Software Engineering
Unit 2

Q1. Software Project Planning

After the finalization of SRS, we would like to estimate size, cost and development time of the project.

Software Project Planning is a crucial phase in the software development lifecycle that involves defining the scope, objectives, resources, schedule, and overall strategy for a software project.

In order to conduct a successful software project, we must understand:
- Scope of work to be done
- Software Project Planning
- The risk to be incurred
- The resources required
- The task to be accomplished
- The cost to be expended
- The schedule to be followed

Software planning begins before technical work starts, continues as the software evolves from concept to reality, and culminates only when the software is retired.

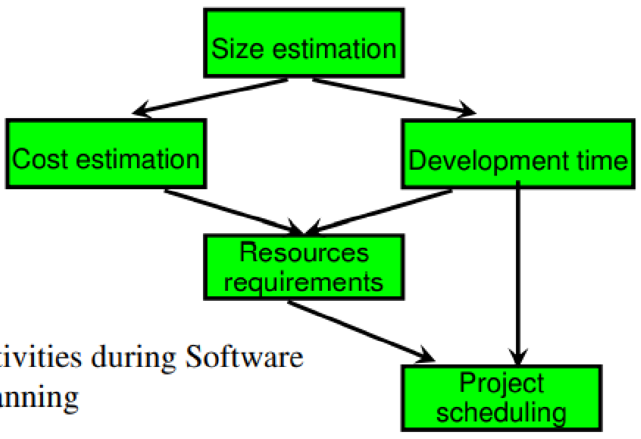


Fig. 1: Activities during Software Project Planning

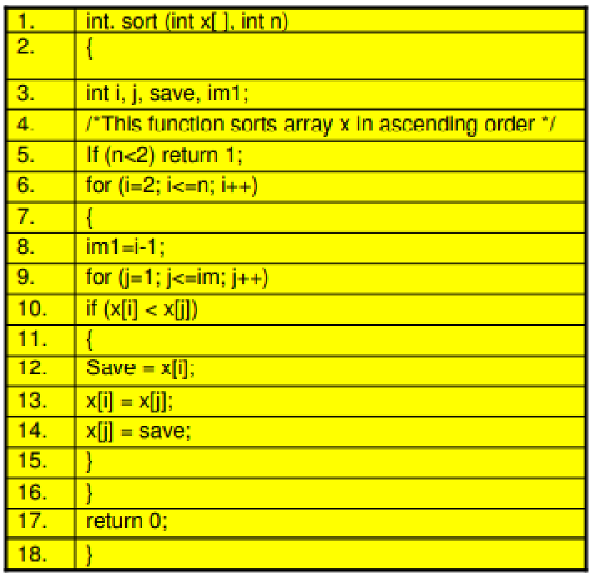
Q2. Size Estimate.

Size estimation in software engineering refers to the process of predicting the size of a software project, typically measured in terms of lines of code (LOC), function points (FP), or story points.

Q3. Lines of code.

if LOC is simply a count of the number of lines then figure shown below contains 17 LOC. Rn Comment

"A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line.



1.	int. sort (int x[ ], int n)
2.	{
3.	int i, j, save, im1;
4.	/*This function sorts array x in ascending order */
5.	If (n<2) return 1;
6.	for (i=2; i<=n; i++)
7.	{
8.	im1=i-1;
9.	for (j=1; j<=im; j++)
10.	if (x[i] < x[j])
11.	{
12.	Save = x[i];
13.	x[i] = x[j];
14.	x[j] = save;
15.	}
16.	}
17.	return 0;
18.	}

This specifically includes all lines containing program header, declaration, and executable and non-executable statements”.

Q4. Function Point Analysis

Alan Albrecht while working for IBM, recognized the problem in size measurement in the 1970s, and developed a technique (which he called Function Point Analysis), which appeared to be a solution to the size measurement problem.

The principle of Albrecht’s function point analysis (FPA) is that a system is decomposed into functional units.

The FPA functional units are shown in figure given below:

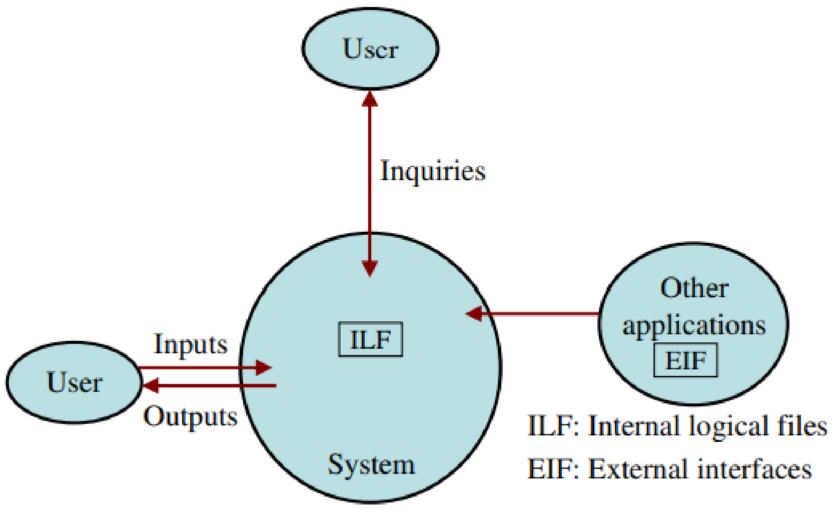


Fig. 3: FPAs functional units System

Function Point Analysis (FPA) measures software size using five functional units:

- **External Inputs (EI)**: Data or control inputs that the system processes (e.g., login forms).
- **External Outputs (EO)**: Processed data or reports provided to users (e.g., reports, invoices).
- **External Inquiries (EQ)**: Data retrieval requests without altering data (e.g., search queries).
- **Internal Logical Files (ILF)**: Internal data stored and maintained by the system (e.g., customer database).
- **External Interface Files (EIF)**: External data files used but not maintained by the system (e.g., shared files).

Special features Software Project Planning

Function point approach is independent of the language, tools, or methodologies used for implementation; i.e. they do not take into consideration programming languages, data base management systems, processing hardware or any other data base technology.

Function points can be estimated from requirement specification or design specification, thus making it possible to estimate development efforts in early phases of development.

Function points are directly linked to the statement of requirements; any change of requirements can easily be followed by a re-estimate.

Software Project Planning Function points are based on the system user's external view of the system, non-technical users of the software system have a better understanding of what function points are measuring.

Counting function points

Functional Units	Weighting factors		
	Low	Average	High
External Inputs (EI)	3	4	6
External Output (EO)	4	5	7
External Inquiries (EQ)	3	4	6
External logical files (ILF)	7	10	15
External Interface files (EIF)	5	7	10

Table 2: UFP calculation table

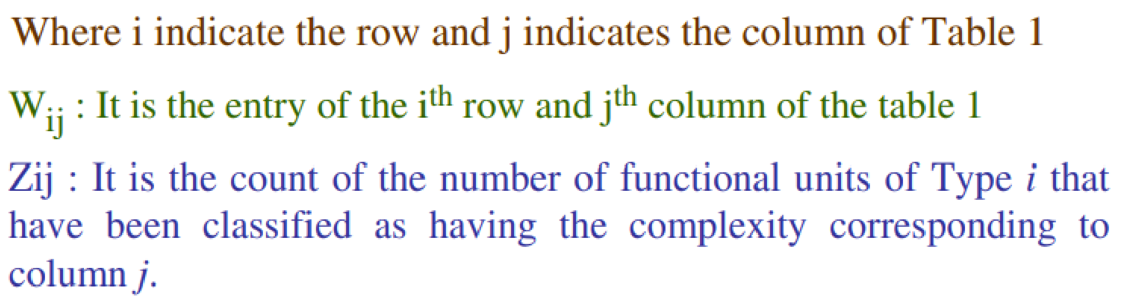
Functional Units	Count Complexity			Complexity Totals	Functional Unit Totals
External Inputs (EIs)		Low x 3	=		
		Average x 4	=		
		High x 6	=		
External Outputs (EOs)		Low x 4	=		
		Average x 5	=		
		High x 7	=		
External Inquiries (EQs)		Low x 3	=		
		Average x 4	=		
		High x 6	=		
External logical Files (ILFs)		Low x 7	=		
		Average x 10	=		
		High x 15	=		
External Interface Files (EIFs)		Low x 5	=		
		Average x 7	=		
		High x 10	=		
Total Unadjusted Function Point Count					

The weighting factors are identified for all functional units and multiplied with the functional units accordingly. The procedure for the calculation of Unadjusted Function Point (UFP) is given in table shown above.

Q5. Unadjusted Function Point

The procedure for the calculation of UFP in mathematical form is given below:

$$UFP = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$



Where i indicate the row and j indicates the column of Table 1

W_{ij} : It is the entry of the i^{th} row and j^{th} column of the table 1

Zij : It is the count of the number of functional units of Type *i* that have been classified as having the complexity corresponding to column *j*.

Q6 **Complexity Adjustment Factor**

CAF stands for **Complexity Adjustment Factor**, and it's used to adjust the **Unadjusted Function Points (UFP)** based on the complexity of the system. It's calculated as part of the Function Point Analysis (FPA) to reflect the overall influence of various system characteristics.

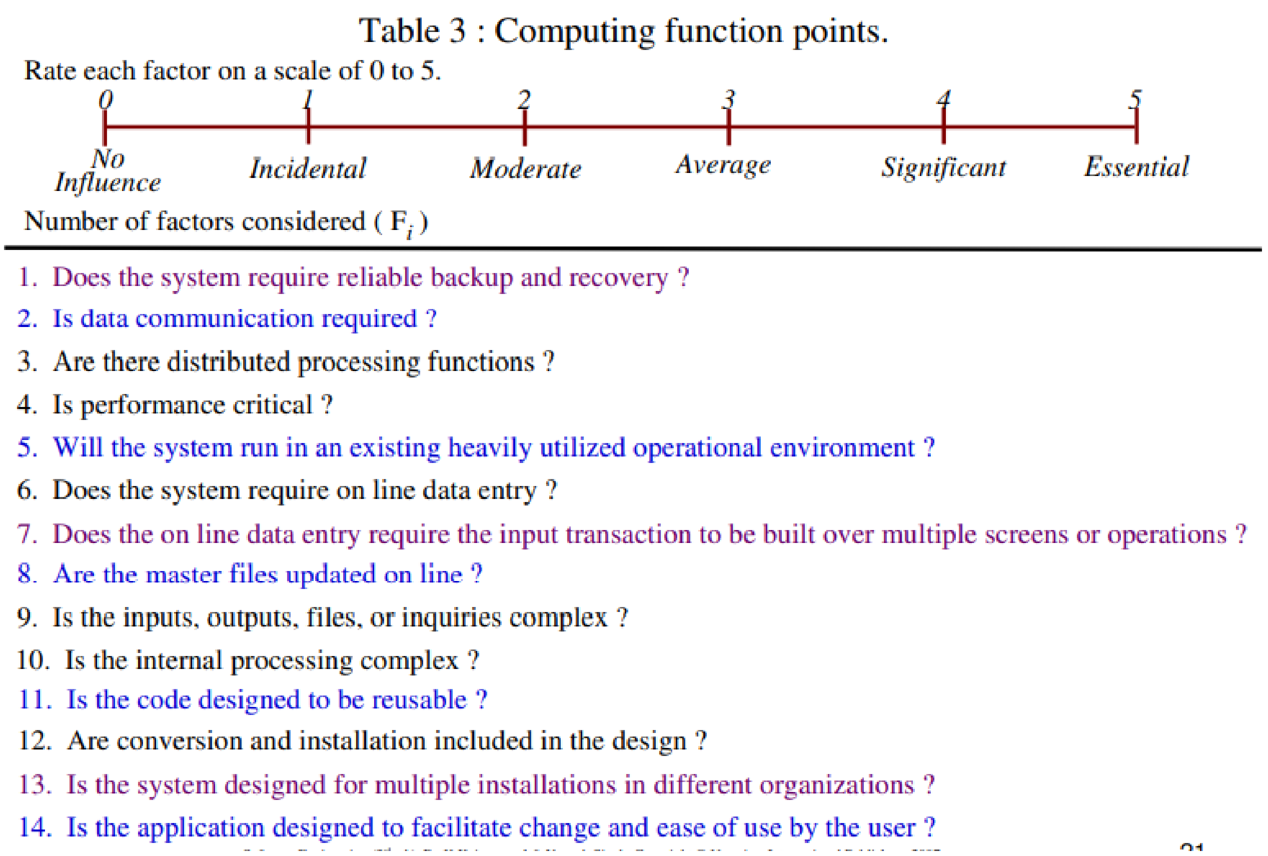


Table 3 : Computing function points.

Rate each factor on a scale of 0 to 5.

0	1	2	3	4	5
No Influence	*Incidental*	*Moderate*	*Average*	*Significant*	*Essential*

Number of factors considered (F_i)

1. Does the system require reliable backup and recovery ?
2. Is data communication required ?
3. Are there distributed processing functions ?
4. Is performance critical ?
5. Will the system run in an existing heavily utilized operational environment ?
6. Does the system require on line data entry ?
7. Does the on line data entry require the input transaction to be built over multiple screens or operations ?
8. Are the master files updated on line ?
9. Is the inputs, outputs, files, or inquiries complex ?
10. Is the internal processing complex ?
11. Is the code designed to be reusable ?
12. Are conversion and installation included in the design ?
13. Is the system designed for multiple installations in different organizations ?
14. Is the application designed to facilitate change and ease of use by the user ?

Q7. Consider a software project with the following parameters:

1. **External Inputs (EI)**:
   - 10 with low complexity
   - 15 with average complexity
   - 17 with high complexity
2. **External Outputs (EO)**:
   - 6 with low complexity
   - 13 with high complexity
3. **External Inquiries (EQ)**:
   - 3 with low complexity
   - 4 with average complexity
   - 2 with high complexity

4. **Internal Logical Files (ILF)**:
    - 2 with average complexity
    - 1 with high complexity
5. **External Interface Files (EIF)**:
    - 9 with low complexity

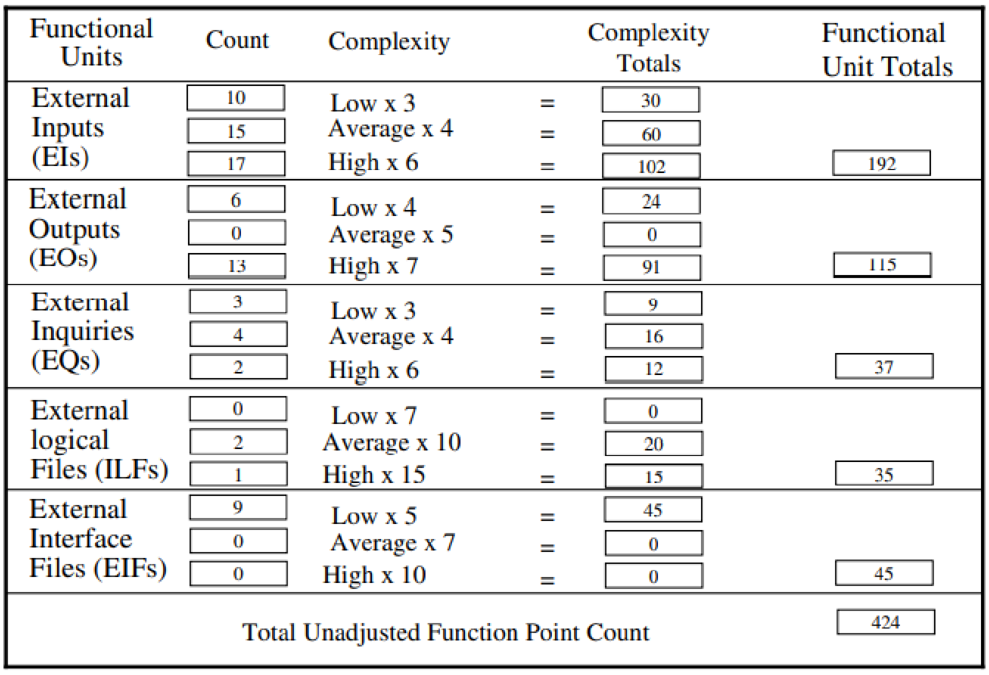
In addition, the system has the following complexity adjustment factors:

1. Significant data communication is required.
2. Performance is very critical.
3. The code is designed to be moderately reusable.
4. The system is **not** designed for multiple installations across different organizations.

All other complexity adjustment factors are treated as average.

Question: Compute the Function Points (FP) for this project.

Solution: Unadjusted function points may be counted using table 2



Functional Units	Count	Complexity		Complexity Totals	Functional Unit Totals
External Inputs (EIs)	10	Low x 3	=	30	
	15	Average x 4	=	60	
	17	High x 6	=	102	192
External Outputs (EOs)	6	Low x 4	=	24	
	0	Average x 5	=	0	
	13	High x 7	=	91	115
External Inquiries (EQs)	3	Low x 3	=	9	
	4	Average x 4	=	16	
	2	High x 6	=	12	37
External logical Files (ILFs)	0	Low x 7	=	0	
	2	Average x 10	=	20	
	1	High x 15	=	15	35
External Interface Files (EIFs)	9	Low x 5	=	45	
	0	Average x 7	=	0	
	0	High x 10	=	0	45
		Total Unadjusted Function Point Count			424

$$\sum_{i=1}^{14} F_i = 3+4+3+5+3+3+3+3+3+3+2+3+0+3=41$$

$$\begin{aligned}\text{CAF} &= (0.65 + 0.01 \text{ x } \Sigma F_i)\\ &= (0.65 + 0.01 \text{ x } 41)\\ &= 1.06\\ \text{FP} &= \text{UFP x CAF}\\ &= 424 \text{ x } 1.06\\ &= 449.44\end{aligned}$$

Hence $\boxed{\text{FP} = 449}$

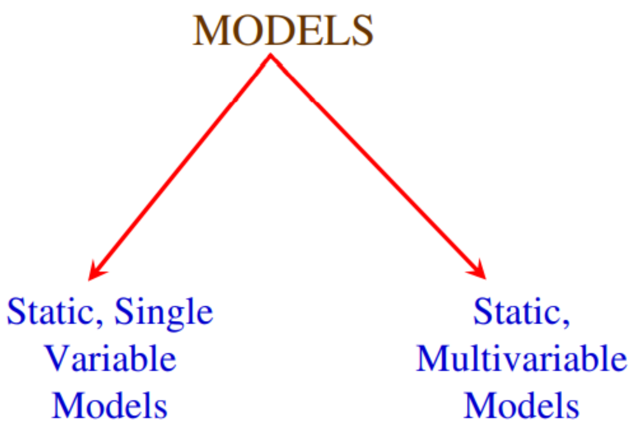
Q8. Cost Estimation

A number of estimation techniques have been developed and are having following attributes in common :

- Project scope must be established in advance
- Software metrics are used as a basis from which estimates are made
- The project is broken into small pieces which are estimated individually

To achieve reliable cost and schedule estimates, a number of options arise:

- Delay estimation until late in project
- Use simple decomposition techniques to generate project cost and schedule estimates
- Develop empirical models for estimation
- Acquire one or more automated estimation tools



Static, Single Variable Models

Methods using this model use an equation to estimate the desired values such as cost, time, effort, etc. They all depend on the same variable used as predictor (say, size). An example of the most common equations is :

$$C = a\,L^{b} \qquad \text{(i)}$$

C is the cost, L is the size and a,b are constants

$$E = 1.4\,L^{0.93}$$

$$DOC = 30.4\,L^{0.90}$$

$$D = 4.6\,L^{0.26}$$

Effort (E in Person-months), documentation (DOC, in number of pages) and duration (D, in months) are calculated from the number of lines of code (L, in thousands of lines) used as a predictor.

Static, Multivariable Models

These models are often based on equation (i), they actually depend on several variables representing various aspects of the software development environment, for example method used, user participation, customer oriented changes, memory constraints, etc.

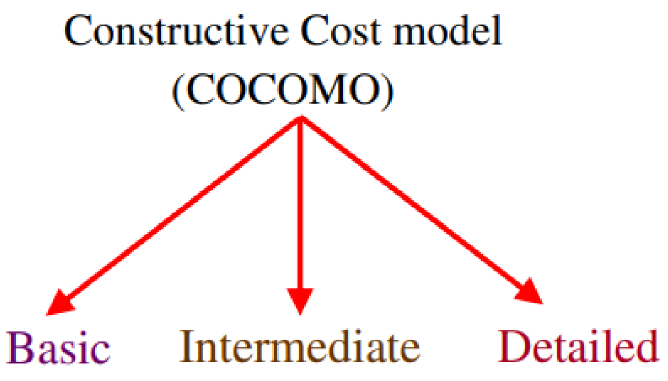
$$E = 5.2\, L^{0.91}$$

$$D = 4.1\, L^{0.36}$$

The productivity index uses 29 variables which are found to be highly correlated to productivity as follows:

$$I = \sum_{i=1}^{29} W_i X_i$$

Q9. The Constructive Cost model (COCOMO)

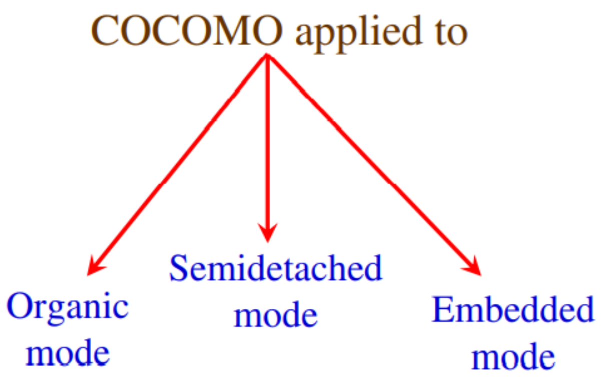


Model proposed by
B. W. Boehm's
through his book
Software Engineering Economics in 1981

The Constructive Cost Model (COCOMO) is a popular cost estimation model used in software engineering to predict project costs based on the size of the software and various project characteristics. Developed by Barry Boehm in 1981, it uses mathematical formulas to estimate the effort, time, and cost required to complete a software project.

Types of COCOMO Models:

1. **Basic COCOMO**: Provides a rough estimate of effort based on the size of the software in terms of **Lines of Code (LOC)**. It is the simplest form of COCOMO and classifies projects into three categories:



- **Organic**: Small teams working on well-understood projects.
- **Semi-detached**: Intermediate-sized projects with mixed teams and moderately complex requirements.
- **Embedded**: Complex projects involving tight hardware or software constraints.

Basic Model

Basic COCOMO model takes the form

$$E = a_b (KLOC)^{b_b}$$

$$D = c_b (E)^{d_b}$$

where E is effort applied in Person-Months, and D is the development time in months. The coefficients a_b, b_b, c_b and d_b are given in table 4 (a).

Software Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 4(a): Basic COCOMO coefficients

When effort and development time are known, the average staff size to complete the project may be calculated as:

$$\text{Average staff size } (SS) = \frac{E}{D} \text{ } Persons$$

When project size is known, the productivity level may be calculated as:

$$\text{Productivity } (P) = \frac{KLOC}{E} \text{ } KLOC / PM$$

2. **Intermediate COCOMO**: Includes additional factors called **cost drivers**, which adjust the basic estimate based on factors like product complexity, experience, and development tools. It provides a more accurate estimate than the Basic COCOMO.

Intermediate COCOMO cost drivers:

1. Product Attributes

- Required Software Reliability (RELY)
- Database Size (DATA)
- Product Complexity (CPLX)

2. Hardware Attributes

- Execution Time Constraint (TIME)
- Main Storage Constraint (STOR)
- Virtual Machine Volatility (VIRT)
- Computer Turnaround Time (TURN)

3. Personnel Attributes

- Analyst Capability (ACAP)
- Programmer Capability (PCAP)
- Personnel Continuity (PCON)
- Application Experience (AEXP)
- Programming Language Experience (LEXP)

4. Project Attributes

- Use of Modern Programming Practices (MODP)
- Use of Software Tools (TOOL)
- Required Development Schedule (SCED)

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Product Attributes						
RELY	0.75	0.88	1.00	1.15	1.40	--
DATA	--	0.94	1.00	1.08	1.16	--
CPLX	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME	--	--	1.00	1.11	1.30	1.66
STOR	--	--	1.00	1.06	1.21	1.56
VIRT	--	0.87	1.00	1.15	1.30	--
TURN	--	0.87	1.00	1.07	1.15	--

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Personnel Attributes						
ACAP	1.46	1.19	1.00	0.86	0.71	--
AEXP	1.29	1.13	1.00	0.91	0.82	--
PCAP	1.42	1.17	1.00	0.86	0.70	--
VEXP	1.21	1.10	1.00	0.90	--	--
LEXP	1.14	1.07	1.00	0.95	--	--
Project Attributes						
MODP	1.24	1.10	1.00	0.91	0.82	--
TOOL	1.24	1.10	1.00	0.91	0.83	--
SCED	1.23	1.08	1.00	1.04	1.10	--

Intermediate COCOMO equations

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

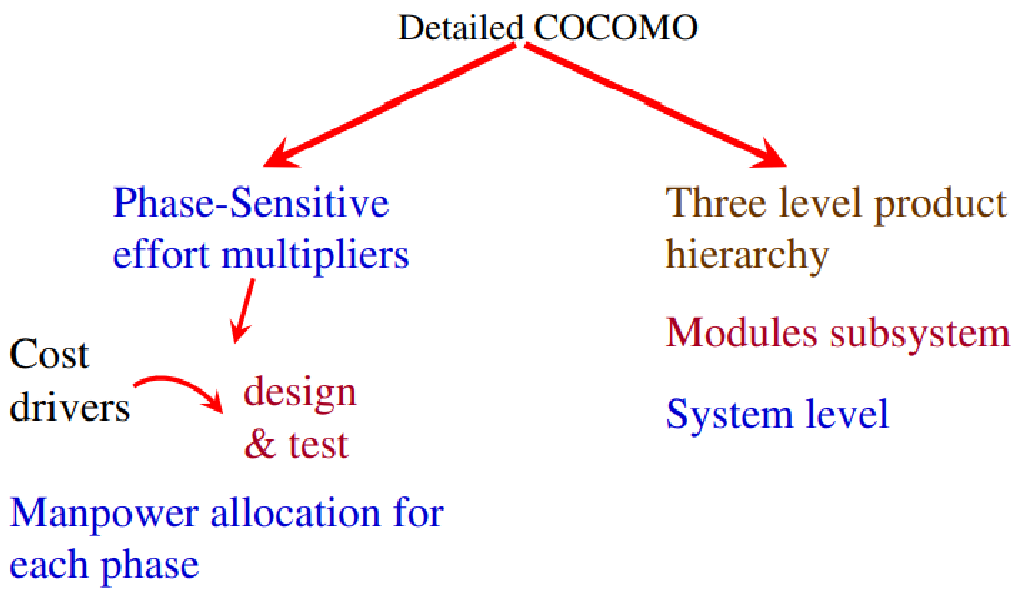
$$D = c_i (E)^{d_i}$$

Project	a_i	b_i	c_i	d_i
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

Table 6: Coefficients for intermediate COCOMO

3. **Detailed COCOMO**: Adds further refinement by breaking the project into smaller components and calculating costs for each component individually. It provides the most detailed and accurate estimates.

Detailed COCOMO Model


Detailed COCOMO
Phase-Sensitive effort multipliers
Three level product hierarchy
Cost drivers
design & test
Modules subsystem
System level
Manpower allocation for each phase

COCOMO-II

The following categories of applications / projects are identified by COCOMO-II and are shown in fig. 4 shown below:

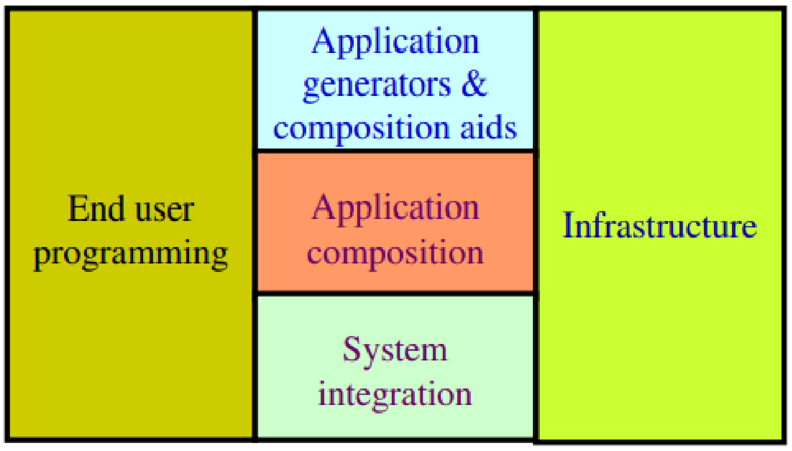
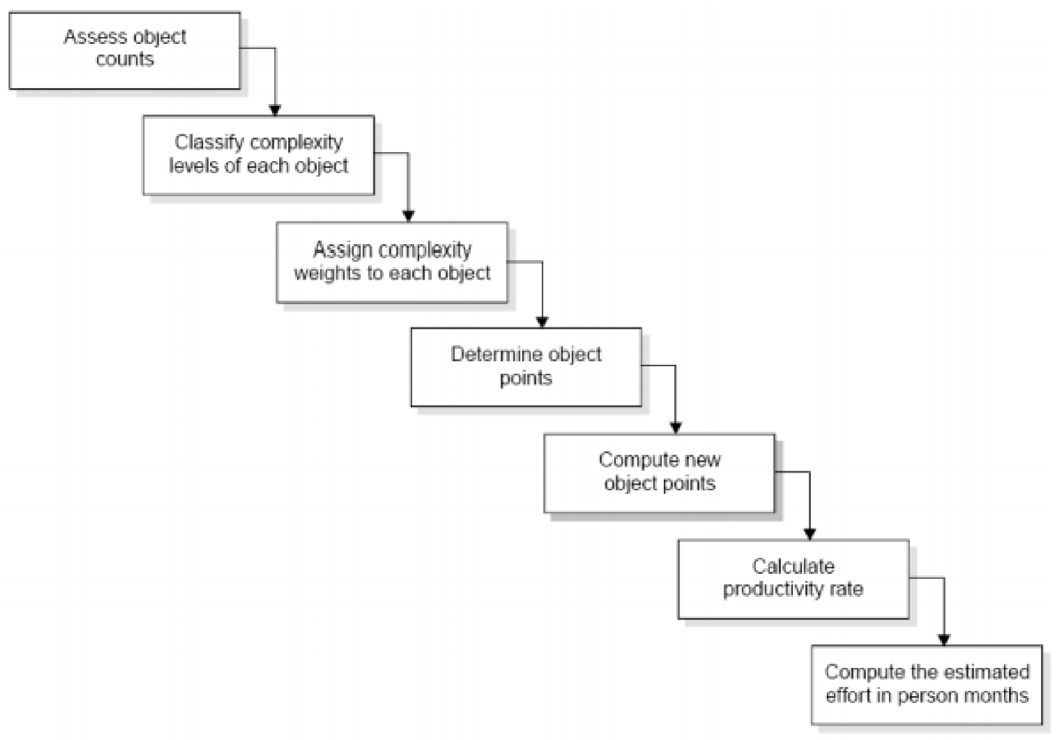


Fig. 4 : Categories of applications / projects

Stage No	Model Name	Application for the types of projects	Applications
Stage I	Application composition estimation model	Application composition	In addition to application composition type of projects, this model is also used for prototyping (if any) stage of application generators, infrastructure & system integration.
Stage II	Early design estimation model	Application generators, infrastructure & system integration	Used in early design stage of a project, when less is known about the project.
Stage III	Post architecture estimation model	Application generators, infrastructure & system integration	Used after the completion of the detailed architecture of the project.

Q10. ACEM

Application Composition Estimation Model


Assess object counts
Classify complexity levels of each object
Assign complexity weights to each object
Determine object points
Compute new object points
Calculate productivity rate
Compute the estimated effort in person months

Q11. Putnam Resource Allocation Model

The **Putnam Resource Allocation Model**, also known as the **Putnam Model** or **SLIM (Software Life Cycle Management)**, is a software cost estimation model developed by Lawrence H. Putnam in the 1970s. It focuses on the relationship between the effort, time, and productivity of software development projects. The model is based on the idea that software development follows a predictable curve, where the allocation of resources over time significantly influences project success.

Putnam Resource Allocation Model

Norden of IBM

Rayleigh curve

Model for a range of hardware development projects.

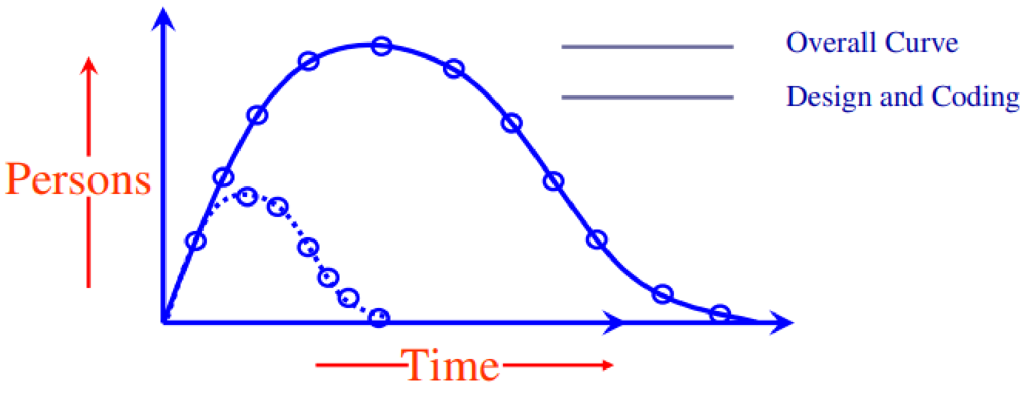


Fig.6: The Rayleigh manpower loading curve

Resource Allocation: The model emphasizes the importance of efficiently allocating resources (i.e., personnel) over the project's lifecycle. It recognizes that the timing and amount of resources directly affect project duration and quality.

- **Effort and Time**: The model establishes a relationship between the total effort (in person-months) and the development time (in months). As development time increases, the effort required decreases and vice versa.
- **Productivity**: Productivity is defined as the amount of software produced per unit of effort. The Putnam Model uses the concept of **"productivity curves,"** which illustrate how productivity changes over time based on resource allocation.
- **Quality vs. Time**: The model suggests that higher quality can be achieved by extending development time and adequately allocating resources, allowing for more thorough testing and refinement.
- **Cumulative Distribution Function**: The model uses a cumulative distribution function to estimate the probability of meeting specific project deadlines based on resource allocation and effort.

Advantages:

- Provides a systematic approach to resource allocation and effort estimation.
- Considers the impact of time and resources on productivity and quality.
- Useful for planning and managing large software projects.

Disadvantages:

- Requires accurate input data, which can be challenging to obtain.
- May not account for all real-world complexities of software development.
- Assumes a predictable development process, which may not always hold true.

The Norden / Rayleigh Curve

The curve is modeled by differential equation

$$m(t) = \frac{dy}{dt} = 2kate^{-at^2} \qquad \text{---------} \ (1)$$

$\frac{dy}{dt}$	= manpower utilization rate per unit time
a	= parameter that affects the shape of the curve
K	= area under curve in the interval [0, ∞]
t	= elapsed time

Putnam observed that

Difficulty derivative relative to time

$$\begin{tikzcd} \text{Difficulty derivative relative to time} \arrow[d, bend right] \\ \text{Behavior of s/w development} \end{tikzcd}$$

If project scale is increased, the development time also increase to such an extent that $\frac{k}{t_d^3}$ remains constant around a value which could be 8,15,27.

$$\begin{tikzcd}[column sep=small] & \text{Project is difficult to develop if} \arrow[dl] \arrow[dr] & \\ \text{Manpower demand is high} & & \text{When time schedule is short} \end{tikzcd}$$

Productivity Versus Difficulty

Productivity = No. of LOC developed per person-month

$$P \propto D^{\beta}$$

Avg. productivity

$$P = \frac{LOC \ produced}{\begin{array}{l} cumulative \ manpower \\ used \ to \ produce \ code \end{array}}$$

$$S = 0.3935\,\phi\, K^{1/3} t_d^{\ 4/3}$$

$$\begin{tikzcd}
0.39\phi \arrow[r] & c \arrow[d] & & \\
& \text{Technology Factor} \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\text{Hardware constraints} & \text{Complexity} & \text{Experience} & \text{Programming environment}
\end{tikzcd}$$

Q12. Development Cycle

Development Subcycle

All that has been discussed so far is related to project life cycle as represented by project curve

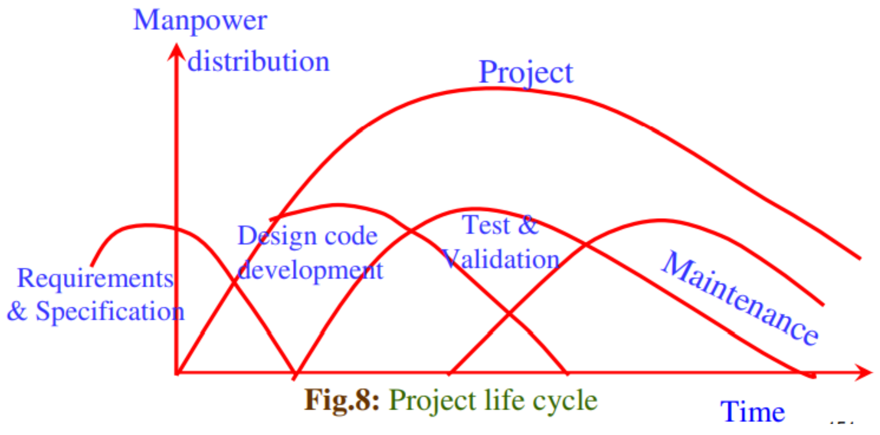
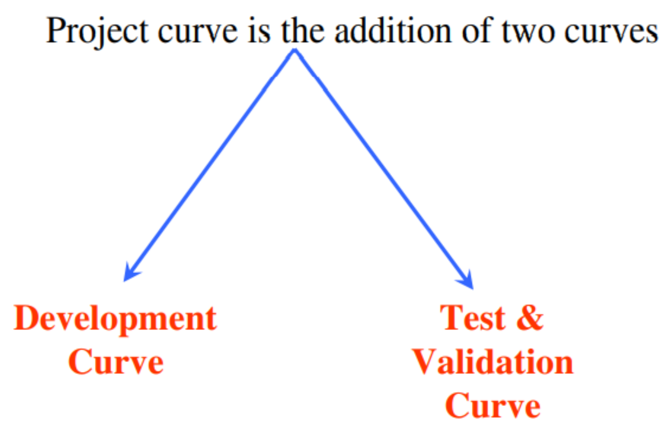
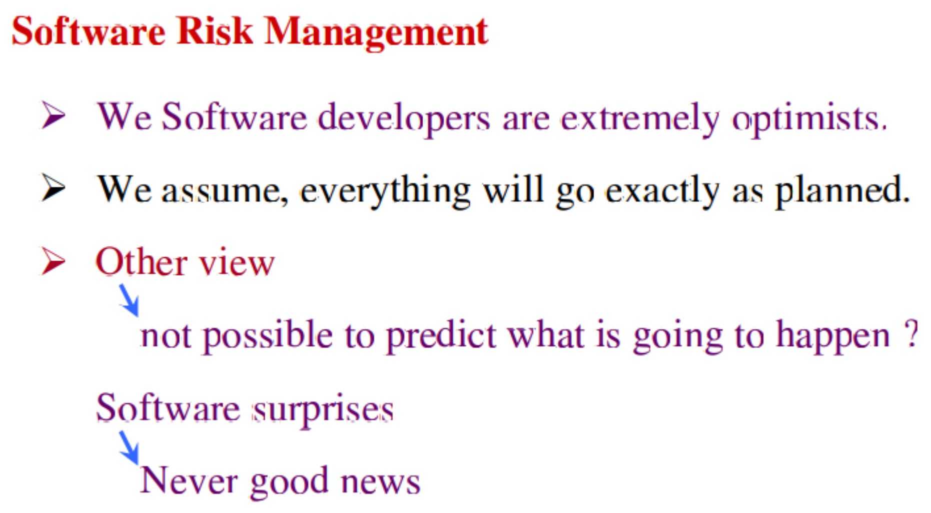


Fig.8: Project life cycle

Project life cycle



Q13. Risk Management



Software Risk Management

- ➢ We Software developers are extremely optimists.
- ➢ We assume, everything will go exactly as planned.
- ➢ Other view

  ↘ not possible to predict what is going to happen ?

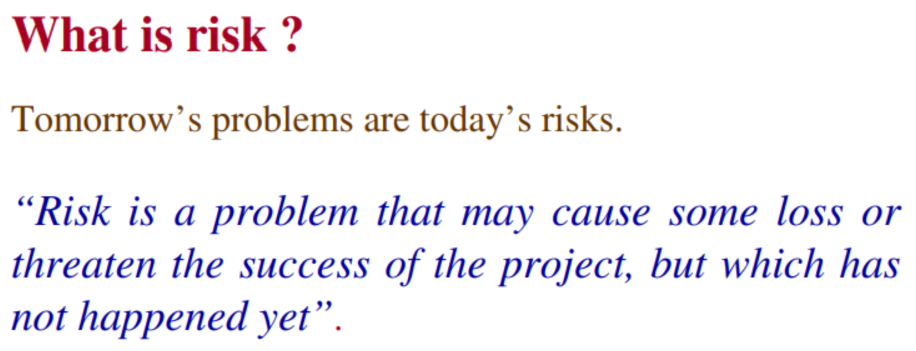
  Software surprises

  ↘ Never good news

Risk management is required to reduce this surprise factor

Dealing with concern before it becomes a crisis.

Quantify probability of failure & consequences of failure.



What is risk ?

Tomorrow’s problems are today’s risks.

“Risk is a problem that may cause some loss or threaten the success of the project, but which has not happened yet”.

Risk management is the process of identifying addressing and eliminating these problems before they can damage the project

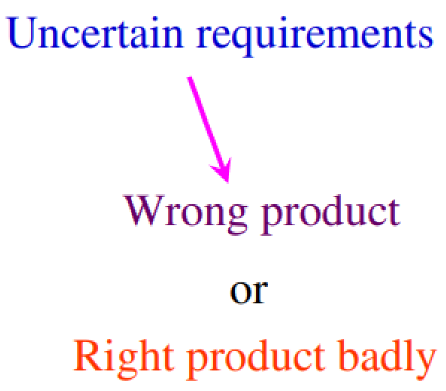
Q14. Typical Software Risk

Capers Jones has identified the top five risk factors that threaten projects in different applications.

1. Dependencies on outside agencies or factors.

• Availability of trained, experienced persons • Inter group dependencies • Customer-Furnished items or information • Internal & external subcontractor relationships

2. Requirement issues



• Lack of clear product vision • Unprioritized requirements • Lack of agreement on product requirements • New market with uncertain needs • Rapidly changing requirements • Inadequate Impact analysis of requirements changes

3. Management Issues

Project managers usually write the risk management plans, and most people do not wish to air their weaknesses in public.

• Inadequate planning • Inadequate visibility into actual project status • Unclear project ownership and decision making • Staff personality conflicts • Unrealistic expectation • Poor communication

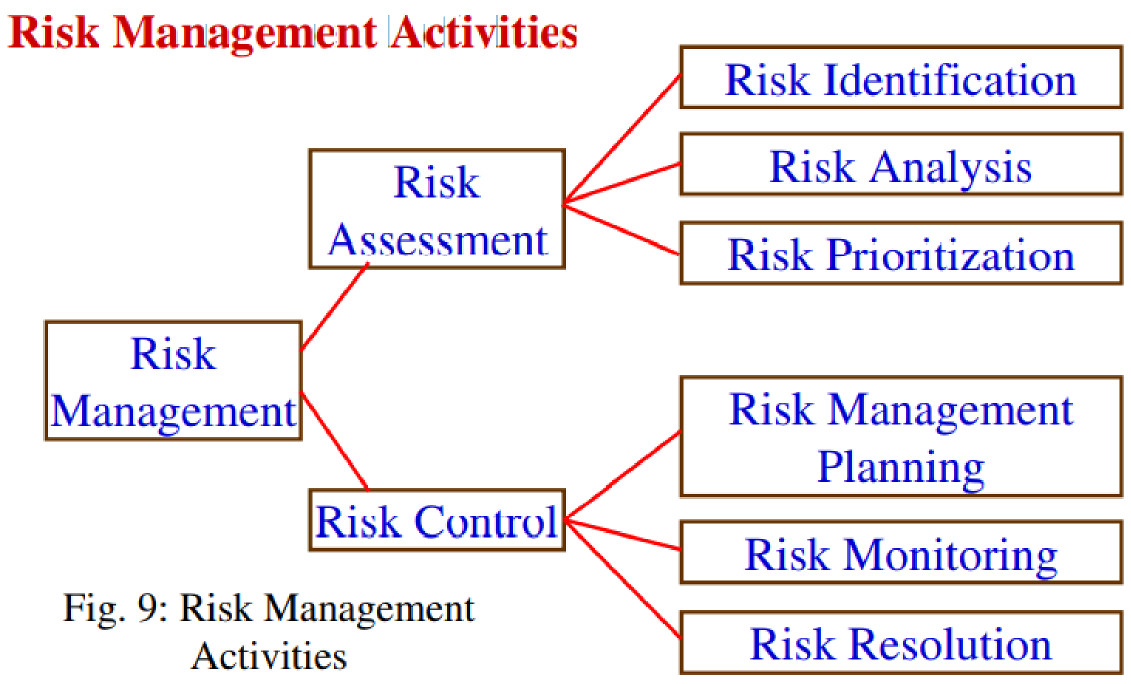
4. Lack of knowledge

• Inadequate training • Poor understanding of methods, tools, and techniques • Inadequate application domain experience • New Technologies • Ineffective, poorly documented or neglected processes

5. Other risk categories

• Unavailability of adequate testing facilities • Turnover of essential personnel • Unachievable performance requirements • Technical approaches that may not work

Q15. Risk Management Activities



Risk Assessment

- **Risk Identification**: Systematically identifying potential risks that could impact the project.
- **Risk Analysis**: Examining how changes in risk input variables might affect project outcomes.
- **Risk Prioritization**: Focusing on identifying and addressing severe risks based on their impact and likelihood.

Risk Controlling

- **Risk Management Planning**: Developing a comprehensive plan to manage each significant risk identified.
- **Risk Monitoring**: Continuously tracking identified risks and assessing their status throughout the project lifecycle.
- **Risk Resolution**: Implementing the planned strategies to effectively manage and mitigate each identified risk.

Q16. Software Design Document (SDD)

A **Software Design Document (SDD)** is a comprehensive document that outlines the design of a software system. It serves as a blueprint for the development team and stakeholders, detailing how the system will be constructed to meet specified requirements
The SDD serves multiple purposes:

- Provides a clear understanding of how the system will be built and how it will function.
- Facilitates communication among stakeholders, including developers, project managers, and clients.
- Acts as a reference throughout the software development lifecycle, ensuring alignment with design goals and requirements.

Q17. Design Framework

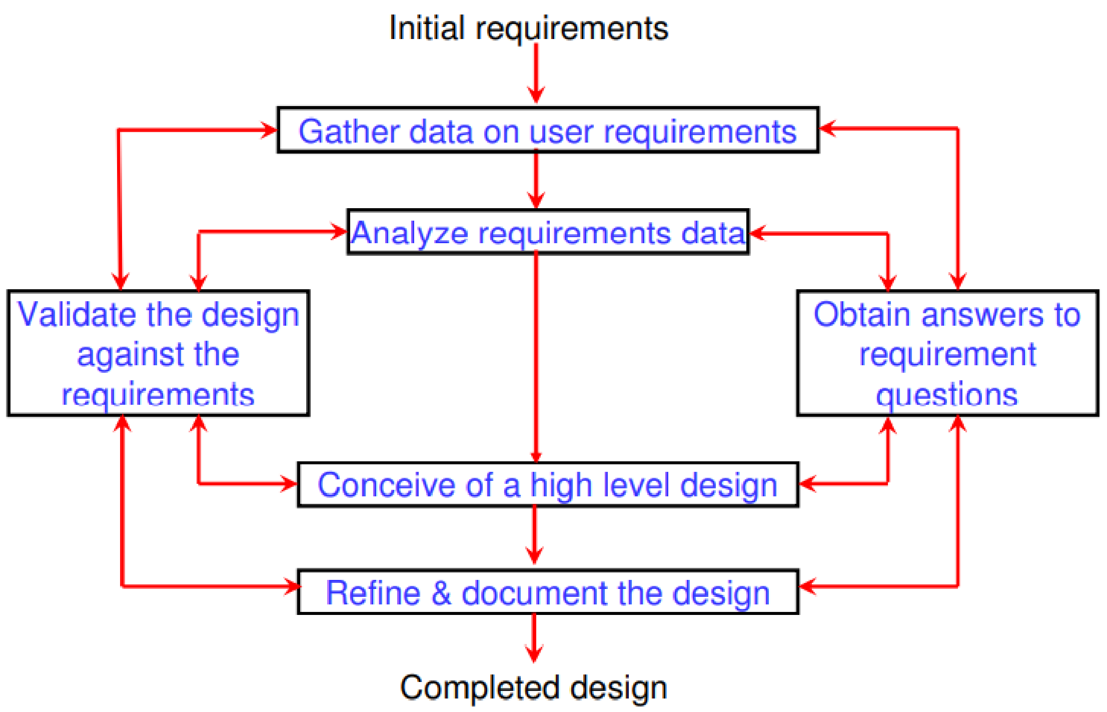

Initial requirements
Gather data on user requirements
Analyze requirements data
Validate the design against the requirements
Obtain answers to requirement questions
Conceive of a high level design
Refine & document the design
Completed design

Fig. 1 : Design framework

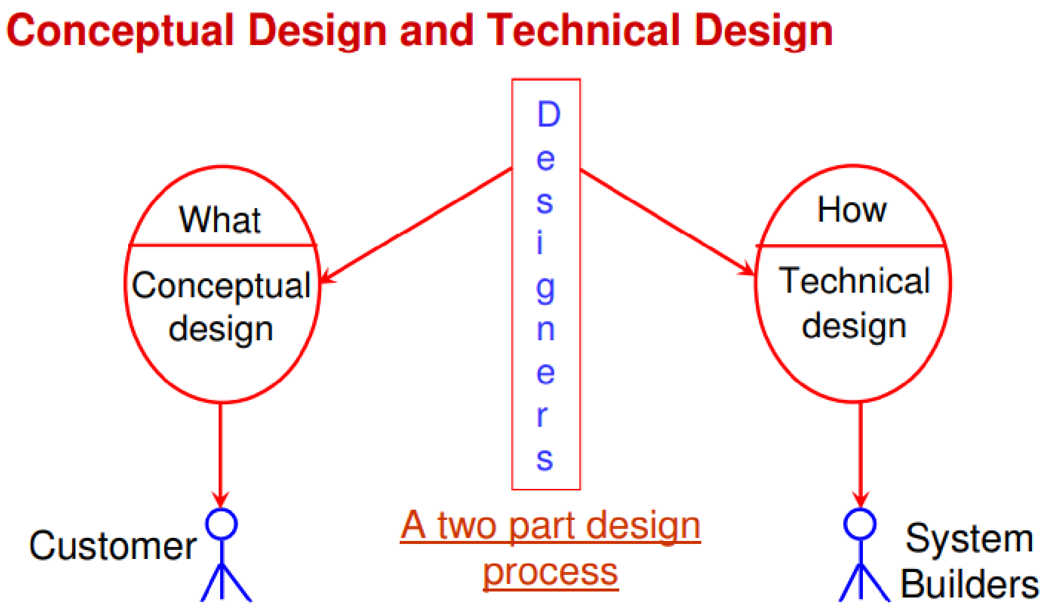


Fig. 2 : A two part design process

The design needs to be

Correct & complete
Understandable
At the right level
Maintainable

Q18. Modularity

A modular system consist of well defined manageable units with well defined interfaces among the units

Properties :
i. Well defined subsystem
ii. Well defined purpose
iii. Can be separately compiled and stored in a library.
iv. Module can use other modules
v. Module should be easier to use than to build
vi. Simpler from outside than from the inside.

Modularity is the single attribute of software that allows a program to be intellectually manageable. It enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of software product.

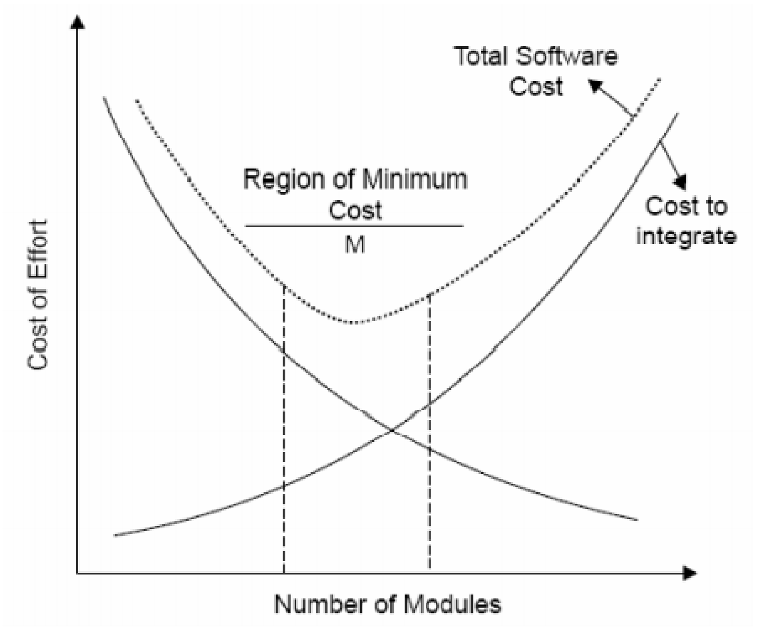
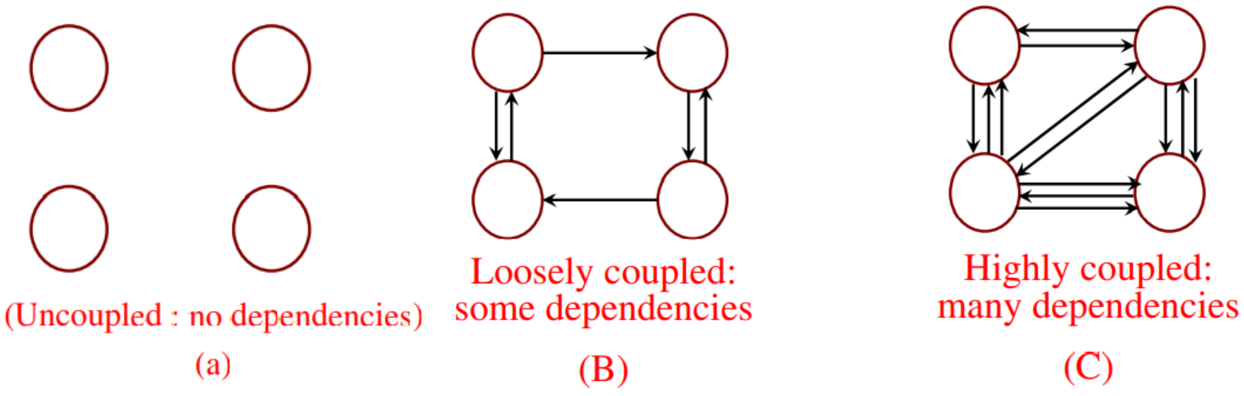


Fig. 4 : Modularity and software cost

Q19 Module Coupling

Coupling is the measure of the degree of interdependence between modules.


(Uncoupled : no dependencies)
(a)
Loosely coupled: some dependencies
(B)
Highly coupled: many dependencies
(C)

This can be achieved as: Controlling the number of parameters passed amongst modules. Avoid passing undesired data to calling module. Maintain parent / child relationship between calling & called modules. Pass data, not the control information.

Q20. Types of Coupling:

Consider the example of editing a student record in a 'student information system'.

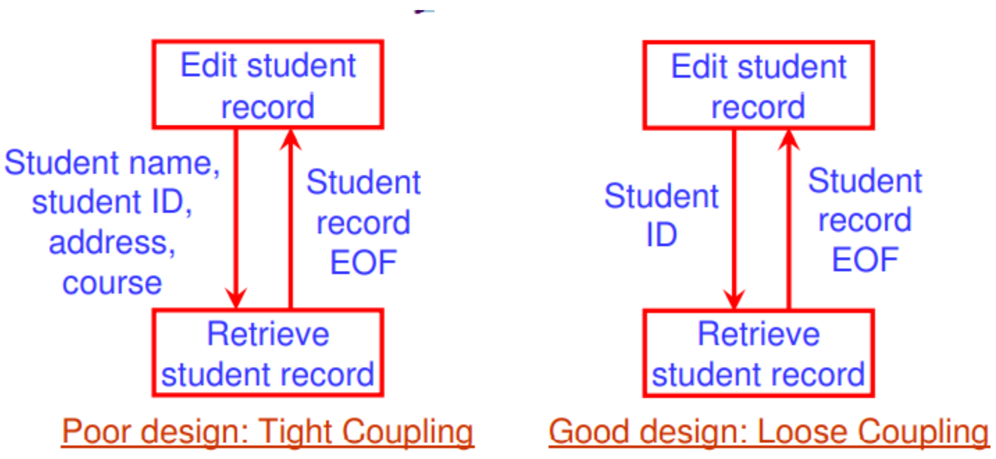
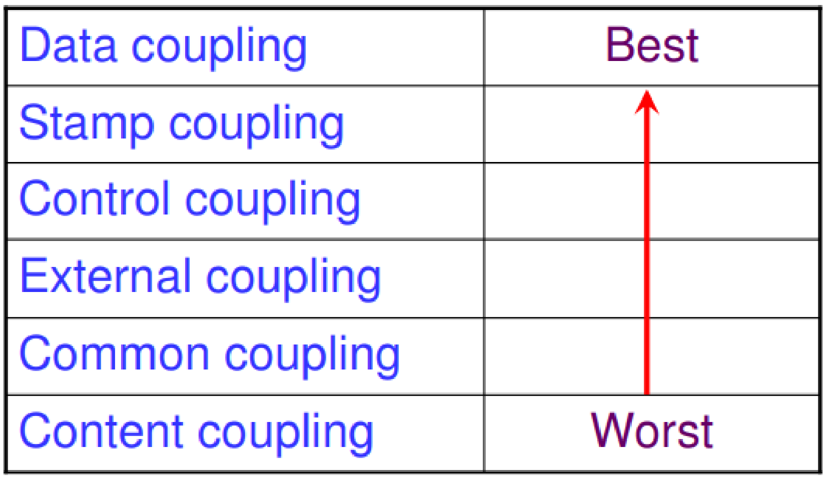


Fig. 6 : Example of coupling



Data coupling	Best
Stamp coupling	↑
Control coupling	
External coupling	
Common coupling	
Content coupling	Worst

Fig. 7 : The types of module coupling

Data coupling

The dependency between module A and B is said to be data coupled if their dependency is based on the fact they communicate by only passing of data. Other than communicating through data, the two modules are independent.

Stamp coupling

Stamp coupling occurs between module A and B when complete data structure is passed from one module to another.

Control coupling

Module A and B are said to be control coupled if they communicate by passing of control information. This is usually accomplished by means of flags that are set by one module and reacted upon by the dependent module.

Common coupling

With common coupling, module A and module B have shared data. Global data areas are commonly found in programming languages. Making a change to the common data means tracing back to all the modules which access that data to evaluate the effect of changes.

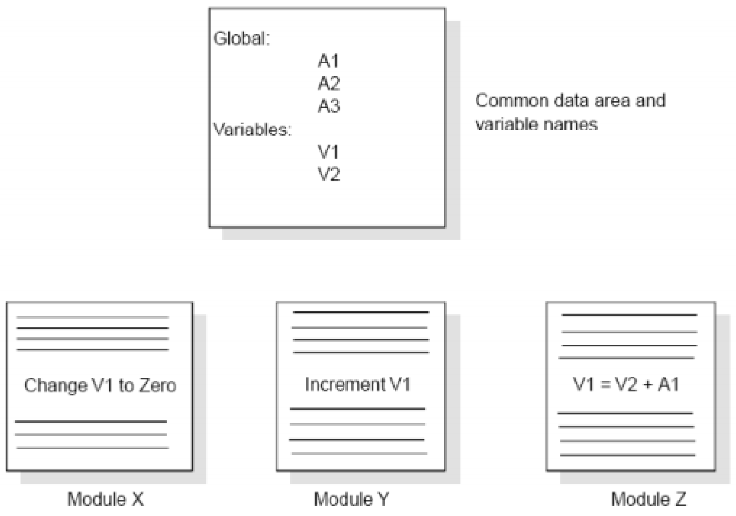


Fig. 8 : Example of common coupling

Content coupling

Content coupling occurs when module A changes data of module B or when control is passed from one module to the middle of another. In Fig. 9, module B branches into D, even though D is supposed to be under the control of C.

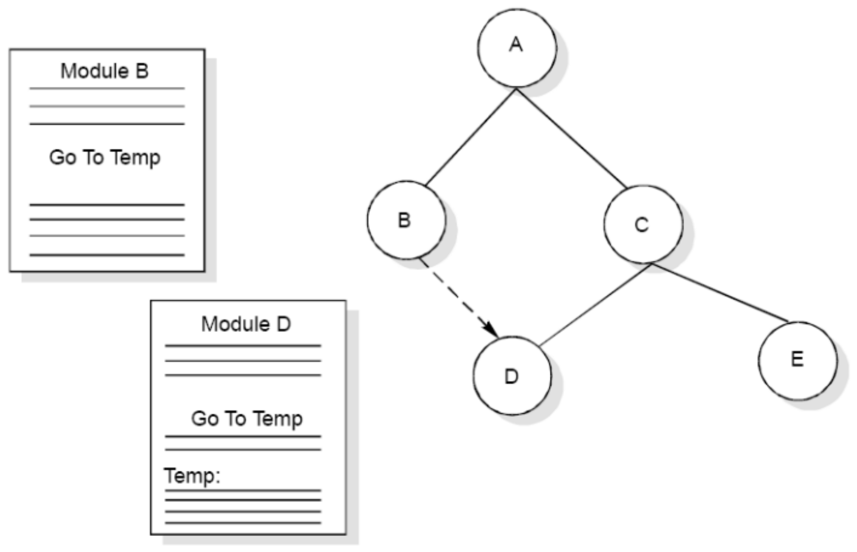


Fig. 9 : Example of content coupling

Q21 Module Cohesion

Cohesion is a measure of the degree to which the elements of a module are functionally related.

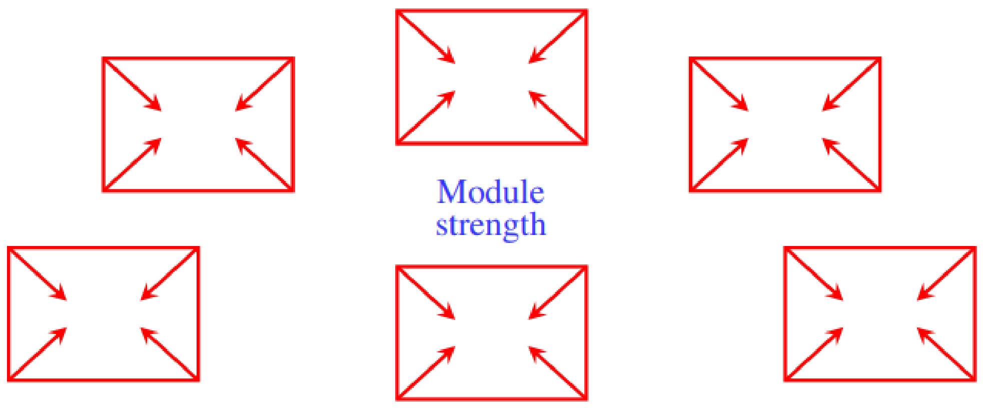


Fig. 10 : Cohesion=Strength of relations within modules

Types of cohesion

- ➢ Functional cohesion
- ➢ Sequential cohesion
- ➢ Procedural cohesion
- ➢ Temporal cohesion
- ➢ Logical cohesion
- ➢ Coincident cohesion

Functional Cohesion	Best (high)
Sequential Cohesion	↑
Communicational Cohesion	
Procedural Cohesion	
Temporal Cohesion	
Logical Cohesion	
Coincidental Cohesion	Worst (low)

Fig. 11 : Types of module cohesion

Functional Cohesion
A and B are part of a single functional task. This is very good reason for them to be contained in the same procedure.

Sequential Cohesion
Module A outputs some data which forms the input to B. This is the reason for them to be contained in the same procedure.

Procedural Cohesion
Procedural Cohesion occurs in modules whose instructions although accomplish different tasks yet have been combined because there is a specific order in which the tasks are to be completed.

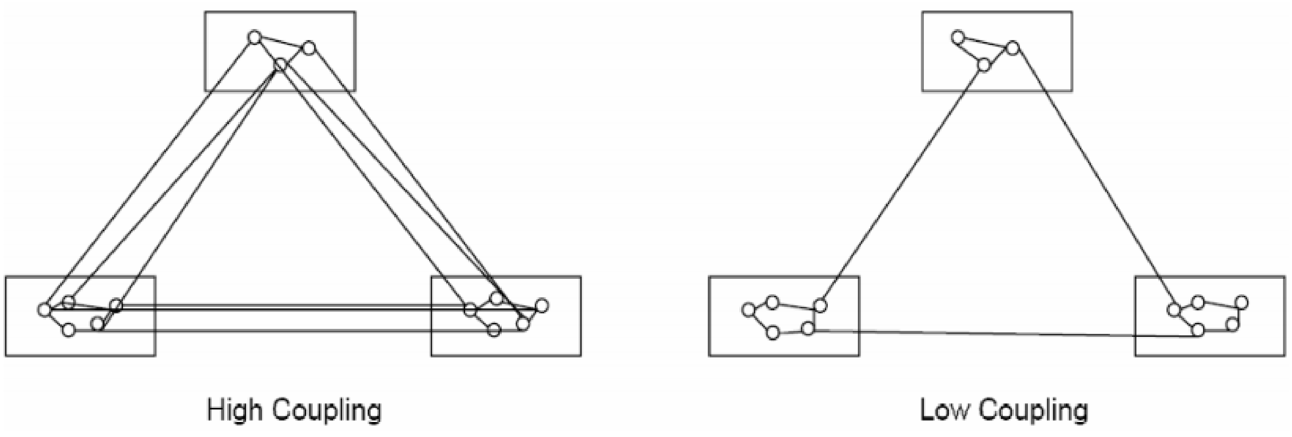
Temporal Cohesion
Module exhibits temporal cohesion when it contains tasks that are related by the fact that all tasks must be executed in the same time-span.

Logical Cohesion
Logical cohesion occurs in modules that contain instructions that appear to be related because they fall into the same logical class of functions.

Coincidental Cohesion
Coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another.

Q22. Relationship between Cohesion & Coupling

If the software is not properly modularized, a host of seemingly trivial enhancement or changes will result into death of the project. Therefore, a software engineer must design the modules with goal of high cohesion and low coupling



Q23. STRATEGY OF DESIGN

A good system design strategy is to organize the program modules in such a way that are easy to develop and latter to, change. Structured design techniques help developers to deal with the size and complexity of programs. Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program. It is important for two reasons:

First, even pre-existing code, if any, needs to be understood, organized and pieced together.

Second, it is still common for the project team to have to write some code and produce original programs that support the application logic of the system.

Q24. Bottom-Up Design:

Bottom-up tree structure These modules are collected together in the form of a “library”.

Bottom-up design is a software design approach where individual modules or components are developed first, starting from the lower-level functionalities. These small, independent modules are created, tested, and combined to form more complex systems, building up towards the higher-level system requirements.

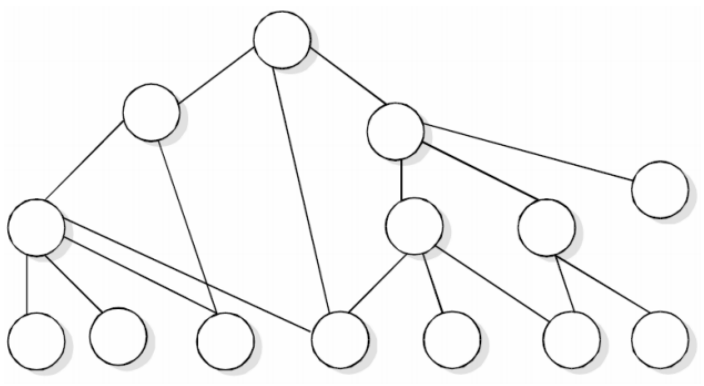
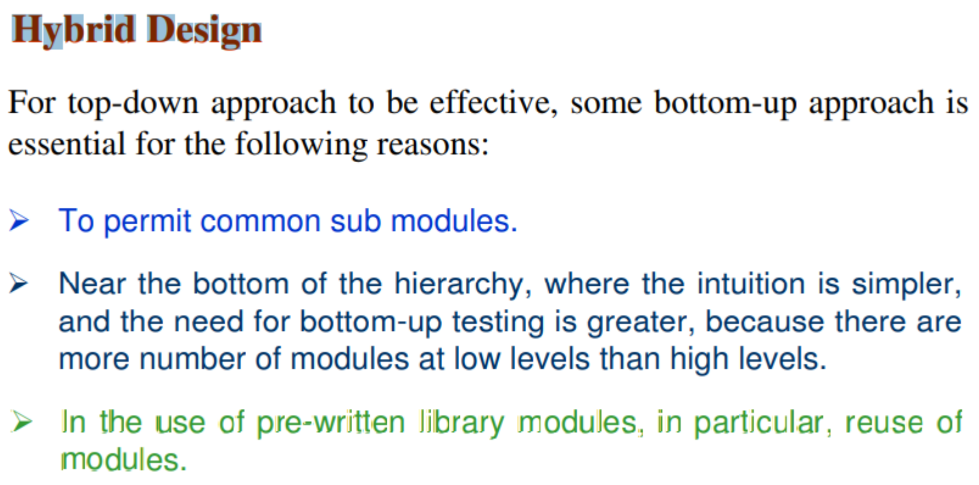


Fig. 13 : Bottom-up tree structure

Q25. Top-Down Design

A top down design approach starts by identifying the major modules of the system, decomposing them into their lower level modules and iterating until the desired level of detail is achieved. This is stepwise refinement; starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly.

Q26. Hybrid Design


Hybrid Design

For top-down approach to be effective, some bottom-up approach is essential for the following reasons:

- To permit common sub modules.
- Near the bottom of the hierarchy, where the intuition is simpler, and the need for bottom-up testing is greater, because there are more number of modules at low levels than high levels.
- In the use of pre-written library modules, in particular, reuse of modules.

Q27. FUNCTION ORIENTED DESIGN

Function Oriented design is an approach to software design where the design is decomposed into a set of interacting units where each unit has a clearly defined function. Thus, system is designed from a functional viewpoint.

Function-oriented design is a software design approach that focuses on decomposing the system into a set of functions or procedures. These functions represent specific tasks or operations that the system performs, often derived from the system's requirements. Key characteristics include:

Q28. Design Notations

Design notations are largely meant to be used during the process of design and are used to represent design or design decisions. For a function oriented design, the design can be represented graphically or mathematically by the following:

- Data flow diagrams
- Data Dictionaries
- Structure Charts
- Pseudocode

Structure Chart

It partition a system into block boxes. A black box means that functionality is known to the user without the knowledge of internal design.

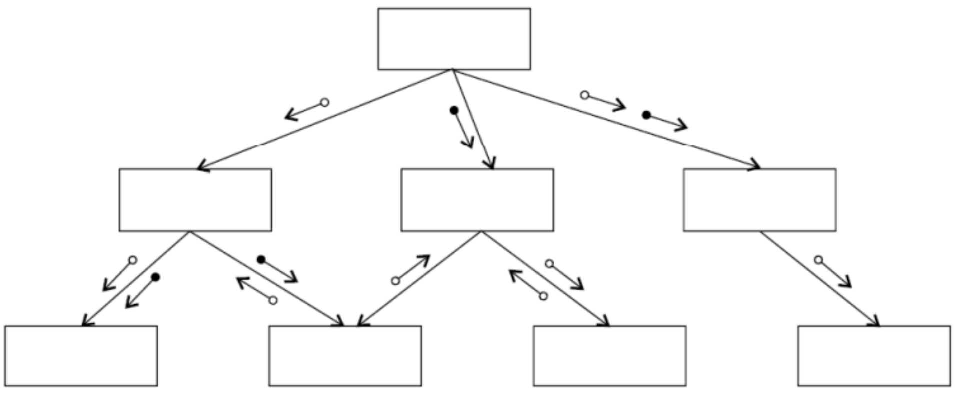


Fig. 16 : Hierarchical format of a structure chart

Pseudocode

Pseudocode notation can be used in both the preliminary and detailed design phases. Using pseudocode, the designer describes system characteristics using short, concise, English language phrases that are structured by key words such as It-Then-Else, While-Do, and End.

Q29. Functional Procedure Layers

Function are built in layers, Additional notation is used to specify details.

- Level 0
  - Function or procedure name
  - Relationship to other system components (e.g., part of which system, called by which routines, etc.)
  - Brief description of the function purpose.
  - Author, date
- Level 1
  - Function Parameters (problem variables, types, purpose, etc.)
  - Global variables (problem variable, type, purpose, sharing information)
  - Routines called by the function
  - Side effects
  - Input/Output Assertions
- Level 2
  - Local data structures (variable etc.)
  - Timing constraints
  - Exception handling (conditions, responses, events)
  - Any other limitations
- Level 3
  - Body (structured chart, English pseudo code, decision tables, flow charts, etc.)

Q30. IEEE Recommended practice for software design descriptions (IEEE STD 1016-1998)

- Scope

  An SDD is a representation of a software system that is used as a medium for communicating software design information.

- References

  i. IEEE std 830-1998, IEEE recommended practice for software requirements specifications.

  ii. IEEE std 610.12-1990, IEEE glossary of software engineering terminology.

- Definitions

  i. **Design entity.** An element (Component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.

  ii. **Design View.** A subset of design entity attribute information that is specifically suited to the needs of a software project activity.

  iii. **Entity attributes.** A named property or characteristics of a design entity. It provides a statement of fact about the entity.

  iv. **Software design description (SDD).** A representation of a software system created to facilitate analysis, planning, implementation and decision making.

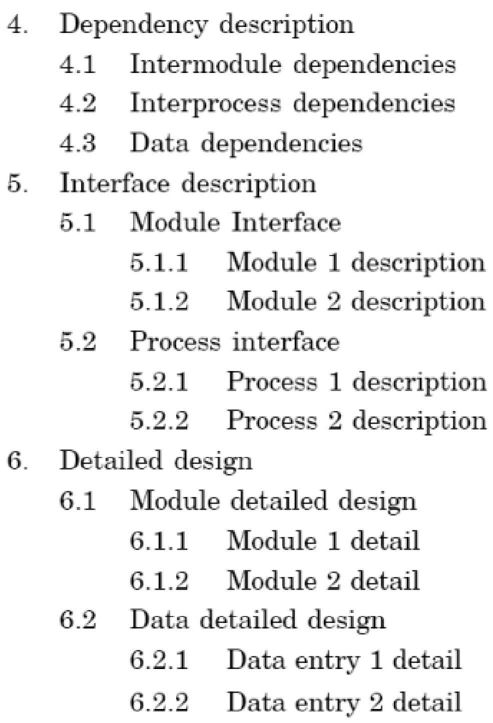
Q31. Design Description Organization

Each design description writer may have a different view of what are considered the essential aspects of a software design. The organization of SDD is given in table 1. This is one of the possible ways to organize and format the SDD. Software Design Design Description Organization

A recommended organization of the SDD into separate design views to facilitate information access and assimilation is given in table

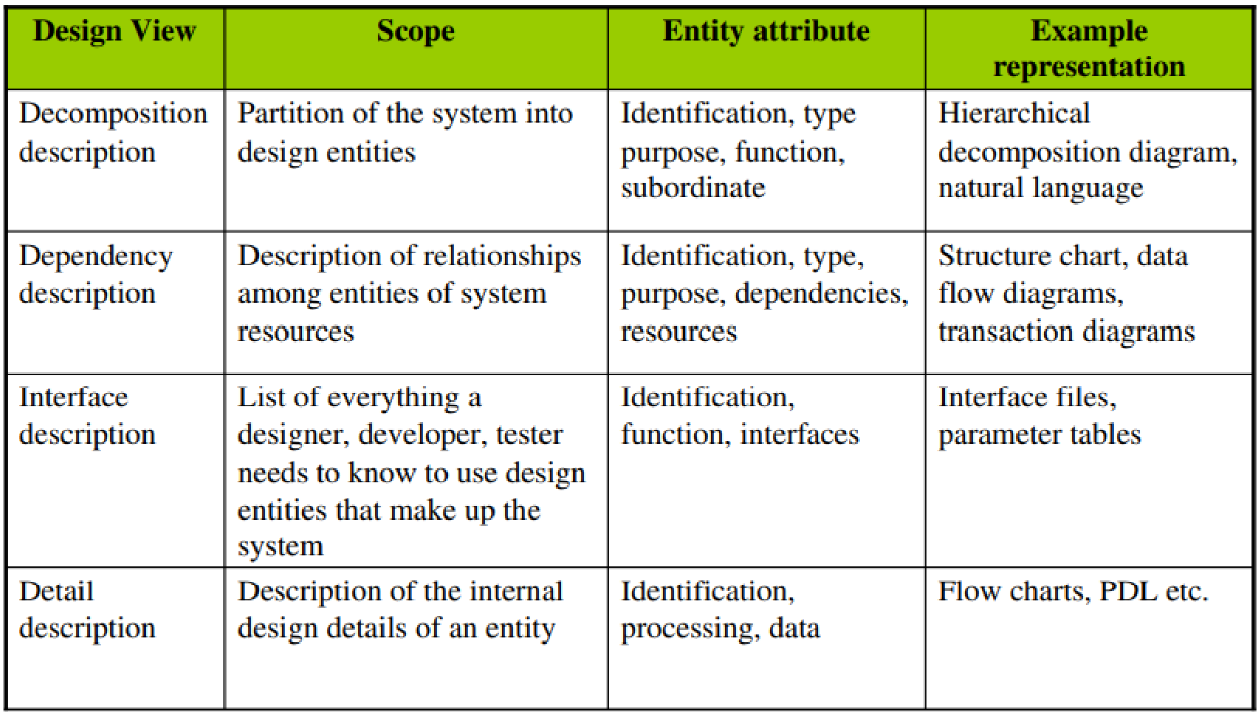
1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions and acronyms
2. References
3. Decomposition description
   - 3.1 Module decomposition
     - 3.1.1 Module 1 description
     - 3.1.2 Module 2 description
   - 3.2 Concurrent Process decompostion
     - 3.2.1 Process 1 description
     - 3.2.2 Process 2 description
   - 3.3 Data decomposition
     - 3.3.1 Data entity 1 description
     - 3.3.2 Data entity 2 description

Cont…



4. Dependency description
   - 4.1 Intermodule dependencies
   - 4.2 Interprocess dependencies
   - 4.3 Data dependencies
5. Interface description
   - 5.1 Module Interface
     - 5.1.1 Module 1 description
     - 5.1.2 Module 2 description
   - 5.2 Process interface
     - 5.2.1 Process 1 description
     - 5.2.2 Process 2 description
6. Detailed design
   - 6.1 Module detailed design
     - 6.1.1 Module 1 detail
     - 6.1.2 Module 2 detail
   - 6.2 Data detailed design
     - 6.2.1 Data entry 1 detail
     - 6.2.2 Data entry 2 detail

Table 1: Organization of SDD



Design View	Scope	Entity attribute	Example representation
Decomposition description	Partition of the system into design entities	Identification, type purpose, function, subordinate	Hierarchical decomposition diagram, natural language
Dependency description	Description of relationships among entities of system resources	Identification, type, purpose, dependencies, resources	Structure chart, data flow diagrams, transaction diagrams
Interface description	List of everything a designer, developer, tester needs to know to use design entities that make up the system	Identification, function, interfaces	Interface files, parameter tables
Detail description	Description of the internal design details of an entity	Identification, processing, data	Flow charts, PDL etc.

Table 2: Design views

Q31. Object Oriented Design

Object oriented design is the result of focusing attention not on the function performed by the program, but instead on the data that are to do manipulated by the program. Thus, it is orthogonal to function oriented design.

Object Oriented Design begins with an examination of the real world "things" that are part of the problem to be solved. These things (which we will call objects) are characterized individually in terms of their attributes and behavior.

The various terms related to object design are:

i. Objects

The word "Object" is used very frequently and conveys different meaning in different circumstances. Here, meaning is an entity able to save a state (information) and which offers a number of operations (behavior) to either examine or affect this state. An object is characterized by number of operations and a state which remembers the effect of these operations.

ii. Messages

Objects communicate by message passing. Messages consist of the identity of the target object, the name of the requested operation and any other operation needed to perform the function. Message are often implemented as procedure or function calls.

iii. Abstraction

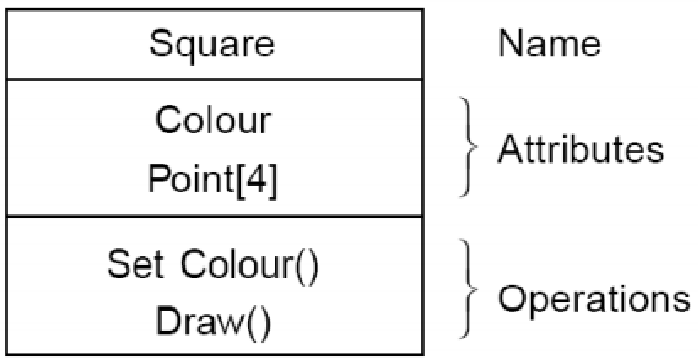
In object oriented design, complexity is managed using abstraction. Abstraction is the elimination of the irrelevant and the amplification of the essentials.

iv. Class

In any system, there shall be number of objects. Some of the objects may have common characteristics and we can group the objects according to these characteristics. This type of grouping is known as a class. Hence, a class is a set of objects that share a common structure and a common behavior.

We may define a class "car" and each object that represent a car becomes an instance of this class. In this class "car", Indica, Santro, Maruti, Indigo are instances of this class as shown in fig. 20.

Class Square



v. Attributes

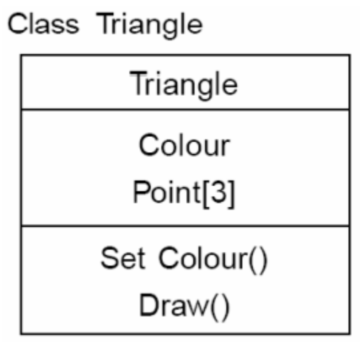
An attributes is a data value held by the objects in a class. The square class has two attributes: a colour and array of points. Each attributes has a value for each object instance. The attributes are shown as second part of the class as shown in fig. 21.

vi. Operations

An operation is a function or transformation that may be applied to or by objects in a class. In the square class, we have two operations: set colour() and draw(). All objects in a class share the same operations. An object "knows" its class, and hence the right implementation of the operation. Operation are shown in the third part of the class as indicated in fig. 21.

vii. Inheritance

Imagine that, as well as squares, we have triangle class. Fig. 22 shows the class for a triangle.



viii. Polymorphism

When we abstract just the interface of an operation and leave the implementation to subclasses it is called a polymorphic operation and process is called polymorphism.

ix. Encapsulation (Information Hiding)

Encapsulation is also commonly referred to as “Information Hiding”. It consists of the separation of the external aspects of an object from the internal implementation details of the object.

x. Hierarchy

Hierarchy involves organizing something according to some particular order or rank. It is another mechanism for reducing the complexity of software by being able to treat and express sub-types in a generic way.

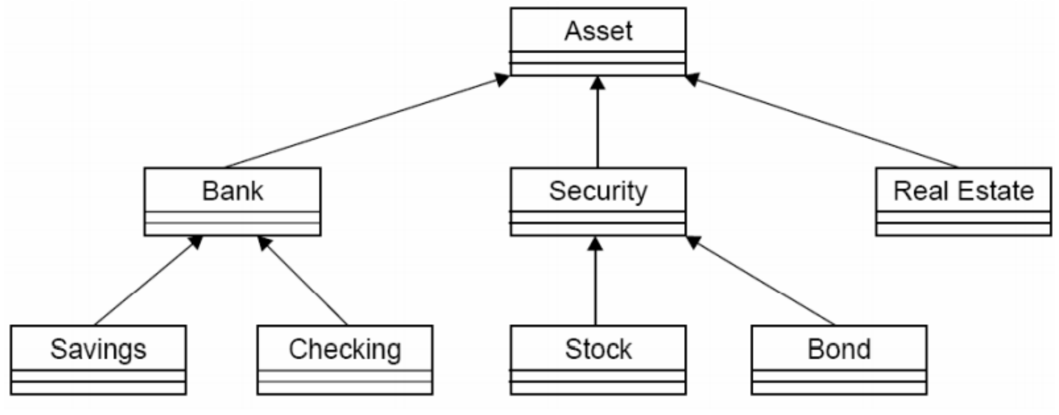


Fig. 24: Hierarchy

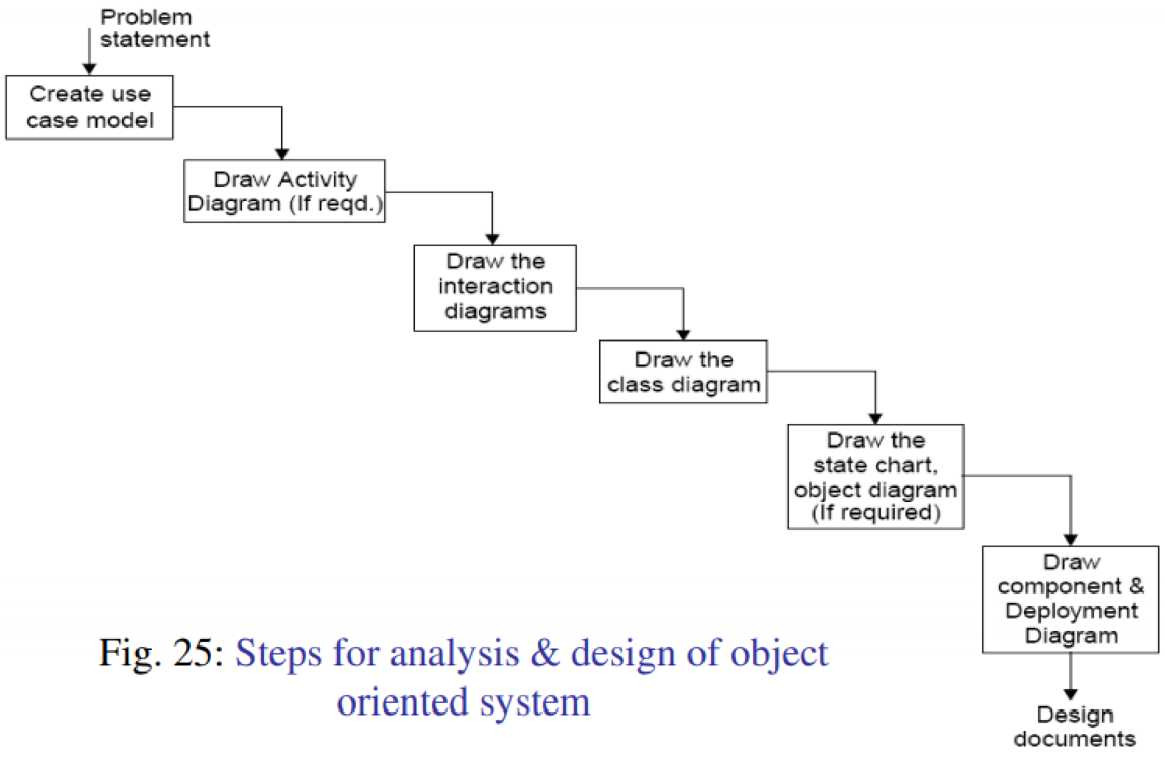

Problem statement
Create use case model
Draw Activity Diagram (If reqd.)
Draw the interaction diagrams
Draw the class diagram
Draw the state chart, object diagram (If required)
Draw component & Deployment Diagram
Design documents

Fig. 25: Steps for analysis & design of object oriented system

Q32. User Interface Design

The user interface is the **front-end application** view to which the **user interacts** to use the software. The software becomes more popular if its user interface is:

1. **Attractive**
2. **Simple to use**
3. **Responsive in a short time**
4. **Clear to understand**
5. **Consistent on all interface screens**

Types of User Interface

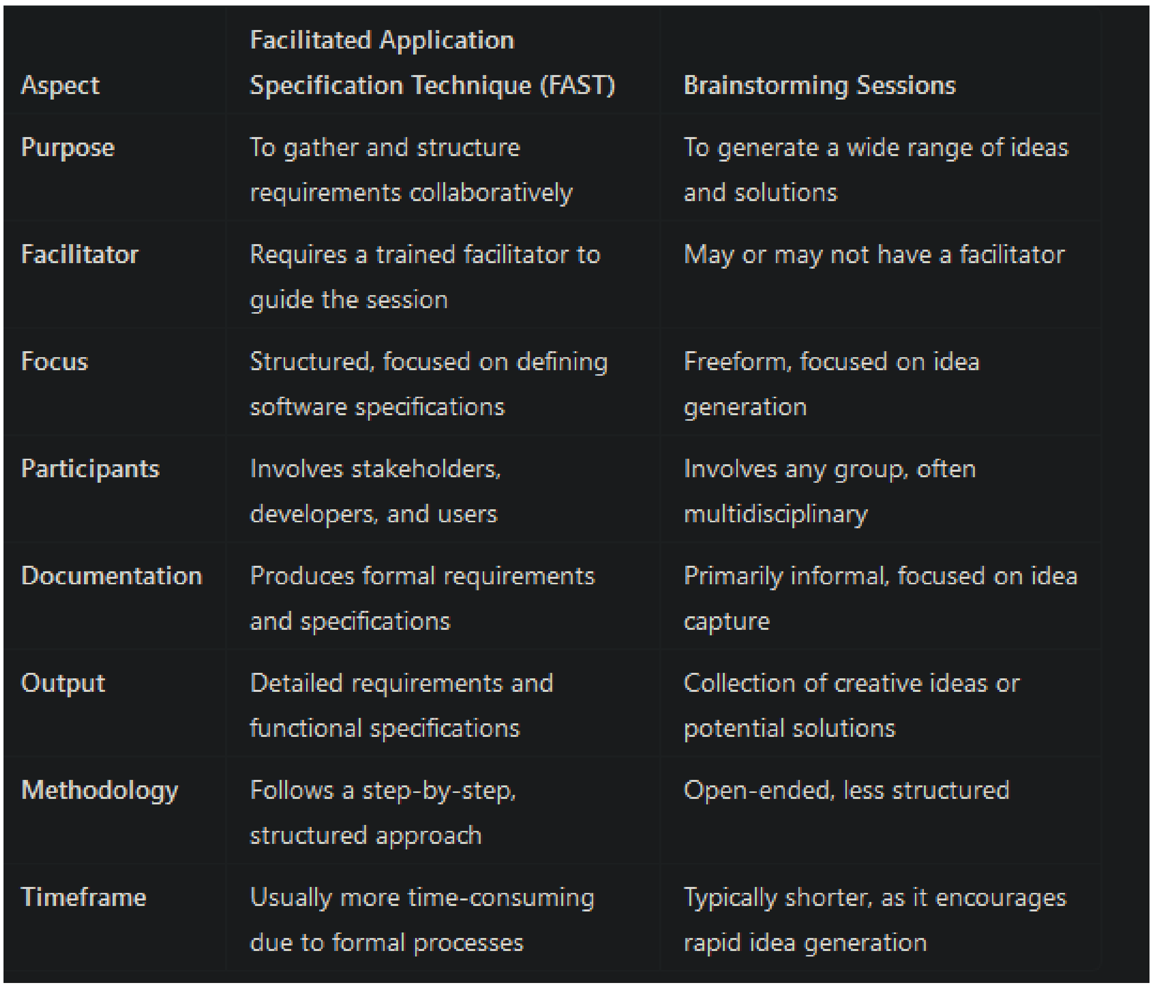
1. **Command Line Interface:** The Command Line Interface provides a command prompt, where the user types the command and feeds it to the system. The user needs to remember the syntax of the command and its use.
2. **Graphical User Interface:** Graphical User Interface provides a simple interactive interface to interact with the system. GUI can be a combination of both hardware and software. Using GUI, the user interprets the software.

PYQ:

Q1. Discuss the need and importance of SRS.

The Software Requirements Specification (SRS) is essential because it clearly defines the software’s functionality, constraints, and interfaces, ensuring alignment between stakeholders and developers. It helps prevent misunderstandings, serves as a reference throughout development, aids in testing, and improves project management and quality control.

Q4 Compare Facilitated application specification technique with brainstorming sessions. (Tabular)



Aspect	Facilitated Application Specification Technique (FAST)	Brainstorming Sessions
Purpose	To gather and structure requirements collaboratively	To generate a wide range of ideas and solutions
Facilitator	Requires a trained facilitator to guide the session	May or may not have a facilitator
Focus	Structured, focused on defining software specifications	Freeform, focused on idea generation
Participants	Involves stakeholders, developers, and users	Involves any group, often multidisciplinary
Documentation	Produces formal requirements and specifications	Primarily informal, focused on idea capture
Output	Detailed requirements and functional specifications	Collection of creative ideas or potential solutions
Methodology	Follows a step-by-step, structured approach	Open-ended, less structured
Timeframe	Usually more time-consuming due to formal processes	Typically shorter, as it encourages rapid idea generation

Q5. What are size metrics? How FP mertic is advantageous over LOC metric?

Size Metrics: These are measures used to estimate or assess the size of a software system. Common size metrics include Lines of Code (LOC) and Function Points (FP). They help in project estimation, cost analysis, and productivity assessment.

FP vs. LOC:

- **FP Metric**: Measures software size based on functionality, considering inputs, outputs, user interactions, and files.
- **Advantages over LOC**:

- Language-independent, making it suitable for comparing systems in different languages.
- Focuses on user requirements, not just code length.
- More accurate for effort estimation in early stages of developmen

Q6. Data Structure Matrices:
Data structure matrices are used to represent relationships between data entities in a structured format, typically in a tabular form. They help visualize connections and dependencies between data components, aiding in understanding data flow and organization within a system.

Q7. Software Prototyping:
Software prototyping is the process of creating a preliminary version (prototype) of a software application to visualize and validate requirements. Prototypes can be low-fidelity (paper sketches) or high-fidelity (interactive models). This approach helps gather user feedback, refine requirements, and reduce risks before full-scale development.

Q8. Time vs. Cost Trade-off in Putnam Resource Allocation Model:
The Putnam model suggests that there is a trade-off between time and cost in software development. If a project is expedited (shortened time), it typically requires more resources (higher costs) to maintain quality and meet deadlines. Conversely, extending the timeline can reduce costs but may increase project risks and complexity. The optimal allocation seeks to balance resource usage and project deadlines for efficiency.

Q9. Write the name phase of software cycle for which IEEE recommended standard IEEE 830-1993 is used.

The IEEE recommended standard IEEE 830-1993 is used in the **Requirements Specification** phase of the software development lifecycle. It provides guidelines for writing software requirements specifications (SRS), ensuring clarity, consistency, and completeness in documenting requirements.

Q10. Write the full form of FAST and QFD techniques.

- **FAST**: **Facilitated Application Specification Technique**
- **QFD**: **Quality Function Deployment**

Q11. Explain the Walston-Felix model.

The **Walston-Felix model** is a software metrics model designed to predict software development effort based on the size of the software and its complexity. It provides a mathematical approach to estimate the resources required for software projects. The model is particularly focused on the relationship between lines of code (LOC) and effort in software development.

Key Aspects of the Walston-Felix Model:

1. **Effort Estimation**: The model estimates the effort required (usually in person-hours) to develop a software project based on its size measured in LOC.

2. **Mathematical Formula**: The basic formula used in the Walston-Felix model is:

$$E = a \times (LOC)^b$$

   - **E**: Estimated effort (in person-hours)
   - **a**: A constant that represents the productivity of the development team
   - **LOC**: The size of the software in lines of code
   - **b**: An exponent that represents the non-linear relationship between size and effort (typically greater than 1).

Q12. What is Risk Exposure? What technique is used to control each risk?

Risk Exposure refers to the potential impact or consequences of a risk on a project, typically quantified as the product of the probability of the risk occurring and the potential loss or damage it could cause. It helps prioritize risks based on their significance to the project's success, enabling project managers to allocate resources effectively for risk management.

The formula for Risk Exposure is:

$$\text{Risk Exposure} = \text{Probability of Risk} \times \text{Impact of Risk}$$

Techniques to Control Risks:

1. **Avoidance**: Altering the project plan to eliminate the risk or protect the project objectives from its impact. For example, choosing a different technology that poses fewer risks.

2. **Mitigation**: Implementing actions to reduce the likelihood or impact of the risk. This could include additional testing, improved training, or enhancing system security.

3. **Transfer**: Shifting the risk to a third party, such as through insurance or outsourcing. This technique is often used to manage financial risks.

4. **Acceptance**: Acknowledging the risk and deciding to proceed without specific action, typically when the risk is minor or the cost of mitigation exceeds the potential impact. This may involve creating contingency plans to manage the consequences if the risk occurs.

5. **Contingency Planning**: Developing backup plans to address risks that are accepted or unavoidable. This ensures that if the risk occurs, there are predefined steps to minimize its impact.