of a flow graph G is equal to the no of predicate (decision) nodes plus 1.

$$V(G) = \Pi + 1.$$

The restriction of this formula is that every predicate node should have only two outgoing edges ie. true or false.

(2) Cyclomatic complexity is equal to the number of regions of flow graph.

⇒ Data Flow Testing :-

Data flow testing is another form of structural

testing. It has nothing to do with the flow diagram. Here, concentration is on the usage of variables and the focal points are:

(i) Statements where variables receive values.
(ii) Statement where these values are used for reference.

* Flow graphs are also used as a basis for the data flow testing as in the case of path testing. Variables are defined and referenced throughout the program. Anomalies are:

(i) A variable is defined but not used.
(ii) A variable is used but never defined.
(iii) A variable is define twice before it is used.

These anomalies can be identified by static analyses if code is analysing code without executing it.

The definitions refers to a program P that has a program graph G(P) and a set of program variables V. The G(P) has a single entry node and a single exit node. The set of all paths in P is PATHS(P).

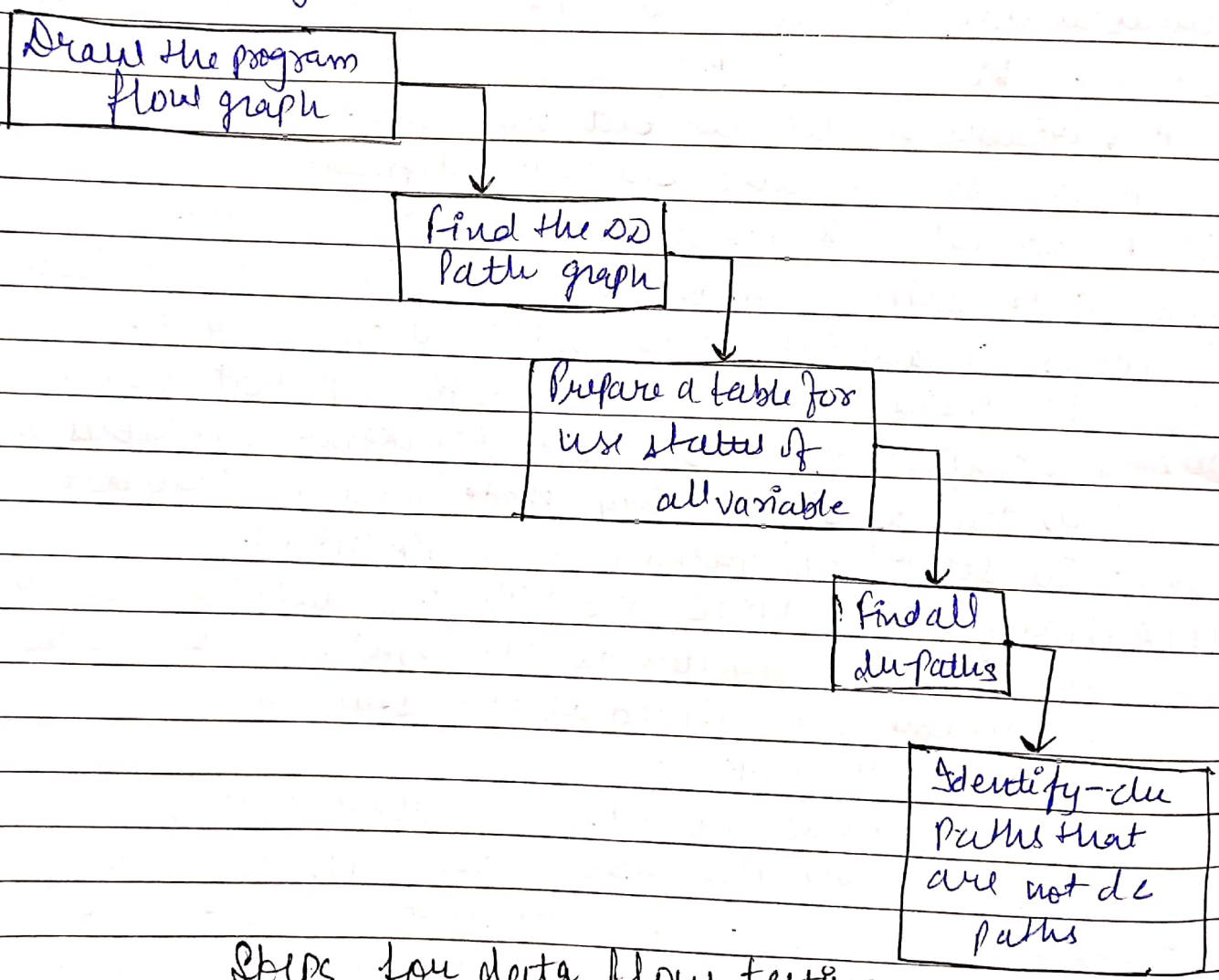
(i) Defining node:- Node $n \in G(P)$ is a defining node of the variable $v \in V$ written as DEF(v,n), iff the value of the variable v is defined at the statement fragment corresponding to node n.

(ii) Usage node:- Node $n \in G(P)$ is a usage node of the variable $v \in V$, written as USE(v,n), iff the value of the variable v is used at the statement fragment corresponding to node n. A usage node USE(v,n) is a predicate use iff statement n is a predicate statement otherwise USE(v,n) is a computation use (denoted as c).

(iii) Definition use: A definition use path with respect to a variable v (denoted du-path) is a path in PATHS(P) such that, for some $v \in V$, there are define and usage nodes

DEF(V, m) and USE(V, n) such that m and n are the initial and final nodes of the path.

iv) Definition clear: A definition clear path with respect to a variable V (denoted dc-path) is a definition use path in PATHS(P) with initial and final nodes DEF(V, m) and USE(V, n) such that no other node in the path is a defining node of v.




Steps for data flow testing.

# Mutation Testing :→

Mutation Testing is a type of software testing where we change certain statements in the source code and check if the test cases are able to find the errors. It is a type of whitebox testing which is mainly used for Unit Testing. The changes in mutant program are kept extremely small, so it does not affect the overall objective of the program. The goal of mutation testing

To assess the quality of the test cases which should be robust enough to fail mutant code. This method is also called as fault-based testing strategy as it involves creating a fault in the program.

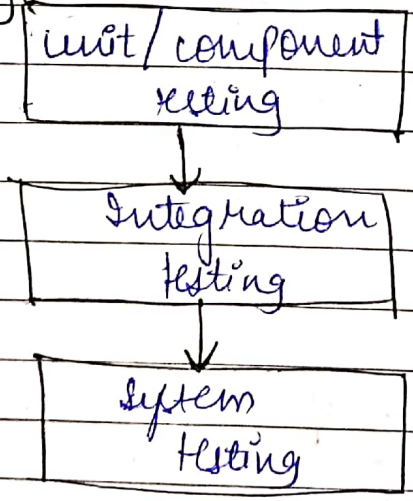
Types of mutation Testing:

(1) Value mutations:- In this type of testing values are changed to detect errors in the program. Basically a small value is changed to a larger value or a larger value is changed to a smaller value. In this testing basically constants are changed.

(2) Decision Mutation:
In decision mutation, logical or arithmetic operators are changed to detect errors in the program.

(3) Statement Mutation:
In statement mutation, a statement is deleted or it is replaced by some other statement.

# Levels of Testing:




(1) Unit/Component testing:- • The most basic type of testing is unit, or component testing.
• Unit testing aims to verify each part of the software by isolating it and then perform tests to demonstrate

that each individual component is correct in terms of fulfilling requirements and the desired functionality. This type of testing is performed at the earliest stages of the development process, and in many cases it is directly executed by the developers themselves before handing the software over to the testing team.
The advantage of detecting any errors in the software early in the day is that by doing so the team minimises software development risks, as well as time and money wasted in having to go back and undo fundamental ~~prog~~ problems in the program once it is nearly completed.

2. Integration Testing:
Integration testing aims to test different parts of the system in the combination in order to assess if they work correctly together. By testing the units in groups, any faults in the way they interact together can be identified. There are many ways to test how different components of the system function at their interface; testers can adopt either a bottom-up or a top-down integration method.
In bottom-up integration testing, testing builds on the results of unit testing by testing higher-level combinations of units in successively more complex scenarios.
It is recommended that testers start with this approach first, before applying the top-down approach which tests higher-level modules first and studies simpler ones later.

3. System Testing:
The next level of testing is system testing. All the components of the software are tested as a whole in order to ensure that the overall product meets the

requirements specified. System testing is a very important step as the software is almost ready to ship and it can be tested in an environment which is very close to that which the user will experience once it is deployed. System testing enables testers to ensure that the product meets business requirements, as well as determine that it runs smoothly within its operating environment. This type of testing is typically performed by a specialized testing team.

# Software re-engineering and Reverse engineering:-

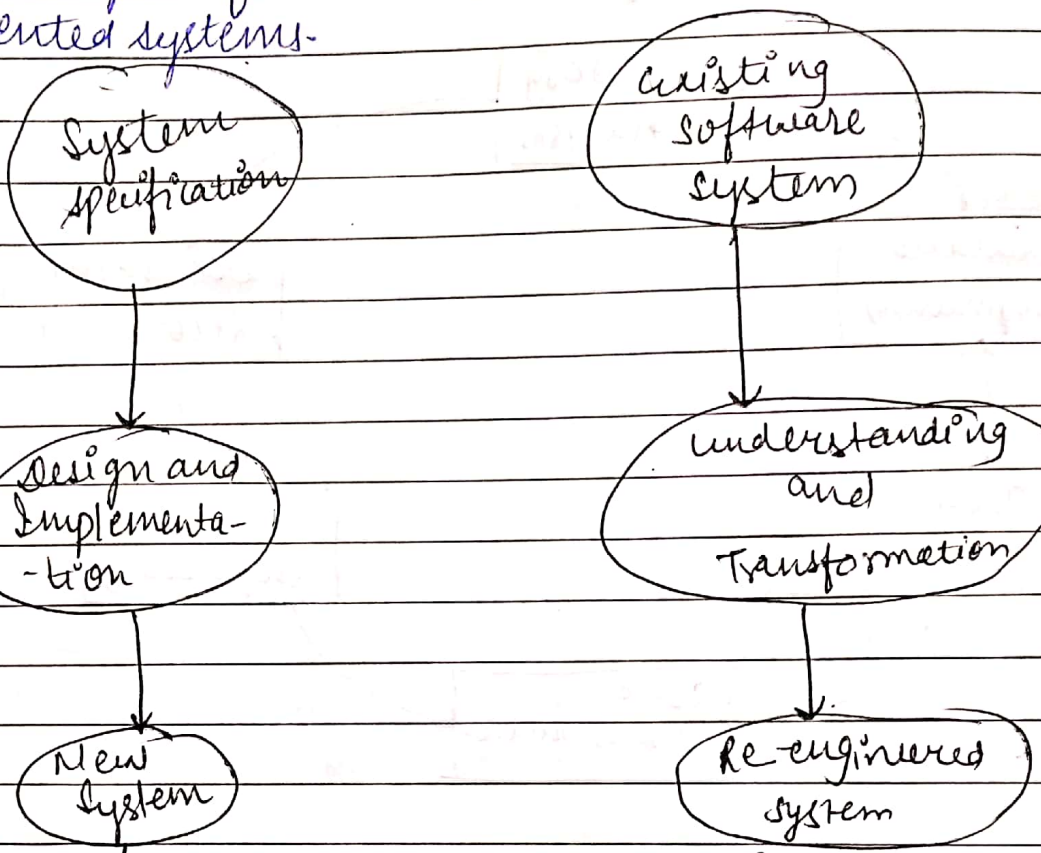
Software Re-engineering is a process of software development which is done to improve the maintainability of a software system. Re-engineering is the examination and alteration of a system to reconstitute it in a new form. This process encompasses a combination of sub processes like reverse engineering, forward engineering, reconstructing etc

Objectives of Re-engineering:

- To describe a cost-effective option for system evolution.
- To describe the activities involved in the software maintenance process.
- To distinguish b/w software and data re-engineering and to explain the problems of data re-engineering.

Software re-engineering is concerned with taking existing legacy systems and re-implementing them to make them more maintainable. As a part of this re-engineering process, the system may be redocumented or restructured. Software reengineering allows us to translate source code to a new language, restructure

old code, migrate to new platform and graphically display design information and re-document poorly documented systems.




a) New S/W development  b) Re-engineering.

Re-engineered cost factors:

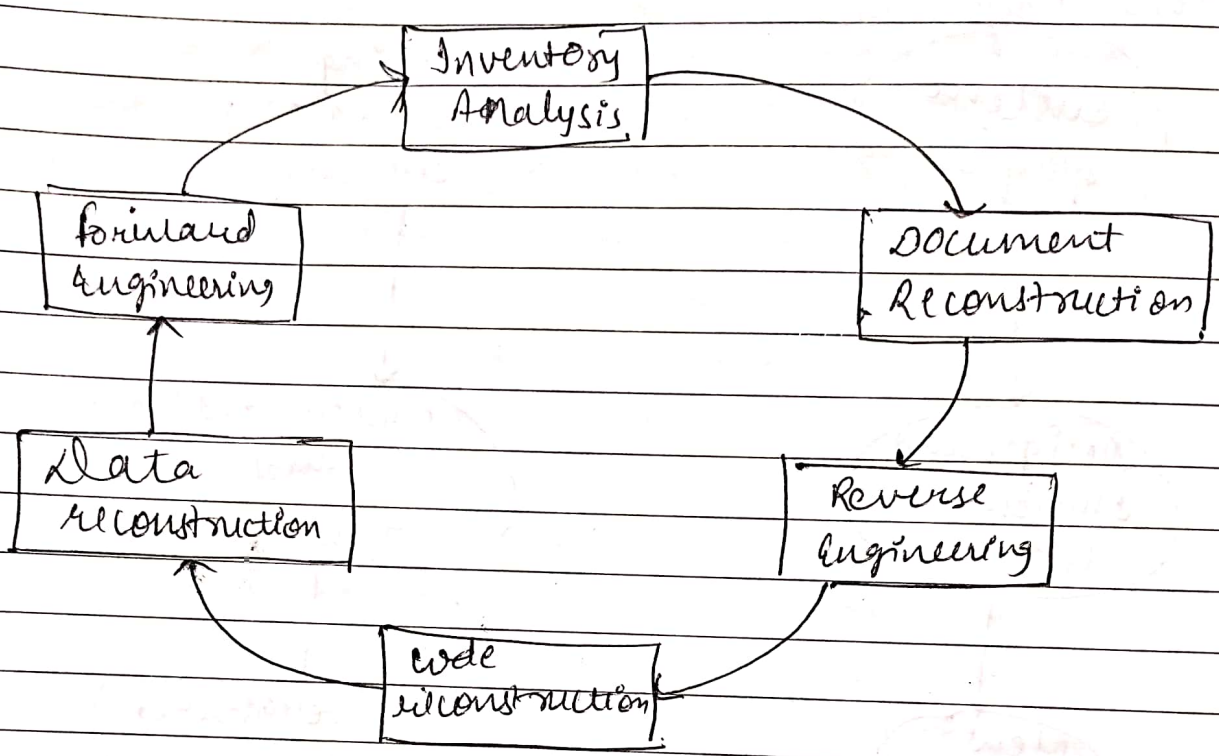
- The quality of the software to be re-engineered.
- The tool support available for re-engineering.
- The extent of the required data conversion
- The availability of expert staff for re-engineering.

Advantages:-

(1) Reduced Risk:
As the software is already existing, the risk is less as compared to new software development. Development problems, staffing problems and specification problems are the lots of problems which may arise in new software development.

(2) The cost of re-engineering is less than the costs of developing new software.

Reverse Engineering:
Steps involved in Re-engineering




Reverse Engineering:-
Reverse engineering is a process of recovering the design, requirement specifications and functions of a product from an analysis of its code. It builds a program database and generates information from this.
The purpose of reverse engineering is to facilitate the maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system.
Goals:-
- Cope with complexity
- Recover lost information
- Detect side effects
- Synthesis higher abstraction
- Facilitate reuse.

Steps:-

(1) Information collection: This step focuses on collecting all possible information (i.e. source design documents) about the software.

(2) Examining the information:
The information collected in step-1 is studied so as to get familiar with the system.

(3) Extracting the structure:
This step concerns with identification of program structure in the form of structure chart where each node corresponds to some routine.

(4) Recording the functionality:
During this step processing details of each module of the structure, charts are recorded using structured language like decision table, etc.

(5) Recording data flow:
From the information extracted, set of DFDs are derived to show the flow of data among the processes.

(6) Recording control flow:
High level control structure of the S/W is recorded.

(7) Review extracted design:
Design document extracted is reviewed several times to ensure consistency and correctness. It also ensures that the design represents the program.

(8) Generate documentation:
Finally, in this step, the complete documentation including SRS, design document, history etc. are recorded for further use.

# Maintenance:-

Software maintenance is a part of Software development life cycle. Its primary goal is to modify and update software application after delivery to correct errors and to improve performance. Software is a model of the real world. When the real would changes, the software require alteration wherever possible.

Software maintenance is an inclusive activity that includes error corrections, enhancement of capabilities, deletion of obsolete capabilities, and optimization.

Need for maintenance:

i) Correct errors
ii) Change in user requirement with time
iii) Changing hardware/ software requirements.
iv) To improve system efficiency.
v) To optimize the code to run faster.
vi) To modify the components

Thus the maintenance is required to ensure that the system continues to satisfy user requirements.

Causes of Software Maintenance Problems:

① Lack of Traceability:-

→ Codes are rarely traceable to the requirements and design specifications.

→ It makes it very difficult for a programmer to detect and correct a critical defect affecting customer operations.

→ Life cycle documents are not always produced even as a part of development project.

② Lack of code process:-

Most of the software system codes lack adequate comments. Lesser comments may not be helpful in

certain situations.

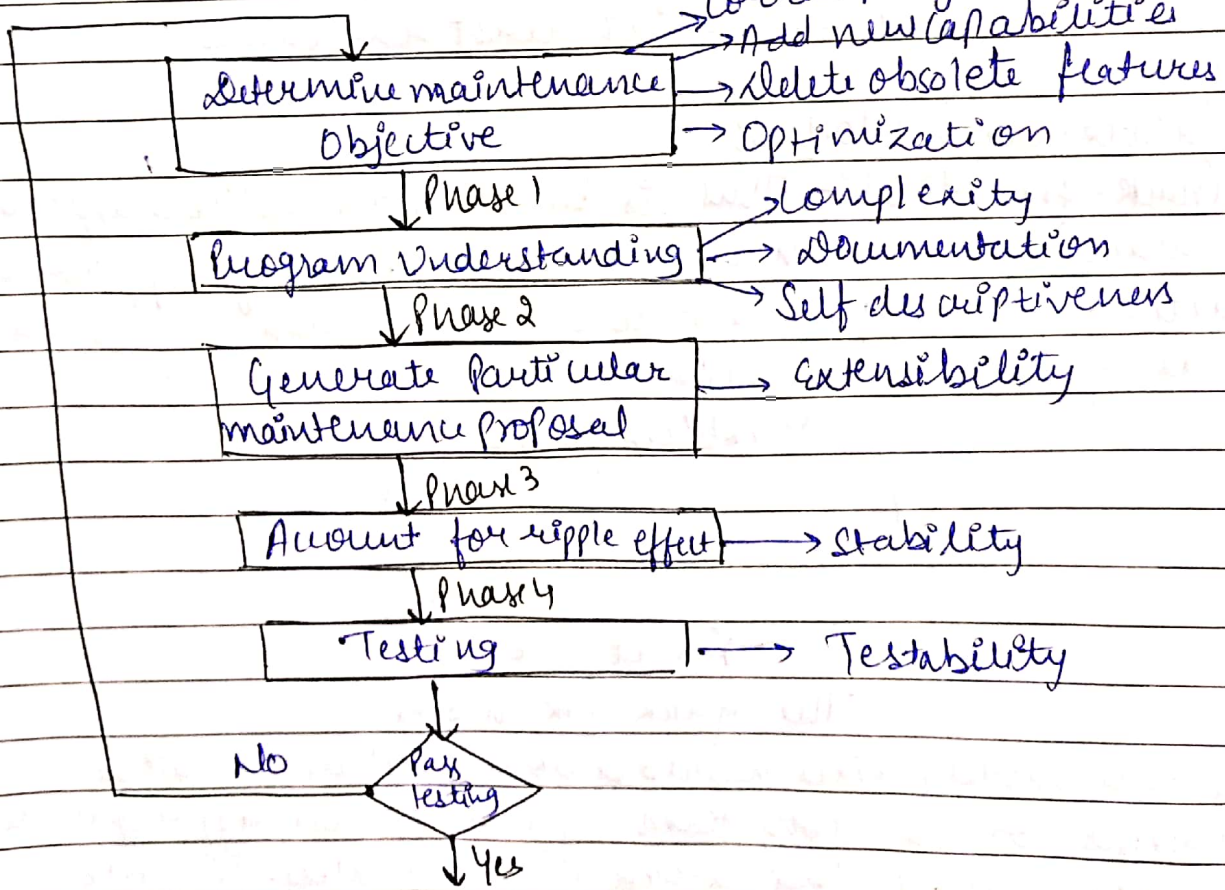
③ Obsolete legacy systems:

→ In most of the countries worldwide, the legacy system that provides the backbone of the nation's critical industries, e.g. telecommunications, medical, transportation were not designed with maintenance in mind.

→ They were not expected to last for a quarter of a century or more.

→ As a consequence, the code supporting these systems is devoid of traceability to the requirements, compliance to design and programming standards and often includes uncommented code, which all make the maintenance task next to the impossible.

Software Maintenance Process:-




Program Understanding:- The first phase consists of

analyzing the program in order to understand it. Several attributes such as the complexity of the program, the documentation and the self-descriptiveness of the program contribute to the ease of understanding the program.

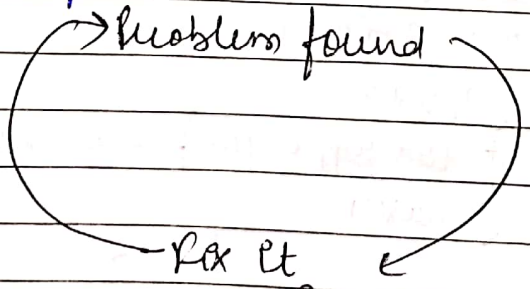
Generating a particular maintenance problem:- The second phase consists of creating a particular maintenance proposal to accomplish the implementation of the maintenance goals.

Ripple effect:- The third step consists of accounting for all of the ripple effects as a consequence of a program modifications.

Modified Program Testing:- The fourth step consists of testing the modified program to ensure that the revised application has at least the same reliability as prior.

# Maintenance Models:

① Quick-fix Model:- This is basically an adhoc approach to maintaining software. It is a fire fighting approach, waiting for the problem to occur and then try to fix it as quickly as possible.




The quick-fix model

In this model, fixes would be done without detailed analysis of the long-term effects. In an appropriate environment it can work perfectly well. If customers are demanding the correction of any error they may

not be willing to wait for the organization to go through detailed and time-consuming stage of risk analysis. The organization may run a higher risk in keeping its customers waiting than it runs in going for the quickest fix.

2) Iterative enhancement Model:-

In this model, it has been proposed that the changes in the software system throughout its lifetime are an iterative process. Originally proposed as a development model but well suited to maintenance, the motivation for this was the environment where requirements were not fully understood and a full system could not be built.

Adapted for maintenance, the model assumes complete documentation as it relies on modification of this as the starting point of each iteration. The model is effectively a 3 stage cycle

1) Analysis
2) Characterization of proposed modifications
3) Redesign and implementation


The existing documentation of each stage is modified starting with the highest-level document effected by the proposed changes. These modifications are propagated through the set of documents and the system redesigned