SOFTWARE ENGINEERING

UNIT – 3

1) What is project scheduling? Explain in brief about the basic principles guiding the s/w project scheduling.

Project scheduling involves separating the total work involved in a project into separate activities and judging the time required by these activities.

Scheduling in software engineering can be considered from two views. First, end date for release has been established for a project and organization is responsible for distributing effort within prescribed time frame. Secondly, rough chronological wounds are discussed and end date is set by organisation.

Thus, project scheduling is an activity that distributes estimated effort across the planned project duration by specific allocation of effort to the specific task in software engineering.

✳ Basic Principles (Simplified):

1. **Compartmentalization**
   - ➤ Break the project into small, manageable tasks and actions.
2. **Interdependency**
   - ➤ Understand how tasks are connected and depend on each other.
3. **Time Allocation**
   - ➤ Give each task a specific amount of time (like 3 days, 1 week, etc.).
4. **Effort Validation**
   - ➤ Make sure you're not assigning more people than available. Check that workloads are realistic.
5. **Defined Responsibilities**
   - ➤ Assign each task to a specific person or team.
6. **Defined Outcomes**
   - ➤ Know what result is expected from each task (a file, document, code, etc.).
7. **Defined Milestones**
   - ➤ Set checkpoints to track progress (e.g., "design complete by day 10").

2) What is the need of project Estimation? What are the steps while estimation of software?

Project estimation is **essential in software engineering** for the following reasons:

🔹 a) Effort and Cost Planning

- Helps estimate how much effort (in person-hours) and cost (budget) is needed.

🔹 b) Time Management

- Determines how long the project will take to complete.

🔹 c) Resource Allocation

- Helps decide how many people, tools, and systems will be needed.

🔹 d) Feasibility Check

- Confirms whether the project is possible within given time, budget, and manpower.

🔹 e) Improves Decision Making

- Helps managers plan and control the project effectively.

🔹 f) Client Communication

- Provides the client with a reliable timeline and cost estimate.

✅ Steps While Estimating Software:

1. **Understand Software Scope** – Clearly define the system's functions, features, and boundaries.
2. **Select Estimation Technique** – Choose an appropriate method like LOC, FP, Use Case, Object Points, or COCOMO.
3. **Decompose the Project** – Break the system into modules, tasks, and subtasks.
4. **Estimate Size** – Measure software size using LOC, FP, or object points.
5. **Estimate Effort** – Calculate effort required using productivity metrics (e.g., hours per FP).
6. **Estimate Time (Duration)** – Estimate time based on effort and number of developers.
7. **Estimate Cost** – Multiply total effort by cost per person-hour.
8. **Review and Refine Estimates** – Use historical data or empirical models to validate and adjust estimates.

3) How are LOC and FP used during Project Estimation? Explain any one with suitable example.

✅ 1. LOC (Lines of Code) Based Estimation:

- LOC refers to the number of lines of source code in a software program.
- It is one of the oldest and simplest techniques used to estimate the **size, effort, and cost** of a project.
- The total number of LOC is predicted early and then used to calculate:
  - Effort = LOC / Productivity (LOC per person-day)
  - Cost = Effort × Cost per person-day
  - Time = Effort / No. of persons

✅ 2. FP (Function Point) Based Estimation:

- FP (Function Point) measures **software functionality** from the user's perspective.
- It is **independent of programming language**, unlike LOC.
- FP is calculated based on:
  - External Inputs (EI)
  - External Outputs (EO)
  - External Inquiries (EQ)
  - Internal Logical Files (ILF)
  - External Interface Files (EIF)

✅ Why LOC and FP are used:

Aspect	LOC	FP
Focus	Code Size	User Functionality
When used	After design or during coding	During requirement analysis
Language Dependent	Yes	No
Suitable for	Small projects	Large, complex systems

✅ Example of FP-Based Estimation:

Let's say a system has:

- 10 External Inputs (average weight = 4) → 10 × 4 = 40
- 5 External Outputs (simple weight = 4) → 5 × 4 = 20
- 3 External Inquiries (complex weight = 6) → 3 × 6 = 18
- 6 Internal Logical Files (average weight = 10) → 6 × 10 = 60
- 4 External Interface Files (simple weight = 5) → 4 × 5 = 20

➡️ **Total UFP (Unadjusted Function Points)** = 40 + 20 + 18 + 60 + 20 = **158**

Assume **CAF (Complexity Adjustment Factor)** = 30

Then,

$$\text{FP} = \text{UFP} \times (0.65 + 0.01 \times CAF) = 158 \times (0.65 + 0.30) = 158 \times 0.95 = \boxed{150.1}$$

Consider an ABC project with some important modules such as

1. User interface and control facilities
2. 2D graphics analysis
3. 3D graphics analysis
4. Database management
5. Computer graphics display facility
6. Peripheral control function
7. Design analysis models

Estimate the project in based on LOC

For estimating the given application we consider each module as separate function and corresponding lines of code can be estimated in the following table as

Function	Estimated LOC
User Interface and Control Facilities(UICF)	2500
2D graphics analysis(2DGA)	5600
3D Geometric Analysis function(3DGA)	6450
Database Management(DBM)	3100
Computer Graphics Display Facility(CGDF)	4740
Peripheral Control Function(PCF)	2250
Design Analysis Modules (DAM)	7980
Total Estimation in LOC	**32620**

- Expected LOC for 3D Geometric analysis function based on three point estimation is -
  - Optimistic estimation 4700
  - Most likely estimation 6000
  - Pessimistic estimation 10000

$$S = [S_{opt} + (4 * S_m) + S_{press}] / 6$$

$$\text{Expected value} = [4700 + (4 * 6000) + 10000] / 6 \rightarrow 6450$$

- A review of historical data indicates -
  1. Average productivity is 500 LOC per month
  2. Average labor cost is $6000 per month

  Then cost for lines of code can be estimated as

  cost / LOC = (6000 / 500) = $12

  By considering total estimated LOC as 32620
- Total estimated project cost = (32620 * 12) = $391440
- Total estimated project effort = (32620 / 500) = 65 Person-months

✅ Conclusion:

- LOC is suitable for **code-based estimation**, especially in later phases.
- FP is useful **early in the project** and preferred in industry for its **accuracy and flexibility**.
- FP is more reliable for modern, user-centric, multi-language software systems.

4) What is the difference between COCOMO and COCOMO II Model?

Point	COCOMO Model	COCOMO II Model
Full Form	Constructive Cost Model	Constructive Cost Model II
Introduced by	Barry Boehm in 1981	Barry Boehm & team in 1995 (updated version)
Type of Projects	Traditional software projects	Modern software projects (object-oriented, web-based, etc.)
Estimation Based On	Lines of Code (LOC)	Object Points, Function Points, and Reuse Factors
Project Types	Organic, Semi-Detached, Embedded	Application Composition, Early Design, Post-Architecture
Phases Covered	Mainly focuses on full project effort	Allows estimation at various stages of development
Reuse Handling	Not well-supported	Includes reuse, reengineering, and software maintenance
Accuracy & Flexibility	Less accurate for modern projects	More accurate and flexible with real-time projects
Scalability	Less scalable	Highly scalable for small to large systems

5) What is the necessity of Estimation? How estimation with Use-cases is performed?

[NECESSITY REFRE QUESTION 2]

🔷 Steps to Perform Use-Case Based Estimation:

1. **Identify Use Cases:**
   - List all use cases from the requirement model.
2. **Classify Each Use Case:**
   - **Simple Use Case** (≤ 3 transactions): Weight = 5
   - **Average Use Case** (4–7 transactions): Weight = 10
   - **Complex Use Case** (≥ 7 transactions): Weight = 15
3. **Count the Number of Actors:**
   - **Simple Actor** (interacts through API): Weight = 1
   - **Average Actor** (interacts through protocol): Weight = 2
   - **Complex Actor** (interacts via GUI): Weight = 3
4. **Calculate Use Case Points (UCP):**

$$\text{UCP} = \text{Unadjusted Use Case Weight} + \text{Unadjusted Actor Weight}$$

5. **Apply Technical and Environmental Factors (Optional):**
   - Adjust UCP based on complexity or team experience.
6. **Convert UCP to Effort:**
   - Multiply UCP by a standard productivity factor (e.g., 20 person-hours/UCP).

✅ Example (Simplified):

- 2 Simple Use Cases = 2 × 5 = 10
- 3 Average Use Cases = 3 × 10 = 30
- 1 Complex Use Case = 1 × 15 = 15
  → **Total Use Case Weight = 55**
- 2 Average Actors = 2 × 2 = 4
  → **Total Actor Weight = 4**

➡️ **UCP = 55 + 4 = 59**

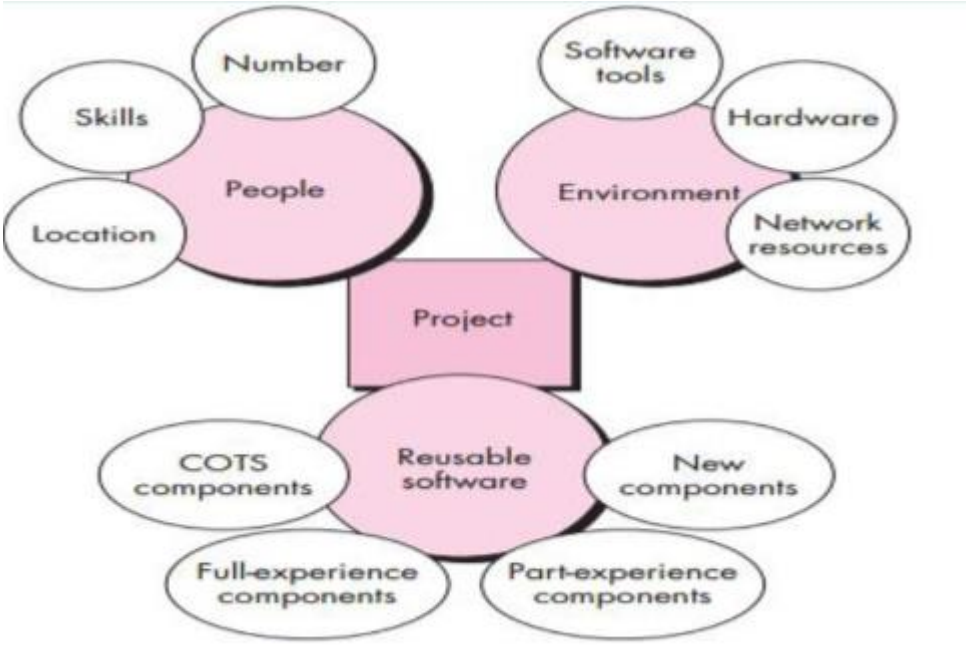
If productivity = 20 hours per UCP:

Effort = 59 × 20 = 1180 person-hours

6) What is the need for defining a software scope? What are the categories of software engineering resources (Project Resources)?

Purpose / Need:

1. **Sets Boundaries** – Defines what is to be included and excluded from the project.
2. **Avoids Scope Creep** – Prevents uncontrolled feature additions later.
3. **Improves Understanding** – Helps developers and stakeholders understand system goals.
4. **Aids Estimation** – Provides the basis for effort, time, and cost estimation.
5. **Defines Interfaces** – Helps identify system interfaces and external interactions.
6. **Supports Feasibility Study** – Confirms whether the desired software can be built within the available time, cost, and resources.



Human Resource

- Planners need to select the number and the kind of people skills needed to complete the project
- They need to specify the organizational position and job specialty for each person
- Small projects of a few person-months may only need one individual
- Large projects spanning many person-months or years require the location of the person to be specified also
- The number of people required can be determined only after an estimate of the development effort

Development Environment Resources

- A software engineering environment (SEE) incorporates hardware, software, and network resources that provide platforms and tools to develop and test software work products
- Most software organizations have many projects that require access to the SEE provided by the organization
- Planners must identify the time window required for hardware and software and verify that these resources will be available

Reusable Software Resources

- Off-the-shelf components
  - Components are from a third party or were developed for a previous project
  - Ready to use; fully validated and documented; virtually no risk
- Full-experience components
  - Components are similar to the software that needs to be built
  - Software team has full experience in the application area of these components
  - Modification of components will incur relatively low risk
- Partial-experience components
  - Components are related somehow to the software that needs to be built but will require substantial modification
  - Software team has only limited experience in the application area of these components
  - Modifications that are required have a fair degree of risk
- New components
  - Components must be built from scratch by the software team specifically for the needs of the current project
  - Software team has no practical experience in the application area
  - Software development of components has a high degree of risk

7) What are the basic principles of software project scheduling. Explain different tasks of project scheduling.
[FOR PRINCIPLES REFER QUESTION 1]

1. Defining Task Set

- List out all the major activities and sub-tasks.
- Tasks should be:
  - Specific
  - Measurable
  - Time-bound

2. Task Sequencing (Task Network)

- Arrange tasks in the order in which they must be performed.
- Identify task **dependencies** (e.g., Task B can start only after Task A).
- Create a **Task Network Diagram** (also called Activity Network) showing the flow of tasks.

3. Time Allocation

- Assign **start and end dates** to each task.
- Use estimation techniques (LOC, FP, Use Case, etc.) to predict task durations.
- Consider **available manpower and productivity**.

🔹 4. Milestone Definition

- Define important **checkpoints** or **milestones** in the project.
- Example: "Design Completed", "Testing Started"

🔹 5. Assigning Resources

- Allocate **human, software, and hardware resources** for each task.
- Avoid overloading a resource with too many tasks at once.

🔹 6. Review and Adjustment

- Continuously monitor progress and adjust the schedule if delays or changes occur.
- Use tools like **Gantt Charts** or **PERT Charts** for tracking.

8) Discuss Empirical Estimation Models. Explain Constructive Cost Model for project estimation with suitable example.

- Empirical estimation models use **real project data and formulas** to estimate effort.
- They usually rely on **LOC (Lines of Code)** or **FP (Function Points)** as inputs.
- The actual values are taken from **tables or past projects**.
- These models are based on **samples of previous software projects**.
- But since no one model fits **all types of software or environments**, you must use them **carefully and wisely**.

✅ Constructive Cost Model (COCOMO):

COCOMO is an **empirical estimation model** developed by **Barry Boehm in 1981**. It is used to **estimate the effort (person-months)** required to develop a software project based on the size of the software.

🔷 Basic COCOMO Model Formula:

$$\text{Effort (E)} = a \times (\text{KLOC})^b$$

Where:

- `E` = Effort in person-months
- `KLOC` = Thousands of Lines of Code
- `a` and `b` = Constants that depend on the type of project

🔷 Types of Projects & Constants:

Project Type	Description	a	b
Organic	Small, simple software	2.4	1.05
Semi-Detached	Medium-sized, mixed teams	3.0	1.12
Embedded	Complex software, real-time constraints	3.6	1.20

✅ Example:

A software project is expected to be **25 KLOC** and falls under the **Organic** category.

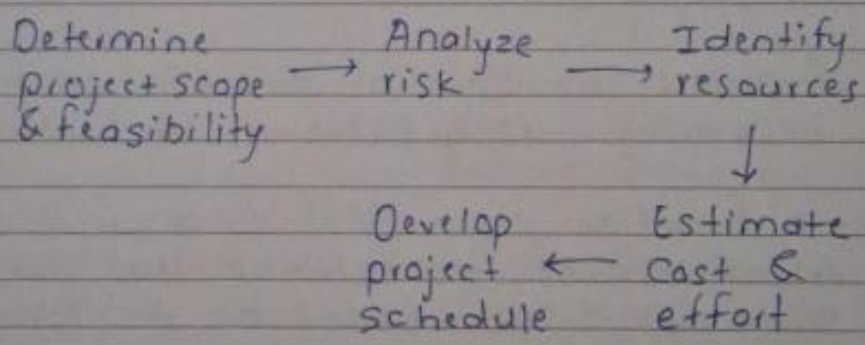
$$\text{Effort} = 2.4 \times (25)^{1.05}$$

$$\text{Effort} \approx 2.4 \times 27.02 \approx \boxed{64.85 \text{ person-months}}$$

➡️ This means approximately **64.85 person-months** of effort is required to complete the project.

9) Explain the various activities during software project planning.

- project planning is an activity in which the project manager makes reasonable estimates of resources, costs & schedule
- following figure represents various activities that are associated with project planning process



i) Step 1: This is an initial stage in which the scope and feasibility of the project is determined. This stage helps to identify the functions and features that can be delivered to the end user.

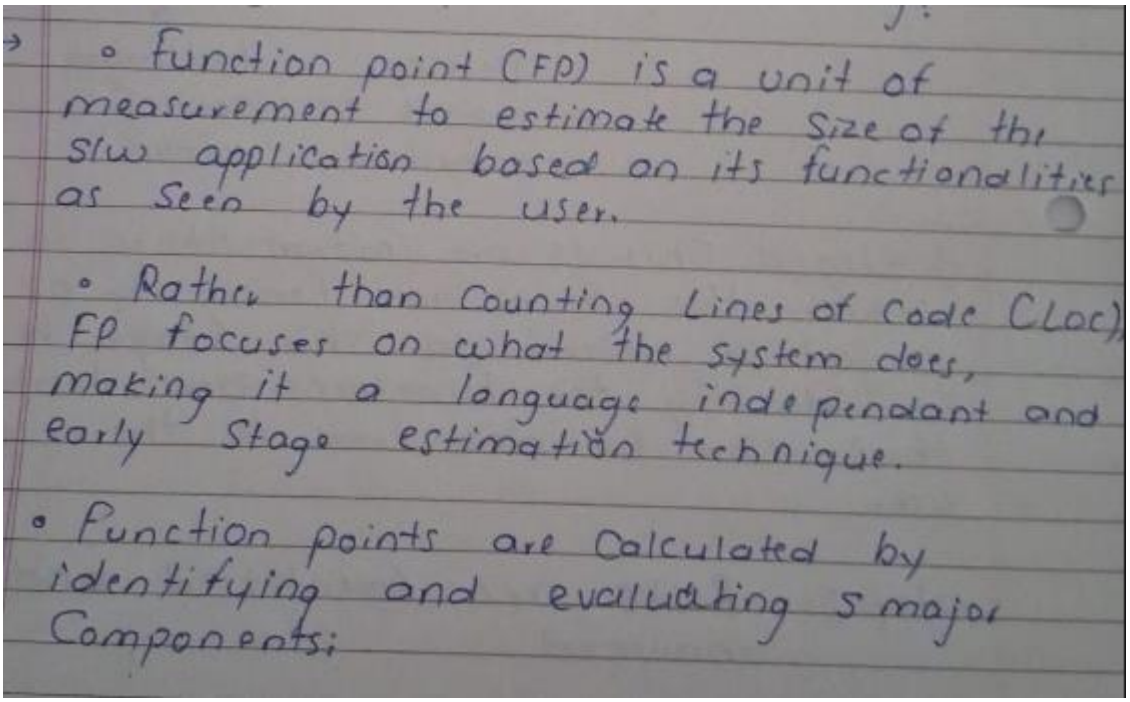
step 2: Risks are identified and analyzed.

Step 3: The required resources, such as human resources, environmental resources and reusable software resources can be determined.

Step 4: The estimation for cost and effort is made. In this stage the problem is first decomposed, then using the size, function point or use cases the estimates are made.

step 5: Using the scheduling tools or task networks the schedule for the project is prepared.

10) Explain the concept of FP. Why FPs are becoming acceptable in industry?



- Function point (FP) is a unit of measurement to estimate the size of the sfw application based on its functionalities as seen by the user.
- Rather than Counting Lines of Code (LOC), FP focuses on what the system does, making it a language independant and early stage estimation technique.
- Function points are Calculated by identifying and evaluating 5 major Components:

Component	Description
External inputs (EI)	Data received from the user.
External outputs (EO)	Data sent to user
External Inquiries (EQ)	User requests that require a response
Internal Logic files (ILF)	Logical groups of Data maintained by the system
External Interface files (EIF)	Data used by the system but maintained externally.

- Steps to Calculate function points:
1. Identify all five Components
2. Assign weights based on Complexity
3. Compute Unadjusted Function points (UFP).
4. Calculate Complexity adjustment Factor (CAF) using 14 general System Characteristics
5. Apply formula:

$$FP = UFP \times (0.65 + 0.01 \times CAF)$$

- FP is Becoming Acceptable in the indust due to following reasons:

1. Language Independant: (oc is tied to Programming language. FP works irrespecti is in C, Java, Python et

2. Early estimation: FP can be used in requirements phase, even before designing or coding starts.

3. User-Oriented: measures functionalities from the users perspective.

4. Improved accuracy: It includes inputs, outputs, files, interfaces, giving a holistic estimate of effort and cost.

5. Better productivity Analysis: Helps Compare productivity across projects by normalizing for size and complexity

- Example:

If a System has:

10 (EI) : (average) : weight = 4 x 10 = 40
5 (EO) : (simple) : weight = 4 x 5 = 20
3 (EQ) : (complex) : weight = 6 x 3 = 18
6 (ILFs) : (average) : weight = 10 x 6 = 60
4 (ELFs) : (simple) : weight = 5 x 4 = 20
Total UFP = 40+20+18+60+20 = 158
Assuming CAF = 30

$$FP = 158 \times (0.65 + 0.01 \times 30)$$
$$= 158 \times 0.95$$
$$= 150.1$$

11) Is it possible to estimate software size before coding? Justify your answer with suitable examples.

Yes, it is possible to estimate software size before coding, but it comes with inherent uncertainties. Software size estimation before coding is typically done using a variety of techniques and tools. Here are some approaches and their justifications:

1. Function Points (FP)

Function points are a widely used metric to estimate the size of software based on its functionality, rather than lines of code (LOC). Function points measure software size in terms of the number and complexity of functions or features the software will have (e.g., user inputs, outputs, data storage, and data retrieval).

- **Example**: For an e-commerce website, you would identify the functions like "user login," "product search," "checkout process," and assign complexity points to each function. Then, using historical data from similar projects, you can estimate the total function points and translate that into a size estimate.
- **Justification**: By estimating the function points early in the project, you can get a rough estimate of the size before the actual coding begins. However, it relies on identifying functional requirements early, which might change during development.

2. Use Case Points

This method estimates the software size based on the complexity of use cases in the system. The method assigns points based on factors like actor types (user types), use case complexity, and the number of use cases.

- **Example**: In the development of a mobile banking application, you would analyze use cases such as "transfer money," "view transaction history," and "check account balance." Each use case is rated for complexity, and the total points give an estimate of size.
- **Justification**: Use case points can be useful when detailed requirements are available, but, like function points, they are still estimates that might change as the project progresses.

12) Illustrate various open-source tools, for scheduling of software activities. List the attributes that are associates with scheduling task for implementing schedule process.

1. **GanttProject**
   - **Description**: GanttProject is a project management tool that uses Gantt charts to schedule tasks. It allows for task dependencies, milestones, and resource management. It is a straightforward tool for planning, scheduling, and tracking the progress of a project.
   - **Key Features**:
     - Gantt chart for visualization of project schedules
     - Task dependencies (start-to-start, finish-to-start)
     - Resource allocation and management
     - Export to PNG, PDF, and Microsoft Project formats
   - **Use Case**: Ideal for managing medium-sized software projects where tasks need to be tracked along with dependencies.
2. **OpenProject**
   - **Description**: OpenProject is an open-source project management software that supports Agile, Scrum, and traditional project management methods. It provides comprehensive scheduling, task tracking, and team collaboration features.
   - **Key Features**:
     - Gantt charts for project planning and scheduling
     - Scrum and Agile boards for task tracking
     - Time tracking and budgeting tools
     - Support for task dependencies and milestones
   - **Use Case**: Well-suited for teams working in Agile or traditional methodologies, offering flexible scheduling options and collaboration.

When scheduling software activities, various attributes need to be considered for an effective and organized process:

1. **Task Name/Description**
   - A clear and concise title/description of the task that explains the work to be done.
2. **Start and End Dates**
   - The planned start and end dates of the task. These dates define the duration of the task and help in determining project timelines.
3. **Task Duration**
   - The amount of time allocated for completing the task. This can be in hours, days, or weeks, depending on the task's scale.
4. **Dependencies**
   - Relationships between tasks (e.g., Task B cannot start until Task A is completed). Dependencies help to understand the flow of the project and schedule tasks accordingly.
5. **Priority**
   - The level of importance or urgency of the task. Tasks can be prioritized as high, medium, or low, or according to a numerical scale (1–5).
6. **Resources Assigned**
   - The personnel or resources (hardware, software, etc.) that are assigned to complete the task. Proper resource management is essential for scheduling and task completion.
7. **Milestones**
   - Significant points or achievements within the project that help in tracking progress. For example, "First prototype delivered" could be a milestone.
8. **Time Estimates**
   - A rough estimate of how much time the task will take to complete. This helps in assessing whether the project is on schedule and if any tasks need to be adjusted.
9. **Slack Time**
   - The amount of time a task can be delayed without affecting the overall project schedule. Slack time is crucial for managing delays and unforeseen issues.
10. **Critical Path**
    - The sequence of tasks that determine the minimum project duration. If a task in the critical path is delayed, the entire project will be delayed.

13) An application has 10 low external inputs, 12 high external outputs, 20 low internal logical files, 15 heigh external interface files, 12 average external inquiries and a value of complexity adjustment factor 1.l0.What are the unadjusted and adjusted function point counts?

Component Type	Low Complexity	Average Complexity	High Complexity
External Inputs (EI)	4 FP	5 FP	7 FP
External Outputs (EO)	5 FP	7 FP	10 FP
Internal Logical Files (ILF)	7 FP	10 FP	15 FP
External Interface Files (EIF)	5 FP	7 FP	10 FP
External Inquiries (EQ)	4 FP	6 FP	8 FP

Now, let's calculate the **Unadjusted Function Points**:

1. **External Inputs (EI)**:
   - 10 low complexity → $10 \times 4 = 40$
2. **External Outputs (EO)**:
   - 12 high complexity → $12 \times 10 = 120$
3. **Internal Logical Files (ILF)**:
   - 20 low complexity → $20 \times 7 = 140$
4. **External Interface Files (EIF)**:
   - 15 high complexity → $15 \times 10 = 150$
5. **External Inquiries (EQ)**:
   - 12 average complexity → $12 \times 6 = 72$

Now, sum up the individual function points:

$$\text{UFP} = 40 + 120 + 140 + 150 + 72 = 522$$

So, the **Unadjusted Function Point Count (UFP)** is **522**.

---

2. Adjusted Function Point (AFP) Count

The **Adjusted Function Point Count** is calculated using the **Complexity Adjustment Factor (CAF)**, which is based on a set of 14 general system characteristics. For this scenario, we are given the **complexity adjustment factor (CAF) value** as **1.10**.

The formula for **Adjusted Function Points (AFP)** is:

$$\text{AFP} = \text{UFP} \times \text{CAF}$$

Now, let's apply the values:

$$\text{AFP} = 522 \times 1.10 = 574.2$$

Since function points are generally rounded to the nearest whole number, the **Adjusted Function Point Count (AFP)** is **574**.

UNIT 4

14) Explain the following design concepts : Abstraction , Modularity, Architecture , Refinement, Pattern, Information hiding, Functional independence ,Refactoring , Design classes

4.4.1 Abstraction

- The abstraction means an ability to cope up with the complexity.
- Software design occurs at different levels of abstraction. At each stage of software design process levels of abstractions should be applied to refine the software solution.
- At the **higher level** of abstraction, the solution should be stated in **broad terms** and in the **lower level** more **detailed description** of the solution is given.
- While moving through different levels of abstraction the **procedural abstraction** and **data abstraction** are created.
- The **procedural abstraction** gives the named sequence of instructions in the specific function. That means the functionality of procedure is hidden. For example : **Search the record** is a procedural abstraction in which implementation details are hidden (i.e. Enter the name, compare each name of the record against the entered one, if a match is found then declare success !! Otherwise declare 'name not found').
- In **data abstraction** the collection of data objects is represented. For example for the procedure **search** the data abstraction will be **record**. The **record** consists of various attributes such as record ID, name, address and designation.

4.4.2 Modularity

- The software is divided into separately named and addressable components that called as **modules.**
- Monolithic software is hard to grasp for the software engineer, hence it has now become a trend to divide the software into number of products. But there is a co-relation between the number of modules and overall cost of the software product.
- Following argument supports this idea -

  "Suppose there are two problems A and B with varying complexity. If the complexity of problem A is greater than the complexity of the problem B then obviously the efforts required for solving the problem A is greater than that of problem B. That also means the time required by the problem A to get solved is more than that of problem B."
- The overall complexity of two problems when they are combined is greater than the sum of complexity of the problems when considered individually. This leads to **divide and conquer strategy** (according to divide and conquer strategy the problem is divided into smaller subproblems and then the solution to these subproblems is obtained).

4.4.3 Architecture

Architecture means representation of **overall structure** of an integrated system. In architecture various components interact and the data of the structure is used by various components. These components are called system elements. Architecture provides the basic framework for the software system so that important framework activities can be conducted in systematic manner.

Refinement:

1. **Refinement** means **explaining things in more detail** step by step.
2. **Stepwise Refinement** is a **top-down design method** introduced by **Niklaus Wirth**.
3. In this method, we start with a **general idea** of the program and slowly **break it down into smaller and detailed parts**.
4. This is similar to how we break down big tasks into smaller ones during **requirement analysis**.
5. **Abstraction** and **refinement** go hand in hand:
   - **Abstraction** hides low-level details.
   - **Refinement** adds those details step by step.

✅ Design Pattern

According to **Brad Appleton**, a **design pattern** is a **named and useful idea** that describes a **proven solution** to a **common problem** that happens in a specific situation.

In simple words, a **design pattern** is like a **ready-made solution** that can be used when a similar type of problem comes up in software design.

✅ Purpose of Design Patterns:

Using design patterns, a designer can decide:

1. If the pattern can be **reused** in other projects.
2. If it is **suitable for the current task**.
3. If it can help **solve similar problems** even with different features.

4.4.6 Information Hiding

- Information hiding is one of the important property of effective modular design.
- The term information hiding means the modules are designed in such a way that information contained in one module cannot be accessible to the other module (the module which does not require this information).
- Due to information hiding only limited amount of information can be passed to other module or to any local data structure used by other module.
- The **advantage** of information hiding is basically in testing and maintenance.
- Due to information hiding some data and procedures of one module can be hidden from another module. This ultimately **avoids** introduction of **errors** module from one module to another. Similarly one can make **changes** in the desired module without affecting the other module.

4.4.7 Functional Independence

- The functional independence can be achieved by developing the functional **modules** with single-minded approach.
- By using functional independence functions may be compartmentalized and **interfaces** are **simplified**.
- Independent modules are easier to maintain with **reduced error propagation**.
- Functional independence is a key to good design and design is the key to software quality.
- The **major benefit** of functional independence is in achieving effective modularity.
- The functional independence is assessed using two qualitative criteria - **Cohesion** and **coupling.**

✅ Cohesion:

- **Cohesion** means how **closely related** the tasks inside a single module are.
- A **highly cohesive** module performs **one specific task** with **minimum interaction** with other modules.
- Cohesion also supports **information hiding**.

🔹 Types of Cohesion:

1. **Coincidental Cohesion**
   ➤ Tasks are loosely related and grouped randomly.
   (Not a good practice)
2. **Logical Cohesion**
   ➤ Tasks are **logically related**, like similar types of operations.
3. **Temporal Cohesion**
   ➤ Tasks are executed together in a **specific time span**, e.g., initialization.
4. **Procedural Cohesion**
   ➤ Tasks must be done in a **specific sequence**.
5. **Communicational Cohesion**
   ➤ Tasks **share the same data** or operate on the same input/output.

✅ **Goal:** Design modules with **high cohesion**.

✅ Coupling:

- **Coupling** means how **strongly one module is connected to another**.
- It is a measure of **interdependence between modules**.
- Lower coupling = better design.

🔹 Types of Coupling:

1. **Data Coupling**
   ➤ Modules share data through **parameters or arguments**.
2. **Control Coupling**
   ➤ Modules share **control information**, like flags or control variables.
3. **Common Coupling**
   ➤ Modules share **global data**.
4. **Content Coupling**
   ➤ One module **directly accesses** or modifies the contents of another.
   (Worst type of coupling)

✅ **Goal**: Achieve **low coupling** to reduce side effects, errors, and maintenance cost.

4.4.8 Refactoring

Refactoring is necessary for simplifying the design without changing the function or behaviour. **Fowler** has defined refactoring as "The process of changing a software system in such a way that the external behavior of the design do not get changed, however the internal structure gets improved".

Benefits of refactoring are -

- The **redundancy** can be achieved.
- **Inefficient algorithms** can be eliminated or can be replaced by efficient one.
- Poorly constructed or **inaccurate data structures** can be removed or replaced.
- Other **design failures** can be rectified.

The decision of refactoring particular component is taken by the designer of the software system.

15) What is the importance of software design? What are types of design classes?

Software design is the **foundation** of all software engineering activities. It provides a **blueprint** for constructing a system and bridges the gap between requirements and implementation.

Importance:

1. **Defines System Structure**
   - Helps in identifying modules, data structures, and control flow.
2. **Improves Quality**
   - Good design ensures better **reliability, maintainability, and performance**.
3. **Facilitates Communication**
   - Design documents act as a common reference for developers, testers, and stakeholders.
4. **Aids in Planning and Cost Estimation**
   - Design helps in **estimating time, resources**, and effort needed.
5. **Reduces Complexity**
   - Breaks system into manageable components using **abstraction and modularity**.
6. **Foundation for Implementation**
   - Acts as a guide for coding and testing phases.

2. Types of Design Classes

(📄 *Page 4-16*)

Design classes are used in **object-oriented design** and represent different roles during software construction.

Types of Design Classes:

1. **User Interface Classes**
   - Handle all user interactions.
   - Example: GUI screens, input forms.
2. **Business Domain Classes**
   - Represent real-world business entities.
   - Example: `Customer`, `Invoice`, `Account`.
3. **Process Classes**
   - Handle business logic and control flow.
   - Act between UI and business classes.
4. **Persistent Classes**
   - Manage database access and storage.
   - Example: Classes for CRUD operations on data.
5. **System Classes**
   - Interface with the operating system or external systems.
   - Example: File handling, network communication.

16) Explain in detail the Architectural design and Component level design elements.

✅ 1. Architectural Design Elements

📄 *Reference: Page 4-19*

Architectural design is the **blueprint** of the software system. It defines the **overall structure**, major components, and their **interactions**.

Elements of Architectural Design:

1. **Software Components**
   - Represent functional units/modules (e.g., login system, payment gateway).
2. **Relationships**
   - Define interactions (data flow, control flow) among components.
3. **Architecture Styles/Patterns**
   - Examples: Layered architecture, client-server, MVC, microservices.
4. **Architectural Decisions**
   - Design choices made regarding structure, technologies, protocols, etc.
5. **System Constraints**
   - Considerations like performance, scalability, and security.

✅ 2. Component Level Design Elements

📄 *Reference: Pages 4-19 to 4-20*

Component level design focuses on the **internal logic** of each component or module. It provides **detailed design** to guide developers during implementation.

Elements of Component Level Design:

1. **Functional Elements**
   - Identify **functions/methods** the component must perform.
   - Example: Validation logic in a "Form Handler" component.
2. **Interface Elements**
   - Define **inputs, outputs, and interaction** with other components.
   - Example: APIs, method signatures.
3. **Data Elements**
   - Include **internal data structures** and variables used.
   - Example: Arrays, lists, objects used for processing.
4. **Behavioral Elements**
   - Define the **control flow or logic** inside the component.
   - Represented using flowcharts, decision tables, or pseudocode.
5. **Reusability and Modularity**
   - Components are designed to be **independent**, **reusable**, and **easy to test**.

17) What is software Architecture? What is architectural context diagram?

Q.8 What is software Architecture? What is architectural context diagram.

→
- The software architecture is the structure of program modules where they interact with each other in a specialized way.
- The system is secured against malicious users by encryption or any other security measures due to layered software architecture.
- It handle request and response of the page in minimum time.
- Architectural design process uses ~~early~~ easily modifiable & replaceable components which is easy to change.
- Avoid critical functionalities in small components & improve communication of the system.
- Architectural design process includes corresponding components, ~~func~~ functionalities for handling the occurrence of any type of errors.
- An architectural context diagram shows how the software system fits into its external environment.
- It highlights the system's interactions with external entities like users, other systems, databases or devices.

- The generic structure of the architectural context diagram is illustrated in Fig. 7.6.1.
- Referring to the Fig. 7.6.1, systems that interoperate with the target system (the system for which an architectural design is to be developed) are represented as :
  - Superordinate systems, these systems that use the target system as part of some higher level processing scheme.

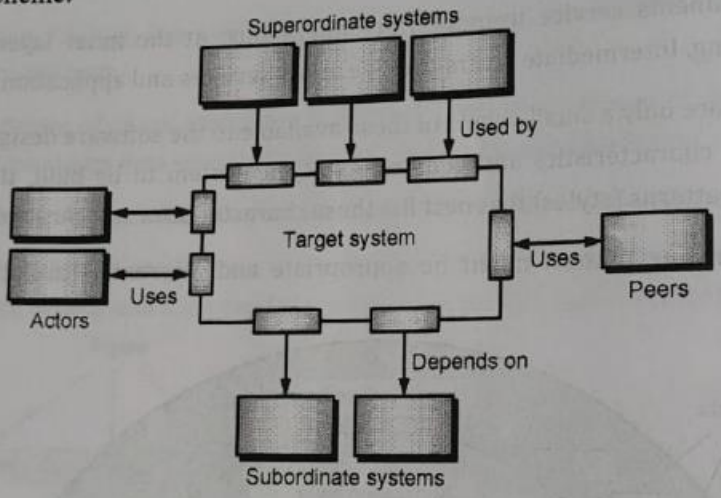


Fig. 7.6.1 : Architectural context diagram

  - Subordinate systems, these systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
  - Peer-level systems, these systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system).
  - Actors, these entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

Each of these external entities

18) Write short note on 'Interface analysis and design models

✅ 1. Interface Analysis

Interface analysis is the process of identifying and defining the **interactions between software components**, and between the software and **external entities** (like users or hardware).

Key Points:

- Focuses on **how modules communicate** with each other.
- Helps in identifying **inputs, outputs**, and **data formats** exchanged between components.
- Clarifies the **control and data flow** between external systems and the software.
- Ensures that **user interface**, **hardware interface**, and **software interface** are clearly understood and specified.

✅ 2. Interface Design Models

Interface design models are used to **specify and represent** how the system's interfaces will appear and function.

Types of Interface Design Models:

1. **User Interface Design**
   - Defines how the system will interact with users (e.g., screens, buttons, menus).
   - Focuses on **usability, consistency, and accessibility**.
2. **Component Interface Design**
   - Specifies the interaction between different software modules or components.
   - Includes **function signatures, input/output formats, and error messages**.
3. **External Interface Design**
   - Defines interaction with **external devices or systems** (e.g., sensors, payment gateways).
   - Ensures compatibility with external communication protocols.

✅ Conclusion

Interface analysis and design are crucial for ensuring that all software components and external systems can **communicate effectively**. Proper interface modeling leads to better **integration, testing, and user experience**.

19) With the help of diagram explain how to translate the requirements model into the design model.

✅ 1. Introduction

Translating the **requirements model** into the **design model** is an important step in software engineering. It involves converting user needs and system requirements into a blueprint for system construction.

- **Requirements model** focuses on *what* the system should do.
- **Design model** focuses on *how* the system will do it.

✅ 2. Elements in Translation Process

Requirements Model	→	Design Model
Use Case Model	→	Analysis Classes
Scenario-Based Elements	→	Sequence and Collaboration Diagrams
Data Flow Diagrams, ER Models	→	Data Design Elements
Behavioral Models (State Charts)	→	Behavioral Design Models

20) With the help of diagram explain how to translate the requirements model into the design model.

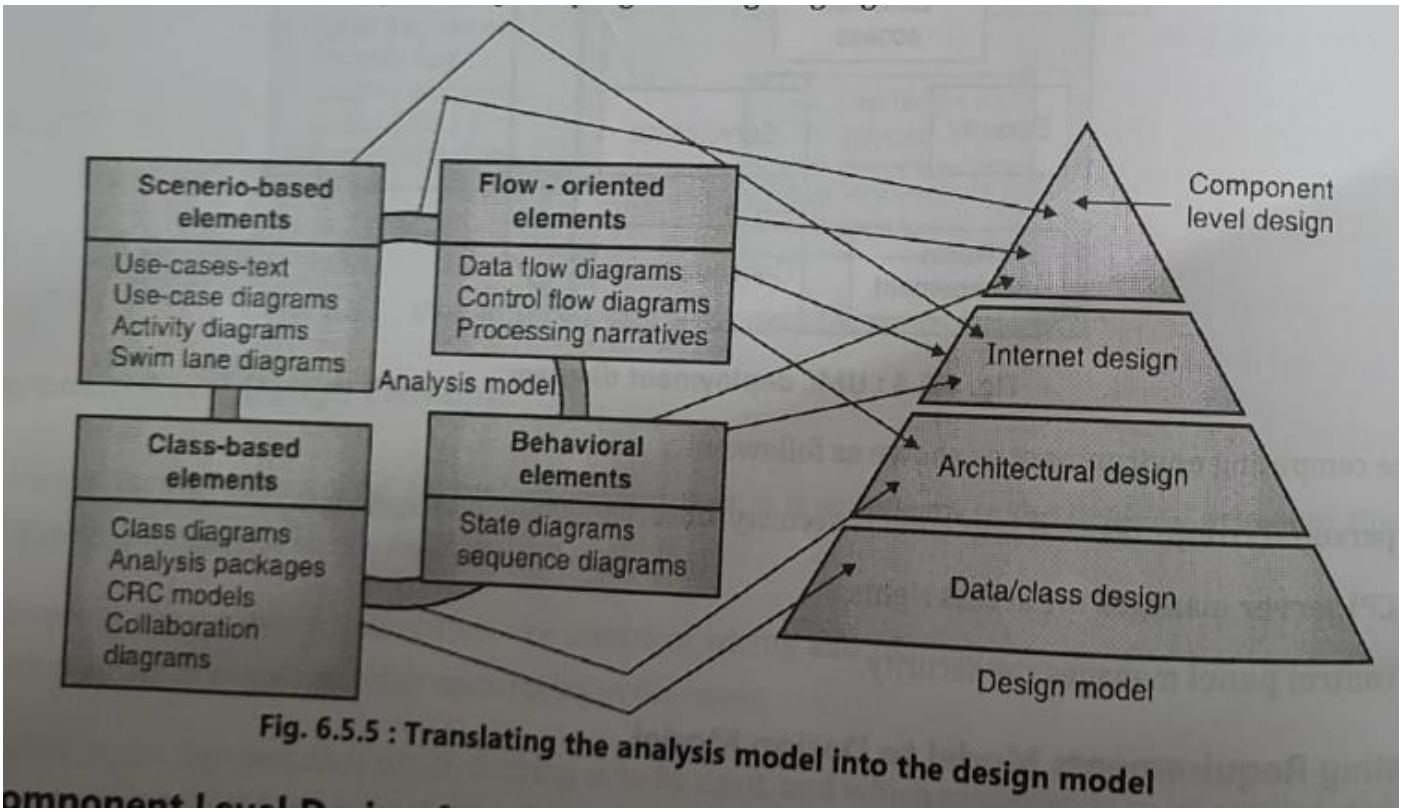


Fig. 6.5.5 : Translating the analysis model into the design model

✅ 1. Introduction

Translating the **requirements model** into the **design model** is an important step in software engineering. It involves converting user needs and system requirements into a blueprint for system construction.

- **Requirements model** focuses on *what* the system should do.
- **Design model** focuses on *how* the system will do it.

1. Analysis Model Elements:

- **Scenario-Based:**
  Use-case diagrams, use-case texts, activity diagrams.
- **Flow-Oriented:**
  Data Flow Diagrams (DFD), control flow, processing narratives.
- **Class-Based:**
  Class diagrams, CRC models, collaboration diagrams.
- **Behavioral Elements:**
  State diagrams, sequence diagrams.

---

2. Design Model Layers:

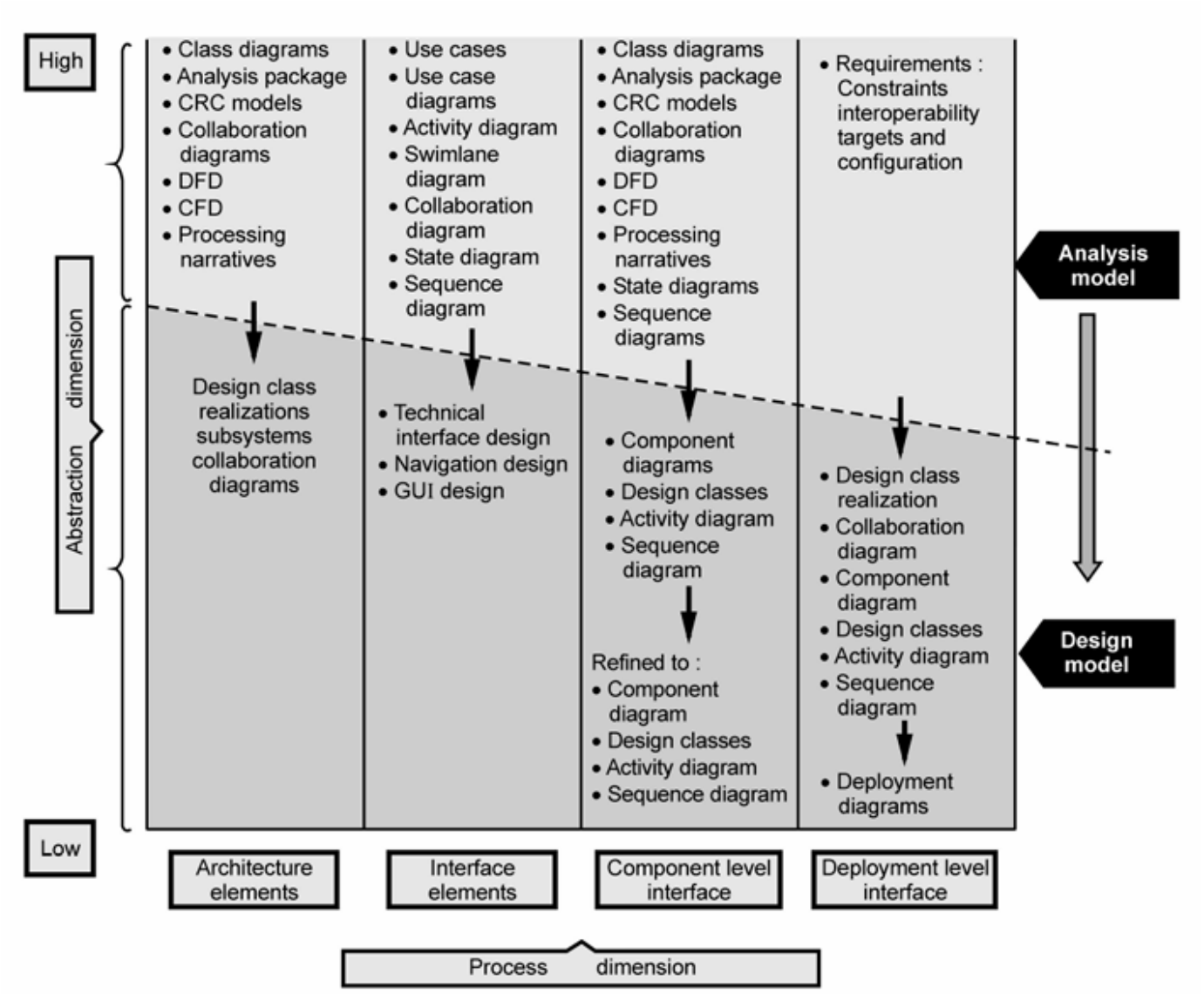
- **Data/Class Design**
- **Architectural Design**
- **Interface Design (Internet Design)**
- **Component-Level Design**

21) Explain dimensions of design model with the help of diagram.

✅ 1. Introduction

A **design model** in software engineering defines the structure and behavior of the software system. It serves as a bridge between the **requirements model** and the final implementation.

The **dimensions of the design model** help ensure that all views of the system are well-represented — from data structures to user interface and system behavior.



2. **Architectural Design**
    - Describes the overall software structure and its major components.
    - Includes the pattern and styles (e.g., layered, client-server).
3. **Interface Design**
    - Defines how the system will communicate with users, hardware, and other systems.
    - Includes user interfaces (UI) and external interfaces (APIs).
4. **Component-Level Design**
    - Describes the internal logic of each component or module.
    - Includes functions, control structures, and internal data.
5. **Deployment-Level Design**
    - Describes how software will be physically deployed on hardware.
    - Includes server configurations, databases, network design.

22) What is software Architecture? Why Architecture is important? What is the use of Architecture Decision Description Template?

✅ 1. What is Software Architecture?

Page 4-25

Software architecture refers to the **structured framework** used to conceptualize software elements, relationships, and properties.

> **Definition**: "Software architecture is the structure or structures of the system, which comprises software components, their externally visible properties, and the relationships among them."

✅ 2. Why is Architecture Important?

A. Provides a High-Level View

- Acts as a blueprint for the system's design and evolution.

B. Supports Communication

- Serves as a common language between stakeholders, developers, and testers.

C. Improves System Quality

- Helps ensure scalability, performance, and maintainability.

D. Enables Reuse

- Encourages modularization and reuse of components.

E. Guides Development and Maintenance

- Assists in decision-making throughout the software lifecycle.

✅ 3. Use of Architecture Decision Description Template

📄 *Page 4-27*

The **Architecture Decision Description Template** is a standardized format used to **document important architectural decisions**.

📌 Purpose:

- Captures **why** a specific architectural decision was made.
- Ensures future maintainers understand the rationale behind the system's structure.

📋 Typical Fields in the Template:

Field	Description
Title	Name of the decision
Context	Background and environment of the decision
Decision	What choice was made
Rationale	Why the decision was made
Implications	Effect on other components or the system
Alternatives	Other options considered
Related Decisions	Other linked architectural decisions
Author/Date	Person and date of documentation

23) Discuss component level and deployment level design elements.

- Component level design refers to the process of breaking down a system into its constituent parts to better understand how they interact and connect.
- In software engineering, it is the phase that focuses on defining and developing the software components that will be used to build the overall system architecture.
- Two main goals:-
  - Identify the components that are needed to build the system. This includes determining the boundaries of each component & how they relate to one another.
  - Define the interfaces between components. This makes the components loosely coupled and independent, allowing them to be developed.

4.7.5 Deployment Level Design Elements

The deployment level design elements indicate how software functions and software subsystems are assigned to the physical computing environment of the software product.

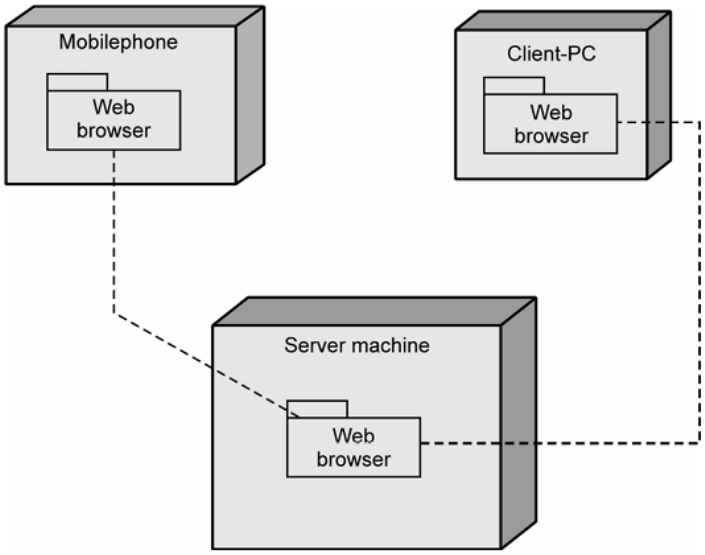


Fig. 4.7.4 Deployment diagram

For example web browsers may work in mobile phoned or they may run on client PC or can execute on server machines.

24) Explain guidelines for component level design.

✅ Guidelines for Component-Level Design

1. **Functional Independence**
   - Each component should perform a **single, well-defined function**.
   - Reduces interdependence (low coupling), making it **easier to test and maintain**.
2. **High Cohesion**
   - All operations within a component should be **closely related in purpose**.
   - Enhances **readability, clarity, and maintainability**.
3. **Low Coupling**
   - Components should have **minimal dependency** on others.
   - Ensures **changes in one component** have little impact on others.
4. **Information Hiding**
   - Internal implementation details should be **hidden from other modules**.
   - Promotes **encapsulation and modularity**.
5. **Well-Defined Interfaces**
   - Each component must define clear **inputs, outputs, and services**.
   - Interfaces should be **minimal and stable** to avoid tight coupling.
6. **Reusability**
   - Design components to be **reused across systems**.
   - Generalize logic and allow parameterization where necessary.
7. **Design for Testability**
   - Components should be **easily testable in isolation**.
   - Support testing with **logging hooks**, stubs, and mock interfaces.

25) Enlist the golden rules of User Interface Design.

✅ Golden Rules of User Interface Design

1. **Strive for Consistency**
   - Use consistent terminology, layouts, colors, and commands across the interface.
2. **Enable Frequent Users to Use Shortcuts**
   - Provide accelerators like keyboard shortcuts, function keys, or command abbreviations.
3. **Offer Informative Feedback**
   - The system should provide clear and timely feedback for every user action.
4. **Design Dialogues to Yield Closure**
   - Group sequences of actions into distinct beginning, middle, and end.
5. **Offer Simple Error Handling**
   - Design the system to prevent errors, and if errors occur, guide users with easy-to-understand messages.
6. **Permit Easy Reversal of Actions**
   - Users should be able to undo or redo their actions when necessary.
7. **Support Internal Locus of Control**
   - The user should feel in control of the interface, not the other way around.
8. **Reduce Short-Term Memory Load**
   - Keep the interface simple and reduce the amount of information the user has to remember.

26) Explain layered system architecture with neat diagram.

4.13.1.5 Layered Architecture

- The layered architecture is composed of different layers. **Each layer** is intended to perform **specific operations** so machine instruction set can be generated. Various components in each layer perform specific operations.
- The **outer layer** is responsible for performing the **user interface** operations while the components in the inner layer perform operating system interfaces.
- The components in **intermediate layer** perform **utility services** and **application software** functions.

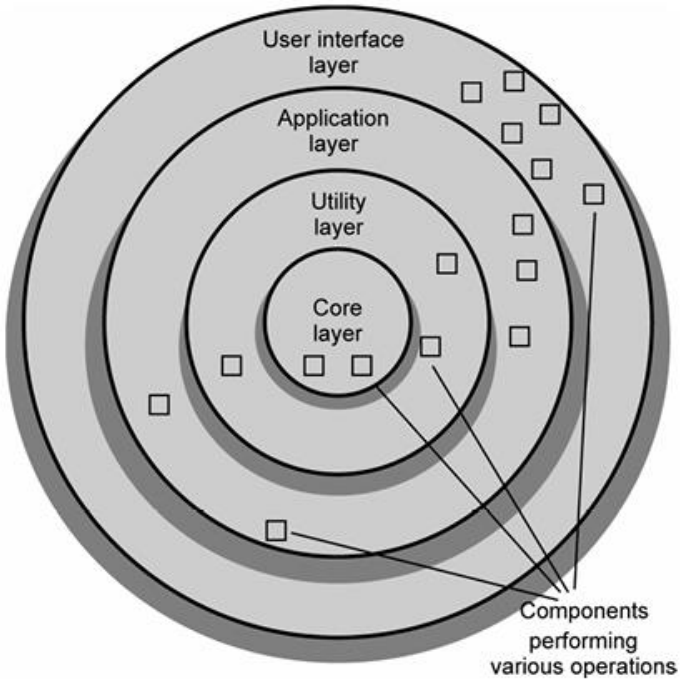


Fig. 4.13.6 Layer architecture of component

27) Describe notations used for deployment diagram. Describe the importance of Deployment diagram.

✅ 1. Notations Used for Deployment Diagram

A **Deployment Diagram** is a UML diagram that shows the **physical layout of hardware and software components** in a system.

📌 Common Notations:

Notation	Description
Node	Represented by a **3D box**; indicates hardware or software execution environment. Example: server, PC.
Component	Represented as a **rectangle with two smaller rectangles** on the side; shows software components deployed on nodes.
Association/Dependency Line	A **dashed line** with an arrow shows **dependency** or **communication** between components or nodes.
Artifact	Denotes **compiled code or configuration file** that resides on a node (e.g., `.exe`, `.jar`).
Stereotypes	Labels like `<<device>>`, `<<executionEnvironment>>` specify the role of nodes/components.

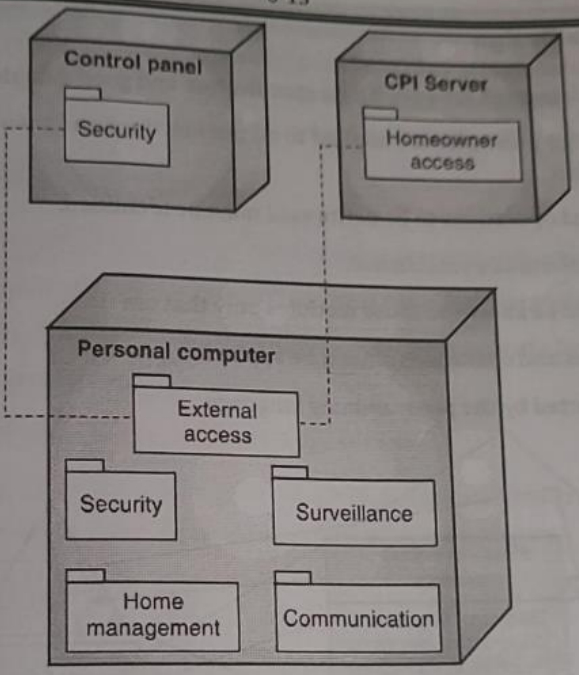


Fig. 6.5.4 : UML deployment diagram

It has three computing environment as shown as follows :

- **The personal computer** that implements security, observation and management.
- **The CPI server** manages the access rights.
- **The control panel** manages the security.

✅ 2. Importance of Deployment Diagram

1. **Represents Physical Architecture**
   - Shows how the software is **deployed across hardware** environments.
2. **Supports System Configuration**
   - Helps in visualizing **servers, databases, clients**, and their interconnections.
3. **Clarifies Resource Allocation**
   - Displays **which component is deployed on which node**, assisting in performance planning.
4. **Useful for Distributed Systems**
   - Particularly important when **multiple nodes** or environments (like cloud, mobile, desktop) are involved.
5. **Assists in Installation and Maintenance**
   - Helps developers and admins in **deployment planning**, system setup, and issue tracking.
6. **Improves Communication**
   - Provides a clear picture to **stakeholders, testers, and devops** teams regarding deployment structure.

28) Explain the following architectural styles with merits/demerits : i) Data-centered Architecture ii) Data-flow architecture

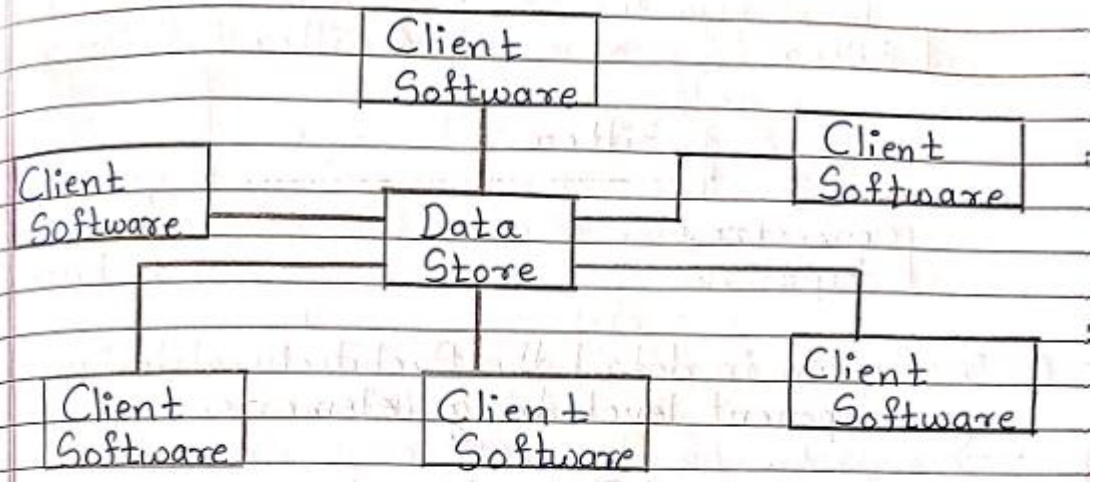
=> • The different architectural styles are:-

1] Data Centered Architecture
2] Data Flow Architecture
3] Object Oriented Architecture
4] Layered Architecture.

• Data Centered Architecture:-

- Data store at the center of this architecture & is accessed frequently by everyone.
- Update, add, delete or modify the data present with the store.
- It is widely used in DBMS, Library Information System etc.
- Advantages:-
  1] Independent of clients
  2] Add additional clients
  3] Modifications can be very easy.
- Disadvantages:-
  1] Duplication or replication is possible.
  2] Changes in data structures highly affect the clients.

- Diagrams :-



• Data Flow Architecture :-

- This architecture is used when input data to be transformed into output data.
- Pipe is a connector which passes the data directionally from one filter to the next.
- Filter is a component reads the data from its input pipes & performs its functions.
- Advantage :-

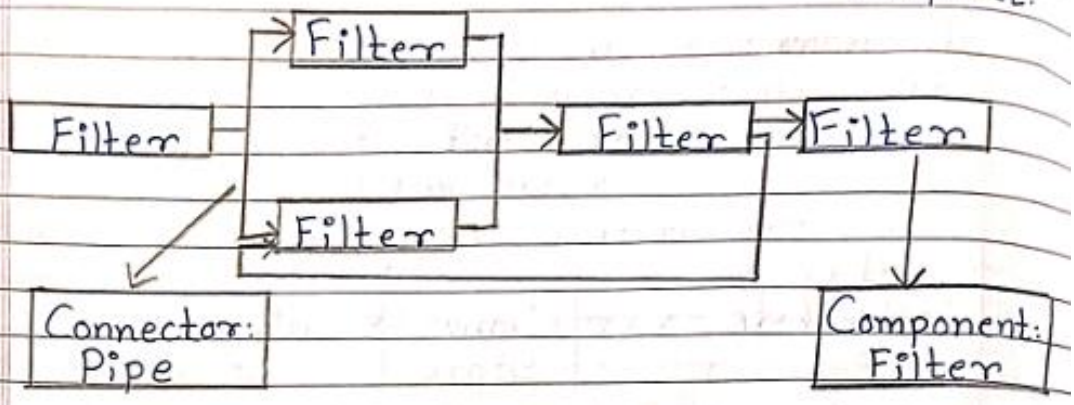
1] Concurrent execution is supported.

- Disadvantage :-

1] Does not allow greater user engagement.

- This data & phases or places the result on all output pipes.
- Diagram :-

~~Componet~~



29) What are the software design quality attributes and quality guidelines?

- The attributes of software design quality are:-

1] Functionality:- - It evaluates the features set and capabilities of the program.

2] Usability:- - It is accessed by the consider-ing the factors such as human factor, consistency and documen-tation.

3] Reliability:- - It is evaluated by measuring Parameters like frequency and security of failure, recovery from failure and output result accuracy.

4] Performance:- - It is measured by considering processing speed, response time, resource consumption & efficiency.

5] Supportability:- - Its ability to extend the program, adaptability, service-ability, testability and compact-ibility this terms are used to make system easily installed & found the problem easily.

• The guidelines of software design quality are :-

1] A design is generated using the recognizable architectural styles & compose a good design characteristics of components.

2] A design should implemented in ~~en~~ evolutionary manner for testing.

3] In design the representation of data, architect -ure, interface & components must be distinct.

4] A design creates an interface which reduce the complexity of connections.

5] Components must show the independent functional characteristics.

6] Design of the software must be modular.

30) Explain the user interface design principles.

✅ User Interface Design Principles

User Interface (UI) design principles help in building software that is **user-friendly, efficient, consistent**, and **visually appealing**.

---

1. User Familiarity

- Interface should use **terms, icons, and concepts familiar** to the user.
- Design should reflect the **user's real-world experiences**.

---

2. Consistency

- Interface elements (buttons, colors, fonts) should be **uniform throughout** the application.
- Reduces user confusion and enhances predictability.

---

3. Minimal Surprise

- Behavior of the system should be **predictable**.
- Users should not be surprised by system responses.

4. User Guidance

- System should provide **assistance, prompts, and feedback** to guide users.
- Examples include tooltips, help messages, or instructions.

5. User Control

- The user should feel **in control of the interface**.
- Allow easy **undo/redo** and avoid forcing unnecessary actions.

6. Error Prevention and Recovery

- Design should **prevent user errors** where possible.
- Provide meaningful error messages and allow **easy recovery**.

7. Visual Clarity

- Interface should be **aesthetically pleasing** and avoid clutter.
- Use proper alignment, spacing, colors, and font sizes.

31) List all the design concepts. Abstraction & refinement are complementary concepts. Justify.

✅ 1. Design Concepts (Fundamental Concepts of Software Design)

The following are the **key design concepts** in software engineering:

1. Abstraction
2. Modularity
3. Architecture
4. Refinement
5. Pattern
6. Information hiding
7. Functional independence
8. Refactoring
9. Design classes

✅ 2. Abstraction & Refinement – Complementary Concepts (Justification)

📌 **Abstraction** and **refinement** are two fundamental but **opposite** processes in software design that work together:

🔹 Abstraction:

- Focuses on **hiding lower-level details** and viewing the system at a higher level.
- Provides a **simplified view** of a component by focusing only on essential characteristics.
- Types: Functional, Data, Control abstraction.

🔹 Refinement:

- Is the **reverse of abstraction**.
- Involves breaking down a system into **more detailed levels** step by step.
- Transforms high-level design into low-level, detailed design or code.

✅ Why They Are Complementary:

Aspect	Abstraction	Refinement
Purpose	Hide complexity	Reveal details
Approach	Top-down (general to specific view)	Stepwise elaboration (detailed modeling)
Outcome	Simplicity, general view	Precision, implementation-level detail
Relation	Used to **understand** the system	Used to **build** the system

🔁 Both work in **opposite directions** but are used **together** to define and construct a complete software system — from **abstract concept to concrete implementation**.

32) Explain the user interface design principles and interface evaluation cycle.

✅ 2. Interface Evaluation Cycle

📄 *Page 4-23*

The **Interface Evaluation Cycle** ensures the UI is assessed and improved **iteratively** during development.

🔁 Phases in the Evaluation Cycle:

1. **Design the Interface**
   - Based on user requirements and principles.
2. **Build the Interface Prototype**
   - Create mockups or working models.
3. **Test with Users**
   - Gather feedback from actual users via observation or questionnaires.
4. **Evaluate Feedback**
   - Analyze usability problems and user behavior.
5. **Refine the Interface**
   - Make necessary changes to improve usability.
6. **Repeat the Cycle**
   - Continue refining until the UI is stable and satisfactory.

33) Enlist and explain Component level design steps in detail.

✅ Component-Level Design – Introduction

Component-level design focuses on the **internal implementation** of software components. It transitions from architectural design to **procedural-level detail** required for coding.

✅ Steps in Component-Level Design

1. Identify Component Interfaces
   - Determine **inputs, outputs, and external dependencies** for each component.
   - Define public methods, data structures, and communication protocols.
2. Define Component Behavior
   - Use **flowcharts, decision tables, or pseudocode** to specify internal logic.
   - Describe how the component will process input and generate output.
3. Design Data Structures
   - Choose appropriate **data types and structures** that the component will use.
   - Ensure they support performance and storage requirements.
4. Develop Algorithms
   - Write or select suitable **algorithms** to perform tasks within the component.
   - Focus on correctness, efficiency, and clarity.
5. Ensure Functional Independence
   - Design each component to perform **only one task** with **minimal interaction** with others.
   - Encourages **high cohesion and low coupling**.
6. Apply Design Notations
   - Use UML diagrams, flowcharts, or structured charts to **visually represent** logic and data flow.
   - Helps in clear communication and review.
7. Review and Refine Design
   - Perform **peer reviews and walkthroughs** to check for logical errors and design flaws.
   - Ensure design aligns with requirements and is ready for implementation.

UNIT - 5

34) Briefly explain the steps involved in risk planning in project development.

Steps Involved in Risk Planning in Project Development

1. **Risk Identification**
   - Identify all potential risks that could affect the project.
   - Analyze aspects such as project scope, technology, resources, and schedule.
   - Prepare a list of software risks (e.g., technical, schedule, cost, personnel risks).
2. **Risk Projection (Risk Analysis)**
   - Assess the likelihood and consequences of each risk.
   - Estimate risk probability and impact.
   - Create a **risk table** indicating the severity and priority of each risk.
3. **Risk Refinement**
   - Break down high-priority risks into sub-risks for better analysis.
   - Identify risk components and their driving factors.
   - Helps in understanding and managing complex risks.
4. **Risk Mitigation, Monitoring, and Management (RMMM)**
   - **Mitigation**: Plan actions to reduce the risk or its impact.
   - **Monitoring**: Observe risk indicators and symptoms.
   - **Management**: Prepare contingency plans in case the risk occurs.
5. **Preparing the RMMM Plan**
   - Document the strategies for mitigating, monitoring, and managing each risk.
   - Assign responsibilities and define actions for dealing with risks.
   - Ensure the plan is followed and updated throughout the project lifecycle.

35) Describe with an example how the effect of risk on project schedule is evaluated using PERT

Program Evaluation and Review Technique (PERT) and Risk Evaluation

PERT is used to estimate **time required** for project tasks considering the **uncertainty** (risk) involved in project scheduling.

1. Three Time Estimates in PERT

To evaluate risk, PERT uses **three types of time estimates**:

- **Optimistic Time (O):** Minimum possible time to complete a task (if everything goes well).
- **Most Likely Time (M):** Normal time required to complete the task.
- **Pessimistic Time (P):** Maximum time assuming things go wrong.

2. Expected Time (TE) Calculation

The formula to calculate expected time:

$$TE = \frac{O + 4M + P}{6}$$

This average gives a **weighted time estimate** based on the impact of risk.

3. Standard Deviation (σ) and Variance

- Standard Deviation (σ):

$$\sigma = \frac{P - O}{6}$$

- Variance (σ^2):

$$\sigma^2 = \left(\frac{P - O}{6}\right)^2$$

These help evaluate the **level of risk or uncertainty** in the schedule.

4. Example

Consider a task with the following estimates:

- Optimistic time (O) = 4 days
- Most Likely time (M) = 6 days
- Pessimistic time (P) = 10 days

Expected Time (TE):

$$TE = \frac{4 + 4(6) + 10}{6} = \frac{4 + 24 + 10}{6} = \frac{38}{6} \approx 6.33 \text{ days}$$

Standard Deviation (σ):

$$\sigma = \frac{10 - 4}{6} = 1$$

Variance (σ^2):

$$\sigma^2 = 1^2 = 1$$

36) Explain Version Control and Change Control Layer in Software Configuration Management in detail.

5.11.2 Change Control

- Changes in any software projects are vital. Sometimes, introducing small changes in the system may lead to big problems in product.
- Similarly, introducing some changes may enhance the capabilities of the system.
- According to **James Bach** too little changes may create some problems and too big changes may create another problems.
- For a large software engineering project, uncontrolled change creates lot of chaos. For managing such changes, human procedures or automated tools can be used.
- The change control process is shown by following Fig. 5.11.1 (See Fig. 5.11.1 on Next page)

Step 1 : First of all there arises a need for the change.

Step 2 : The change request is then submitted by the user.

Step 3 : Developers evaluate this request to assess technical merits, potential side effects, and overall impact on system functions and cost of the project.

Step 4 : A change report is then generated and presented to the Change Control Authority (CCA).

Step 5 : The change control authority is a person or a group of people who makes a final decision on status or priority of the change.

Step 6 : An Engineering Change Order **(ECO)** is generated when the change gets approved. In ECO the change is described, the restrictions and criteria for review and audit are mentioned.

Step 7 : The object that needs to be changed is checked out of the project database.

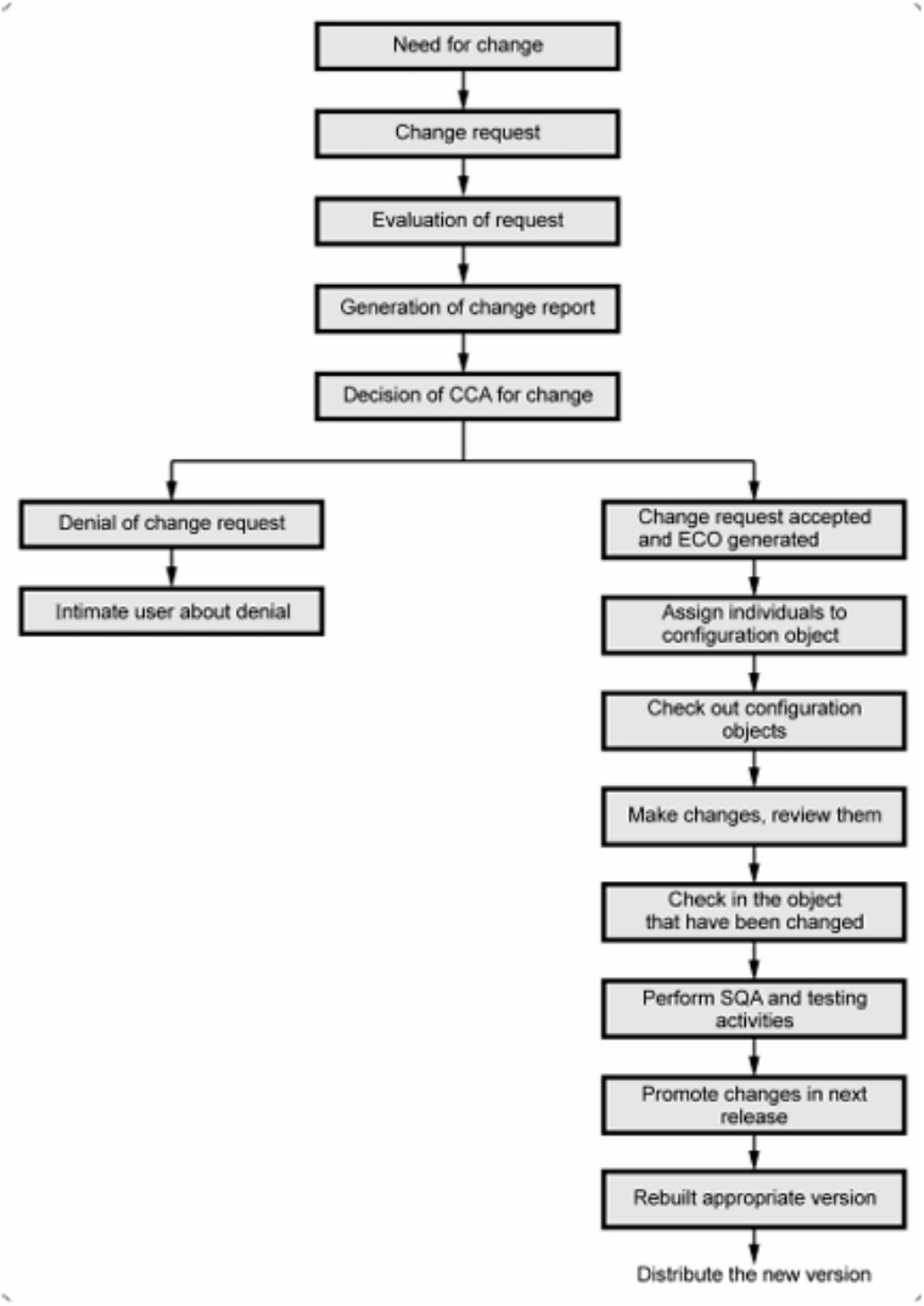
Step 8 : The changes are then made on the corresponding object and appropriate SQA activities are then applied.

Step 9 : The changed object is then checked in to the database and appropriate version control is made to create new version.

5.11.3 Version Control

- Version is an instance of a system which is functionally distinct in some way from other system instances.
- Version control works to help manage different versions of configuration items during the development process.

- The configuration management allows a user to specify the alternative configurations of the software system by selecting appropriate version.
- Certain attributes are associated with each software version. These attributes are useful in identifying the version. For example : The attribute can be ‘date’, ’creator’, ’customer’, ‘status’.
- In practice the version needs an associated name for easy reference.
- Different versions of a system can be shown by an evolution graph as shown in Fig. 5.11.2.
- Each version of software system is a collection of software configuration items.



37) Discuss Software Configuration Management in detail.

Definition : Software configuration management is a set of activities carried out for identifying, organizing and controlling changes throughout the lifecycle of computer software.

- During the development of software change must be managed and controlled in order to improve quality and reduce error.
- Hence Software Configuration Management is a **quality assurance activity** that is applied throughout the software process.
- The **origins of changes** that are requested for software are -
  1. New business or market positions cause the changes in the requirements. Due to which the changes need to occur.
  2. New stakeholders may require some changes in the existing requirements.
  3. Due to business growth or project extension, it is essential to makes changes in the project.
  4. Sometimes due to schedule or budge constraints, changes are must in project.
- The software configuration management is concerned with managing evolving software systems.
- Software configuration management is a set of tracking and control activities that begin when a software development project begins and terminates when the software is taken out of operation.

5.9.2 Elements of Configuration Management System

Susan Dart identified four important elements for software configuration management system. These elements are -

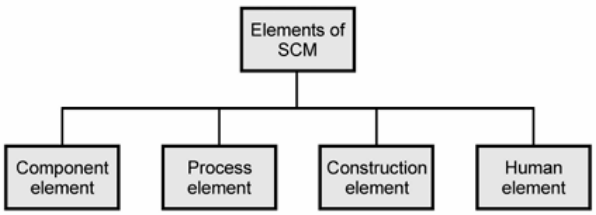


Fig. 5.9.1 Elements of SCM

1. **Component Elements :** It consists of collection of tools that are used for file management system. For example - databases.
2. **Process Elements :** It consists of actions and tasks used during change management and use of software.
3. **Construction Elements :** It is a collection of tools that automate the construction of software.
4. **Human Elements :** It consists of set of tools that are used by software team to implement software configuration management.

38) Define Software Risk in detail. What are different types of Software Risk?

1. Definition of Software Risk

- A **software risk** refers to a condition or event that may **adversely affect** a software project.
- Risk is a **potential problem** — it might happen or it might not.
- If it does occur, it can impact **project cost, schedule, performance, or quality**.

2. Characteristics of Software Risk

- Risks involve **uncertainty**.
- Risks can be **identified**, **analyzed**, and **managed**.
- Risk management focuses on **minimizing** the probability and impact of negative events.

Different types of risk

1. **Project risk**

   Project risks arise in the software development process then they basically affect budget, schedule, staffing, resources, and requirements. When project risks become severe then the total cost of project gets increased.

2. **Technical risk**

   These risks affect quality and timeliness of the project. If technical risks become reality then potential design implementation, interface, verification and maintenance problems gets created. Technical risks occur when problem becomes harder to solve.

3. **Business risk**

   When feasibility of software product is in suspect then business risks occur. Business risks can be further categorized as

   i) Market risk - When a quality software product is built but if there is no customer for this product then it is called market risk (i.e. *no market for the product)*.

   ii) Strategic risk - When a product is built and if it is not following the company's business policies then such a product brings strategic risks.

   iii) Sales risk - When a product is built but how to sell is not clear then such a situation brings sales risk.

   iv) Management risk - When senior management or the responsible staff leaves the organization then management risk occurs.

   v) Budget risk - Losing the overall budget of the project is called budget risk.

39) Discuss the RMMM plan in detail.

- The RMMM plan is a document in which all the risk analysis activities are described.
- Sometimes project manager includes this document as a part of overall project plan.
- Sometimes specific RMMM plan is not created, however each risk can be described individually using risk information sheet.
- Typical template for RMMM plan or Risk information sheet can be,

<table>
<tr><td colspan="4">Risk information sheet</td></tr>
<tr><td colspan="4">Project name <enter name of the project for which risks can be identified></td></tr>
<tr><td>Risk id <#></td><td>Date
<date at which risk is identified ></td><td>Probability
<risk probability></td><td>Impact
<low/medium/high></td></tr>
<tr><td colspan="2">Origin
<the person who has identified the risk></td><td colspan="2">Assigned to
<who is responsible for mitigating the risk></td></tr>
<tr><td colspan="4">Description
<Description of risk identified></td></tr>
<tr><td colspan="4">Refinement/Context
<associated information for risk refinement></td></tr>
<tr><td colspan="4">Mitigation/Monitoring
<enter the mitigation/monitoring steps taken></td></tr>
<tr><td colspan="4">Trigger/Contingency plan
<if risk mitigation fails then the plan for handling the risk></td></tr>
<tr><td colspan="4">Status
<Running status that provides a history of what is being done for the risk and changes in the risk. Include the date the status entry was made></td></tr>
<tr><td colspan="2">Approval
<name and signature of person approving closure>.</td><td colspan="2">Closing date
<date></td></tr>
</table>

RMMM stands for **risk mitigation**, **monitoring** and **management**. There are three issues in strategy for handling the risk is

1. Risk avoidance
2. Risk monitoring
3. Risk management.

Risk mitigation

Risk mitigation means preventing the risks to occur(risk avoidance). Following are the steps to be taken for mitigating the risks.

1. Communicate with the concerned staff to find of probable risk.
2. Find out and eliminate all those causes that can create risk before the project starts.
3. Develop a policy in an organization which will help to continue the project even though some staff leaves the organization.
4. Everybody in the project team should be acquainted with the current development activity.
5. Maintain the corresponding documents in timely manner. This documentation should be strictly as per the standards set by the organization.
6. Conduct timely reviews in order to speed up the work.

Risk monitoring

In risk monitoring process following things must be monitored by the project manager,

1. The approach or the behaviour of the team members as pressure of project varies.
2. The degree in which the team performs with the spirit of "team-work".
3. The type of co-operation among the team members.
4. The types of problems that are occurring.
5. Availability of jobs within and outside the organization.

The project manager should monitor certain mitigation steps. For example.

If the current development activity is monitored continuously then everybody in the team will get acquainted with current development activity.

The **objective** of risk monitoring is

1. To check whether the predicted risks really occur or not.
2. To ensure the steps defined to avoid the risk are applied properly or not.
3. To gather the information which can be useful for analyzing the risk.

Risk management

- Project manager performs this task when risk becomes a reality.
- If project manager is successful in applying the project mitigation effectively then it becomes very much easy to manage the risks.
- **For example**, consider a scenario that many people are leaving the organization then if sufficient additional staff is available, if current development activity is known to everybody in the team, if latest and systematic documentation is available then any 'new comer' can easily understand current development activity. This will ultimately help in continuing the work without any interval.

40) Explain Risk and management concern with the help of diagram.

- **Definition of risk :** The risk denotes the uncertainty that may occur in the choices due to past actions and risk is something which causes heavy losses.
- **Definition of risk management :** Risk management refers to the process of making decisions based on an evaluation of the factors that threats to the business.
- Various **activities** that are carried out for **risk management** are -
  1. Risk identification
  2. Risk projection
  3. Risk refinement
  4. Risk mitigation, monitoring and management.

✅ 3. Types of Risks

- **Project Risks**: Delay in schedule, budget issues.
- **Technical Risks**: Technology limitations, integration issues.
- **Business Risks**: Market changes, stakeholder withdrawal.

✅ 4. Risk Management Process (RMMM Plan)

📍 *As per textbook (Pg. 5-10)*

A. Risk Identification

- Recognize potential risks before they become problems.

B. Risk Projection

- Analyze likelihood and impact.
- Build a **Risk Table** (Risk, Probability, Impact, Description).

C. Risk Refinement

- Break down large risks into manageable parts.

D. Risk Mitigation

- Prepare strategies to reduce or avoid risk.
- E.g., add buffer time, assign experienced staff.

E. Risk Monitoring

- Continuously observe risk indicators and progress.

F. Risk Management

- Take corrective actions when risk becomes reality.

41) What are the advantages of SCM Repository? Explain functions performed by SCM Repository.

- Software configuration Items (SCI) are maintained in project repository or project library.
- The software repository is basically a database that acts as a center for both accumulation and storage for software engineering information.
- The software engineer interacts with repository using tools that are integrated within it.

5.10.1 Role of Project Repository

- Software repository is a **collection of information** accessed by software engineers to make appropriate changes in it.
- This repository is handled using all the modern database management functions.
- It must maintain important properties such as data integrity, sharing and integration.
- The repository must maintain uniform structure and format for software engineering work products.

5.10.2 Features

The Software Configuration Management can be performed by interacting with repository. The repository must have toolset that provides support for following features –

1. **Versioning :** As project progresses various versions of individual work products may get created. The repository must be able to maintain all these versions and permit the developer to go back to previous versions during testing and debugging.
2. **Dependency Tracking and Change Management** : The data elements stored in the repository are related to each other. The repository must have an ability to keep track of these relationship and to maintain data integrity.
3. **Requirements Tracing :** If the constructed components, its designed is tracked, then particular requirement can be traced out. The repository must have this ability to trace the requirement from constructed components.
4. **Configuration Management :** The repository must be able to keep track of series of configurations representing project milestones and production releases.
5. **Audit Trails :** The audit trail

✅ 2. Advantages of SCM Repository

1. **Centralized Storage**
   - All documents, code, and design items are stored in one central location.
2. **Change Control**
   - Tracks changes made to software artifacts over time.
3. **Version Control**
   - Maintains different versions of files and allows rollback to earlier versions.
4. **Access Control**
   - Restricts access to authorized users only.
5. **Audit Trail**
   - Maintains historical data of changes for auditing and review.
6. **Improved Team Collaboration**
   - Enables multiple team members to work in parallel without conflicts.
7. **Supports Reusability**
   - Reusable components can be stored and retrieved easily.

42) Write short note on SCM Process

✅ 1. Definition of SCM Process

- The **Software Configuration Management (SCM) Process** refers to a set of activities required to systematically **control changes**, **track versions**, and **maintain integrity** of software products throughout the development life cycle.

✅ 2. Objectives of SCM Process

- Manage software changes efficiently.
- Ensure version control and traceability.
- Maintain integrity and consistency of software products.

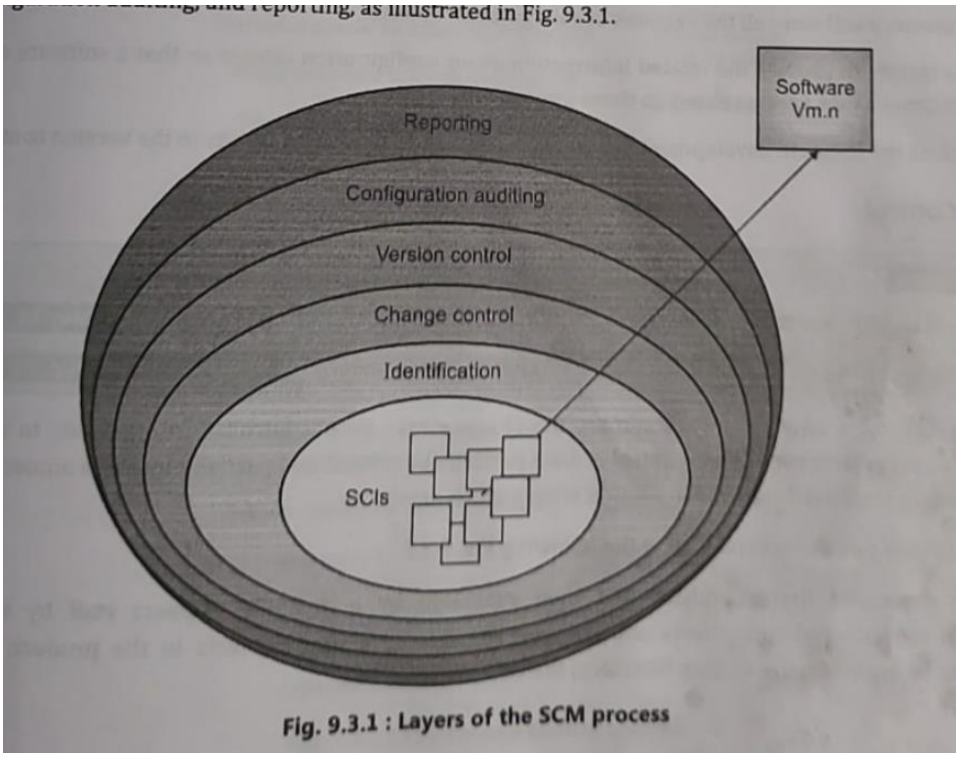


Fig. 9.3.1 : Layers of the SCM process

✅ 3. Activities in SCM Process

🔹 1. **Identification of Configuration Items**

- Identify and label items such as code, documents, designs, and test cases that need to be tracked.

🔹 2. **Change Control**

- Handle change requests.
- Evaluate the impact and implement approved changes.

🔹 3. **Version Control**

- Maintain different versions of each configuration item.
- Enable rollback to earlier versions if required.

🔹 4. **Configuration Audit**

- Review configuration items to verify compliance with requirements.

🔹 5. **Status Reporting**

- Track and report the current status of configuration items.

✅ 4. Benefits of SCM Process

- Prevents unauthorized changes.
- Facilitates team collaboration.
- Supports traceability and rollback.
- Improves product quality and reliability.

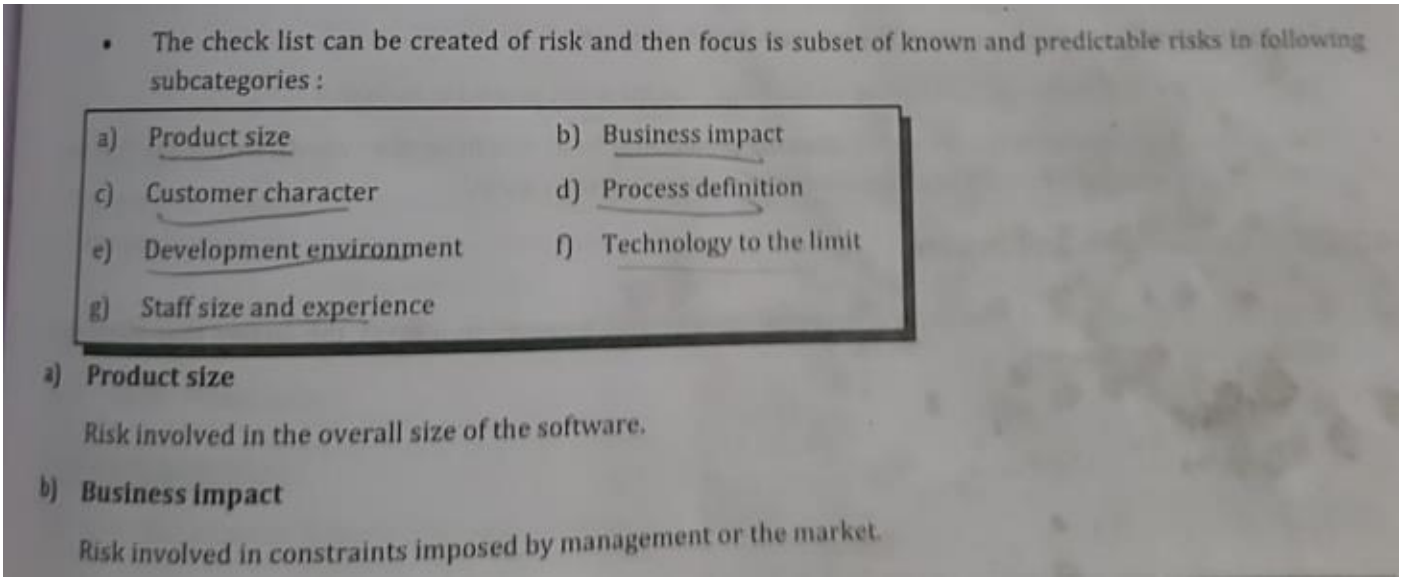
43) What is Risk Identification? What are different categories of Risk?

✅ 1. Definition of Risk Identification

- **Risk Identification** is the first step in the **Risk Management Process**.
- It involves recognizing and documenting all **possible risks** that could affect the software project.
- The goal is to create a comprehensive list of **potential problems** that might impact project **schedule, cost, quality**, or **performance**.

The risk identification is based on **two approaches**

1. **Generic risk identification -** It includes potential threat identification to software project.
2. **Product-specific risk identification -** It includes product specific threat identification by understanding people, technology and working environment in which the product gets built.



- The check list can be created of risk and then focus is subset of known and predictable risks in following subcategories :

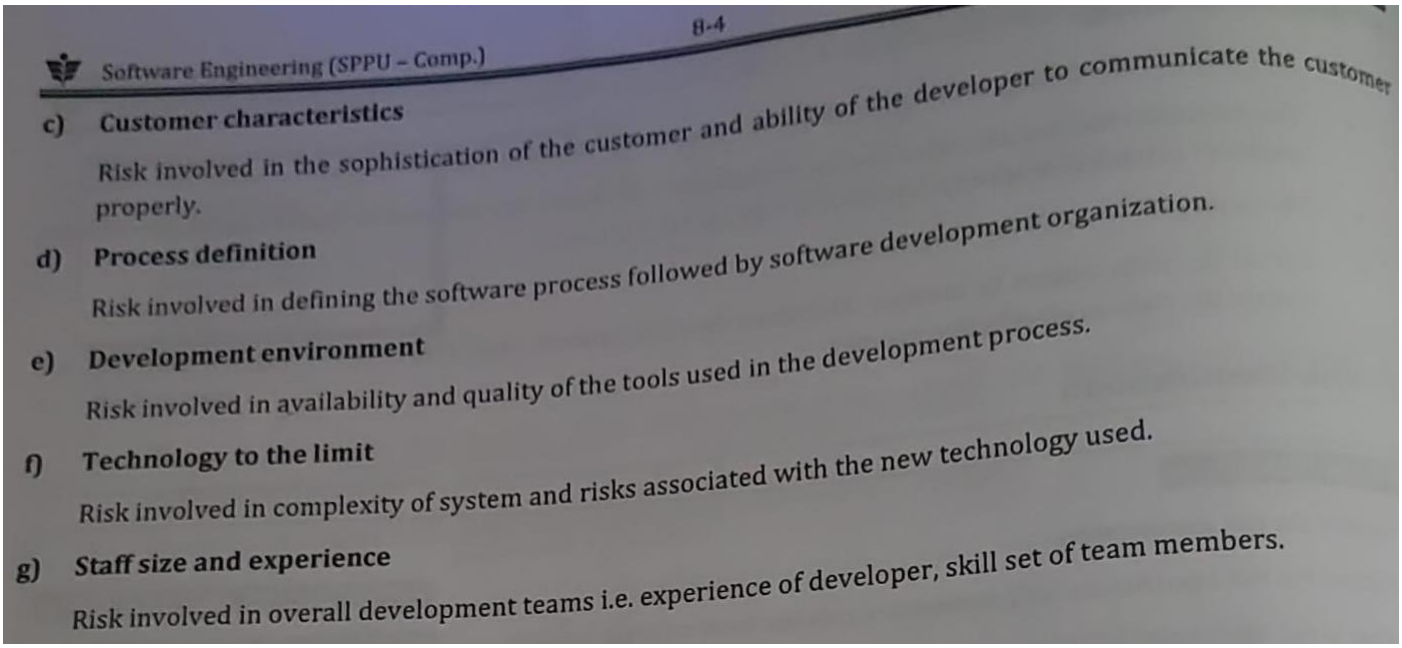
a) Product size	b) Business impact
c) Customer character	d) Process definition
e) Development environment	f) Technology to the limit
g) Staff size and experience	

a) **Product size**

Risk involved in the overall size of the software.

b) **Business impact**

Risk involved in constraints imposed by management or the market.



c) **Customer characteristics**

Risk involved in the sophistication of the customer and ability of the developer to communicate the customer properly.

d) **Process definition**

Risk involved in defining the software process followed by software development organization.

e) **Development environment**

Risk involved in availability and quality of the tools used in the development process.

f) **Technology to the limit**

Risk involved in complexity of system and risks associated with the new technology used.

g) **Staff size and experience**

Risk involved in overall development teams i.e. experience of developer, skill set of team members.

44) What is risk projection? How risk projection is carried out using risk table?

- Risk projection is interchangeably called as Risk estimation also. The risk projection rates the risk in following two ways :
  - The probability of risk occurrence and
  - The consequences of the risk occurred.
- Following are four important risk projection activities that every manager, developer should perform :
  - Make a scale to measure the likelihood of the risk.
  - Describe the consequences of the risks.
  - Estimate its impact and
  - Write down overall accuracy of the risk projection to avoid misunderstandings.

Q. Write short note on : Risk table.

- A risk table gives very useful technique to project managers for the risk projection. Following Table 8.3.1 is an example of risk table.

Table 8.3.1 : A sample risk table

Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse then planned	PS	70%	2	
End-users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirement	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	

- In the Table 8.3.1 the impact values are as follows :
  - For the disastrous situations → 1
  - For some critical situations → 2
  - For the average situations → 3
  - For the negligible situations → 4
- In the above table the categories used are as follows :
  - PS used for project size risk

BU used for business risk
CU used loss of funds etc.

45) Prepare RMMM plan for late delivery of software product to the customer.

✅ Risk Information Sheet

Project name	**Library Management System**
Risk id	**R-101**
Date	10-May-2025
Probability	High
Impact	High
Origin	**Project Manager – Mr. A. Sharma**
Assigned to	Development Lead – Ms. R. Patel

Description

There is a high probability of delay in delivery due to underestimation of development time and limited resources.

Refinement/Context

The team is currently behind schedule by 2 weeks due to unresolved technical dependencies and unexpected staff absences.

Mitigation/Monitoring

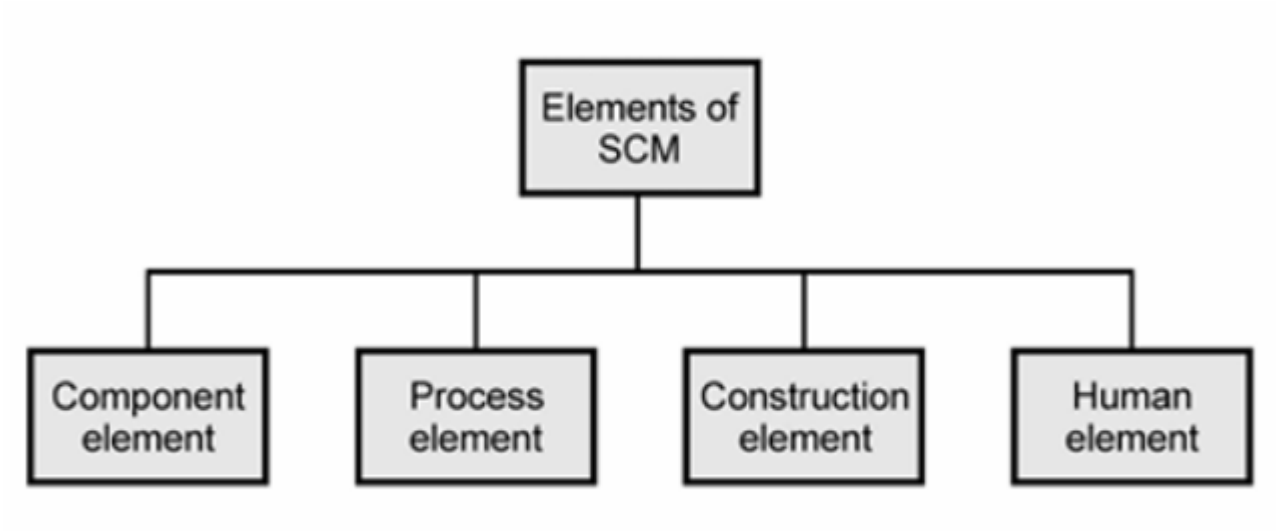
- Additional developers will be temporarily assigned.
- Daily stand-up meetings to track progress.
- Critical tasks are being prioritized and rescheduled.
- Weekly progress reviews with client. |

Trigger/Contingency Plan

- If delay extends beyond 3 weeks, inform client with revised timeline.
- Negotiate for phased delivery (MVP first).
- Engage temporary contract staff to speed up backlog tasks. |

Status	
• **10-May-2025:** Risk identified and logged. • **12-May-2025:** 2 developers added to the backend team. • **14-May-2025:** First milestone delivery date rescheduled with client approval. |	
Approval	**A. Sharma (Project Manager)**
Closing date	*To be updated upon resolution*

46) What are the elements that exist when an effective SCM system is implemented? Discuss each briefly.



✅ Elements of Software Configuration Management (SCM)

The major **elements of SCM** as shown in the diagram are:

🔹 1. Component Element

- Refers to all **configuration items** used in the software project.
- Includes:
  - Source code
  - Documents
  - Test data
  - Design models
- Each component is uniquely identified and tracked.

🔷 2. Process Element

- Defines the **procedures and workflows** for managing changes to software.
- Includes:
  - Change control process
  - Baseline management
  - Configuration audits

🔷 3. Construction Element

- Concerned with **building and assembling** the software.
- Tools and techniques that ensure:
  - Repeatable builds
  - Consistent releases
- Examples: Build scripts, compilers, CI/CD tools

🔷 4. Human Element

- Refers to the **people** involved in SCM activities.
- Roles include:
  - Configuration manager
  - Developers
  - Reviewers
- Human interaction is crucial for change approval, audits, and decision-making.

47) Explain Risk Refinement in detail.

✅ 1. Definition of Risk Refinement

- **Risk Refinement** is the process of breaking down a general (broad) risk into **more specific and manageable sub-risks**.
- It helps in better **understanding**, **analyzing**, and **responding** to the risk.
- It is used when a risk is too complex or vague to address directly.

✅ 2. Purpose of Risk Refinement

- To **sharpen the focus** on a particular risk.
- To make risk **assessment more accurate**.
- To allow the creation of **precise mitigation plans**.

✅ 3. Example of Risk Refinement

Broad Risk: **"Product requirements may be misunderstood."**

Refined into sub-risks:

1. Communication gap between customer and analyst.
2. Requirements documentation is incomplete or ambiguous.
3. Frequent changes in requirements without proper tracking.
4. Lack of domain knowledge among team members.

> These refined risks can be individually assessed and mitigated.

48) What are the guidelines those lead to a successful software testing strategy?

✅ 1. Introduction

A **software testing strategy** provides a roadmap for performing testing activities effectively. A well-planned strategy ensures early defect detection, proper validation, and delivery of quality software.

✅ 2. Guidelines for a Successful Testing Strategy

According to the textbook, the following guidelines ensure a successful software testing strategy:

🔹 1. Testing Should Begin at the Module Level

- Start testing at the **smallest unit** (like functions or classes).
- Progressively move towards integration and system-level testing.

🔹 2. Use Independent Testing Team

- Testing should be done by a **separate team**, not the developers.
- Ensures **unbiased** detection of defects.

🔹 3. Test Planning Should Begin Early

- Prepare test plans **in parallel** with development.
- Helps in allocating proper time, resources, and scope for testing.

🔹 4. Testers Should Understand the Software and Its Goals

- The testing team must have **domain knowledge** and understand:
  - User needs
  - System behavior
  - Performance expectations

🔹 5. Tests Should Be Planned Before Testing Begins

- Identify **test cases, inputs, outputs, and expected results** in advance.
- Enables organized and efficient testing.

🔹 6. Testing Should Be Risk-Focused

- Prioritize testing for **high-risk modules or components**.
- Improves defect detection where it matters most.

🔹 7. Automation Where Appropriate

- Use automation tools (e.g., Selenium, JUnit) to save time and improve coverage.
- Especially useful for **regression and performance testing**.

49) What is meant by integration testing? Explain top down and bottom up integration testing.

✅ 1. Definition of Integration Testing

- **Integration Testing** is a **level of software testing** where **individual modules** are combined and tested as a group.
- The goal is to identify **interface defects** between modules.
- It ensures that different parts of the system work **together correctly**.

✅ 2. Purpose of Integration Testing

- To verify **data communication**, **control flow**, and **interaction** between integrated units.
- To detect issues such as:
  - Incorrect data passed between modules
  - Incomplete or wrong interface logic

🔷 A. Top-Down Integration Testing

- Modules are integrated from **top (main module)** to **bottom (sub-modules)**.
- Major control modules are tested first, followed by lower-level modules.

📌 **Steps:**

1. Main control module is tested first.
2. Stubs are used for sub-modules not yet developed.
3. Sub-modules are added and tested step by step.

✔ **Advantages:**

- Early verification of high-level logic and control flow.
- Major design flaws are detected early.

❌ **Disadvantages:**

- Requires many **stubs**.
- Low-level modules are tested late.

🔷 B. Bottom-Up Integration Testing

- Integration starts from **lowest-level modules** and moves upward.
- Sub-modules are tested first using **drivers** to simulate higher modules.

📌 **Steps:**

1. Build and test independent sub-modules first.
2. Integrate upward toward the main control module.

✔ **Advantages:**

- Lower-level modules are tested early.
- No stubs are needed; only drivers.

❌ **Disadvantages:**

- High-level logic is tested late.
- Requires test drivers for each module set.

Integration testing is a crucial step in ensuring that **individual modules work together** properly. Both **Top-Down** and **Bottom-Up** approaches have their pros and cons, and the choice depends on the project structure and priorities.

Top-Down Integration Testing

- **Advantages:**
  1. Major design flaws are identified early.
  2. Important modules are tested first.
  3. Supports early system prototype development.
- **Disadvantages:**
  1. Lower-level modules may be tested late.
  2. Requires development of numerous stubs (dummy modules).
  3. Missing lower modules can delay the testing process.

Bottom-Up Integration Testing

- **Advantages:**
  1. Lower-level modules are tested first, ensuring basic functionality.
  2. No need for stubs, only drivers (simpler to implement).
  3. Errors in critical lower modules are identified early.
- **Disadvantages:**
  1. Design flaws may be detected late.
  2. Major module combinations may be tested last.
  3. Final product prototype is achieved late in the process.

50) What is the difference between verification and validation?

Sr. No.	Verification	Validation
1.	Verification refers to the set of activities that ensure software correctly implements the specific function.	Validation refers to the set of activities that ensure that the software that has been built is traceable to customer requirements.
2.	After a valid and complete specification the verification starts.	Validation begins as soon as project starts.
3.	Verification is for prevention of errors.	Validation is for detection of errors.
4.	Verification is conducted using reviews, walkthroughs, inspections and audits.	Validation is conducted using system testing, user interface testing and stress testing.
5.	Verification is also termed as white box testing or static testing as work product goes through reviews.	Validation can be termed as black box testing or dynamic testing as work product is executed.
6.	Verification finds about 50 to 60 % of the defects.	Validation finds about 20 to 30 % of the defects.
7.	Verification is based on the opinion of reviewer and may change from person to person.	Validation is based on the fact and is often stable.
8.	The verification verifies the problem statements, decisions taken during the development and execution paths.	The validation validates the requirements, functionalities and features of the product.
9.	Verification is about process, standard and guideline.	Validation is about the product.

51) Differentiate between black box testing and white box testing

Point	Black Box Testing	White Box Testing
Definition	Testing based on **inputs and outputs**. Internal code is not visible.	Testing based on **internal code logic and structure**.
Focus	Verifies **functionality** against requirements.	Verifies **internal logic**, loops, paths, and conditions.
Tester Knowledge	No knowledge of internal code is required.	Requires full knowledge of source code.
Performed By	Testers or QA team.	Developers or white-box testers.
Techniques Used	Equivalence Partitioning, Boundary Value Analysis, Decision Table, etc.	Statement Coverage, Branch Coverage, Loop Testing, etc.
Type of Testing	**Functional testing**.	**Structural testing**.
Tools Used	Selenium, QTP, etc.	JUnit, NUnit, etc.

52) Explain how Object oriented software testing is different from conventional software testing.

Conventional Software Testing is applied to procedural programming (e.g., C), where programs are structured as a sequence of procedures or functions.

Object-Oriented Software Testing (OOST) is tailored for object-oriented languages (e.g., Java, C++) where the focus is on **objects, classes, inheritance**, and **encapsulation**.

Aspect	Conventional Testing	Object-Oriented Testing
Testing Focus	Focuses on **functions, procedures, and data flow.**	Focuses on **classes, objects, and their interactions.**
Testing Levels	Unit → Integration → System → Acceptance.	Class testing → Cluster testing → System testing.
Test Units	Individual procedures or modules.	Classes and methods (with data and behavior).
Inheritance & Polymorphism	Not applicable or limited.	Must test inherited methods, overridden behavior.
Encapsulation	Functions and data are separate, easy to isolate.	Data is hidden; testing requires access methods (getters/setters).
Test Design	Relatively simpler, procedural logic.	More complex due to inter-object interactions and state behavior.

53) Explain Unit Testing and Integration Testing with respect to the Object Oriented Context

6.7.1 Unit Testing in OO Context

- **Class** is an encapsulation of data attributes and corresponding set of operations.
- The **object** is an instance of a class. Hence objects also specify some data attributes and operations.
- In object oriented software the focus of unit testing is considered as classes or objects.
- However, operations within the classes are also the testable units. In fact in OO context the operation can not be tested as single isolated unit because one operation can be defined in one particular class and can be used in multiple classes at the same time. For example consider the operation **display()**. This operation is defined as super class an at the same time it can be used by the can be multiple derived classes. Hence it is not possible to test the operation as single module.
- Thus class testing for OO software is equivalent to unit testing for conventional software. In conventional software system, the algorithmic details and data that flow are units of testing but in OO software the encapsulated classes and operations (that are encapsulated within the class and state behavior of class) are the unit of testing.

6.7.2 Integration Testing in OO Context

- There are two strategies used for integration testing and those are -
  1. **Thread based testing** 2. **Use-based testing.**
- The **thread based testing** integrates all the classes that respond to one input or event for the system. Each thread is then tested individually.
- In **use-based testing** the independent classes and dependent classes are tested. The **independent classes** are those classes that uses the server classes and **dependant classes** are those classes that use the independent classes. In use based testing the independent classes are tested at the beginning and the testing proceeds towards testing of dependent classes.

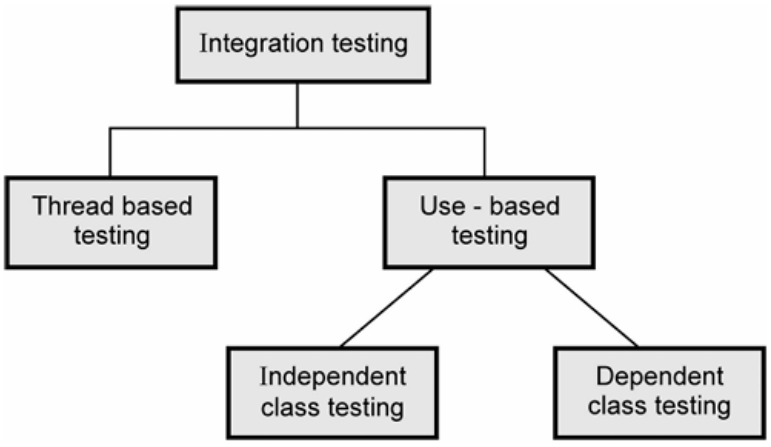


Fig. 6.7.1 Integration testing in OO context

- **Drivers and stub :** In OO context the driver can be used to test operations at lowest level

The **cluster testing** is one step in integration testing in OO context. In this step all the collaborating classes are tested in order to uncover the errors.

54) Define testing? Explain graph based functional testing techniques with suitable graph notation diagram.

Software Testing is defined as a critical element of software quality assurance that represents the ultimate review of specification, design, and coding. The main objective of software testing is to ensure that the software functions appear to work according to the specified requirements and performance expectations .

Graph-Based Functional Testing Techniques:

Graph-Based Testing is a technique in which a graph of objects present in the system is created. This graph is essentially a collection of nodes and links:

- **Nodes** represent the objects involved in the software system.
- **Links** represent the relationships among these objects.
- **Node Weight** represents the properties of an object.
- **Link Weight** represents the characteristics of the relationship between objects.

After creating the graph, the important objects and their relationships are tested. This technique is used to ensure that each part of the system and their interactions are correctly functioning .

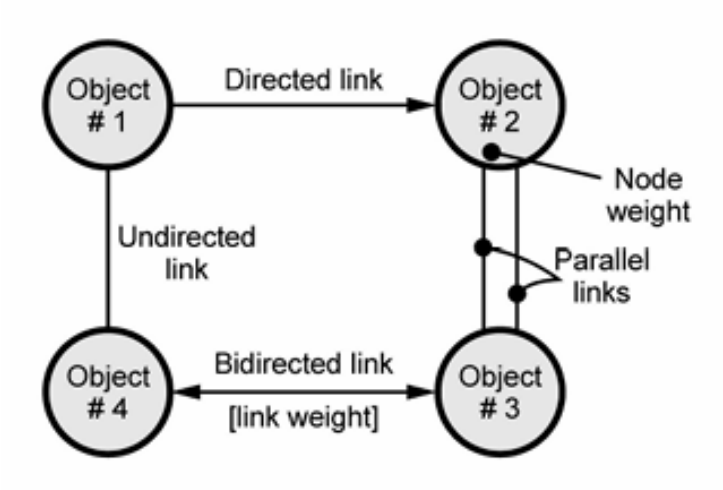


Fig. 6.12.2 Graph notations

55) Discuss User Acceptance Testing

User Acceptance Testing (UAT) is a type of acceptance testing conducted to ensure that the software works correctly in the user's work environment. It is typically the final phase of the software testing process and is performed by the end users or clients to verify that the software meets their requirements and is ready for deployment.

Types of User Acceptance Testing:

1. **Alpha Testing**:
   - Performed at the developer's site.
   - Conducted in a controlled environment under the supervision of developers.
   - The customer tests the complete software in a natural setting.
2. **Beta Testing**:
   - Performed at the customer's site.
   - Conducted without the developer being present.
   - The customer tests the software in an uncontrolled environment.
   - Any issues found are reported to the developer for correction .

Sr. No.	Alpha testing	Beta testing
1	Performed at developer's site	Performed at end user's site
2	Performed in controlled environment as developer is present	Performed in uncontrolled environment as developer is not present
3	Less probability of finding of error as it is driven by developers.	High probability of finding of error as end user can use it the way he wants.
4	It is not considered as live application	It is considered as live application
5	It is done during implementation phase of software	It is done as pre-release of software
6	Less time consuming as developer can make necessary changes in given time.	More time consuming. As user has to report bugs if any via appropriate channel.

56) Discuss Software Testing Life Cycle

Software Testing Life Cycle (STLC)

The **Software Testing Life Cycle (STLC)** is a sequence of specific activities conducted during the testing process to ensure software quality. The primary goal of STLC is to identify bugs, errors, and defects in the software before it goes live.

Phases of STLC:

1. **Requirement Analysis:**
   - Understand the testing requirements.
   - Identify testable requirements.
2. **Test Planning:**
   - Prepare a test plan and strategy.
   - Identify resources and tools required.
   - Define test estimation and schedule.
3. **Test Case Development:**
   - Write detailed test cases.
   - Prepare test data for testing.
4. **Environment Setup:**
   - Set up the testing environment.
   - Configure necessary hardware and software.
5. **Test Execution:**
   - Execute test cases.
   - Log defects if any are found.
6. **Test Closure:**
   - Prepare test summary reports.
   - Archive test results and documentation.

Entry and Exit Criteria:

- **Entry Criteria:** Conditions that must be met to start a particular phase of testing.
- **Exit Criteria:** Conditions that must be fulfilled to complete a phase of testing, like ensuring all critical test cases are passed and all high-priority defects are fixed .

57) What is system testing? Explain any three types system testing

System Testing is a series of tests conducted to fully evaluate the complete software system. It is performed to ensure that the integrated system meets the specified requirements. This type of testing focuses on both the functional and non-functional aspects of the system.

Main Objectives of System Testing:

- To ensure that the system meets the specified requirements.
- To validate the overall performance of the system.
- To check the system's ability to work under various conditions.

Three Types of System Testing:

1. **Recovery Testing**:
   - Verifies the system's ability to recover from failures.
   - The software is intentionally forced to fail, and the recovery process is observed.
   - Checks for reinitialization, checkpoint mechanisms, data recovery, and restart capabilities .
2. **Security Testing**:
   - Ensures that the system's protection mechanisms prevent unauthorized access or data alteration.
   - It verifies that protection mechanisms prevent intrusion (internal or external) and ensure data integrity .
3. **Stress Testing**:
   - Evaluates the system's performance under extreme conditions.
   - It checks how the system behaves when it is subjected to high load or volume.
   - This includes testing the system with a heavy number of transactions, large amounts of data, or high user traffic .

58) Explain with suitable diagram Drivers and stubs in unit test environment.

Drivers and Stubs in Unit Test Environment:

- **Drivers:**
  - These are dummy code used when the upper-level code is not developed.
  - They are used to test lower-level modules.
  - A driver is essentially a main program that calls other modules and helps test them independently.
- **Stubs:**
  - These are dummy code used when the lower-level code is not developed.
  - They are used to test upper-level modules.
  - A stub accepts values from the calling module and returns a null or test value.

Example and Diagram:

- In a **Top-Down Integration Testing**, the main control module is used as a **Test Driver**, and the stubs are substituted for all modules directly subordinate to it.
- In a **Bottom-Up Integration Testing**, lower modules are combined and tested using a **Driver Program**, while stubs simulate higher modules.

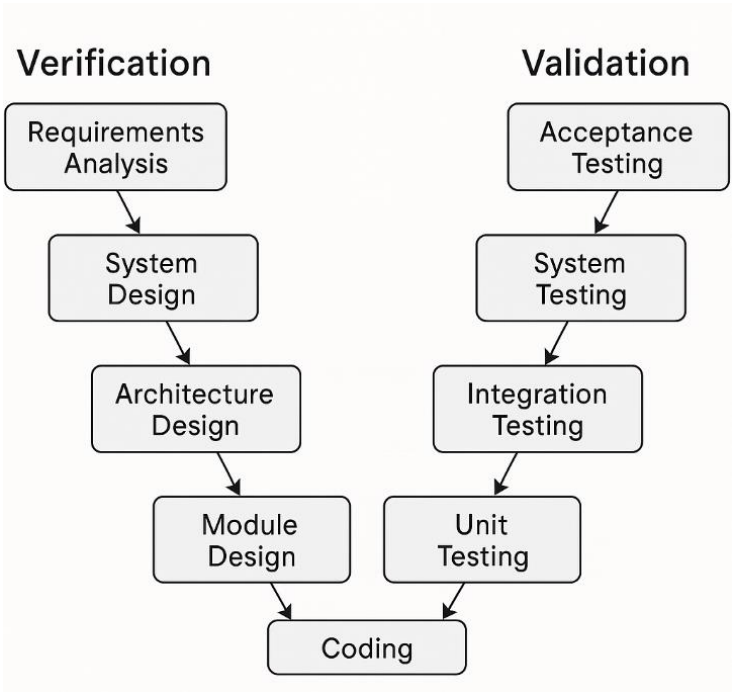
59) Explain phases in Verification and Validation model with suitable diagram.

Verification and Validation (V-Model) Phases:

The **V-Model (Verification and Validation Model)** is a software development model where processes are executed in a sequential manner, and each development phase has a corresponding testing phase. It is called the V-Model because of its V-shaped structure.

Phases in V-Model:

1. **Requirements Analysis:**
   - Verification Phase: Understanding and analyzing customer requirements.
   - Validation Phase: Acceptance Testing (Ensures the system meets customer requirements).
2. **System Design:**
   - Verification Phase: Defining system architecture and design.
   - Validation Phase: System Testing (Validates system functionality and performance).
3. **Architecture Design:**
   - Verification Phase: Designing high-level architecture.
   - Validation Phase: Integration Testing (Tests interface between modules).
4. **Module Design:**
   - Verification Phase: Designing detailed module structure.
   - Validation Phase: Unit Testing (Tests individual modules).
5. **Coding Phase:**
   - This is the base of the V-model, where the actual implementation of software modules is done.



60) Compare manual testing and Automation testing

Aspect	Manual Testing	Automation Testing
Definition	Human testers execute test cases without using tools or scripts.	Uses tools and scripts to automatically execute test cases.
Process	Test cases are executed step-by-step by the tester.	Tests are executed automatically based on predefined scripts.
Speed	Slower, as each test case requires human effort.	Faster, as tests are run automatically.
Cost	Lower initial cost, but more expensive over time for large projects.	Higher initial cost due to setup and script creation, but more cost-effective over time.
Accuracy	Can be prone to human errors.	More accurate, as scripts follow predefined steps exactly.
Flexibility	Flexible for ad-hoc testing, exploring new features, and finding new bugs.	Less flexible, requires script updates when there are changes.
Maintenance	Minimal, but tester needs to track test cases and results.	Requires ongoing maintenance to update scripts for changes.
Test Coverage	Limited by tester's capacity and time.	Can handle a large number of test cases across platforms.
Types of Testing	Best for functional, exploratory, usability, and ad-hoc testing.	Best for regression, performance, and large-scale testing.

All The Best !!

- **Karan Salunkhe ☺**
- **Anish Joshi ☺**