Types of Agile Software Development:

- ✓ Extreme Programming (XP)
- ✓ ~~Scrum~~ Scrum
- ✓ Adaptive Software Development (ASD)

- **Extreme Programming (XP):**

In Oct 1999, Kent Beck published the book ~~on~~ Extreme Programming.

XP Values:

Beck defines a ~~set~~ of 5 values, which established a ~~set of~~ foundation for all ~~worked~~ works performed as part of XP.

i) Communication

ii) Simplicity

iii) Feedback

iv) ~~Courage~~ Courage

v) Respect

i) Effective Communication:

XP emphasises close, yet informal (verbal) collaboration between customers and developers avoiding voluminous documentation as a communication medium.

ii) Simplicity:

XP restricts developers to design only for immediate needs rather than considering future needs

iii) Feedback:

Feedback is derived from 3 sources —

a) the implemented software itself via unit test results
b) customer
c) other software team members.

iv) Courage/Discipline:

There is often significant pressure to design for future requirement. In order to manage this pressure, an Agile XP team must have the discipline (courage) to design for today.

v) Respect:

By having each of these values, the Agile team inculcates respect among its members, between other stakeholders and team members and the software itself.

- Scrum Software Development:

Scrum is a framework for managing work with emphasis on software dev. It is designed for a team of 3-9 dev. who break their work into actions that can be completed within time boxed iterations called sprints (typically 2 weeks) and track progress and replan 15 min standard meetings called daily scrums.

Roles: There are 3-4 roles in the scrum framework. These are ideally co-located to deliver potentially shippable product implemented in every

scrum Team :—

i) Product Owner: Represents the product's stakeholder and the voice of the customer and is accountable for ensuring that the team delivers value to the business.

ii) Development Team: Responsible for delivering potentially shippable product increments every sprint.

iii) Scrum Master: It facilitates a scrum, and is accountable for removing impediments to the ability of the team to deliver product goals and deliverables. The scrum master is not a traditional team lead or project manager but acts as a buffer between the team and any distracting influences.

Scrum Workload:

Sprint: A sprint (iteration) is the basic unit of development in scrum. A sprint is a time-boxed effort that is restricted to a specific duration. The duration is fixed for each sprint and is normally between 1 week and 1 month, with 2 weeks most common.

Sprint planning: At the beginning of a sprint, the scrum team holds a sprint planning event to: i) Mutually discuss and agree on the scope of work that is intended to be done during that sprint.

ii) Select product backlog items that can be completed in 1 sprint.

iii) Prepare a sprint backlog that includes the work needed to complete the selected product backlog item.

iv) the recommended duration is 4 hrs for a sprint.

v) Once the dev. team has prepared the sprint backlog, they prepare usually a voting to decide which tasks will be delivered with the sprint.

Daily Scrum: Each day with a sprit, a team holds a stand-up.

All members of the dev. team come prepared. The daily scrum starts precisely on time even if some dev. team members are absent.

It should happen at the same time and place everyday. Is limited (time-boxed) with 15 mi. Anyone is welcome, though only dev. team

member should contribute.

Sprint Review and Retrospective: At the end of a sprint the team holds two events — the sprint review and retrospective.

At the end of the sprint review, the team reviews the work that was completed and plans the work that was not completed. Give demo to the stakeholders. The team members and stakeholders collaborate on what to work on next.

Retrospective; The team reflects on the past sprints, identifies and agrees on continuous process improvement actions.

Structure Chart

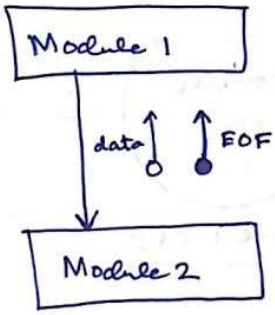
It represents the software architecture, ie. various modules making up the system, the dependency (ie. which module calls which other modules), and the parameters passed among diff. modules.

Basic building blocks:

Rectangular boxes: Represents a module.

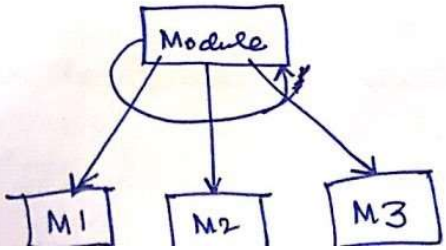
Module invocation arrows: Control is passed from one module to another module in the direction of connecting arrow.

Data flow arrows: Arrows with an empty circle are annotated with data name. Named data passes from one module to another in the direction of arrows.



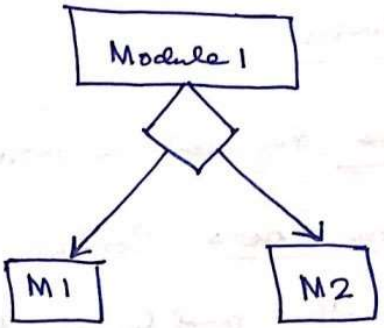
Control flow: A directed arrow with filled circle at the end.

Loop: A curved arrow represents loop in the module.



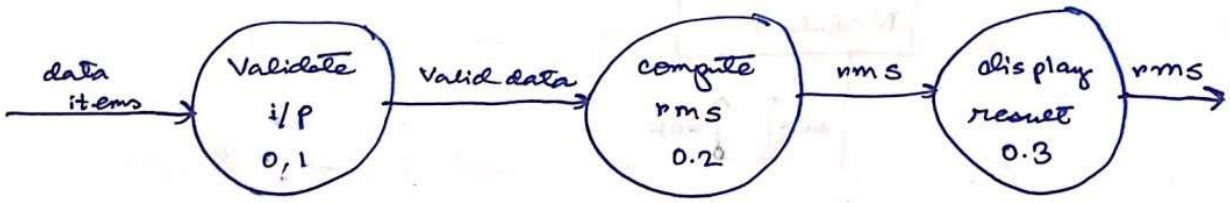
All submodules covered by the loop repeat execution after the module.

Selection of the Condition: Represented by a small diamond at the base of module.

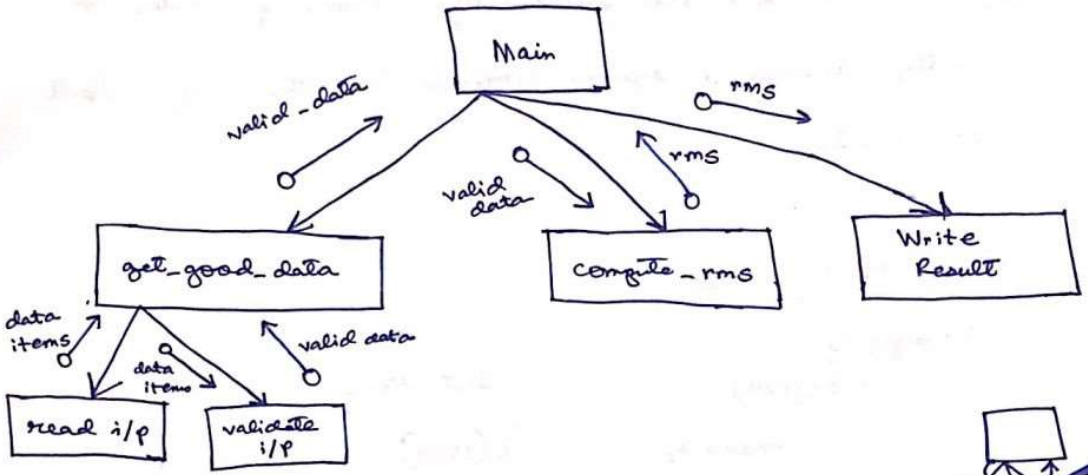


It depicts that control module can select any of the subroutine based on some condition.

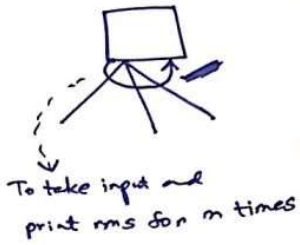
Structure Chart of RMS Software



Level 1 DFD



Structure Chart



Testing

Testing a program consists of subjecting the program to a set of test inputs (or test cases) and observing if the program behaves as expected.

Terminologies:

1) **Test Case:** It is the triplet [I, S, O] where I is the input, S is the state of the system at which the data is input and O is the expected output of the system.

2) **Test Suite:** It is the set of all test cases with which a given software product is to be tested.

3) **Verification:** It is the process of determining whether the output of one phase of software dev. conforms with the previous phase.

4) Validation: It is the process of determining whether a fully developed system conforms to the requirement specification.

Design of Test Cases:

Example (1):

```
if (x>y)
    max = x;
else
    max = y;
```

Test Suite 1:

{(10,9), (7,3), (6,1), (0,-1), (5,0), (5,4), (129,11)}

Test Suite 2:

{(2,1), (1,3)}

Test Suite 2 is better because it checks the if as well as the else condition.
Test Suite 1 never goes into the else part.

1) Exhaustive Testing is non-trivial and ~~for~~ practical. Therefore we must design an optimal ~~set~~ test suite that is of reasonable size.

Consider Example (1).

Designing Test Suite:

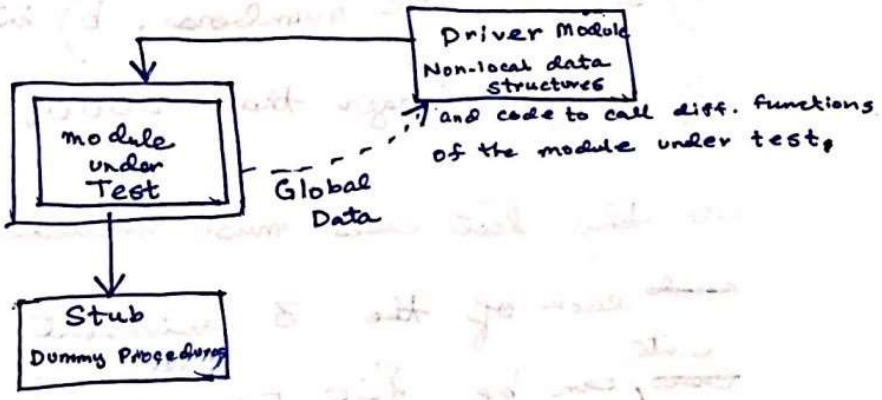
1) Black Box Testing
2) White Box Testing

Three levels of Testing →

1) Unit Testing
2) Module Testing
3) Integration Testing
4) System Testing

1) Unit Testing:

In order to test a single module, we need a complete environment to provide all that is necessary for execution of the module.



Black-Box Testing

- Equivalence class Partitioning
- Boundary Value Analysis

Equivalence class Partitioning:

The domain of input ~~nums~~ to a program ~~is~~ is partitioned into a set of equivalent classes. This partitioning is done such that the behaviour of the program is similar for every input data belonging to the same equivalence class. Eg: A software that computes the square root ~~on~~ of an integer that can assume ~~values~~ values between 0 and 5000. There are 3 equivalent classes, a) set of -ve numbers, b) integers between 0 & and 500 c) integers larger than 5000.

So the test cases must include representative values from ~~each~~ each of the 3 equivalent classes. A possible test ~~case~~ suite can be $\{-5, 500, 6000\}$.

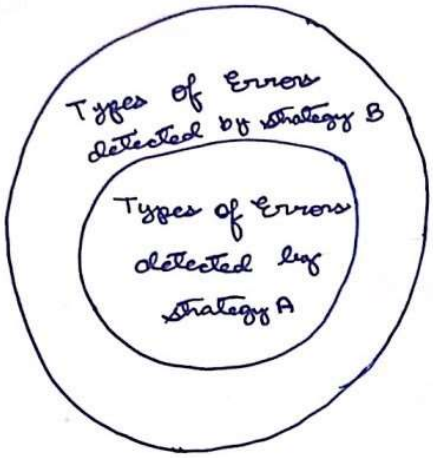
Boundary Value Analysis:

Use of $<$ instead of $\leq$ or vice versa.

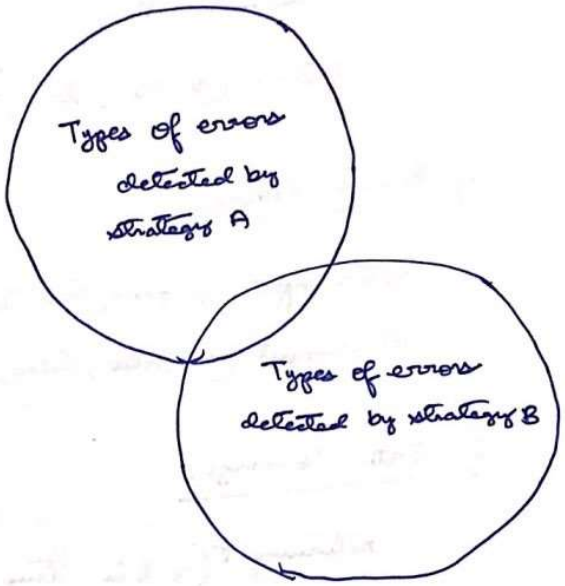
Boundary value analysis leads to selection of test cases at the boundaries of different equivalent classes. For the above mentioned software test cases must include ~~the~~ the following cases —

$\{0, -1, 5000, 5001\}$.

White-Box Testing



Strategy B is stronger than strategy A



Strategy A is complementary to strategy B

Coverage Issues:

Types of test coverages:

1) Statement Coverage
2) Branch Coverage
3) Path Coverage

1) Statement Coverage:

```
int returninput (int input, boolean condition1, boolean condition2,
                                                boolean condition3)
{   int x = input;
    if (condition 1) x++;
    if (condition 2) x--;
    if (condition 3) x = x + 10;
    return x;
}
```

1) Statement Coverage:

return (x, true, true, true)

2) Branch Coverage:

returninput (x, true, true, true)

returninput (x, false, false, false)

3) Path Coverage:

returninput (x, true, true, true)

returninput (x, true, true, false)

⋮ } all 8 combinations of true/false, true/false, true/false

Definitions:

Statement Coverage:

The statement coverage strategy aims to design test cases so that every statement in a program is executed at least. By choosing the test set in the ~~eg~~ example we can exercise the program such that all statements are executed once. ~~Branch coverage~~

Branch Coverage:

Eg: Shown before

Path Coverage:

returnInput (x, true, true, true)

⋮ → 8 combinations (See examples)

Control Flow Graph Drawing

Step 1: Number all statements of a program.

Step 2: Different numbered statements will serve as the nodes of the CFG.

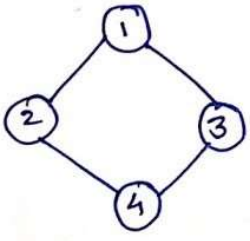
Step 3: An edge from one node to another node exists if the execution of the statement representing the first node can result in transfer of control to the other node.

Sequence:

```
1. a = 5;
2. b = a * 2 - 1;
```

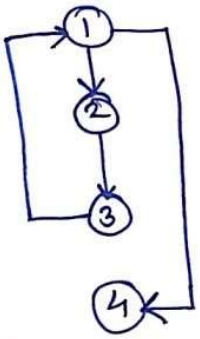
Selection

```
1. if (a > b)
2.     c = 3;
3. else c = 5;
4. c = c * c;
```

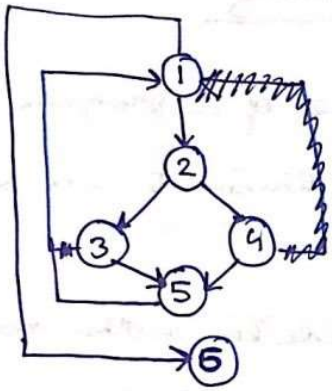


Iteration

```
1. while (a > b) {
2.     b = b - 1;
3.     b = b * a; }
4. c = a + b
```



```
1. while (x!=y){
2.         if (x>y)
3.             x=x-y;
4.         else y=y-x;
5. }
6. return x;
```



Fig(i)

Linearly Independent Path: It is any path through the program that introduces at least one new edge that is not included in any other linearly independent paths. Thus a path that is a subpath of another path is not a linearly independent path.

Mc Cabe's Cyclomatic Complexity

It defines an upper bound on the no. of independent paths in a program. Given a CFG, G of a program, the cyclomatic complexity $V(G)$ can be computed as $V(G) = E - N + 2$ where E → no. of edges, N → no. of nodes

⇓

for the previous graph $V(G) = 3$.

(fig i)

Method 2:

$V(G)$ = Total no. of bounded areas + 1

Derivations of Test Cases:

i) Draw the CFG

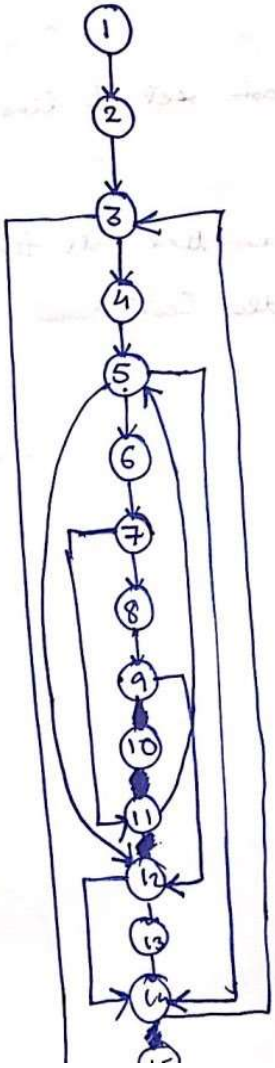
ii) Determine $V(G)$.

iii) Determine the basic set of linearly independent paths.

iv) Prepare the test cases that will force execution of the basic path in the test cases.

Code for series of prime numbers upto n:

```
1. findprime(n)
2. {
3.     for(i=1; i<=n; i++)
4.     {
5.         flag=0;
6.         for(j=2; j<i; j++)
7.         {
8.             if(i%j==0)
9.             { flag=1;
10.              break;
11.            }
12.        }
13.        if(flag!=1)
14.            printf("%d", i);
15.    return; }
```



$V(G) = E - N + 2$

$= 18 - 15 + 2$

$= 5$

{1, 2, 3, 15}

{1, 2, 3, 4, 5

Integration Testing

It is the phase of software testing in which individual modules are combined and tested as a group. During integration testing diff. modules of a system are tested in a planned manner using an integration plan.

Types —

1) Big Bang Approach: Simplest integration testing approach where all the modules making up a system are integrated in a single step. All the modules of the system are simply put together and tested. This technique is applicable only for small systems. The main prob. with this approach is that it is difficult to identify the error as it may belong to any of the modules being integrated.

2) Bottom-up Approach: Here each subsystem is tested seperately, then the full system is tested. A sub system consists of many modules which communicate among each other through well defined interfaces. The primary purpose of each sub system is to test the various modules making up the subsystem.

3) Top-Down Integration Testing: It starts with the main routine and one or two subordinate routines in the system. After the top level skeleton has been tested, the immediate subroutines

of the skeleton are combined with it and tested. TDIBT requires the use of program stubs to simulate the effect of lower level routines that are called by the routines under test. However a pure top down integration does not require any driver routine.

4) Mixed Integration Testing: It follows a combination of top down and bottom up testing approaches. In top down approach, testing can start only after the top level modules have been coded and unit tested. Similarly bottom up testing can be started only if the modules are ready.
Similarly in MIT, testing can start as and when modules become available. Therefore this is one of the most commonly used integration testing approaches.

System Testing

They are designed to validate a fully developed system to assure that it meets user requirements.

1) α-Testing: Carried out by the test team within the developing organization.

2) β-Testing: Performed by a selected group of friendly customers.

3) Acceptance Testing: It is the system testing performed by the customer to determine whether they should accept the delivery of the system.

Software Project Management

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

Software Project Management Complexities:

1) Invisibility: Difficult to assess the progress.
2) Changeability: Changes in requirements.
3) Uniqueness: Every software is associated with unique features.
4) Enactness of the solution.
5) Complexity: Data coupling, control dependency, file sharing etc.

Responsibilities of a Software Project Manager:

1) Project Planning: This is done after feasibility study and before requirement analysis phase. It involves estimating several characteristics of a project and planning the project.

- Estimating: 1) Project size 2) cost, 3) duration, 4) effort
- Scheduling manpower and other resources
- Staffing
- Risk identification, analysis and abatement of plan

~~Software Configuration Management~~

Once project planning is complete, software project management plan is written.

2) Project ~~Monit~~ Monitering and Control: To ensure ~~that~~ software development proceeds as per plan.

Metric for Software Project Size Estimation

1. Lines of Code (LOC):

i) Simplest among all metrics available to estimate project size.

ii) Project size is estimated by counting the no. of source instructions excluding the ~~comments~~ comments and header lines.

iii) However determining LOC count at the beginning of the project is very difficult.

iv) Project managers usually divide the problem into modules and each modules into sub modules and so on until the sizes of different leaf level modules can be approximately predicted.

v) To be able to do this, past experience in developing similar product is helpful.

Shortcomings of LOC Metric:

i) Focusses on coding activity alone.

ii) LOC can vary with coding style.

1. `for (i=0; i<=100; i++) printf("Hello");`

PLOC = 1 (Physical LOC) LLOC = 2 (Logical LOC)

2.
```
for(i=0; i<=100; i++)
{
    printf("Hello");
}
```

PLOC = 4 LLOC = 2

3.
```
for (i=0; i<=100; i++)
    printf("Hello");
```

PLOC = 2 LLOC = 2

iii) Penalizes higher level programming languages, code reuse etc

iv) Measures textual or lexical complexities only, does not address the issues of structural or logical complexity.

v) ~~Lacks~~ Correlates poorly with the quality and efficiency of the code.

vi) Very difficult to ~~estimate~~ accurately estimate lines of code of the final program from problem specification

2. Function Point:

Since its inception function point metric is gaining popularity. In contrast to LOC Metric where the size can be estimated only after the product is fully developed FP Metric can be used to estimate the size directly from product specification.

The size of a product in FP is estimated as the weighted sum of 5 diff. characteristics —

i) no. of inputs
ii) no. of outputs
iii) no. of inquiries
iv) no. of files
v) no. of interfaces

FP Metric Computation:

Step 1: Compute the UFP (unadjusted FP) using a heuristic expression.

(write ω_1 instead)

$$UFP = (\text{no. of i/ps}) * (4)^{\omega_1} + (\text{No. of o/ps}) * (5)^{\omega_2} + (\text{No. of inquiries}) * (4)^{\omega_3} + (\text{No. of files}) * (10)^{\omega_4} + (\text{No. of interfaces}) * (10)^{\omega_5}$$

• See Bresman

$w_1 \in \{3, 4, 5\}$
$w_2 \in \{4, 5, 6\}$
$w_3 \in \{3, 4, 5\}$
$w_4 \in \{9, 10, 11\}$
$w_5 \in \{7, 10, 15\}$

~~Step 1~~

no. of inputs: Data item input by the user different from ~~inquiries~~ inquiries.

no. of outputs: Reports printed, ~~screen~~ screen outputs, error msgs.

inquiries: It is a user command (without any data input and only requires some actions to be performed by the system) eg. print account balance

no. of files: Physical files and data structures.

no. of interfaces: Different mechanisms to exchange information with ~~other~~ other system.

The weights associated with these characteristics are determined empirically through data gathered from any project.

Step 2: Refine Parameters.

Refine UFP using technical complexity factor (TCF) ~~using~~ to reflect the actual complexity of diff. parameters used in UFP calculation.

Step 3: Compute FP by further refining UFP.

Constructive Cost Estimation Model

(COCOMO) Barry W. Boehm [1981]

Three types of software project proposed by Barry W. Boehm.—

a_1, a_2, b_1, b_2 are constants for each category of software.

KLOC mean Kilo lines of code.

T_{dev} is the estimated time to develop expressed in months

Effort → total effort required to develop the software in person month.

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

$$T_{dev} = b_1 \times (Effort)^{b_2} \ Months$$

3 Types of Project:

1) **Organic:** A development project can be considered as organic type if the project deals in developing a well-understood application program, the size of the dev. team is reasonably small and the team members are experienced in developing similar kinds of project.

2) **Semi-detached:** Mixture of experienced and inexperienced staff.

3) **Embedded:** If the software is completely coupled with hardware or if stringent regulations on operational procedures exist.

Estimation of Development Effort

Organic: $\text{Effort} = 2.4 \times (\text{KLOC})^{1.05}$ PM

Semidetached: $3.0 \times (\text{KLOC})^{1.12}$ PM

Embedded: $3.6 \times (\text{KLOC})^{1.20}$ PM

Estimation of Development Time

Organic: $T_{dev} = 2.5 (\text{Effort})^{0.38}$ months

Semidetached: $T_{dev} = 2.5 (\text{Effort})^{0.35}$ months

Embedded: $T_{dev} = 2.5 (\text{Effort})^{0.32}$ months

Example Problem:

An organic type software has 32,000 lines of source code. If the average salary of a software engg. is 15,000 per month..... Estimate cost, effort.

$\therefore$ Effort $= 2.4 \times (32)^{1.05}$

$= 91.33$ PM (Ans)

$\therefore T_{dev} = 2.5 (91.33)^{0.38}$ months

$= 13.89$ ~~month (Ans)~~

≈ 14 months

$\therefore$ Cost $= 14 \times 15,000$

$= 210,000/-$

Problems with basic COCOMO Model:

1) The algorithmic complexity are not considered.
2) The characteristics of the developers are not clearly defined.
3) Problem similar to LOC metric are there.

Intermediate COCOMO

It computes software development effort as function of program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel and project attributes. This extension considers a set of 4 cost drivers each with a no. of subsidiary attributes.

A) Product Attributes

1) Required software reliability
2) Size of application database
3) Complexity of the product

B) Hardware Attributes

1) Runtime performance constraints
2) Memory constraints
3) Volatility of the virtual machine environment
4) Required Turnaround Time

C) Personnel Attributes

1) Analyst Capability

2) Software Engineering Capability

3) Application's experience

4) Virtual Machine Experience

5) Programming Language Experience

D) Project Attributes

1) Software Tools

2) Application of software engg. methods

3) Required Development schedule

Each of the 15 attributes receives a rating on a 6 point scale that ranges from "very low" to "extra high" in importance or value.

An effort multiplier from the table below applies to the rating. The product of all effort multipliers results in an effort adjustment factor (EAF).

Typical values for EAF range from 0.9 to 1.4.

Cost drivers	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
Required s/w Reliability	0.75	0.88	1.00	1.15	1.40	
Size of app database		0.94	1.00	1.08	1.16	
Complexity of the product	0.70	0.85	1.00	1.15	1.30	1.65
Hardware Attributes						
Runtime Perf. Constraints			1.00			
⋮			1.00			
⋮						

Formula:

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

- $(KLOC)^{b_i}$ → Basic COCOMO
- (EAF) → comes from the table
- a_i, b_i depends on the project type
- $\prod_{k=1}^{15}$ of the attributes found → Multiplication of the values from the table

Complete or Detailed COCOMO

Detailed COCOMO ~~uses~~ incorporates all the characteristics of the intermediate version with an assessment of the cost drivers in each step of the software engg. process.

The detailed ~~process uses~~ model uses different effort multiplier for each cost driver attribute.
These phase sensitive effort multipliers are used to

determine the effort required to complete each phase. In detailed cocomo the whole software is divided into modules and then we estimate cocomo on the different modules then, to sum the efforts.

Maintainence

Software maintainence is the modification of a software product after delivery to correct faults, to improve performance or other attributes. A common perception of maintainence is that it merely involves fixing defects. However one study indicated that over 80% of the maintainence effort is used for non-corrective action. This perception is perpetuated by users submitting problem reports that in reality are functionality enhancements to the system.

Necessity of Software Maintainence

3) When the hardware platform is changed and the software perform some ~~some~~ low level fuction, maintainence is necessary.

4) When support environment is changed, the software product requires rework, ~~for instance~~

5) For instance a Soft. prod. may need to be maintained when the OS changes.

6) Thus every soft. prod. continues to evolve after its development through maintainence effort.

Software Maintainence Tasks:

3 Types —

1) Corrective
2) Adaptive
3) Perfective

1) Corrective Maint. of a soft. prod. is necessary to rectify the bugs observed while the system is in use.

2) Adaptive: A soft. prod. might need maint. when the customer needs the soft. to run on new platforms, new OS or they need the # product to interface with new hardware or software.

3) Perfective: A soft. prod. needs maint. To support new features that the user want it to

support, to change the functionalities of the system according to customer demand & to enhance the system for performance.

Problems with software maintainence

1) Soft. Maint. work is typically much more expensive than what it should be and takes more time than required. 2) In softw. org. maint. work is mostly carried out using ad-hoc techniques.

3) There are no systematic and planned activities for maintaining software.

4) Soft. Maint. has a very poor image in industry. Therefore an org. often cannot employ bright engineers to carry out maint. work.

5) Even though maint. suffers from a poor image, the work involved in maintainence is often more challenging than development work.

6) During maint. it is necessary to thoroughly understand someone else's work and then carry out the required modifications and extensions.

7) Another prob is that the majority of soft. prod. needing maintainance are legacy prod.

Legacy Software Products

A legacy system is hard to maintain. The typical problems associated are— Poor documentation, unstructured code and lack of personal knowledgable in the product. Many of the systems were developed long time back, but recently developed system having poor design and documentation are considered as legacy system.

Factors on which Software System Maintainance Activities depend—

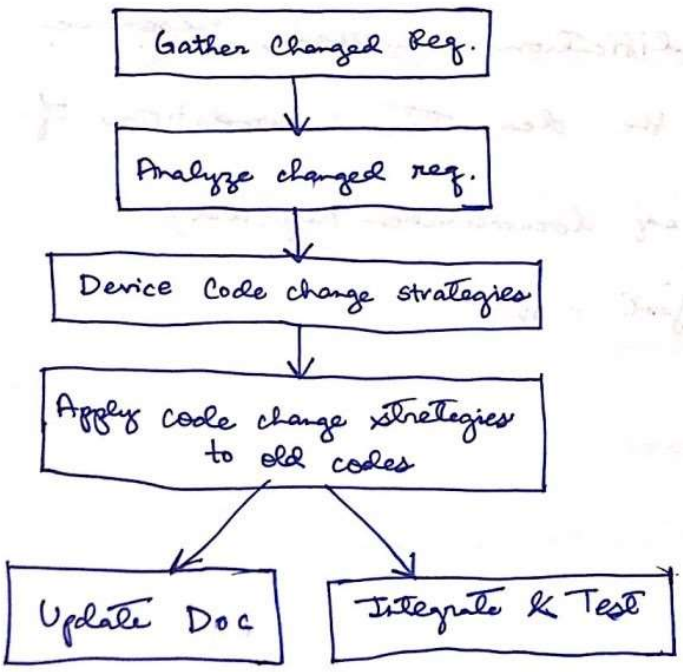
The activities involved in a soft. system. maint. project depends on several factors such as extent of modification required, resource available to the dev. team, conditions of existing prod, eg. documentation required, * expected project risks.

Software Reverse Engineering

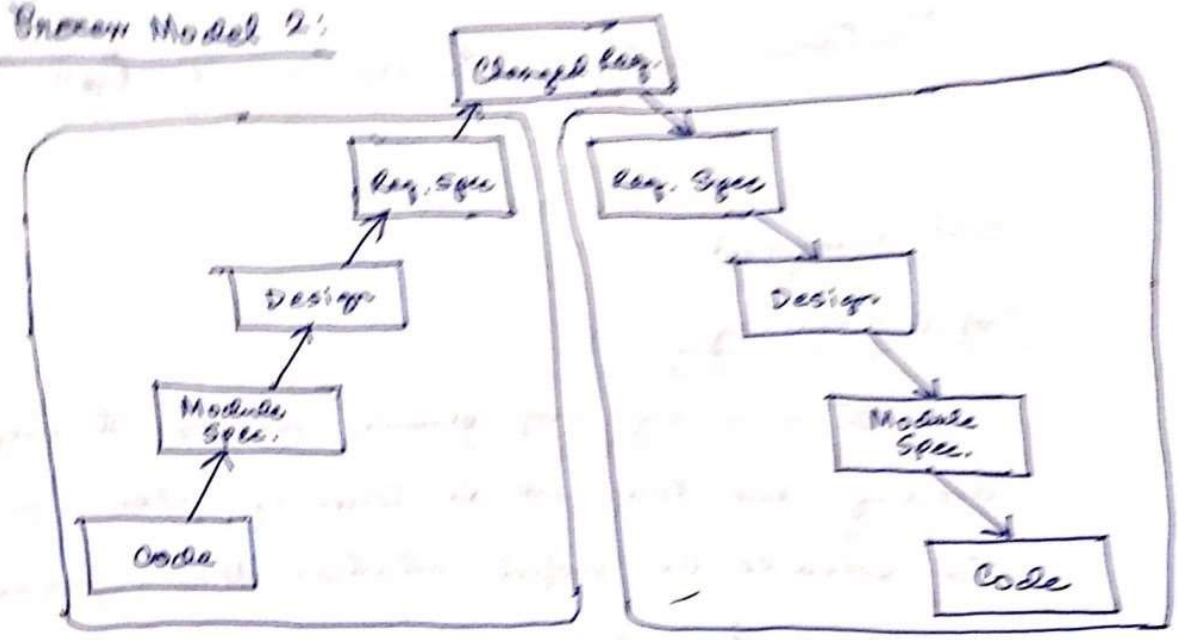
It is the process of recovering the design and requirement specification of a product from an analysis of its code. The purpose of reverse engg. is to facilitate maintainance work by improving the understandability of a system and to produce the necessary doc. for a legacy system. Rev. Engg. is becoming imp. since legacy soft. prod. lack proper documentation and are highly unstructured.

Software Maintainence Process Model

Process Model No 1:



Waterfall Model 2:



Estimation of Approximate ~~Estimate~~ Maintainence Cost

It is well known that Maintainence cost takes up about 60% of Total life cycle cost.

For embedded system, the maint. cost can be 2-4 times the dev. cost. Boehm (1991) proposed a formula for estimating the cost for his COCOMO cost estimation model. Boehm's maint. estimation cost is made in terms of a quantity called Annual Change Traffic (ACT). When defined ACT ~~at~~ as fraction of a soft. prod. source instruction which undergo a change during a Typical year either through addition or deletion.

$$ACT = \frac{(KLOC\ added + KLOC\ deleted)}{KLOC\ Total}$$

Maintainence cost = ACT × Development Cost

Project Management

Project Scheduling

It is an imp. proj planning activity. It helps in deciding which task will be taken up when. In order to schedule the project activities the proj. manager need to do the following—

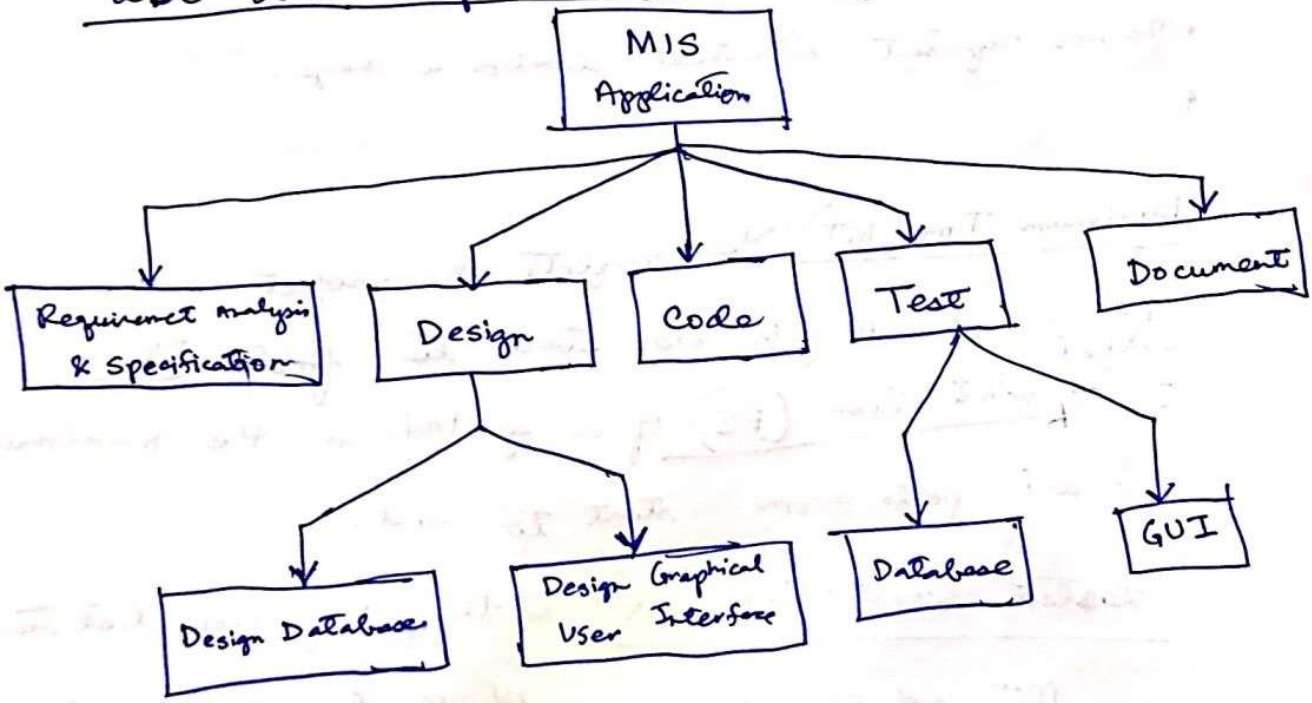
1) Identify all the tasks needed to complete the project
2) Break down large tasks into small one.
3) Determine dependency among small activities.
4) Establish the most likely estimates for the time durations necessary to complete the activities.
5) ~~Allo~~ Allocate resources to activities.
6) Plan the start and end dates for various activities
7) Determine the critical path. It is the chain of activities that determine the duration of the project.

① and ②

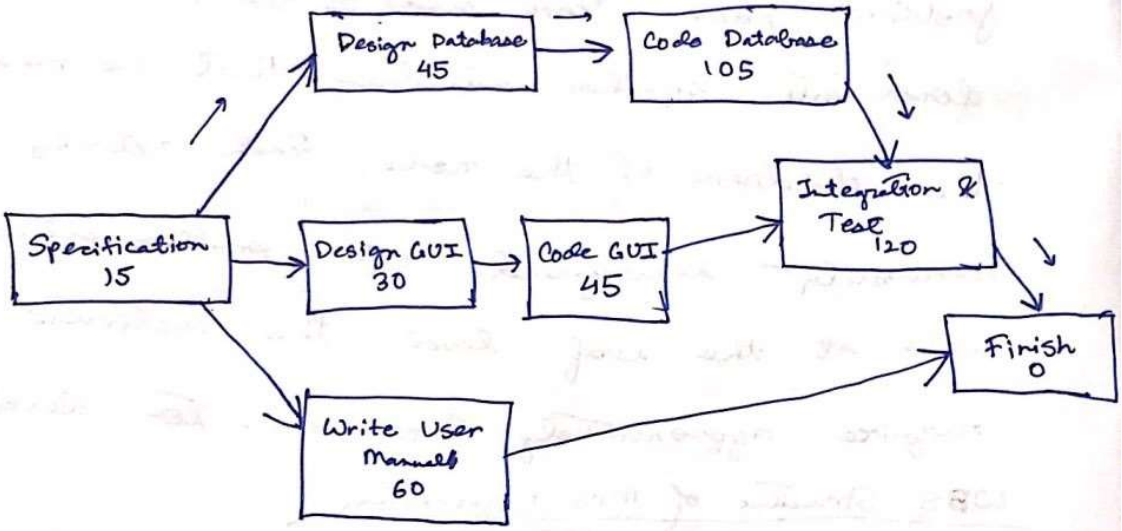
Work Breakdown Structure (WBS)

WBS is used to decompose a given Task set into small activities. WBS provides a notation for representing the major tasks need to be carried out in order to solve a problem. The root of the tree is labelled by the problem name. Each node of the tree is broken down into smaller activities that are made of the children of the node. Each activity is recursively decomposed into smaller subactivities until at the leaf level, the activities require approximately two weeks to develop.

WBS Structure of MIS Application



WDS representation of a process is transformed into an activity network by representing activities identified in WDS along with their dependencies. An activity network shows the different activities making up a project, their estimated durations and interdependencies.



• The nos represent estimated duration in days.

4

Minimum Time (MT): To complete the project is the maximum of all paths from start to finish. The earliest start time (ES) of a task is the maximum of all paths from start to end.

Latest Start Time (LS) is the difference between MT and the maximum of all paths from this task to the finish.

Earliest Finish Time (EF) of a task is the sum of the earliest start time of the task and the duration of the task.

Latest Finish Time (LF) of a task can be obtained by subtracting minimum of all paths from this task to finish from MT.

The slack Time (ST) is ~~LS - EF~~ LS - ES and equivalently can be written as LF - EF. The slack time (or float time) is the total time that a task may be delayed before it will affect the end time of the project. The slack time indicates the flexibility in starting and completion of tasks. A critical task is one with zero slack time. A path from the start node to the finish node containing only critical tasks is called a critical path.

Task	ES	EF	LS	LF	ST	
Specification	0	15	0	15	0	*
Design Database	15	60	15	60	0	*
Design GUI	15	45	90	120	75	
Code Database	60	165	60	165	0	*
Code GUI	45	90	120	165	75	
Integrate & Test	165	285	165	285	0	*
Write User Manual	15	75	225	285	210	

* Critical Tasks

MT = Maximum of all paths from start to finish

= 15 + 45 + 105 + 120 = 285

LF (Design GUI) = MT - Maximum of all paths from this task to the finish

= 285 − (45 + 120) = 120

LS (Design GUI) = LF - Duration of the current task

= 120 − 30

= 90

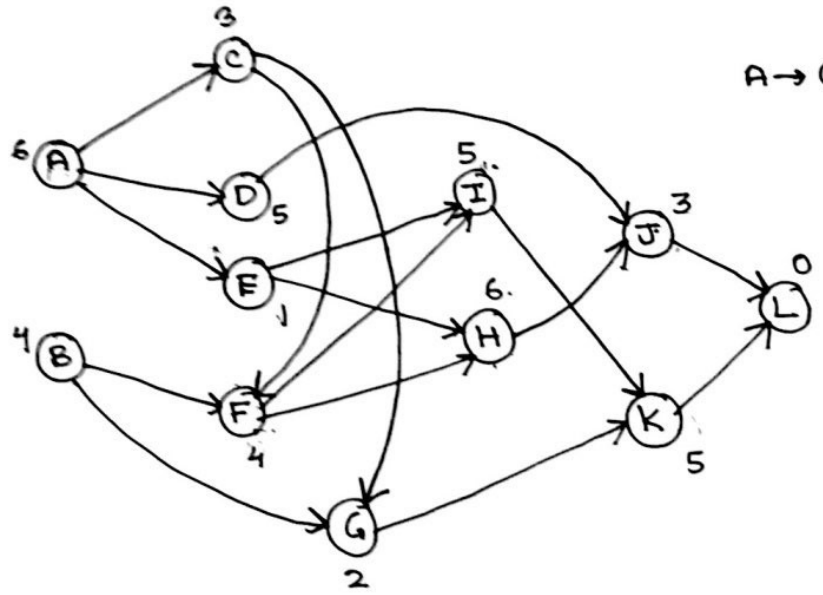
285 − 270 = 15

285 − 225 = 60

285 − 120 = 165

Activity	Immediate Predecessor	Time (days)
A	—	6
B	—	4
C	A	3
D	A	5
E	A	1
F	B, C	4
G	B, C	2
H	E, F	6
I	E, F	5
J	D, H	3
K	G, I	5
L	J, K	0

Take



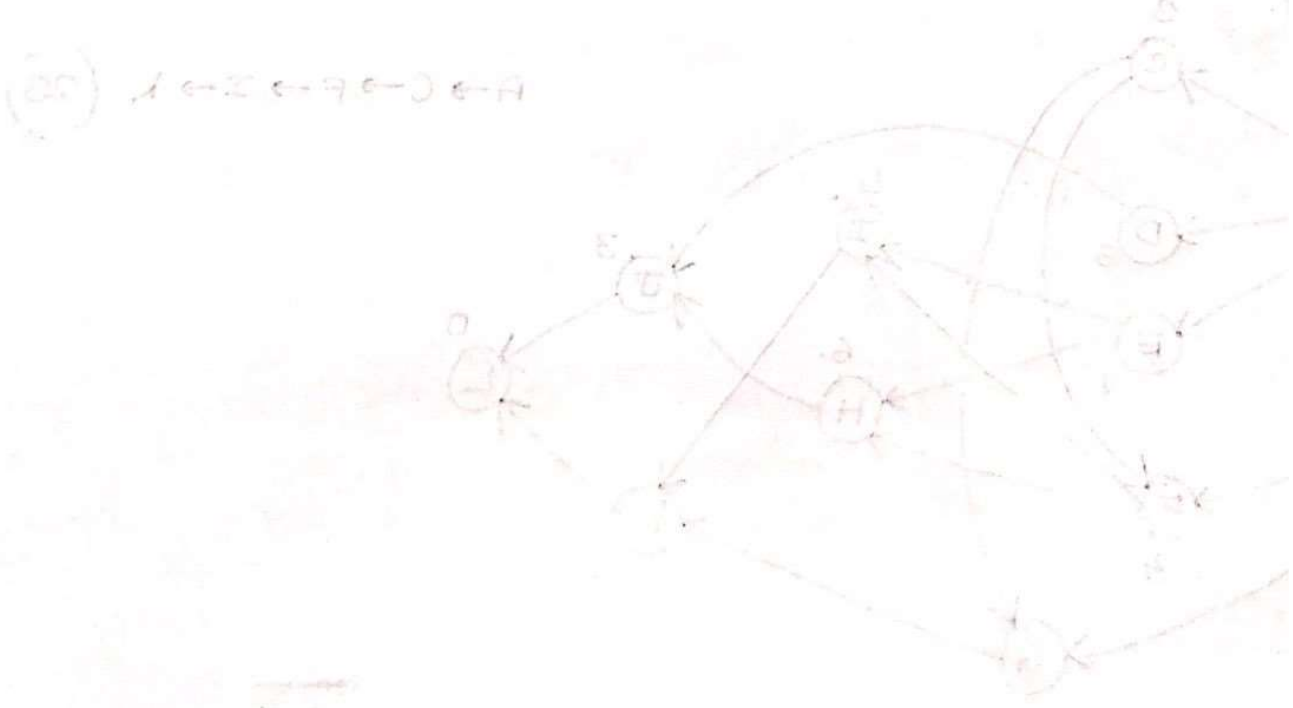
A → C → F → I → K (23)

16/17

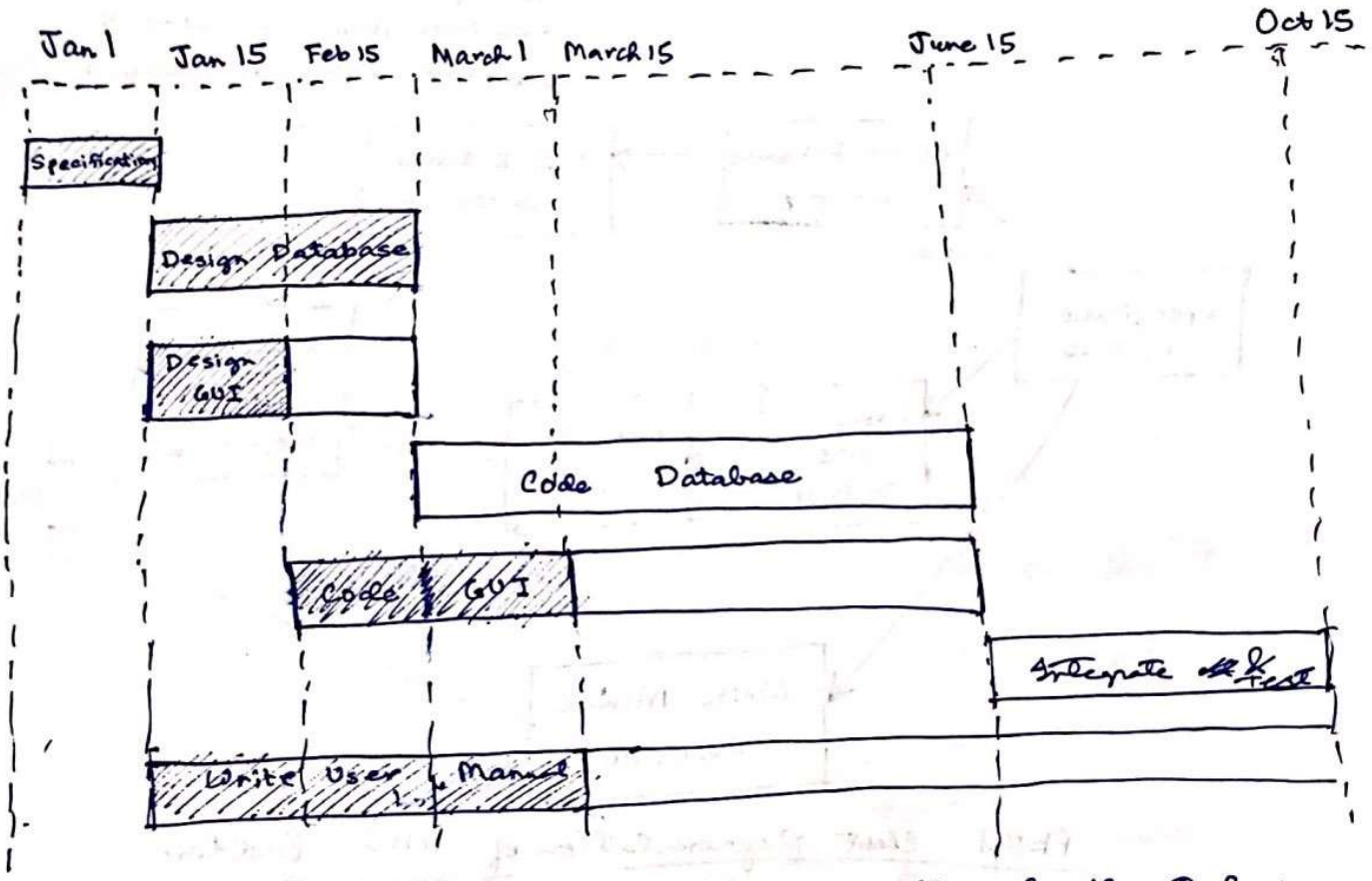
MT → 23

23 - ∞

	ES	EF	LS	LF	ST
A	0	6			
B	0	4			
C	6	9			
D	6	11			
E	6	7			
F	9	13			
G	9	11			
H	13	19		20	
I	13	18			
J	19	22			
K	~~18~~	~~23~~			



Gantt Chart



Shaded part of the bar represents the time this task is estimated to take.

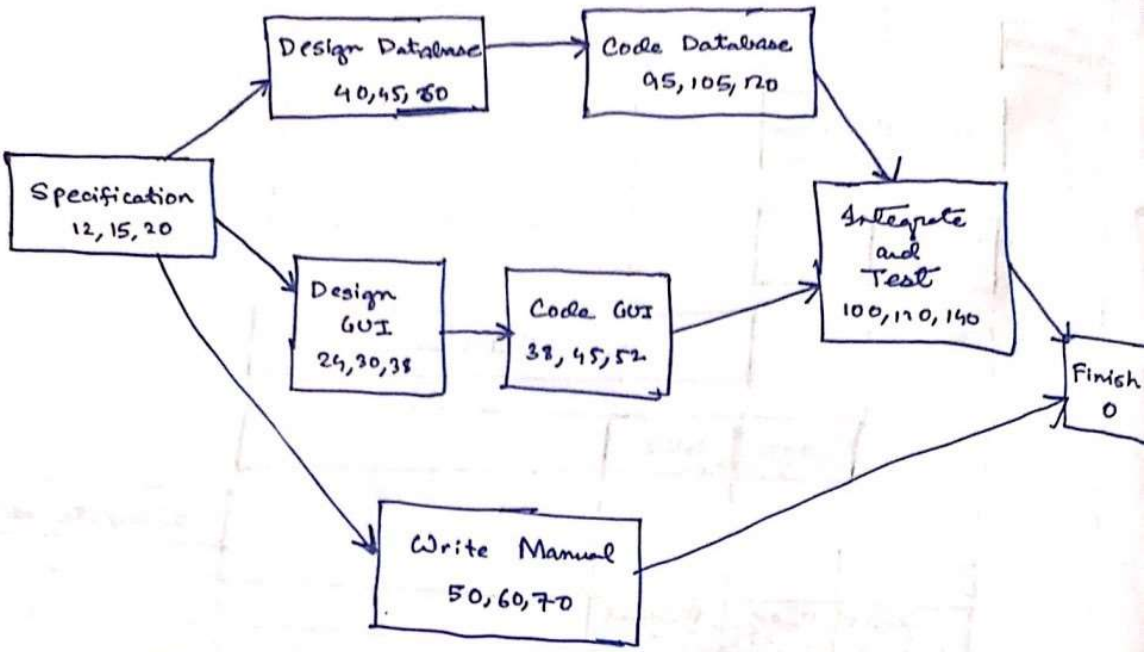
White part represents slack time or float time of this part.

Use:

1) Visually aid better
2) Resource Scheduling

PERT Charts (Program Evaluation and Review Technique)

1st → Optimistic → weightage: 1
2nd → Most likely → weightage: 4
3rd → Pessimistic → weightage: 1



PERT Chart Representation of MIS Problem

PERT Chart represent the statistical variations in the project estimates assuming a normal distribution. Thus in a PERT Chart instead of making a single estimate for each task, pessimistic, likely and optimistic estimates are made. The boxes of PERT Charts are usually annotated with the optimistic, likely and pessimistic estimates for every task.

~~PERT~~

Optimistic Time: The minimum possible time required to accomplish an activity or a PERT assuming everything proceeds better than normally expected.

Pessimistic Time: The maximum possible time to accomplish an activity or a path assuming everything goes wrong.

Most Likely Time: The time required to accomplish an activity or a path assuming everything proceeds as normal.

Expected Time: The best estimate of the time required to accomplish an activity or a path accounting for the fact that things don't always proceed as normal. Expected Time required to accomplish an activity is calculated by the heuristic $t_e = (O + 4m + P)/6$,

where O represents optimistic time,
m represent most likely time,
P represent pessimistic time.

Standard Deviation of Time: The variability of the time for accomplishing an activity represented by $\sigma_{te} = (P - O)/6$.

Gantt chart representation of a project schedule is helpful in planning the utilization of resources, while PERT chart is useful for monitoring ~~Program Evaluation and Review Techni~~ the timely progress of Activities.

Also it is easier to identify parallel activities in a project using a PERT Chart.

Software Configuration Management (SCM)

Raju Mall Presman

SCM also called change management is a set of activities designed to manage change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing diff. versions of these work product, controlling the changes imposed and auditing and reporting on the changes made.

SCM activities are developed to —

1) Identify change,
2) Control Change
3) Ensure that change is being properly

4) Report changes to others who may have an interest. It is important to make a clear distinction between software support and software configuration management.

Support is a set of soft. engg. activities that occurs after soft. has been delivered to the customer and put into operation.

Software Config. Management is a set of tracking and control activities that are initiated when a soft. engg. project begins and terminates only when the software is taken out of operation.

Origin of Changes

1) New business or market conditions dictate changes in product requirements or business rules.

2) New Stakeholders need demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer based system.

3) Re-organization or business growth/ downsizing causes changes in project priorities or soft. engg. team structure.

4) Budgetary or scheduling constraints cause a redefinition of the system or product.

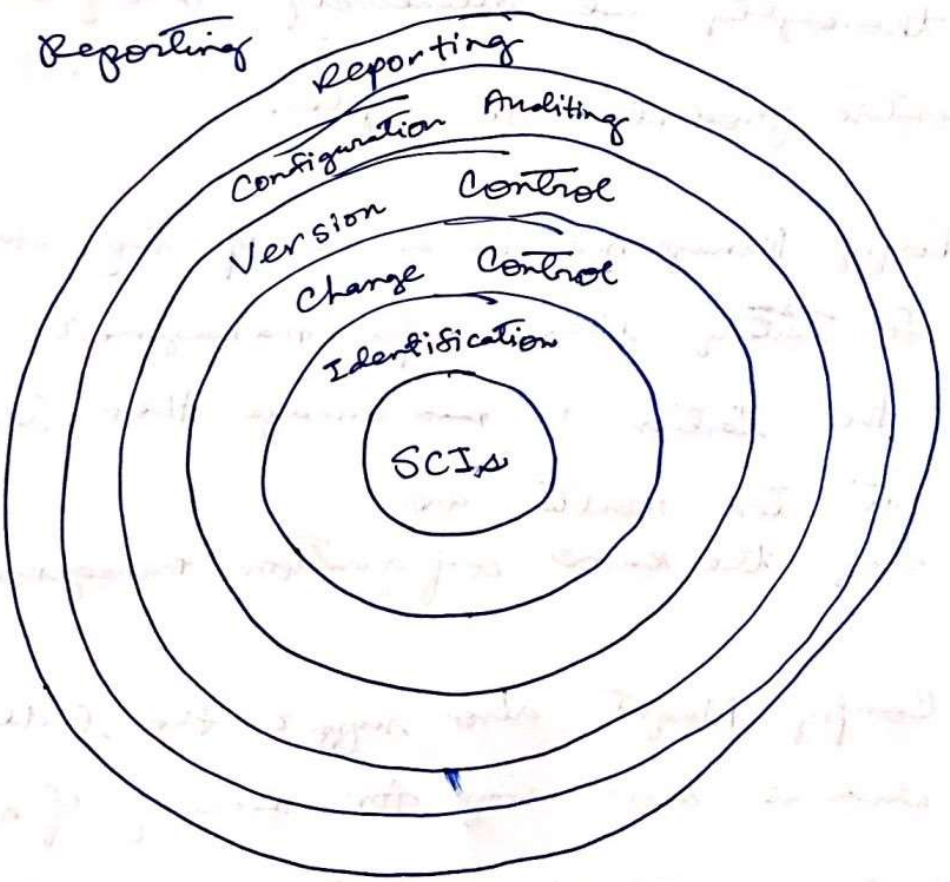
Software Config. items (SCI)

An SCI (Soft. Config. item) is all or part of a work product (eg. a document, an entire suite of test cases, or a main program component). The results (also called as the deliverables), ~~the results~~ of a large development effort typically consists of a large no. of objects, eg. source code, design doc, SRS doc, test doc, user's manual etc. These objects are usually referred to and modified by a ~~no~~ no. of soft. engg. throughout the life cycle of the software. The state of all these objects at any point of time is called the configuration

of the soft. product. The state of each deliverable object changes as development progresses and also as bugs are detected and fixed.

SCM Tasks

1) Identification
2) Version Control
3) Change Control
4) Configuration Auditing
5) Reporting



SCM Features

1) Configuration Management determines clearly about items that make up the software or system. These items include source code, test scripts, third party software, hardware, data and both development and test documentation.

2) Config. Management is also about making sure that these items are managed carefully, thoroughly and attentively throughout the entire product life cycle.

3) Config. Management has a no. of imp. implications for testing like config. management allows the testers to manage their testwares at test results are using the same configuration management.

4) Config. Mngnt also supports the build process which is also imp for delivery of a test release into the test environment.

5) Simply sending zip archives by email will not be sufficient because there

archives to become polluted with undesirable contents.

Config Mngmt is a topic that is very complex, so advance planning is very imp. To make this work. During the project planning stage ad perhaps as part of your own test plan make sure that config. mngmt ~~tools~~ procedures and tools are selected. As the project proceeds the config. procedures and mechanisms must be implemented at the key interfaces to the rest of the development process should be documented.