Unit -2

System Model

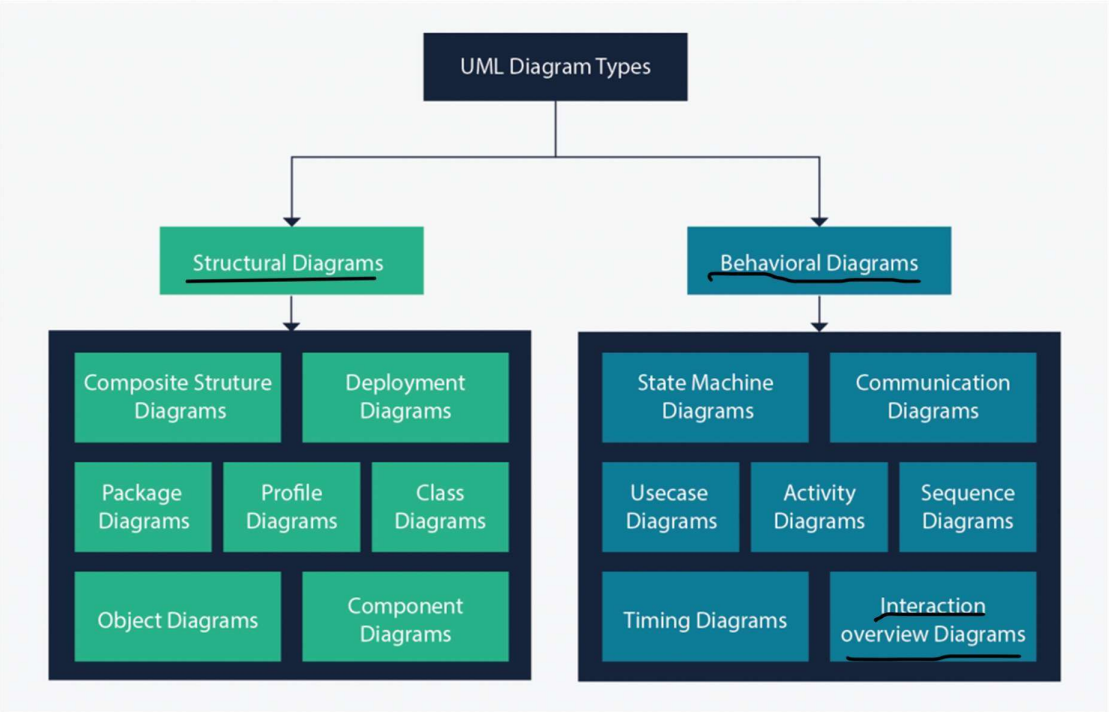
- **Definition**: System modelling is the process of creating simplified, abstract representations of complex systems.
- **Purpose**: Helps understand, analyze, and communicate the **structure, behavior, and interactions** of a system.
- **Process**: Developing abstract models of a system that show different perspectives.
- **Example**: Using **UML (Unified Modeling Language)** diagrams.

Two Views of UML System:

1. **Static (Structural View)**
   - Emphasizes static structure using **objects, attributes, operations, relationships**.
   - Example: **Class Diagram**
   - Describes: System architecture, entities, properties.
2. **Dynamic (Behavioral View)**
   - Emphasizes **dynamic behavior** of system (object interactions, state changes, event flow).
   - Examples: **Sequence Diagram, Activity Diagram, State Diagram, Use Case Diagram**
   - Describes: How system **responds and behaves in different scenarios**.

Graphical Modelling

- **Definition**: Representing system information using diagrams, charts, graphs, and symbols.
- **Why**: Makes it easier to understand concepts and interactions in a visual way.



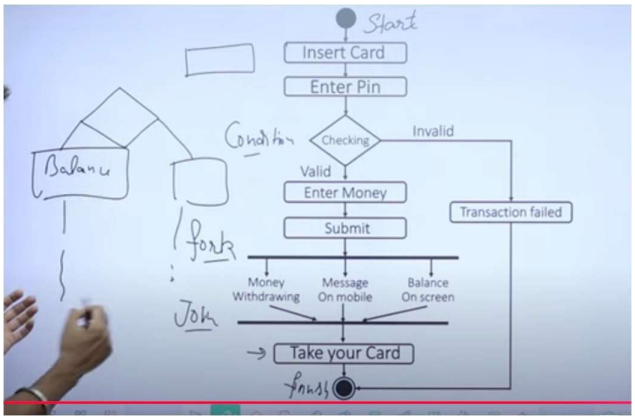
• All UML models are **graphical models**. But not all graphical models are UML (example: flowcharts, ER diagrams, DFDs are graphical models but not UML)

Diagrams used in System Modelling (Software Engineering):

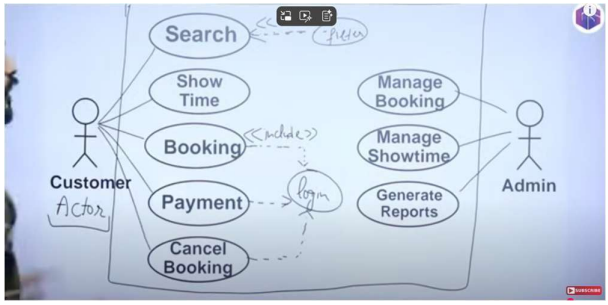
1. Activity Diagram

- Shows activities and processes.
- Focus: flow of tasks and their relationships.

Steps: Draw activity flow of system, Show sequence of activities and Show parallel, branched, concurrent flows



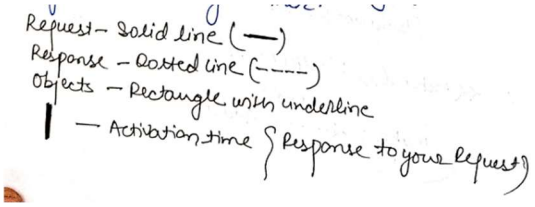
2. **Use-Case Diagram**
    - Shows interactions between a system and its external environment (actors, users).
    - Focus: what actions users can perform.

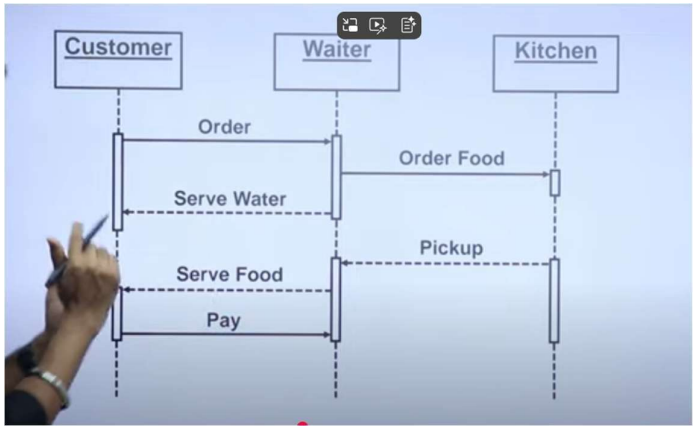


3. Sequence Diagram

- Shows interactions between objects, actors, and system components over time.

- Focus: the order of operations.

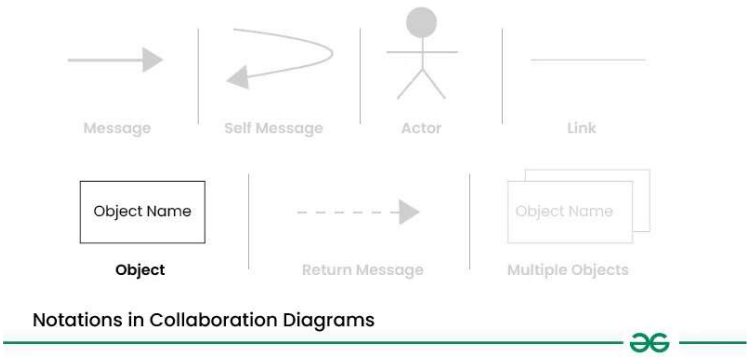




4. **Collaboration Diagram/Communication Diagram**
   - Purpose: Shows interactions between objects, actors, and system components.
   - It is behavioral diagram
   - Focus: the **structural organization** of objects and the **messages exchanged**.
   - Key:
     - Objects are represented as nodes.
     - Links connect objects (showing relationships).
     - Messages are numbered to indicate **sequence/order** of interaction.
   - Used to visualize how objects **work together** to accomplish a task.

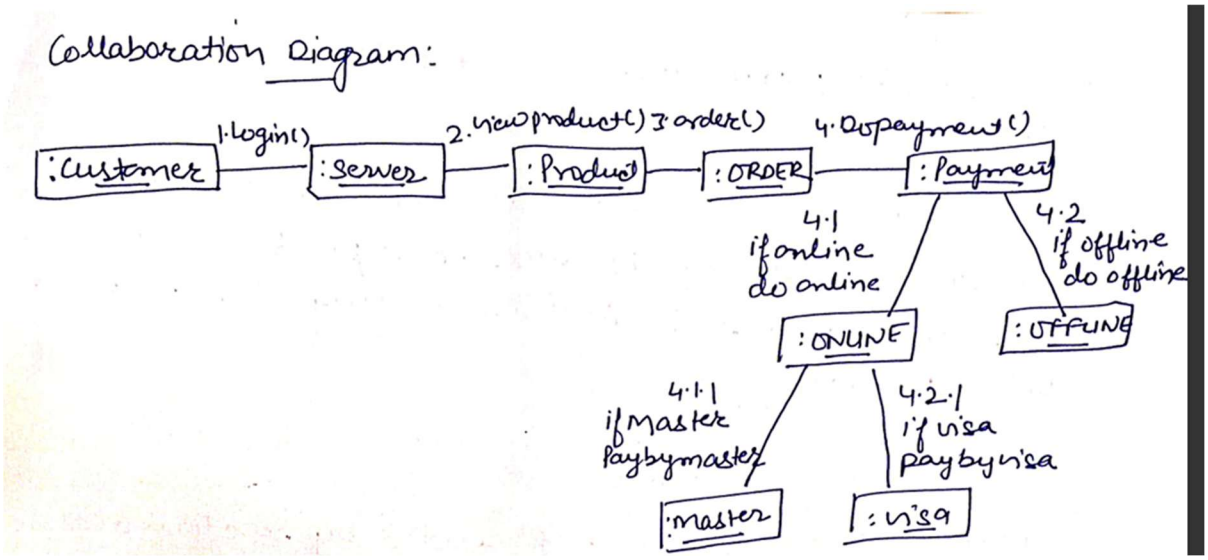
   Note:
   - **Sequence Diagram** → emphasizes **time/order of messages**.
   - **Collaboration Diagram** → emphasizes **relationships between objects**.

   Notations in collaboration diagram



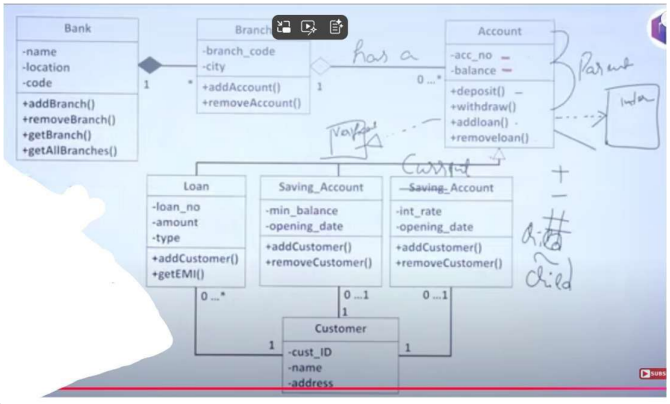
   - **Message** → Solid arrow between objects/actors showing communication.
   - **Self Message** → Arrow looping back to the same object (object calls itself).
   - **Actor** → Stick figure representing an external user/system.
   - **Link** → Straight line connecting objects/actors (possible communication path).
   - **Object** → Rectangle with the object's name inside.

- **Return Message** → Dashed arrow showing a reply/response.
- **Multiple Objects** → Overlapping rectangles indicating a group of objects.

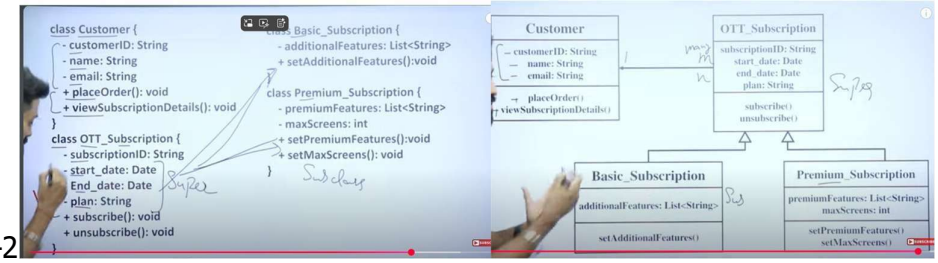


5. Class Diagram

- Represents classes and relationships between them.
- **Class Diagram** → Classes (C), Attributes (A), Relationships (R)
- Focus: **structure** of a system.



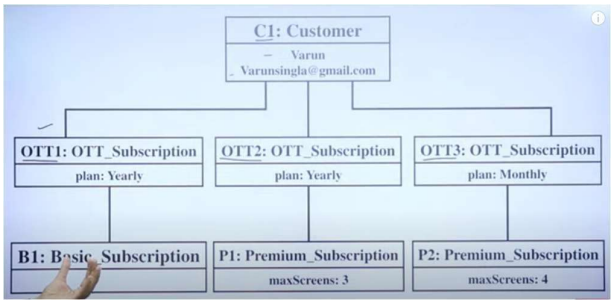
Ex-1



Ex-2

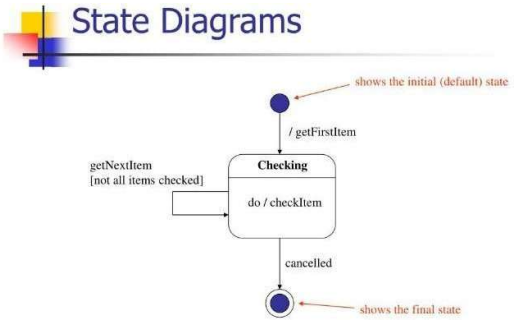
6. Object Diagram (UML)

- **Definition**: An **object diagram** shows a **snapshot of objects (instances)** of classes at a particular moment in time, along with their **attributes and links**.
- It is basically a **"runtime instance"** of a **Class Diagram**.



7. State Diagram

- Shows how a system/object responds to internal and external events.
- Focus: different states and transitions.



Initial State (black filled circle)

- Marks where the state machine starts.
- In this diagram: system starts when /getFirstItem event happens.

State (rounded rectangle)

- Represents a condition or situation in the lifecycle.
- Example: **Checking** (state).
- Inside: do/checkItem means this action is performed while in that state.

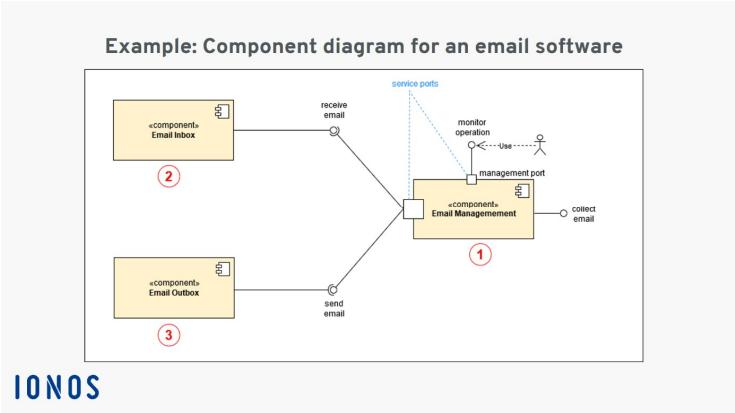
Transitions (arrows)

- Show movement from one state to another, triggered by events.
- Example:
  - getNextItem [not all items checked] → loop transition.
  - cancelled → exit transition.

Final State (bull's eye: black circle inside another circle)

- Marks the end of the lifecycle.
- In this diagram: system ends after *cancelled*.

8. **Component Diagram**
   - Purpose: Represents how system components are organized.
   - Focus: structural organization and implementation details.
   - Key: Components communicate via interfaces; essential for designing complex systems.



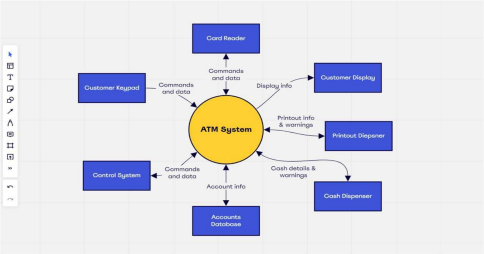
- **Component** 📦
  - Drawn as a rectangle with the component name.
  - May include the UML component icon (two small rectangles on the side).
- **Interface (Provided)** ⭕ (*lollipop symbol*)
  - Shows services a component **offers**.
- **Interface (Required)** 🔌 (*socket symbol*)
  - Shows services a component **needs**.
- **Dependency / Connector**
  - Line connecting required and provided interfaces.
- **Port** ■
  - Small square on a component boundary.
  - Represents a specific interaction point.
- **Node / Subsystem (optional)** 🖥️
  - Used when showing deployment or larger subsystems

In software engineering / system modelling, the main types of models (as in your notes) are:

---

1. Context Model

- Shows the system as a **black box** and how it interacts with external entities (users, other systems).

- The context model is an **external perspective model**
- Example: Context diagram.
- 

---

2. Interaction Model

- Shows interactions between components, users, and the system.

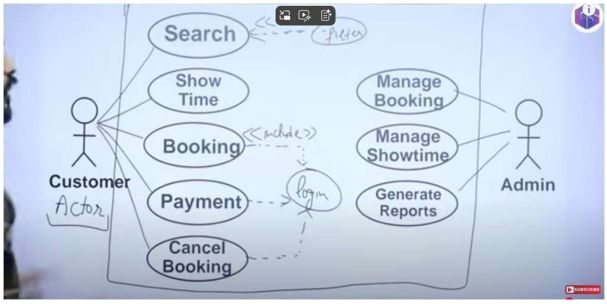
- **Interaction Model** comes under the **Behavioral Model**
- Example: Use-case diagram, sequence diagram.

Use Case Model Components (functional view)

Used in **Use Case Diagrams**

- **System Boundary (big rectangle):** The box labeled **POST** represents the system (Point of Sale Terminal), Everything *inside* is part of the system functionality, everything *outside* are external actors.
- **Actors (stick figures): Cashier** and **Customer** are **actors**.Actors are **external entities** that interact with the system.
- **Use Cases (ovals): Buy Item**, **Log In**, **refund a Purchased Item** are use cases (functionalities the system provides).
- **Associations (lines):** Lines between actors and use cases show **interactions**, Example: Customer interacts with *Buy Item*, *Refund Item*.Cashier interacts with *Log In*.

- **Include** → mandatory reuse of another use case
- **Extend** → optional/conditional extension of another use case



---

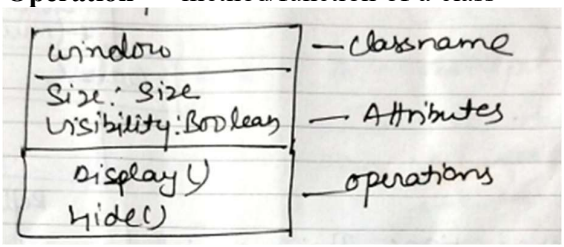
3. Structural Model

- Shows the **organization and structure** of the system's components.
- Example: Class diagram, component diagram.

Structural Model Components (static structure)

Used in Class, Object, Component, Deployment, Package diagrams

- **Class** → blueprint of objects
- **Object** → instance of a class
- **Attribute** → data member of a class (e.g., name, age)
- **Operation** → method/function of a class



Constraints in UML

- A **constraint** is a **condition or restriction** that must be true for the system to be valid.
- It's like a **rule** applied to UML elements (class, attribute, operation, relationship, etc.).
- Written inside **curly braces { }**.
- Expressed in **natural language** or in **OCL (Object Constraint Language)**.

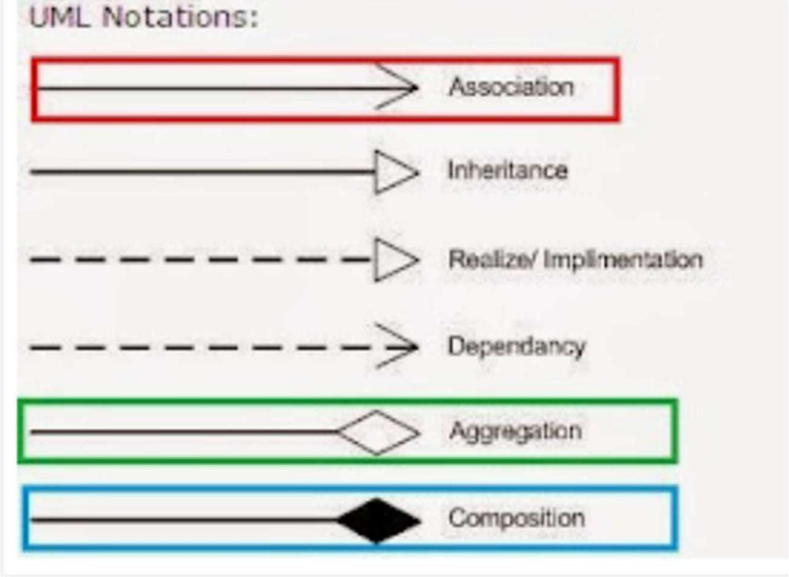
Examples:

1. Attribute constraint
   - {age > 0} → attribute *age* must always be positive.
2. Multiplicity constraint
   - 1..* → at least one object must exist.
3. Relationship constraint
   - {ordered} → an association must maintain order.
   - {unique} → no duplicate values allowed.
4. General rule constraint
   - {salary <= manager.salary} → An employee's salary must be less than or equal to their manager's salary.

◆ Rules vs Constraints

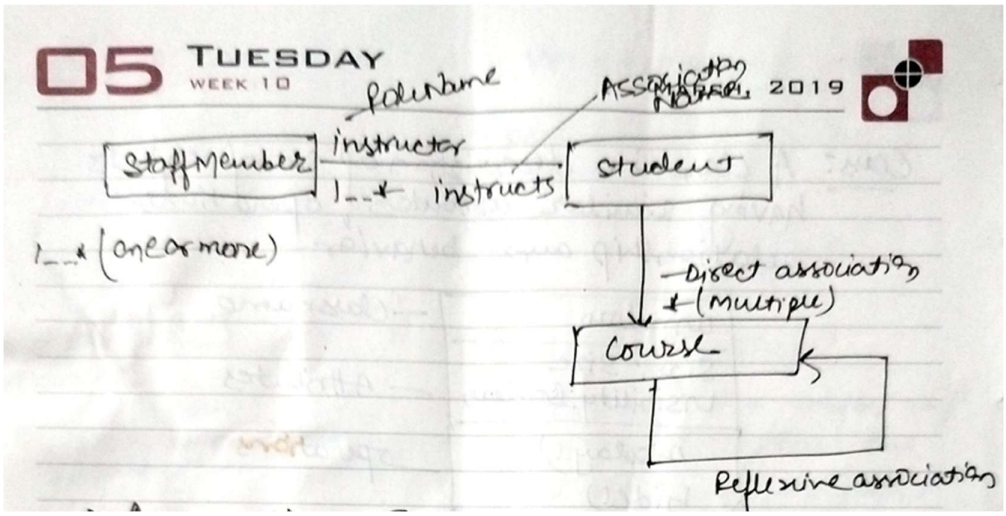
- **Rules** = general guidelines or policies of the system (business logic).
- **Constraints** = strict formal conditions that enforce those rules in UML models.

Relationships in Class Diagrams



1. Association

- Definition: A structural relationship that represents how two classes are connected.



types

◆ 1. Directed Association (Navigable Association)

- Shows which direction the relationship can be followed.
- Arrowhead points from the source class to the target class.
- Example:
  Customer → Order (Customer *places* Order)

◆ 2. Reflexive Association

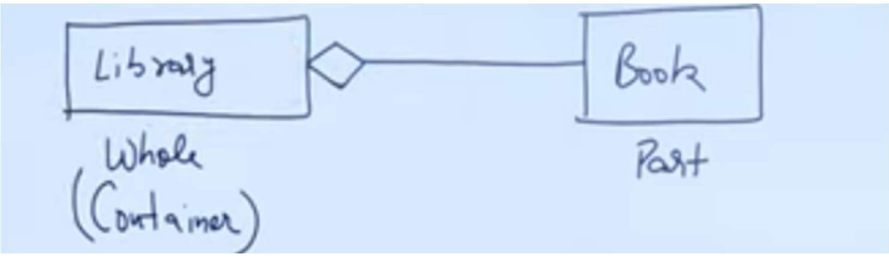
- A class is associated with itself.
- Used when objects of the same class are related.
- Example:
  Employee → Employee (Employee *manages* another Employee)

◆ 3. Multiplicity Association

- Defines **how many objects** can participate in the relationship.
- Shown as numbers at the ends of the association line.
- Common notations:
  - 1 (exactly one)
  - 0..1 (zero or one)
  - * (many)
  - 1..* (one or more)
- Example:
  Customer 1 --- * Order
  (One Customer can place many Orders)

2.Aggregation

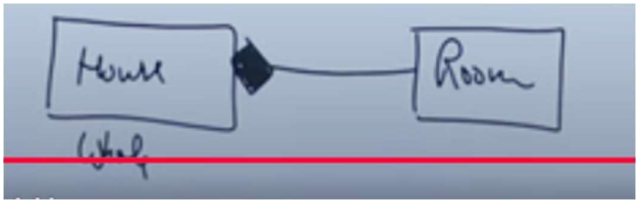
- Definition: A special form of Association.
- Represents a **whole–part relationship** where parts can exist independently of the whole.
- Example:
  - *Car has Wheels* (wheels can exist without the car).
- Notation: Hollow diamond at the "whole" end.



3.Composition

- Definition: A **stronger form of Aggregation**.

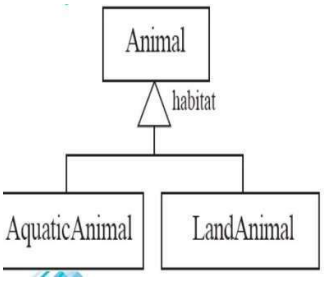
- The parts' lifetimes depend on the whole (if the whole is destroyed, parts are destroyed).
- Example:
  - *House has Rooms* (if house is destroyed, rooms also go).
- Notation: Filled (black) diamond at the "whole" end.



---

4. Generalization (Inheritance)

- Definition: A hierarchical relationship.
- One class (child) inherits attributes and operations from another (parent).
- Example:
  - *Dog and Cat inherit from Animal*.
- Notation: A line with a hollow triangle pointing to the parent class.



---

5. Dependency

A **weak relationship** between two elements.

Means: *one element depends on another*, but only temporarily (not permanently linked).

Shown with a **dashed arrow**.

Example:

A ReportGenerator class depends on a Database class to fetch data.

If the database changes, the report generator might also be affected.

Think of it as: *"uses temporarily"*.

6. Realization

A relationship between an **interface** and a **class** (or component) that **implements** that interface.

Shown with a **dashed line + hollow triangle arrowhead**.

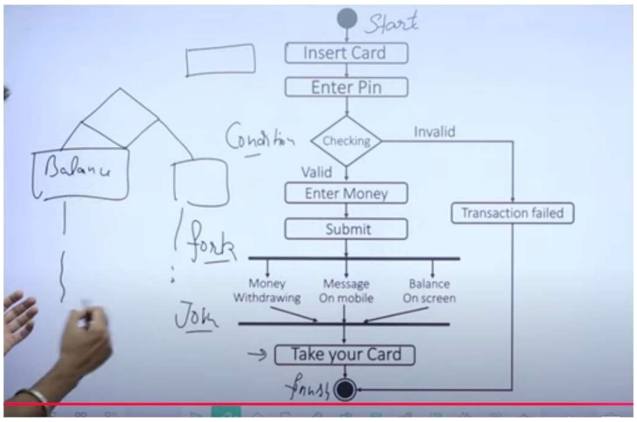
Example:

> PaymentService (interface) ← PayPalPayment (class implements it)
>
> PayPalPayment *realizes* PaymentService.

Think of it as: *"implements / fulfills the contract"*.

---

4. Behavioral Model

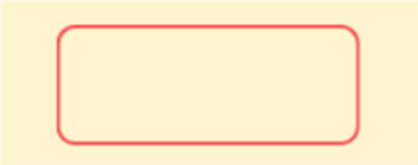
- Shows the **dynamic behavior** of the system in response to inputs/events.
- Example: State diagram, activity diagram.
- 

Behavioral Model Components (dynamic behavior)

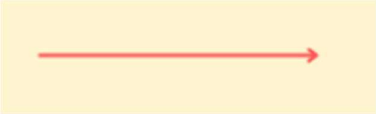
- Covers Activity, Sequence, Communication, State, Timing diagrams

a) Activity Diagram

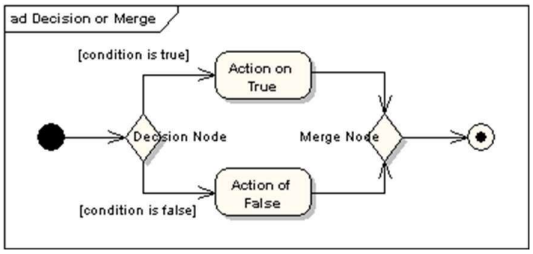
- **Activity Diagram** → Rounded rectangle that Represents a task to be performed. Example: *Login*, *Process Payment*.



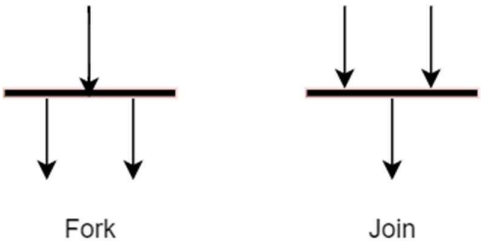
- **Control/Transition (Flow)** → Arrow that Shows the flow of control from one activity to the next. Example: After *Login*, flow goes to *View Dashboard*.



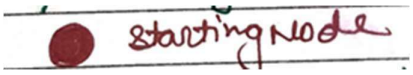
- **Decision Node** → branching (if/else)
- **Merge Node** → merge branches back



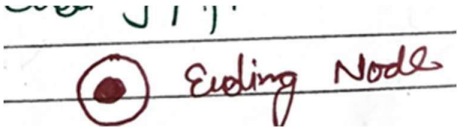
- **Fork Node** → split into parallel flows
- **Join Node** → synchronize parallel flows back into one



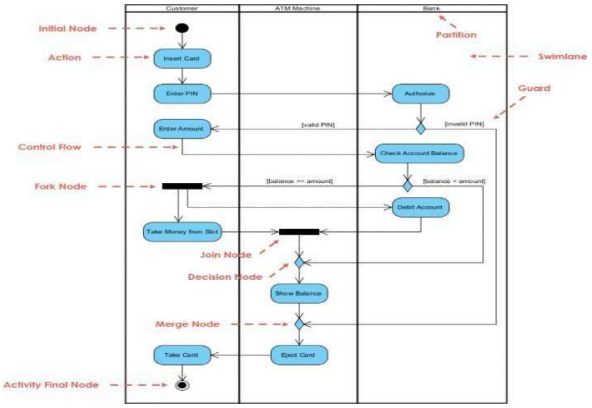
- **Initial Node** → start (filled black circle)



- **Final Node** → end (bullseye circle)



- **Swimlane** → partitions showing responsibility (User vs System)



b) State Machine Diagram

- **State** → condition of an object
- **Transition** → movement between states
- **Event/Trigger** → causes transition

c) Interaction Diagrams (Sequence/Communication)

- **Lifeline** → represents an object/actor
- **Message** → communication between lifelines

---

5. Model-Driven Engineering (MDE)

- ## Definition:
  MDE is an approach to software development where **models** (not programs) are the **main outputs** of the development process.
- ## Key Idea:
  Instead of focusing on writing source code, developers create **abstract models** that describe the system.
- ## Benefits:
  - Raises the **level of abstraction** in software engineering.
  - Engineers don't need to worry too much about programming language details.
  - Reduces dependency on specific execution platforms.

🟩 In short:
MDE = *Models are the primary focus → Code can be generated automatically from models → Engineers focus more on design, less on low-level programming detail*

UNIT -3

◇ Software Architecture

Definition:
Blueprint/structure of a software system → shows **components, relationships, data/control flow, and dependencies**.

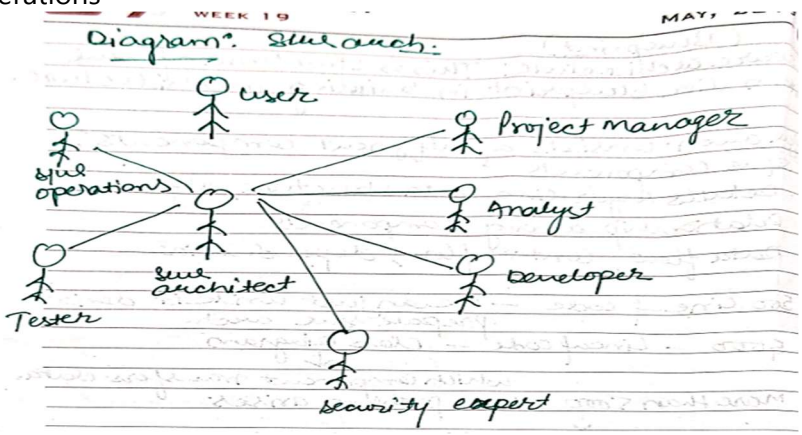
Key Elements

1. **Components** → building blocks (objects, algorithms, functions).
2. **Connectors** → links that enable communication among components.
3. **Configuration/Topology** → arrangement of components + connectors.
4. **Design Models** → represent architecture (UML, class diagrams, use-case diagrams).

Why Important?

- Provides **clarity** → structured, reusable, maintainable code.
- Ensures **functionality, performance, flexibility, scalability, security**.
- Essential for **large programs**.

Stakeholders in Software Architecture
User, Project Manager, Analyst, Developer, Tester, Security Expert, System Architect, Site Operations

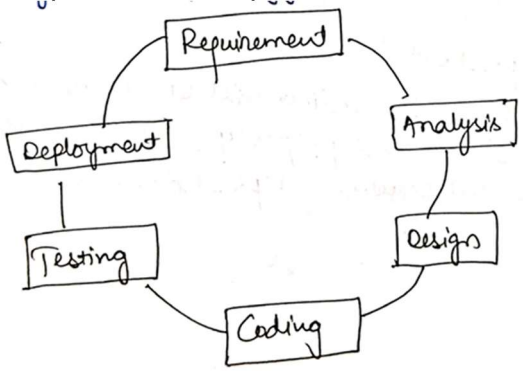


➡️ Each has different concerns but all influence architecture.

Software Design Process

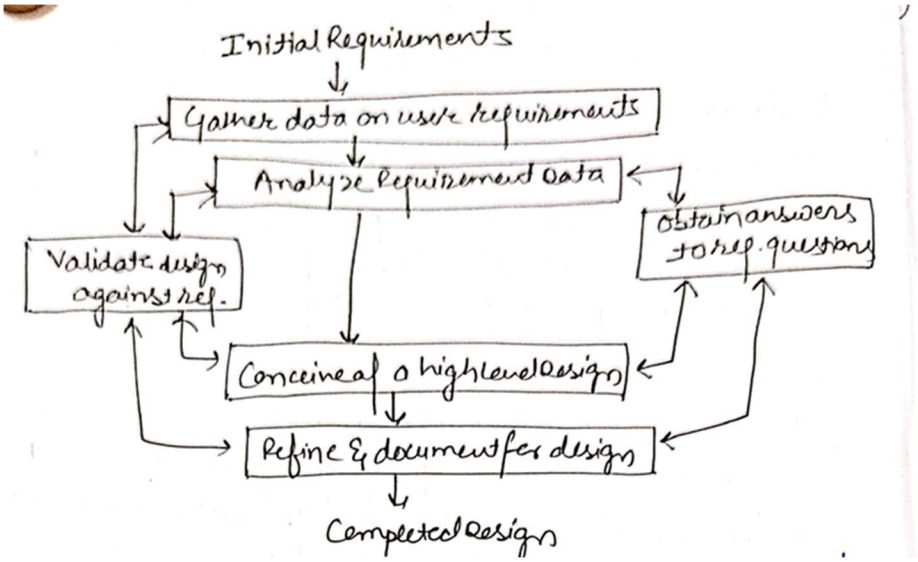
Software design is the step in software engineering where we **translate requirements (what the system should do)** into a **blueprint (how the system will do it)**. It comes

after requirements analysis and before coding.



Steps in the Design Process:

1. **Requirement Analysis Review**
    - Revisit the SRS (Software Requirement Specification).
    - Clarify functional and non-functional requirements.
2. **System Design (High-Level Design)**
    - Divide the system into major modules or components.
    - Define architecture, data flow, control flow, and interfaces.
3. **Detailed Design (Low-Level Design)**
    - Describe each module in detail (algorithms, data structures, database schema).
    - Create class diagrams, ER diagrams, pseudo-code, etc.
4. **Design Review**
    - Evaluate design for correctness, efficiency, and maintainability.
    - Identify risks and possible improvements.
5. **Documentation**
    - Maintain design documents (HLD, LLD, interface specifications).
    - Ensure it's ready for implementation.



◇ Objectives/Characterstics of Software Design

The **main objective** is to transform "what" needs to be built into "how" it will be built.
More specifically:

1. **Correctness** → The design should correctly implement all requirements.
2. **Efficiency** → Optimize performance, resource usage, and response time.
3. **Modularity** → Divide the system into independent, reusable modules.
4. **Maintainability** → Make the system easy to modify, debug, and extend.
5. **Reusability** → Encourage reuse of existing components and patterns.
6. **Scalability** → Ensure the system can handle growth in data, users, or features.
7. **Reliability & Security** → Protect against failures and vulnerabilities.
8. **Simplicity & Clarity** → Keep the design understandable for developers.

Levels of Software Design process

1. Interface Design

- Specifies **interaction between system and environment** (users, devices, other systems).
- System treated as a **black box** (internals ignored).
- Includes:
    - Incoming events/messages (inputs).
    - Outgoing events/messages (outputs).
    - **Data formats** of inputs/outputs.
    - **Ordering & timing** of inputs and outputs.

---

2. Architectural Design

- Specifies **major system components** and their **interactions**.
- Focus on **structure**, not internal details.
- Includes:
    - Decomposition into components.
    - Allocation of responsibilities.
    - Component **interfaces**.
    - Performance, scalability, reliability.
    - Communication between components.

---

3. Detailed Design

- Specifies **internal details** of each component.
- Translates architecture into **ready-to-code design**.
- Includes:
    - Decomposition into **program units**.
    - Allocation of functions to units.
    - **User interfaces**.
    - **States and transitions**.
    - Data packaging, scope & visibility.
    - **Algorithms and data structures**.

◇ Software Design Quality

A **good software design** should not only meet requirements but also be efficient, maintainable, and reliable.
High-quality design ensures:

- **Correctness** → satisfies requirements.
- **Efficiency** → uses resources optimally.
- **Maintainability** → easy to modify, fix, extend.
- **Usability** → simple for end-users and developers.
- **Scalability & Reusability** → supports growth and reuse.

◇ Characteristics of a Good Software Design

A design is considered good if it has the following characteristics:

1. **Correctness** → Implements all specified requirements.
2. **Understandability** → Easy to read and comprehend.
3. **Efficiency** → Minimizes time, memory, and resource usage.
4. **Modularity** → System is divided into independent modules.
5. **Flexibility** → Easy to modify without major changes.
6. **Maintainability** → Supports debugging, testing, and updates.
7. **Reusability** → Components can be reused in other projects.
8. **Simplicity** → Avoids unnecessary complexity.
9. **Security & Reliability** → Protects data and ensures stable operation.
10. **Scalability** → Handles increasing load or future expansion.

◇ Guidelines for Good Software Design

To achieve high-quality design, follow these guidelines:

1. **Follow modularity** → Break system into independent, well-defined modules.
2. **Use abstraction** → Hide unnecessary details, show only essential features.
3. **Ensure high cohesion & low coupling** → Each module should focus on one task (cohesion), and dependencies between modules should be minimal (coupling).
4. **Follow design principles** → Such as *Separation of Concerns*, *Single Responsibility*, and *Information Hiding*.
5. **Use standard design notations** → e.g., UML diagrams, flowcharts, DFDs.
6. **Anticipate future changes** → Keep the design flexible and maintainable.
7. **Design for testing** → Make sure modules are testable independently.
8. **Balance trade-offs** → Between performance, cost, and simplicity.
9. **Ensure consistency** → Uniform naming, style, and documentation.
10. **Document design properly** → High-Level Design (HLD) + Low-Level Design (LLD).

◇ Software Design Concepts / **Principles of Design**.

Software design concepts are the **principles and strategies** that help in creating **high-quality, maintainable, and efficient software systems**.

---

1. Abstraction

- Focus on **essential details**, hide unnecessary complexity.
- Types:
  - **Procedural Abstraction** → what a function does.
  - **Data Abstraction** → defines data + operations, hides representation.
  - **Control Abstraction** → hides control flow details.

---

2. Refinement

- Stepwise detailing from **high-level design** → **low-level design**.
- Example: "Process Order" → "Check Stock" → "Calculate Bill" → "Generate Invoice".

---

3. Modularity

- Divide system into **independent modules**.
- Benefits: easy to develop, test, maintain, and reuse.

---

4. Architectural Models

1. **Structural Model**
   - Shows the **static structure** of the system.
   - Components, modules, layers, interfaces.
   - Example: UML Class Diagram, Component Diagram.
2. **Framework Model**
   - Provides **standards, templates, and guidelines** for architecture.
   - Examples: TOGAF, Zachman, 4+1 View Model.
3. **Dynamic Model**
   - Shows the **runtime behavior** of the system.
   - How objects/components interact.
   - Example: UML Sequence Diagram, Activity Diagram, Statechart.
4. **Process Model**
   - Defines **how architecture is developed** (lifecycle).
   - Examples: Waterfall, Spiral, Iterative, Agile models.

---

5. Information Hiding

- Each module hides **internal details**, exposes only what's necessary.
- Reduces dependency → improves security and maintainability.

---

6. Functional Independence

1. Modules should work as independently as possible.
2. Achieved by:
    - **High Cohesion** (strong relation within a module).
    - **Low Coupling** (minimal dependency between modules).

7.**Pattern**

A pattern in general refers to a reusable solution to a commonly recurring problem.
In the design process, a pattern captures proven approaches that can be applied when similar issues arise.
Each design pattern tries to focus on:

- **Applicability – whether the pattern is relevant to the current work.**
- **Reusability – whether it can be adapted to solve similar functional challenges.**
- **Guidance – serving as a blueprint or reference for developing solutions with consistent structure and functionality.**

---

8. Design Classes (in OOP)

- Represent elements in object-oriented design.
- Types:

1. Entity Classes (Business Classes)

- Represent **real-world objects / business data**.
- Store attributes and business rules.
- Examples: *Customer, Product, Order, Invoice*.

---

2. Boundary Classes (User Interface Classes)

- Manage **interaction with external agents** (users, devices, other systems).
- Provide input/output between system and environment.
- Examples: *LoginForm, DashboardUI, API Interface*.

---

3. Control Classes (Process Classes)

- Handle **workflow, logic, and state transitions**.
- Act as a **mediator** between Boundary and Entity classes.
- Examples: *OrderProcessor, PaymentController*.

---

4. Persistence Classes

- Deal with **data storage and retrieval**.
- Encapsulate database or file operations.
- Examples: *DatabaseManager, FileHandler, ORM classes*.

---

5. System Classes

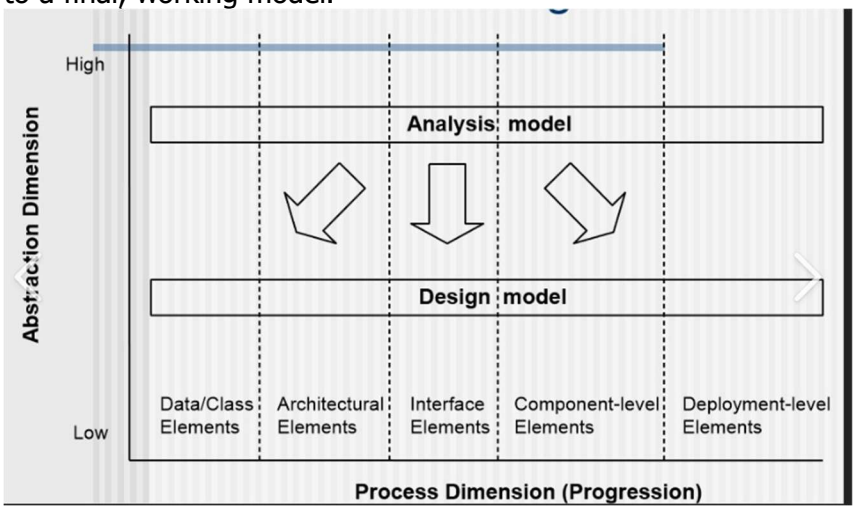
- Provide **general system services** (utilities, security, logging).
- Support overall system operation.
- Examples: *Logger, SecurityManager, NotificationService*.

9. Refactoring

- **Improving existing code design** without changing functionality.
- Makes code more readable, efficient, and maintainable.

Design Model

The design model is a blueprint that translates requirements into a detailed plan for software development. It helps systematically evolve a project from high-level requirements to a final, working model.



Two Dimensions of the Design Model

- **Process Dimension:** Describes how the design model evolves over time during software development. For example, starting with a high-level analysis model, which is gradually developed, refined, and detailed into the final model.
- **Abstraction Model:** Describes how detailed or abstract each design element is at any given point. Initially, designs are abstract (high-level), and become more detailed as development progresses.

Five Components of the Design Model

1. **Data Design:** Explains how data flows through the system, including data structures, databases, and relationships. Example: a student portal.

2. **Architectural Design:** Describes the overall structure—modules, layers, and how components interact. Example: a web application with presentation/UI, business logic, and data access layers.
3. **Interface Design:** Describes how users will interact with the system. Example: login screens or backend APIs.
4. **Component Level Design:** Focuses on the internal logic of each module. Example: the logic for calculating CGPA in a student portal.
5. **Deployment Level Design:** Describes how the system will be deployed across hardware systems, including server setups, cloud, database locations, etc.

Architectural Design Model

The architectural design model acts as a blueprint for a system, helping guide early design decisions regarding software assets, architectural styles, and system patterns. It involves selecting hardware and software components, connectors, integration conditions, and semantic models that clarify the system's properties.i.e

- Specifies required hardware/software modules, such as databases and function-specific components.image.jpg
- Describes connectors for coordination and communication between modules.
- Details how components integrate to form the overall system.
- Provides semantic models to help designers understand system properties.

Importance of Architectural Design

- **Security:** Includes safeguards like encryption against malicious users.image.jpg
- **Performance:** Addresses how the system handles requests and responses efficiently.
- **Maintainability:** Ensures components are modular and easily replaceable for future changes.
- **Safety:** Prevents errors and improves communication between system parts.
- **Availability:** Prepares the system to handle various errors through well-designed components.

Key Decisions in Architectural Design

- Distribution of the system across networks.
- Approaches for structuring the system (which styles to use).
- Methods for documenting the architecture.
- Decomposing the system into modules.
- Control strategies for component operation.
- Analyzing the architectural design.

Taxonomy of Architectural Styles/ Types of Software Architectural Styles

---

1. Data-Centered Architecture

- A **central data store (repository/blackboard)** is accessed by multiple client components.
- Clients perform **read/write/update** operations on the data.
- **Blackboard mechanism** → notifies clients when relevant data changes.

✅ Advantage:

- Centralized repository makes data easy to manage.
- Easy to add or modify client components.

❌ Disadvantage:

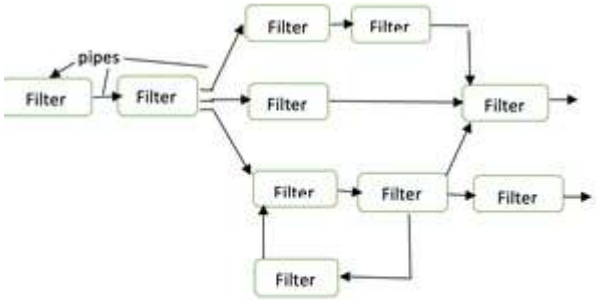
- Single point of failure.
- Repository may become a bottleneck.

📌 **Example:** Database systems, Blackboard systems.

---

2. Data-Flow Architecture

- Input data is transformed into output data via a **series of processing components (filters)** connected by **pipes**.
- Each filter works independently (no knowledge of others).
- **Batch Sequential** → if it becomes a single sequence of transformations.



✅ Advantages:

- Supports reuse of filters (independent components).

- Encourages parallel/concurrent execution.

❌ Disadvantages:
- Not suitable for highly interactive applications.
- Synchronizing multiple data streams is difficult.

📌 **Example:** Compilers, Data Processing Systems, Signal Processing.

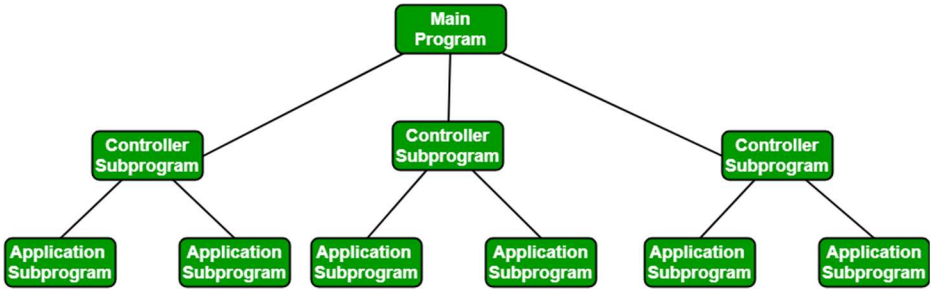
---

3. Call-and-Return Architecture
- Structure is based on **procedure calls**.
- Easy to scale and modify.
- Two subtypes:
    - **Main Program–Subprogram Architecture:** hierarchical control, subprograms invoked by main program.
    - **Remote Procedure Call (RPC):** procedure calls executed across different computers in a network.

✅ Advantage:
- Easy to understand and implement.
- Supports modular program design.

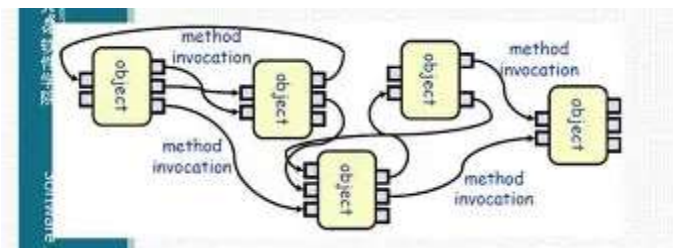
❌ Disadvantage:
- Tight coupling between caller and callee.
- Rigid hierarchy makes changes difficult.

📌 **Example:** Traditional programming languages (C, Pascal), Distributed systems using RPC.



---

4. Object-Oriented Architecture
- System is a collection of **objects** that encapsulate both **data and behavior**.
- Objects interact using **message passing**.



✅ **Characteristics:**

- Objects protect system integrity.
- Objects are independent (don't know others' internal details).

✅ Advantage:

- Promotes reusability through inheritance/polymorphism.
- Easy to maintain and extend.

❌ Disadvantage:

- Can be complex for small/simple systems.
- Performance overhead due to abstraction.

📌 **Example:** OOP-based systems (Java, C++, Python applications).

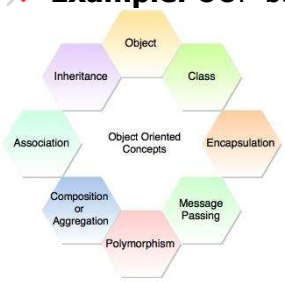


Fig. Object Oriented Concepts

Core Concepts of oops

1. **Object**
    - Instance of a class, contains **state (attributes)** + **behavior (methods)**.
    - Real-world entity (e.g., *Student, Book*).
    - State example: A *Book* object → "available", "checked out".
2. **Class**
    - Blueprint/template for objects.
    - Defines **attributes** and **behaviors** shared by all instances.
3. **Encapsulation**
    - Hiding internal details, exposing only via interfaces.
    - Data + methods bundled in a single unit.
4. **Polymorphism**
    - "Many forms" → same operation behaves differently for different objects.
    - Example: *draw()* method for *Circle, Rectangle, Triangle*.
5. **Inheritance**
    - Reuse by deriving new classes (child/subclass) from existing ones (parent/superclass).
    - Subclass inherits attributes & methods, can extend/override them.

- Example: *Shape* → subclasses *Rectangle*, *Triangle*.

6. **Message Passing**
    - Objects communicate by **sending messages** (method calls with parameters).
7. **Association**
    - Relationship between objects (1–1, 1–many, many–many).
    - Example: *Teacher* ↔ *Student*.
8. **Aggregation vs Composition**
    - **Aggregation**: "Has-a" relationship, weaker bond (child can exist independently).
        - Example: *Department* has *Professors*.
    - **Composition**: Stronger bond, child object's lifecycle depends on parent (death relationship).
        - Example: *House* → *Rooms*.

---

5. Layered Architecture

- System is divided into **layers**, each layer provides services to the higher one.
- **Outer layers:** user interface.
- **Middle layers:** utility services, application logic.
- **Inner layers:** OS interaction, hardware communication.
- Clear separation of concerns.
- Enhances portability and reusability.

❌ Disadvantage:

- Performance overhead (many layers).
- Difficult to assign responsibilities to layers.

📌 **Example:** OSI-ISO Network Model, Operating Systems, Web Application stacks (UI → Service → Database).

Unit -4

☑ Strategic Approach to Software Testing

A **software testing strategy** is a high-level plan describing *how* testing will be organized, *what* will be tested, *when*, and *by whom*. A good strategy ensures quality, reduces risk, and aligns testing with project goals.

---

1. Understand the Product and Requirements

Before designing tests, gather:

- Functional requirements
- Non-functional requirements (performance, security, usability, etc.)
- Architecture and design documents
- Constraints & risks

Goal: Identify *what needs to be validated* and *potential failure points*.

---

2. Identify Risks (Risk-Based Planning)

Testing resources are always limited. Prioritize based on:

- Complexity of modules
- Impact of failure
- Likelihood of defects
- Integration dependencies

High-risk areas get deeper and earlier testing.

3. Define the Overall Testing Approach

Decide the types of testing needed:

✓ Functional testing:

- Unit
- Integration
- System
- User acceptance testing (UAT)

✓ Non-functional testing:

- Performance / Load
- Security
- Reliability
- Compatibility
- Usability

---

4. Establish the Test Levels

Level 1 — Unit Testing

Performed by developers to validate small components.
Tools: JUnit, NUnit, pytest.

Level 2 — Integration Testing

Ensures modules work together correctly.
Approaches:

- Top-down
- Bottom-up
- Big-bang
- Sandwich (hybrid)

Level 3 — System Testing

End-to-end validation of the entire software.

Level 4 — Acceptance Testing

Performed by stakeholders/business to verify readiness.

---

5. Choose Test Design Techniques

Black-box testing

- Boundary value analysis
- Equivalence partitioning
- State transition testing
- Use case testing

White-box testing

- Path coverage
- Branch coverage
- Loop testing

Experience-based

- Exploratory testing
- Error guessing

---

6. Define the Test Environment & Tools

- Identify hardware/software platform
- Test data strategy
- CI/CD integration (Jenkins, GitHub Actions)
- Automation tools (Selenium, Cypress, Playwright)

---

7. Define Entry and Exit Criteria

Entry criteria examples:

- Requirements approved
- Test environment ready
- Unit testing completed

Exit criteria examples:

- All critical defects resolved
- Test coverage goals met
- Successful regression results

---

8. Test Execution & Defect Management

- Execute according to a test plan

- Log defects with severity & priority
- Verify fixes and perform regression testing
- Track metrics (pass %, defect arrival rate, burn-down)

---

9. Test Reporting & Continuous Improvement

- Test summary reports
- Lessons learned
- Process improvements for future projects

⭐ Common Test Strategies for Conventional Software

Below are typical strategies used in standard (non-AI, non-ML) software testing:

---

1. Analytical Strategy

Testing is based on:

- Requirements analysis
- Risk analysis
- Model-based testing

Used when formal documentation exists.

---

2. Model-Based Strategy

Uses models such as:

- State machines
- Activity diagrams
- Decision tables
- UML Use cases

Tests are derived systematically.

---

3. Methodical Strategy

Follows industry standards or checklists:

- Security checklists
- Accessibility standards (WCAG)
- Payment card standards (PCI-DSS)

---

4. Process-Compliant Strategy

Aligns testing with a process model:

- Waterfall testing stages
- V-Model
- Formal QA cycles

---

5. Reactive Strategy

Tests are created *after* the software is delivered.

Used when:

- Requirements are unclear
- Agile deployments where features evolve

---

6. Consultative/Directed Strategy

Subject matter experts or stakeholders decide priorities.

---

7. Regression-Avoidance Strategy

Use of automation to minimize regression risk.
Examples:

- Automated smoke tests
- Daily end-to-end automated suites

Blackbox testing vs white box testing

Aspect	White Box Testing	Black Box Testing
Knowledge Required	Requires knowledge of code and internal logic	No knowledge of code; focuses on functionality

Aspect	White Box Testing	Black Box Testing
Focus	Internal structure, paths, and logic	External behavior and outputs
Performed By	Developers / technically skilled testers	QA testers / end users
Test Basis	Source code, algorithms, control flow	Requirements, specifications, use cases
Test Levels	Unit and integration testing	System and acceptance testing
Type of Errors Found	Logical errors, hidden bugs, code faults	Functional defects, missing features, usability issues
Coverage	High coverage of internal code	Limited internal coverage
Tools Used	Debuggers, code analyzers, coverage tools	Test management tools, automation tools
Advantages	Optimizes code and internal quality	Represents real user scenarios
Disadvantages	Time-consuming and requires coding skills	Cannot detect internal logic flaws

---

✅ **Example of White Box Testing**

Example:

A function in code calculates the sum of two numbers:

```
def add(a, b):
    return a + b
```

A white-box tester will:

- Check all possible internal paths (e.g., what happens with positive, negative, or zero values).
- Verify that the return statement correctly adds the values.
- Ensure no unreachable code is present.

This test is based on the internal code.

✅ **Example of Black Box Testing**

Example:

A login page requires:

- Correct username
- Correct password

A black-box tester will test:

- Enter valid username & password → login should succeed
- Enter invalid username → login should fail
- Enter invalid password → login should fail

This test is based on inputs/outputs, without seeing the code.

Debugging

- Debugging means **finding and fixing errors (bugs)** in a program.
- It usually starts after a **test fails**, helping locate the exact cause of the error.

- Debugging tools let you **run the program step-by-step**, inspect variables, and observe behaviour to identify issues.

Types of Errors

1. Syntax Errors: Mistakes in the code structure (e.g., missing semicolons).
2. Runtime Errors: Problems that occur while the program is running.
3. Logical Errors: The program runs but produces incorrect results due to a logic mistake.

Debugging Tools

- IDE Debuggers: Tools like CodeBlocks, VSCode, Eclipse.
- Print Statements: To check the values of variables at different points.
- Error Messages: Information from the compiler/interpreter to locate issues.

Steps to Debug

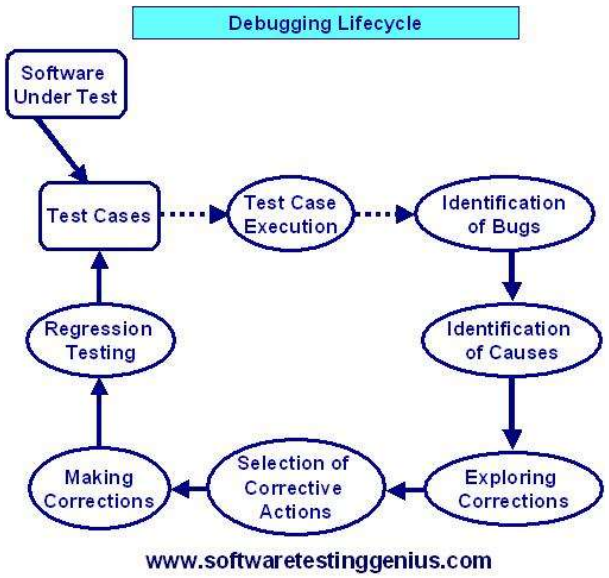
1. Identify the problem from error output.
2. Reproduce the error and note where/how it happens.
3. Fix the bug.
4. Retest the program to confirm it works.

Approaches to Debugging

1. Brute Force Debugging:
   Run programs, check lists, logs, and traces to let the computer "find the error." Common but least efficient.
2. Backtracking:
   Manually trace the source code backward from where the error occurred. More practical for small programs.
3. Cause Elimination:
   Make a hypothesis about what might cause the error, then test changes by eliminating possible causes until the real one is found.

Debugging Process Flow

- Run test cases.
- See test result.
- If there's an error, debug to find and identify the cause.
- Apply fixes and rerun tests (additional tests).
- Repeat until successful.



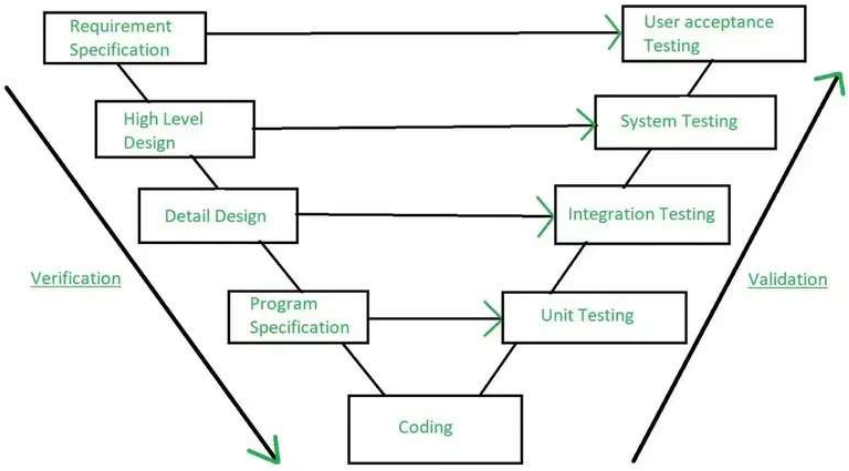
Debugging as an art

- **Creative thinking** to find hidden issues
- **Pattern recognition** to spot clues
- **Attention to detail** for tiny mistakes
- **Iterative refinement** like improving a sketch
- **Patience and intuition** guiding each step

Software Testing

Software testing is the process of **checking a software system to find defects**, ensure quality, and verify that it works as expected. It includes activities like executing the software, checking outputs, and reporting bugs.

Software Testing is divided into two major parts:



✅ **Verification**

- **Verification** means **"Are we building the product, right?"**
- It checks whether the software **meets the specified requirements, designs, and standards**.
  It focuses on **process, documents, and design**, not actual running code.
- **Verification = Check before building**
- **Done by developer**
- It is a **static testing** process—meaning the software is not executed.

✓ **Activities in Verification:**

1. **Inspections**
   Careful examination of design, documents, and code to find errors early.
2. **Reviews**
   Team-based evaluation of requirements, design, and code.
3. **Walkthroughs**
   Developer informally explains work to the team for feedback.
4. **Desk Checking**
   Developer manually checks his/her own code before formal testing.

Examples of Verification:

- Reviewing requirement documents
- Checking design diagrams
- Code reviews
- Static analysis

➡️ Verification **does not require executing the program**.

✅ **Validation**

- **Validation** means **"Are we building the right product?"**

- It checks whether the **final software actually meets the user needs and expectations**.
- It focuses on **testing the working software**.
- **Validation = Check after building**
- **Done by tester**
- It is a **dynamic testing** process—software must be executed.

✓ Activities in Validation:

1. **Black Box Testing**
   Focus on input-output behavior; tester doesn't know internal code.
2. **White Box Testing**
   Tester checks internal logic and code paths.
3. **Unit Testing**
   Tests individual components or functions.
4. **Integration Testing**
   Ensures combined modules interact correctly.

Examples of Validation:

- Running test cases
- User acceptance testing (UAT)
- Functional testing
- System testing

➡️ Validation **requires executing the program**.

What is Validation Testing?

Validation Testing is a crucial phase in the software quality assurance process that verifies whether the **final software product meets the customer's actual requirements and expectations**.
It answers the question:
➡️ **"Are we building the *right* product?"**
This stage ensures the delivered system satisfies:

- User needs
- Business goals
- Stakeholder expectations

Validation testing is performed **after unit and integration testing**, at the **system level**, once the entire application is assembled.

Purpose / Objectives of Validation Testing
1. Fulfilling User Requirements
Ensures all features and behaviors requested by the customers are implemented correctly.
2. Meeting Business Needs
Checks that the final software aligns with the organization's business processes and strategic goals.

3. Completeness Verification
Validates that **all specified features** and functionalities exist, work properly, and nothing is missing.
4. Functionality Confirmation
Evaluates the software from an **end-user perspective**, ensuring real-world workflows and functions behave as expected.
5. Building Stakeholder Confidence
Provides assurance to clients, users, and stakeholders that the product is stable, accurate, and ready for deployment.

Key Aspects & Activities in Validation Testing
1. System Testing

- Performed after integration testing
- Validates the **complete, integrated software system**
- Covers **functional AND non-functional** requirements
- Ensures the overall system behaves exactly as specified

2. Acceptance Testing (Final Phase)
This determines whether the product is ready for release and meets acceptance criteria. Acceptances tests include:
a) User Acceptance Testing (UAT)

- Conducted by **actual end-users**
- Uses real-world business scenarios
- Verifies the system works in a production-like environment

b) Business Acceptance Testing (BAT)

- Conducted by business stakeholders
- Ensures the product aligns with business rules, workflows, and policies

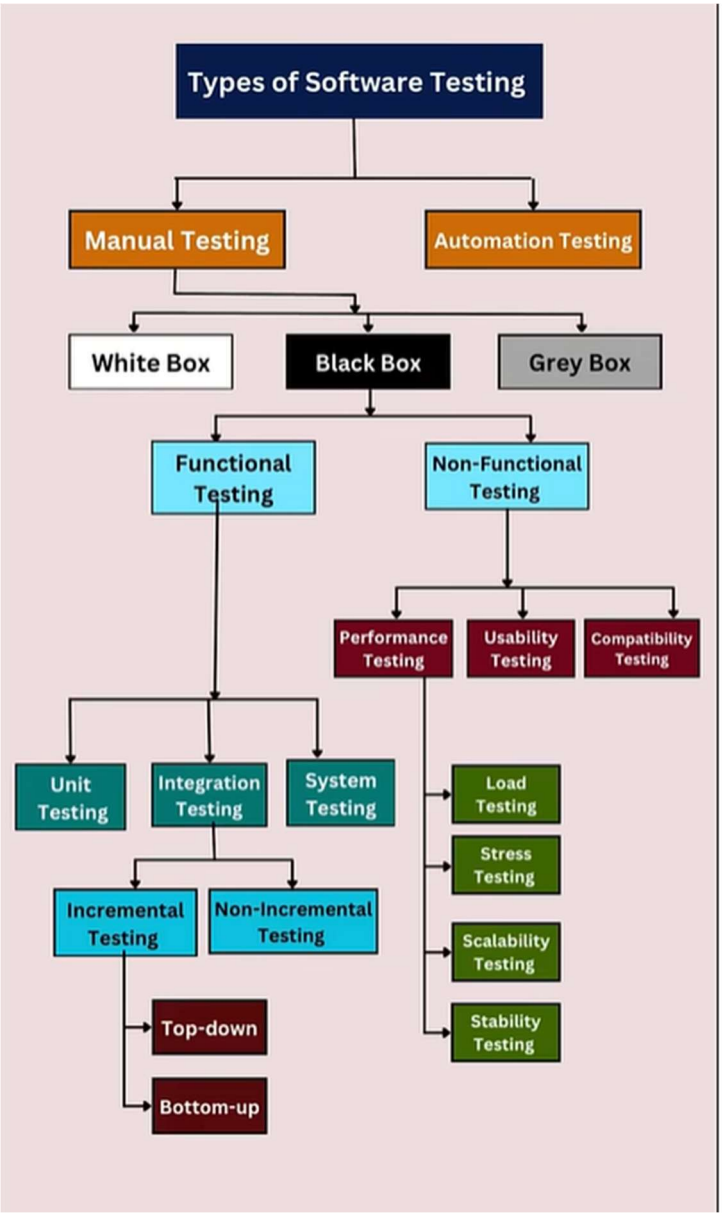
c) Contract Acceptance Testing (CAT)

- Ensures the product meets **contractual requirements**
- Performed to confirm deliverables match agreed terms between client and developer

Alpha, Beta, and Customer Acceptance (Context)

- Alpha Testing: Internal testing performed by developers or QA before release to spot bugs.
- Beta Testing: Limited release to select users or stakeholders, real-world feedback but still pre-release.
- Customer Acceptance: May refer to a client or customer checking deliverables, potentially part of UAT or contract testing.

Types of Software Testing



⭐ Complete Explanation of All Types of Software Testing

🔵 1. Manual Testing

Manual Testing is the process where testers execute the test cases **without using automation tools**.
They manually check the application's functionality, usability, and behavior.

✓ Features:

- Human-driven
- Useful for exploratory, usability, and ad-hoc testing
- Time-consuming but essential in early phases

🔵 2. Automation Testing

Automation Testing is the process of executing test cases **using automation tools** (like

Selenium, QTP, Cypress, Playwright).

✓ **Features:**

- Faster execution
- Reusable scripts
- Best for regression, load, and repetitive tests

🟧 **Manual Testing → Types**

Manual Testing includes:

A) White Box Testing

Testing done by developers where they know the **internal code structure**.

✓ **What it tests:**

- Code paths
- Conditions
- Loops
- Logic

✓ **Techniques:**

- Statement coverage
- Branch coverage
- Path coverage

B) Black Box Testing

Testing the software **only from the user's perspective**, without knowing the internal code.

Black Box Testing is further divided into:

🟦 **3. Functional Testing (under Black Box)**

Functional Testing checks whether the software **functions according to requirements**.

Includes:

1) Unit Testing

- Tests **individual units/modules** of code.
- Done by **developers**.
- Ensures each unit works correctly.

---

2) Integration Testing

Integration Testing checks whether **combined modules** work correctly.

Two main approaches:

A) Incremental Testing

Modules are integrated **step-by-step**.

Types:

• **Top-Down**

- Starts from **higher-level modules**.
- Use **stubs** for lower modules.

- **Bottom-Up**
  - Starts from **lower-level modules**.
  - Use **drivers** for higher modules.

Sandwich testing/mixed testing: It is a way to check if all parts of a big software project work well together.:

- It mixes two methods: Top-down (testing from the top layer downward) and bottom-up (testing from the bottom layer upward).
- Both ends are tested at the same time: You start testing high-level parts and low-level parts, then work towards the middle where they meet.
- This helps you spot problems early and makes sure everything is working smoothly, from top to bottom and bottom to top.

Sandwich testing is especially good when your software is built in layers or levels, because it checks all connections between them at once

3) **Regression Testing**

Regression testing is a type of software test that checks whether changes made to the system—such as bug fixes or new features—do not impact the previously working functionality.

Types of Regression Testing

1. Full Regression Test:
   - Tests the entire application from start to finish after changes have been made.
   - Ensures that everything in the system still works properly after those changes.
2. Partial Regression Test:
   - Tests only those parts of the application that were affected by the changes.
   - Checks that changes haven't broken the impacted areas, without retesting unaffected sections.

B) Non-Incremental Testing (Big Bang)

- All modules combined at once.
- Test everything together.
- Hard to identify defects.

---

3) System Testing

The entire system is tested **end-to-end** to ensure it meets requirements.
Performed after integration testing.

🟩 4. Non-Functional Testing (under Black Box)

Non-functional testing checks **quality attributes** like speed, usability, compatibility, etc.
It includes:

A) Performance Testing
Ensures the system performs well under expected load.
Types:
1. Load Testing
- Checks behavior under expected user load.

2. Stress Testing
- Tests system under **extreme load** to check breaking point.

3. Scalability Testing
- Checks if the system can **scale up** (more users, data).

4. Stability Testing
- Ensures system remains stable during **long-duration usage**.

B) Usability Testing
Tests how easy and user-friendly the application is.
✓ **Focus:**
- UI design
- Navigation
- User experience

C) Compatibility Testing
Ensures an application works on:
- Different devices
- Browsers
- Operating systems
- Network conditions

Done after the application becomes stable.

🔵 **5. Grey Box Testing**
Combination of **White Box + Black Box**.
✓ **Tester has:**
- Partial knowledge of internal structure
- Tests functionality + internal concepts

Useful for:
- Integration testing
- Web applications
- Security testing

---

what is Software Quality?

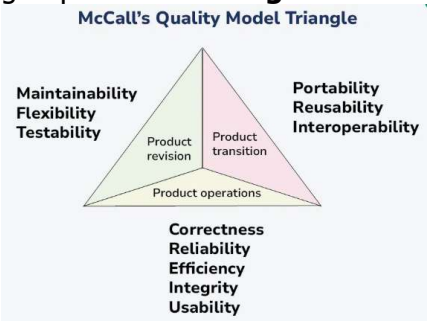
Software quality measures how well a software product meets:

- User requirements
- Business needs
- Reliability, efficiency, and maintainability standards

High-quality software is **reliable, easy to use, maintainable, and meets customer expectations**.

McCall's Software Quality Factors

James McCall proposed a model in the 1970s to measure software quality using **11 factors** grouped into **3 categories**:



---

1. Product Operation Factors (How well the software works)

1. **Correctness** – Does the software do what it is supposed to do?
2. **Reliability** – Can the software run without failure under given conditions?
3. **Efficiency** – Does the software use resources (CPU, memory) effectively?
4. **Integrity** – Is the software secure from unauthorized access?
5. **Usability** – Is it easy for users to learn and operate?

---

2. Product Revision Factors (How easy it is to change)

6. **Maintainability** – Can the software be easily fixed when defects are found?
7. **Flexibility** – Can the software adapt to changes in requirements?
8. **Testability** – Can the software be easily tested to ensure it works correctly?

---

3. Product Transition Factors (How well it adapts to a new environment)

9. **Portability** – Can the software run on different platforms and environments?
10. **Reusability** – Can parts of the software be reused in other projects?
11. **Interoperability** – Can the software work with other systems or products?

What are Product Metrics?

Product metrics are measurements used to evaluate the **quality, performance, and effectiveness** of a software product. They help teams understand **how good the product is** and identify areas for improvement.
Think of them as **numbers that tell you how well the software is working**.

Types of Product Metrics

1. **Size Metrics**

- Measure the size or scale of the software.
- Examples: Lines of Code (LOC), Number of modules, Number of features

2. **Complexity Metrics**

- Measure how complicated the software is.
- Examples: Cyclomatic complexity, Code dependencies

3. **Quality Metrics**

- Measure how reliable and defect-free the product is.
- Examples: Number of bugs, Defect density, Failure rate

4. **Performance Metrics**

- Measure how well the software performs under certain conditions.
- Examples: Response time, Throughput, Resource usage

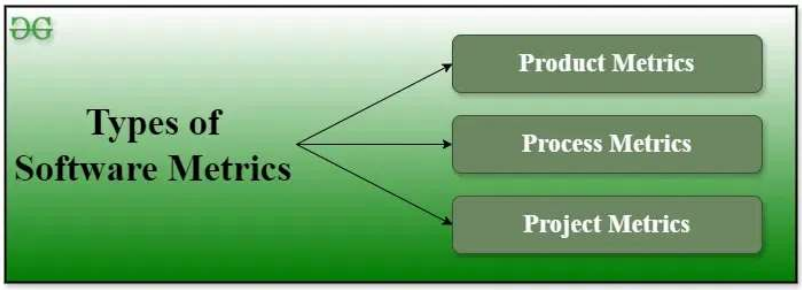
5. **Customer/User Metrics**

- Measure user satisfaction and usability.
- Examples: Customer satisfaction score (CSAT), Number of support tickets, Usage statistics

---

Why Product Metrics Matter

- Help **improve software quality**
- Guide **decision-making** for development and maintenance
- Track **progress over time**
- Ensure the product meets **user expectations and business goals**



Software Metrics:
Software metrics are quantitative standards for measurement, applied to evaluate software products, development processes, and project management. Good use of metrics enables effective tracking, assessment, and improvement throughout the software lifecycle.

Types of Software Metrics

Type	Definition	Key Attributes and Examples	Purpose / Uses
Product	Measures characteristics of the final software product	Size, complexity, design features, performance, quality level, reliability, functionality, function points, coupling, number of classes/objects, number of relationships	Assesses software quality, guides testing, indicates stability, evaluates procedural and architectural complexity
Process	Measures characteristics of development and maintenance activities	Effort required, time to produce, effect of techniques/tools, number of defects found, quality, productivity, test coverage, defect removal efficiency, failure rate	Tracks and improves SDLC activities, documents process effectiveness, helps continuous improvement
Project	Measures characteristics and execution of the software project	Number of developers, staffing patterns, cost, schedule, productivity, estimates vs. actuals	Manages project resources, budgets, timelines, and progress; supports planning & decision-making

---

Product Metrics:

- Indicate how well the software meets design goals.
- Help evaluate the clarity, scope, and coupling of components.
- Examples:
    - Functionality delivered: How much functional value is present.
    - Number of functions identified: Scope of major capabilities.
    - Design features: How many unique or advanced features.
    - System size: Total use cases/scenarios/classes/relationships in models.
    - Function coupling: Extent of interdependence among functions.

Process Metrics:

- Used to optimize development and maintenance processes:

    - Effort: Resources required for activities.
    - Time to produce: Speed to deliver outputs.
    - Defect management: How effectively issues are found and remedied.
    - Productivity, quality, failure rate: Indicates team and process health.
- Example metrics: defect removal efficiency, test case coverage, development speed, test execution coverage.

Project Metrics:

- Evaluate project execution aspects:
    - Staffing patterns: Developer allocation throughout project phases.
    - Cost and schedule: Track estimates and actuals for resource and time usage.
    - Productivity: Gauges outcome per input (e.g., code written per developer per month).
- Project metrics are crucial for comparing with benchmarks, forecasting risks, and maintaining schedule and budget discipline

Software Product Metrics
Purpose

- Evaluate analysis and design model effectiveness.
- Indicate complexity in procedural design and source code.
- Facilitate effective testing strategies.
- Assess stability and delivered quality of products.

Product Metrics Taxonomy

1. Matrix for Analysis Model
    - *Functionality Delivered*: Scope and clarity of implemented functions.
    - *Number of Functions Identified*: Major system or module capabilities.
    - *Functionality Completeness*: Portion of requirements mapped to system functions.
    - *Function Coupling*: Degree of interdependence between functions (impact on maintainability and design simplicity).
2. System Size
    - *Number of Use Cases*: Total modeled user scenarios.
    - *Number of Scenarios per Use Case*: Covers comprehensiveness for each use case.
    - *Number of Classes/Objects*: Size and scope of object-oriented analysis/models.

- *Number of Relationships*: Connectivity among model components, showing design integration.

Functional Count and Function Point Analysis (FPA)

they are important techniques in software engineering for measuring and estimating software size, complexity, and development effort—especially during early phases before code is written.
FPA is a type of product metric focused on measuring functional size and complexity.

Functional Count

- *Functional Count* refers to the process of quantifying the number of distinct functions user expects from the system.
- It focuses on **counting user-recognizable functionalities** rather than internal technical components.
- This provides a basis for objective measurement, independent of programming language or technical design.

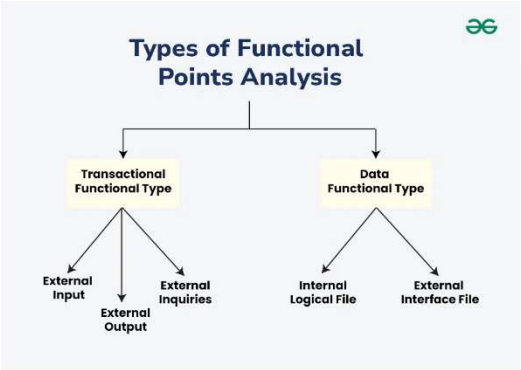
Special Features

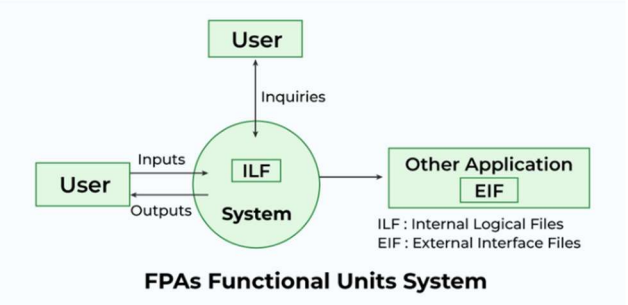
1. Independent of language: Functional count does not depend on the programming language used.
2. Estimate development effort in early process: It helps to predict how much work will be needed, even at the beginning of software development.
3. Directly linked with requirements: The count reflects the actual functional requirements of the system.
4. Based on users' external view of the system: Functional count is determined by what the user experiences or expects from the system, not by its internal design or coding

---

Function Point Analysis (FPA)
Function Point Analysis is a systematic method to evaluate the functional size of a software application by analyzing user requirements and decomposing them into measurable units called **function points**.
Key Principles

- FPA decomposes a system into **five functional components:**





Transactional Functional Type: "what does the system do for the user?" through actions and immediate feedback.

1. **External Inputs (EI):** Information entering the system.
2. **External Outputs (EO):** Information leaving the system.
3. **External Inquiries (EQ):** User requests for instant information.

Data Functional Type: "what data does the system store or use?" through ongoing management and reference.

1. **Internal Logical Files (ILF):** Data stored and maintained within the system.
2. **External Interface Files (EIF):** Data accessed from other systems.

- Each component is classified and counted based on complexity—Low, Medium, High—using weighting factors.
- FPA is independent of programming language and technical design choices.
- It helps estimate **development effort, cost, and project timeline** early in the software lifecycle.

14 General System Characteristics (GSCs) in FPA

- **Data Communications:** Does the application require communication with other systems or devices?
- **Distributed Data Processing:** Is data processed across multiple locations or platforms?
- **Performance:** Are there performance requirements (response time, throughput, transaction speed, etc.)?
- **Heavily Used Configuration:** Will the software be used on a heavily loaded or high-traffic hardware configuration?
- **Transaction Rate:** Are high transaction rates required?
- **On-Line Data Entry:** Is on-line, interactive data entry required?
- **End-User Efficiency:** Does the application require features to enhance efficiency for end-users (GUI, shortcuts, etc.)?
- **On-Line Update:** Does the system need to update the data in real time, as opposed to batch updates?
- **Complex Processing:** Is there complex internal processing (sorting, calculations, rules)?
- **Reusability:** Will components be reused in other software systems?
- **Installation Ease:** Are there specific requirements for the ease of installation and setup?
- **Operational Ease:** Does the system need to be easy to operate (clear error messages, help systems)?
- **Multiple Sites:** Will the system be used at multiple physical locations?

- **Facilitate Change:** Is the software designed to easily accommodate future changes/enhancements?

Calculation Steps

1. **Count each functional component** (EI, EO, EQ, ILF, EIF) and determine their complexity.
2. **Compute Unadjusted Function Point (UFP):**
    - Multiply count by respective weights for complexity.
    - Example weights (Low/Avg/High): EI (3/4/6), EO (4/5/7), EQ (3/4/6), ILF (7/10/15), EIF (5/7/10).
3. **Calculate Value Adjustment Factor (VAF):**

    Rate 14 general system characteristics (GSCs) relating to system functionality (like reliability, performance, ease-of-use, etc.).
    Formula:
    VAF=0.65+0.01×∑Fi
    (Fi = scores for each GSC, typically ranging from 0–5 per characteristic)
4. **Adjusted Function Point (AFP):**
    - $AFP = UFP \times VAF$
    - This gives the total functional point count reflecting system complexity.

Special Features of FPA

- Language independent.
- Based on user requirements—external system view, not internal code.
- Directly aids in early project estimation and planning.

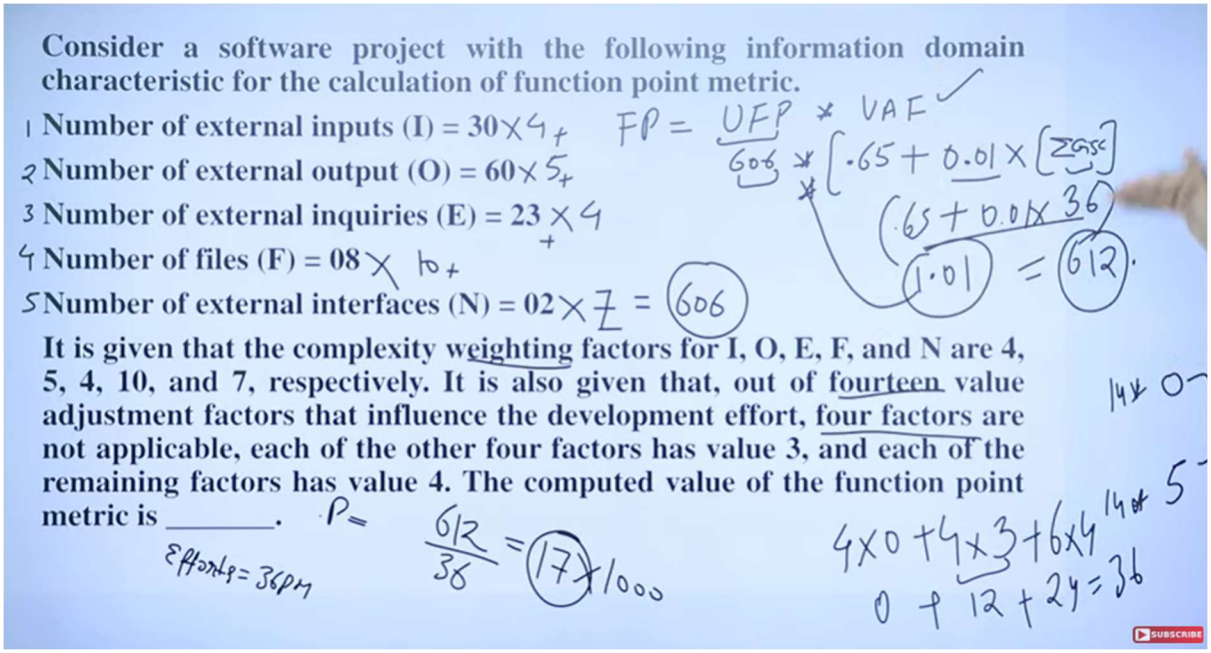
Example

Suppose a system has:

- 10 EIs (Low), 13 EOs (High), 4 EQs (Average), 2 ILFs (Average), and 9 EIFs (Low).
- Using provided complexity weights, calculate UFP:
    - EI: $10 \times 3 = 30$
    - EO: $13 \times 7 = 91$
    - EQ: $4 \times 4 = 16$
    - ILF: $2 \times 10 = 20$
    - EIF: $9 \times 5 = 45$
    - **Total UFP = 30 + 91 + 16 + 20 + 45 = 202**
- Calculate CAF from question responses, e.g., if $\sum Fi = 15$, CAF = 0.65 + 0.01 × 15 = 0.80.
- AFP = $202 \times 0.80 = 161.6$

Purpose and Use

- **Functional Count and FPA** provide standardized quantitative estimates for software projects.
- Widely used for effort estimation, project management, and productivity analysis in both academics and industry.



P=producitivity

Software Testing Metrics

Measurements used to check the progress, quality, and effectiveness of the testing process so issues can be found early, resources can be used properly, and overall testing can improve

Purpose:

- Assess software quality and testing performance.
- Identify defects and trends early.
- Optimize testing resources.
- Support decision-making for future testing.

Types of Software Testing Metrics

1. **Process Metrics**
    - Measure characteristics of the **testing process**.
    - Help improve the SDLC process.
2. **Product Metrics**
    - Measure **size, performance, complexity, quality** of the software.

- Help in improving product quality.

3. **Project Metrics**
    - Assess **overall project quality**, resources, productivity, and defects.

Manual Test Metrics

1. Base Metrics – Collected during test case development and execution:

- Total number of test cases.
- Number of test cases completed.

2. Calculated Metrics – Derived from base metrics:

- Provide progress tracking at module, tester, and project level.

Other Important Metrics

Metric	Simple Purpose
Defect Metrics	Check how good the software is by counting problems.
Schedule Adherence	See if the project is on time or delayed.
Defect Severity	Show how serious each problem is.
Test Case Efficiency	Check how well test cases find problems.
Defects Finding Rate	Track how fast defects are being found.
Defect Fixing Time	Measure how long it takes to fix each problem.
Test Coverage	Show what percentage of the software has been tested.
Defect Cause	Find out why defects happened.

Test Metrics Life Cycle

1. **Analysis** – Identify metrics.
2. **Communication** – Inform stakeholders and testing team.
3. **Evaluation** – Collect and verify data, calculate metrics.
4. **Reporting** – Present results and gather feedback.

Key Formulas for Software Testing Metrics

1. Percentage of Test Cases Executed

Executed % = (No. of test cases executed / Total test cases written) × 100

2. Test Case Effectiveness

Effectiveness % = (Defects detected / Test cases run) × 100

3. Passed Test Cases Percentage

Passed Test Cases % = (Test cases passed / Test cases executed) × 100

4. Failed Test Cases Percentage
Failed Test Cases % = (Test cases failed / Test cases executed) × 100

5. Blocked Test Cases Percentage
Blocked Test Cases % = (Test cases blocked / Test cases executed) × 100

6. Fixed Defects Percentage
Fixed Defects % = (Defects fixed / Total defects reported) × 100

7. Accepted Defects Percentage
Accepted Defects % = (Defects accepted by dev team / Total defects reported) × 100

8. Defects Deferred Percentage
Defects Deferred % = (Defects deferred for future releases / Total defects reported) × 100

Example Calculation

Metric	Value
Total test cases written	200
Test cases executed	164
Test cases passed	100
Test cases failed	60
Test cases blocked	4
Total defects reported	20
Defects fixed	12
Accepted defects	15
Deferred defects	5

Calculations:

1. Executed % = (164/200) × 100 = **82%**
2. Test Case Effectiveness = (20/164) × 100 = **12.2%**
3. Failed Test Cases % = (60/164) × 100 = **36.59%**
4. Blocked Test Cases % = (4/164) × 100 = **2.44%**
5. Fixed Defects % = (12/20) × 100 = **60%**
6. Accepted Defects % = (15/20) × 100 = **75%**
7. Deferred Defects % = (5/20) × 100 = **25%**

Test Metrics Classification

- **1. Process Metrics:**
  Used to measure test preparation and execution activities, such as productivity, coverage, and outcome of test cases.image.jpg
- **2. Product Metrics:**
  Used for defect analysis and test effectiveness regarding the software product itself.image.jpg

Key Process Metrics

1. **Test Case Preparation Productivity**
    - Measures how many test cases are prepared per unit effort.
    - Formula:

$$\text{Productivity} = \frac{\text{Number of test cases}}{\text{Effort spent preparing test cases}}$$

2. **Test Design Coverage**
    - Assesses what percentage of requirements are covered by test cases.
    - Formula:

$$\text{Design Coverage} = \frac{\text{Requirements mapped to test cases}}{\text{Total requirements}} \times 100$$

3. **Test Execution Productivity**
    - Number of test cases executed per unit effort.
    - Formula:

$$\text{Execution Productivity} = \frac{\text{Number of test cases executed}}{\text{Effort spent executing}}$$

4. **Test Execution Coverage**
    - Percentage of planned test cases actually executed.
    - Formula:

$$\text{Execution Coverage} = \frac{\text{Test cases executed}}{\text{Test cases planned}} \times 100$$

5. **Test Cases Passed / Failed / Blocked**
    - Measure outcome percentages:
        - Passed: $\frac{\text{Test cases passed}}{\text{Test cases executed}} \times 100$
        - Failed: $\frac{\text{Test cases failed}}{\text{Test cases executed}} \times 100$
        - Blocked: $\frac{\text{Test cases blocked}}{\text{Test cases executed}} \times 100$.image.jpg+1

Key Product Metrics

1. **Error Discovery Rate:**
    - Measures effectiveness of test cases in finding defects.
    - Formula:

$$\text{Error Discovery Rate} = \frac{\text{Defects found}}{\text{Test cases executed}} \times 100$$

2. **Defect Fix Rate:**
    - Measures how many reported defects are resolved.
    - Formula:

$$\text{Defect Fix Rate} = \frac{\text{Defects fixed}}{\text{Defects reported + new bugs from fix}} \times 100$$

3. **Defect Density:**
    - Ratio of identified defects to requirements.
    - Formula:

$$\text{Defect Density} = \frac{\text{Defects identified}}{\text{Actual requirements}}$$

4. **Defect Leakage:**
    - Percentage of defects missed before a major test phase (like UAT).
    - Formula:

$$\text{Defect Leakage} = \frac{\text{Defects found post-UAT}}{\text{Defects found before UAT}} \times 100$$

5. **Defect Removal Efficiency (DRE):**
    - Efficiency of development team at removing defects.
    - Formula:

$$DRE = \frac{\text{Defects removed by dev}}{\text{Defects found at measurement time}}$$

Metrics for Design Model (Complete)

Measurements used to evaluate the structure, quality, and complexity of a software design to ensure it is easy to build, understand, and maintain.

---

1. Architectural Design Metrics
Focus on overall program structure and how modules connect (no code details needed).
3 Design Complexity Measures (Card & Glass): Card & Glass proposed three design complexity measure's structure, data and system complexity
Structural Complexity
Shows how modules are linked and how independent they are.
Common structures:

- **Tree Structure:** Parent–child hierarchy
- **Linear Structure:** Modules arranged in a straight sequence

- **Hierarchy Structure:** Multi-level arrangement of modules and submodules
- $S(i) = \int f_{out}^2(j)$ $f_{out}(j)$ represents the number of modules invoked by module j

---

Data Complexity
Describes how data moves in and out of modules and how global/static data is handled.

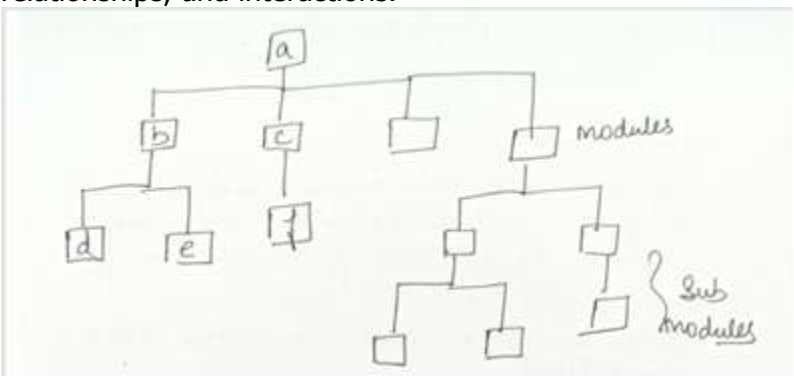
System Complexity
Overall complexity combining structural and data complexity.

$$C(i,j) = \frac{V(i)}{f_{out}(j)+1} + f_{out}(j)^2$$

Where $V(i)$= variables, $f_{out}(j)$= modules invoked by module j.

---

2. Class-Oriented Design Metrics
Used in object-oriented design; measure design complexity using class diagrams, relationships, and interactions.



---

3. Information Flow Metrics (Henry & Kafura)
Measure complexity based on how information moves between modules.

- **Fan-in:** Number of flows into a procedure (inputs) + global data read
- **Fan-out:** Number of flows out of a procedure (outputs) + global data updated

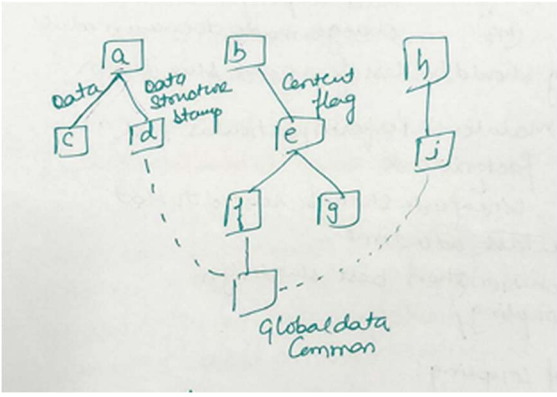
Procedure Complexity: $Cp(IF) = (Fan\_in \times Fan\_out)^2$

---

4. Coupling and Cohesion
Coupling (Connection Between Modules)
Lower coupling = better design
Types (best → worst):

1. **Data Coupling** – passing required parameter only add(a,b) only a and b is passed
2. **Stamp Coupling** – passing whole data structures
3. **Control Coupling** – passing flags/control data
4. **External Coupling** – using external interfaces

5. **Common Coupling** – shared global data
6. **Content Coupling** – one module changes another's internal data (worst)



Cohesion (How Well a Module's Functions Fit Together)
High cohesion = a module does one clear task.
Leads to easier maintenance and better modularity.

Additional Notes

- Very high **fan-in** or **fan-out** usually means poor cohesion.
- Good designs aim for **high cohesion** and **low coupling** for easier maintenance and better structure.

Metrics for Source Code

- Measurements used to evaluate the size, quality, and complexity of source code to ensure it is easy to understand, maintain, and improve.
- These metrics are gathered after writing code or completing the design phase of a software project.

METRICES:

1) **Halstead Metrics:** A set of measurements used to judge the size, complexity, and effort needed to understand or maintain a program by counting operators and operands in the code.

- **Distinct Operators (n_1)**: Number of distinct operators in the program.
- **Distinct Operands (n_2)**: Number of distinct operands in the program.
- **Total Operators (N_1)**: Total occurrences of operators.
- **Total Operands (N_2)**: Total occurrences of operands.

Key Formulas

- **Program Length (N)**: The total number of elements in the program, usually counting operators + operands (from Halstead metrics).
    - Length: N=N1+N2
    - Vocabulary: n=n1+n2

- Estimated Program Length ($\hat{N}$)(Halstead's estimate):

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2.$$

- **Purity Ratio/ Succinctness Ratio/ Optimality Ratio of program:** Higher ratio = cleaner, more understandable code.

Formula

$$\text{Purity Ratio} = \frac{\hat{N}}{N}$$

Where:

- $\hat{N}$ = Estimated program length (ideal length by Halstead)
- N = Actual program length (real count of operators + operands)

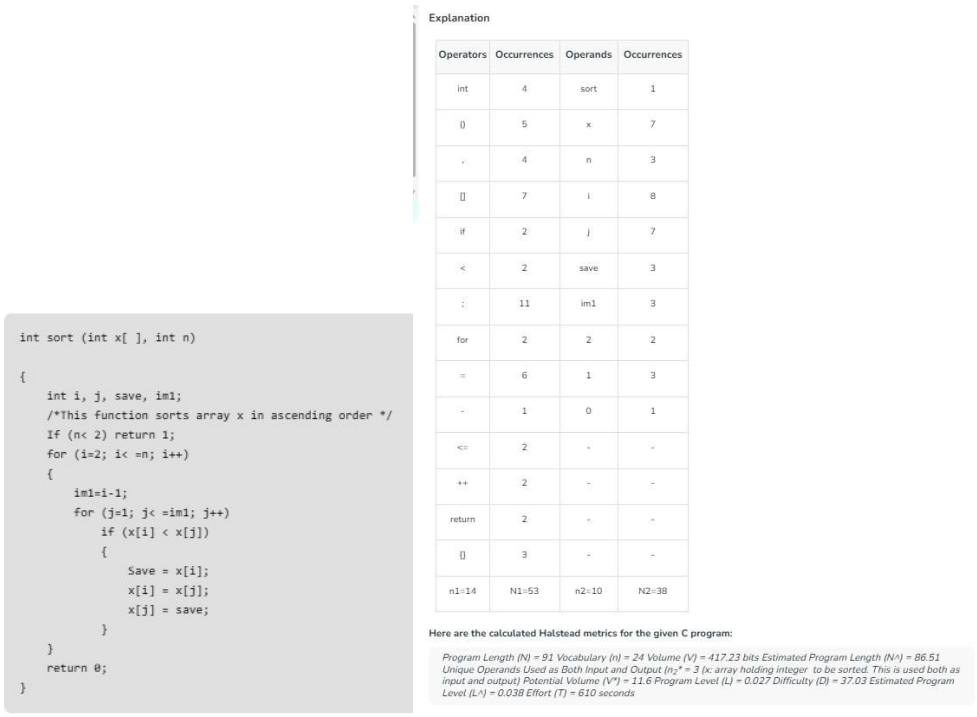
- **Program Volume (V)**: A measure of how big the program is based on its size and information content.
  - $V = N \log_2 n$
- **Halstead Difficulty (D):** Measures how hard the code is to write or understand.

$$D = \frac{1}{L}$$

- **Language Level (Level of Abstraction, L)**: How "high-level" a programming language is. Higher level = easier to write code (e.g., Python). Lower level = more detailed work needed (e.g., Assembly)

$$L = \frac{2}{n_1} \times \frac{n_2}{N_2}.$$

- **Effort (E)**: The amount of work or time needed to write, understand, or maintain a program
  - Defined as $E = V * D$



```
int sort (int x[ ], int n)

{
    int i, j, save, im1;
    /*This function sorts array x in ascending order */
    If (n< 2) return 1;
    for (i=2; i< =n; i++)
    {
        im1=i-1;
        for (j=1; j< =im1; j++)
            if (x[i] < x[j])
            {
                Save = x[i];
                x[i] = x[j];
                x[j] = save;
            }
    }
    return 0;
}
```

Explanation

Operators	Occurrences	Operands	Occurrences
int	4	sort	1
()	5	x	7
,	4	n	3
[]	7	i	8
if	2	j	7
<	2	save	3
;	11	im1	3
for	2	2	2
=	6	1	3
-	1	0	1
<=	2	-	-
++	2	-	-
return	2	-	-
{}	3	-	-
n1=14	N1=53	n2=10	N2=38

Here are the calculated Halstead metrics for the given C program:

Program Length (N) = 91 Vocabulary (n) = 24 Volume (V) = 417.23 bits Estimated Program Length ($N^{\wedge}$) = 86.51 Unique Operands Used as Both Input and Output (n_3 = 3 (x: array holding integer to be sorted. This is used both as input and output) Potential Volume (V*) = 11.6 Program Level (L) = 0.027 Difficulty (D) = 37.03 Estimated Program Level ($L^{\wedge}$) = 0.038 Effort (T) = 610 seconds*

---

2) **McCabe's Cyclomatic Complexity**
McCabe's Cyclomatic Complexity is a metric that measures the complexity of a program by counting the number of independent paths through its source code.
How Cyclomatic Complexity is Calculated

- Formula:

$$M = E - N + 2P$$

where:

- E: Number of edges in the control flow graph
- N: Number of nodes in the control flow graph
- P: Number of connected components (usually 1 for a single program).

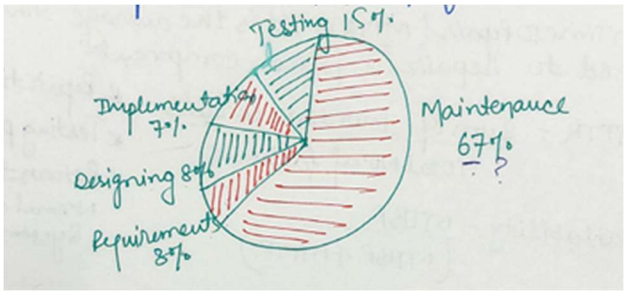
Software maintenance: it focuses on modifying and updating applications after delivery to correct errors, improve performance, fix bugs, and add new features.

Metrics for Maintenance

Measurements used to evaluate the **quality, cost-effectiveness, and reliability of maintenance activities**
Activities Included:

- Fixing errors
- Removing outdated features
- Adding or improving features
- Making the system faster and better performing

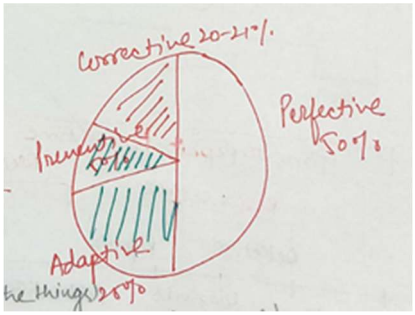
Pie Chart (Development Phases):



- Maintenance: 67%
- Implementation: 7%
- Testing: 15%
- Designing of requirements: 8%

Types of Maintenance:

1. **Corrective:** Fixing problems after they happen
2. **Adaptive:** Updating software so it works in new environments
3. **Preventive:** Making changes to avoid future issues
4. **Perfective:** Improving features, performance, and overall quality
5. **Inspection-based:** Updating code based on reviews to prevent future failures



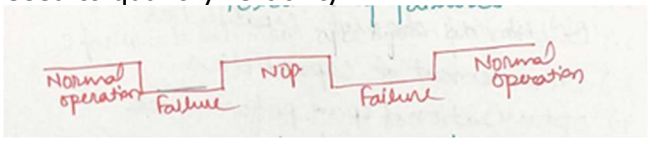
Key Maintenance Metrics

1. Mean Time Between Failure (MTBF)

Average time for which a system/component operates before failing:

$$MTBF = \frac{\text{Sum of operational time}}{\text{Total Number of failures}}$$

Used to quantify reliability.



2. Mean Time To Repair (MTTR)

Average time required to repair a failed component:

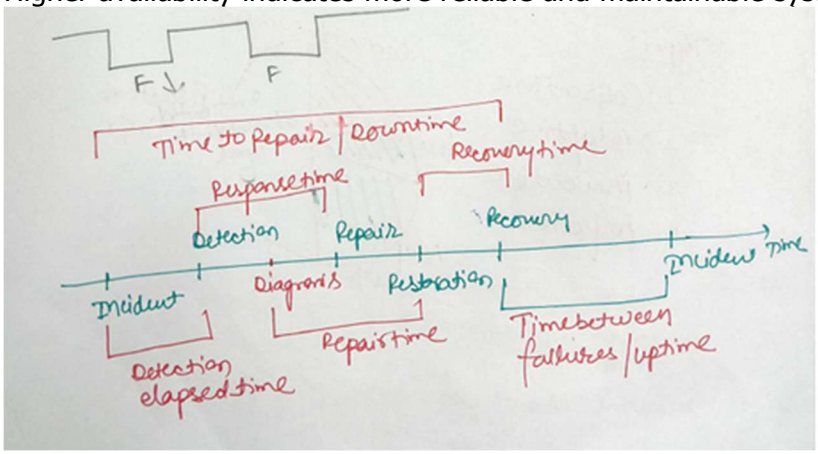
$$MTTR = \frac{\text{Sum of downtime periods}}{\text{Total Number of failures}}$$

Includes time for reporting, testing, and returning the system to operation.

3. Availability
Measures the proportion of time a system is functioning:

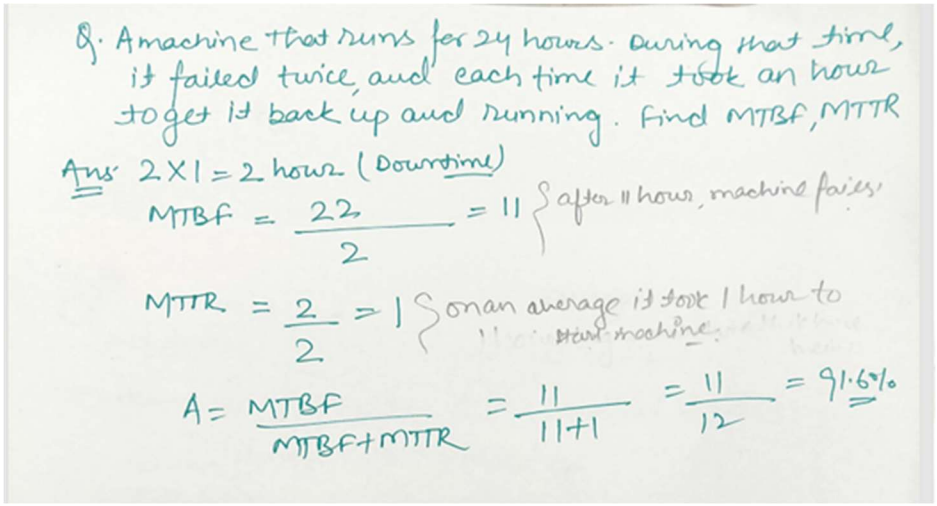
$$Availability = \frac{MTBF}{(MTBF + MTTR)}$$

Higher availability indicates more reliable and maintainable systems.



Maintenance Timeline

- Incident ➔ Detection (elapsed time)
- Diagnosis (downtime) ➔ Repair ➔ Recovery
- Detection's elapsed time, repair time, and recovery time make up total downtime
- Uptime is the time between failures



Metrics for Analysis Model

What They Are

- Measurements used to judge **quality, size, and complexity of requirements** and early diagrams.
- Applied during the **system analysis** phase before design and coding.

Key Types

1. **Function Point Analysis (FPA)**
    - Measures system size by counting **inputs, outputs, queries, files, interfaces**.
    - Helps estimate **effort, cost, and time** early.
2. **Information Flow Metrics**
    - **Fan-in**: number of modules sending information to a module.
    - **Fan-out**: number of modules receiving information from a module.
    - Higher values → **higher complexity**.
3. **Data Structure Metrics**
    - Measures **number of entities/objects** and their **relationships**.
    - Helps predict system **complexity and maintainability**.
4. **Requirement Metrics**
    - **Requirement Coverage**: It shows which requirements are included (covered) in the analysis and which ones are not included (missed).
    - **Requirement Change Rate**: how frequently requirements change.

Why They Are Useful

- Early **effort and cost estimation**.
- Helps detect **unclear or missing requirements**.
- Reduces project **risk** and improves planning.
- Ensures analysis is **complete** and meets user needs.

Unit -5

Quality Concepts

1. Software Quality

- Software quality means **How well the software works and how well it meets the user's needs.**
- If the software is **easy to use, fast, reliable, and has fewer bugs**, then it has good quality
- It is usually checked by **third-party organizations** like international standard groups.

---

2. Types of Software Quality

1. Quality of Design

- This depends on **how good the design is**.
- It includes the **tools used, architecture, features**, etc.

2. Quality of Conformance

- This means **how correctly the software is built according to the design**.
- Basically: *Did the developers follow the design properly?*

---

3. Quality Assurance (QA)

- QA is about **planning and following proper processes**.
- Its goal is to give **confidence that the final product will be reliable**.
- QA focuses on **preventing defects**.

---

4. Quality Control (QC)

- QC means **checking the product** to make sure it meets quality standards.
- It includes **inspection, reviews, and testing** (manual or automated).
- QC focuses on **finding defects**.

---

5. Cost of Quality (CoQ)

- CoQ is the **total money spent to make sure the software is good**.

Types of Costs:

1. **Prevention Cost**
    - Money spent to **avoid defects**.
    - Examples: Quality planning, First Time Right (FTR), training.
2. **Appraisal Cost**
    - Money spent on **checking quality**.
    - Examples: Testing, inspections.
3. **Failure Cost**
    - Money spent because of **defects**.
    - **Internal failure:** Defects found before the customer sees them.
    - **External failure:** Defects found *after* the customer uses the product.

Software Quality Assurance (SQA)

- Software Quality Assurance (SQA) is a way to ensure quality in software.
- It consists of activities that ensure processes, procedures, and standards are suitable for the project and implemented correctly.
- SQA works **parallel to software development** and focuses on **preventing problems** before they become major issues.
- It acts as an **umbrella activity**, applied throughout the software process.

Focus of Software Quality Assurance

1. **Software's Maintainability:** Ease of modification, updating, or extension.
2. **Software's Correctness:** Ability to produce desired results under specific conditions.
3. **Software's Portability:** Ability to run on different platforms without major modifications.
4. **Software's Usability:** Ease of use and intuitive interaction for users.
5. **Software's Reusability:** Designing components that can be reused in multiple projects.
6. **Software's Error Control:** Mechanisms to detect, handle, and recover from errors.

Major Activities in SQA

1. **SQA Management Plan**
   → Make a plan for how all SQA (quality assurance) activities will be done during the project.
2. **Set Checkpoints**
   → Decide specific points during the project where progress and quality will be checked.
3. **Measure Change Impact**
   → When changes or fixes are made, check how they affect the whole project.
4. **Multi Testing Strategy**
   → Use different types of testing, not just one method, to find more defects.
5. **Manage Good Relations**
   → Work smoothly with other teams so the project doesn't get delayed. (Avoid misunderstanding)

6. **Maintain Records and Reports**
   → Keep written details of test cases, defects found, changes made, and testing cycles.
7. **Review Software Engineering Activities**
   Identify, document, and verify processes and products.
8. **Formalize Deviation Handling**
   → Follow a proper procedure to record and fix anything that goes wrong or differs from the plan.(Report → Analyze → Fix → Re-test → Close)

Statistical Software Quality Assurance (SQA)

Definition:
Statistical SQA focuses on identifying defects in software, analyzing their root causes, and implementing corrective measures to improve quality.

Steps Involved:

1. **Collect Software Defects**
   Gather all reported defects from testing, user feedback, or production incidents.
2. **Categorize the Defects**
   Classify defects based on type, severity, module, or other relevant factors.
3. **Trace Each Defect**
   Identify the origin or source of each defect.
4. **Apply Pareto Principle**
   - Principle: 80% of problems are caused by 20% of causes.
   - Analyze defects to identify the most impactful 20% that lead to the majority of issues.
5. **Find Solutions for Key Defects**
   Focus on resolving the top 20% of defects that contribute most to problems.

Key Terms:

- **Cause:** The reason a defect occurred.
- **Consequence:** The result or impact of the defect.

Six Sigma

Six Sigma isa widely used strategy in statistical SQA which is a quality improvement method used in software development to **reduce defects** and **make processes more efficient**.

Key Features (in simple words)

- **Reduces defects:**
  It gives techniques to find mistakes in the software process and fix them.
- **Improves processes:**
  It helps teams work in a better, faster, and more organized way.
- **Improves overall quality:**
  By controlling errors and improving workflow, the final software becomes more reliable.
- **Gives measurable results:**
  Uses data and numbers to show actual improvement (not guesswork).

☑ Software Metrics

A metric is a **quantifiable measure** used to assess a software product, process, or project.

Functions of Software Metrics

1. **Planning**
2. **Organizing**
3. **Controlling**
4. **Improving**

✅ **Advantages of Software Metrics**

1. **Cost Saving** – Helps reduce costs and stay within budget.
2. **Quality Improvement** – Improves software product quality.
3. **Better Planning** – Supports planning and decision-making.
4. **Team Productivity** – Helps manage workload and team efficiency.
5. **Time Saving** – Reduces software development time.

Mnemonic: CQPTT → Cool Quality Plans Take Time

✅ **Disadvantages of Software Metrics**

1. **Expensive/Difficult** – Can cost a lot or be hard to implement.
   Example: Buying complex testing tools may be too costly for small teams.
2. **Ignores Individuals** – Measures product, not individual performance.
3. **Misleading Data** – Metrics may not show true software quality.
   Example: Few reported defects doesn't always mean high quality; some bugs may be hidden.
4. **Time Wasting** – Collecting unnecessary data wastes effort.
   Example: Tracking every small change in code may slow the team down.
5. **Wrong Decisions** – Bad or misinterpreted metrics lead to poor decisions..

Mnemonic to Remember: "EIMTW" → Every Issue May Trigger Worry

---

Software Quality Metrics

- Software quality metrics are quantitative measures used to evaluate the quality of software products, processes, and maintenance activities. They help in monitoring, controlling, and improving software quality.
- Software quality metrics are **closely associated with product and process quality** rather than general project metrics.

---

1. Product Quality Metrics

- Focus on the **quality of the software product itself**.
- Measure characteristics such as functionality, reliability and usability
- **Examples:**
    - **Defect Density:** Number of defects per 1000 lines of code.
    - **Mean Time Between Failures (MTBF):** Average time software operates without failure.
    - **Response Time:** Speed of system response to user requests.

---

2. In-Process Quality Metrics

- Focus on the **software development process** to ensure quality is built in during development.
- Help identify problems early to prevent defects in the final product.
- **Examples:**
    - **Defect Discovery Rate:** The number of bugs found in each development phase, e.g., if 10 bugs are found during coding and 5 during testing, coding has a higher rate.
    - **Defect Removal Efficiency (DRE):** The percentage of defects fixed before release, e.g., if 100 defects are found and 90 are fixed, DRE = 90%.
    - **Test Coverage:** The portion of code or requirements tested, e.g., if 100 lines exist and 80 are tested, coverage = 80%.

---

3. Maintenance Quality Metrics

- Focus on **software quality after deployment**.
- Measure how well the software can be maintained, enhanced, and corrected.
- **Examples:**
    - **Number of Post-Release Defects:** Defects reported by users after release.
    - **Mean Time to Repair (MTTR):** Average time taken to fix a defect.
    - **Change Impact Analysis:** Assessing the effect of updates or modifications on the system.

Software Review

A Software Review is a process in which software work products (like requirements, design, code, or documentation) are examined to detect defects, improve quality, and ensure compliance with standards.

Purpose:

- Identify defects **early in the software development life cycle**.
- Improve **quality and reliability** of the software.
- Ensure compliance with **project standards and requirements**.
- Facilitate knowledge sharing among team members.

Types of Software Reviews

1. **Informal Review**
    - Quick and unstructured.
    - Performed by peers or colleagues.
    - No formal documentation required.
    - Examples: casual walkthroughs, desk checks.
2. **Formal Review / Formal Technical Review (FTR)**
    - Structured and well-documented review process.
    - Conducted by a **review team** following defined procedures.
    - Aimed at **finding defects and ensuring adherence to standards**.
    - Often required in **safety-critical or high-quality projects**.

Formal Technical Review (FTR)

A **Formal Technical Review** is a **systematic examination of software work products by a team of qualified personnel** to detect defects, ensure quality, and verify compliance with requirements and standards.

FTR Process Steps:(Example context: A team is developing a simple login feature for a web app)

1. **Planning:**
    - Decide what to review → The **login module code**.
    - Choose review team → Alice (author), Bob (reviewer), Carol (review leader), Dave (recorder).
    - Schedule meeting → Friday 10 AM.
2. **Overview Meeting (Optional):**
    - Reviewers look at the login module briefly to understand how it works.
3. **Preparation:**
    - Each reviewer checks the code individually.
    - Example defects found:
        - Password field doesn't mask input.
        - No check for SQL injection.
4. **Review Meeting:**
    - Team discusses findings.
    - Carol (review leader) ensures discussion stays focused.
    - Dave (recorder) notes defects and suggestions.

5. **Rework:**
    - Alice fixes the defects:
        - Masks password input.
        - Adds SQL injection check.
6. **Follow-up:**
    - Carol verifies that all issues are fixed.
    - Confirms that the login module is ready to integrate.

Roles in FTR (with Example)

- **Author:** Alice → wrote the login code.
- **Review Leader / Moderator:** Carol → runs the meeting.
- **Reviewers:** Bob → checks code for mistakes.
- **Recorder / Scribe:** Dave → writes down defects and suggestions.

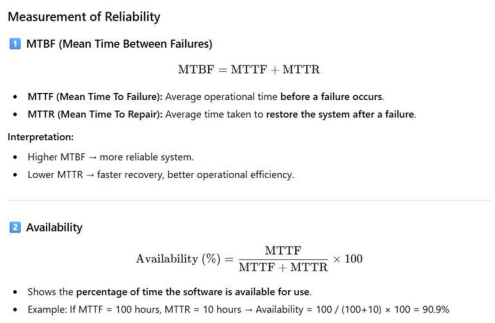
Software Reliability (Operational Reliability)

Definition:
Software Reliability is the **ability of a software system or component to perform its required functions under specified conditions(hardware,os,network like vc on 5g phone work good but not on 3g phone) for a specified period of time** without failure.

- Also called **Operational Reliability**.
- It measures how consistently the software behaves as expected.

Key Concepts

1. **Failure:**
   When software **does not perform its intended function**, it is considered a failure.
2. **Error/Bug:**
   A **defect in the code** or design that may lead to failure.
3. **Reliability vs. Availability:**
    - **Reliability:** Focuses on **failure-free operation over time**.
    - **Availability:** Focuses on **percentage of time the system is operational**, considering both failures and repair time.



Importance of Software Reliability

- **Critical for mission-critical systems:** Banking, aerospace, healthcare.
- Reduces **downtime and maintenance costs**.
- Improves **user trust and satisfaction**.
- Supports **risk management** in projects.

☑ Reactive vs Proactive Risk Strategies

Here's a **simple difference table** for **Reactive vs Proactive Risk Strategies**:

Feature	Reactive Risk Strategy	Proactive Risk Strategy
Definition	Deals with risks **after they happen**	Deals with risks **before they happen**
Approach	Corrective / Damage control	Preventive / Planning ahead
Timing	After the risk occurs	Before the risk occurs
Focus	Solving problems caused by risk	Preventing risks from happening
Cost	Can be higher because fixing problems is expensive	Usually lower because prevention is cheaper than correction
Example	Fixing a software bug after it affects users	Performing testing and code review to avoid bugs

Software Measurement

- Process of **quantifying attributes/measure feature** of a software product or process.
- Helps understand **quality, size, complexity, performance, and progress**.
- Follows **ISO standards** for consistency and reliability.

Principles of Software Measurement

1. **Formulation** – Decide what to measure and define metrics.
2. **Collection** – Gather raw data for metrics.
3. **Analysis** – Use math/statistics to compute metrics.
4. **Interpretation** – Understand what the results say about quality or performance.
5. **Feedback** – Share results to improve future work.

Need of Software Measurement

- Check current quality of product/process
- Make data-driven decisions
- Predict future quality
- Improve software quality

- Increase productivity and efficiency
- Ensure compliance with standards
- Control budget and schedule
- Identify bottlenecks and areas to improve

Types of Software Measurement

1. Direct Measurement – Can be measured directly using standard scales.
Examples:

- Lines of Code (LOC)
- Execution time
- Number of defects
- CPU/memory usage

2. Indirect Measurement – Measures attributes that **cannot be observed directly**, uses related factors.
Examples:

- Complexity
- Maintainability
- Reliability
- Quality

ISO 9000 Models

- **ISO (International Standards Organization)** is a group of 63 countries that works to create and promote standards.
- The **ISO 9000 series** was introduced in **1987**.
- It acts as a **reference standard** for agreements between independent parties.
- ISO 9000 gives **guidelines for managing a quality system**.
- It defines **processes and organizational practices(how production should be managed)**, not what the product should be.

Types of ISO 9000 Quality Standards
1. ISO 9001

- For organizations involved in **design, development, production, and servicing**.
- This is the main standard used by **software development companies**.

2. ISO 9002

- For organizations that **do not design products** but only **manufacture** them.
- Examples: steel industry, car manufacturing (using external designs).
- Not applicable to software development.

3. ISO 9003

- For organizations involved only in **installation and testing**.
- Example: gas companies.

ISO 9000 Certification – Simplified Steps

1. **Application:** Organization applies for registration.
2. **Pre-Assessment:** Registrar makes an initial/rough review.
3. **Document Review:** Registrar checks submitted documents and suggests improvements.
4. **Compliance Audit:** Registrar checks whether improvements were completed.
5. **Registration:** Certification is granted if all steps are successful.
6. **Continuous Inspection:** Registrar monitors the organization periodically.

-`

What is a Risk?

A **risk** doesn't mean danger but an uncertain event or condition that may or may not occur, and if it does, it can have a **positive or negative** effect on a project or organization's objectives.

Risks can affect

- Money/Finances
- Operations/Processes
- Reputation
- Employees
- Assets/Resources

Main Characteristics of a Risk

- **Uncertainty:** A risk is not guaranteed to occur—there are *no 100% certain risks*.
- **Impact/Loss:** If a risk occurs, it can cause undesirable consequences (financial loss, delays, safety issues).
  (Positive risks/opportunities may bring benefits. Like experienced developer can do project faster getting that developer is uncertain)

☑ What is Risk Management?

Risk management is the process of:

1. **Finding risks** that could affect a project or organization.
2. **Analyzing their impact**.
3. **Planning how to handle them**.
4. **Monitoring** them continuously.

Common Sources of Risk

1. **Financial instability** – Not enough money or poor budget planning.
2. **Cybersecurity threats** – Hackers, data breaches, or malware.
3. **Accidents** – Unexpected mishaps during work.
4. **Natural disasters** – Floods, earthquakes, storms, etc.
5. **Legal or regulatory issues** – Breaking laws or not following rules.
6. **Human error** – Mistakes by employees.
7. **Strategic planning failures** – Wrong decisions about direction or goals.

Project-Specific Sources of Risk

1. **Misunderstanding customer requirements** – Not knowing what the customer really wants.
2. **Misestimation of team skills** – Wrong assumptions about team capability or coordination.
3. **Incorrect budget estimation** – Underestimating costs or resources needed
4. **Unrealistic promises to customers** – Saying things you can't deliver.
5. **Incorrect assumptions about system design** – Overestimating system strength or flexibility.
6. **Uncontrolled changes to requirements** – Constant changes that disrupt the project.
7. **Misjudging new methods** – Risks from adopting new technologies or processes.

Main goal:
To **anticipate possible risks** and develop effective responses to minimize negative impacts and maximize opportunities.

☑ Why Is Risk Management Important?

Risk management helps organizations:

- Prepare for unexpected events
- Protect financial stability

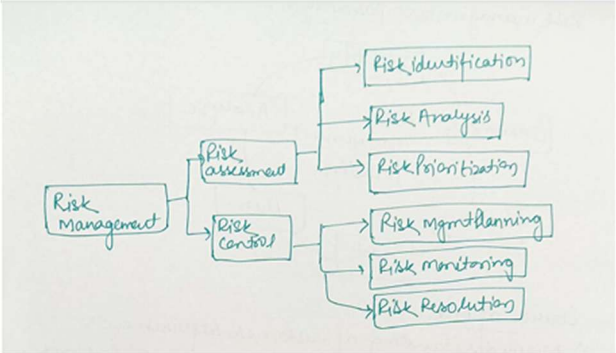
- Ensure continuity of operations
- Reduce losses
- Improve decision-making
- Increase the likelihood of meeting objectives

☑ Risk Management Paradigm

The **Risk Management Paradigm** includes the following key steps:

1. **Identify:** Discover and document potential risks that could affect the project.
2. **Analyse:** Assess the probability, impact, and root causes of each risk.
3. **Plan:** Develop strategies to avoid, mitigate, transfer, or accept risks.
4. **Track:** Continuously monitor the status of risks as the project progresses.
5. **Control:** Implement corrective actions, adjust plans, and ensure risks remain manageable.
6. **Communicate:** Share risk information with team members and stakeholders to maintain awareness and alignment.

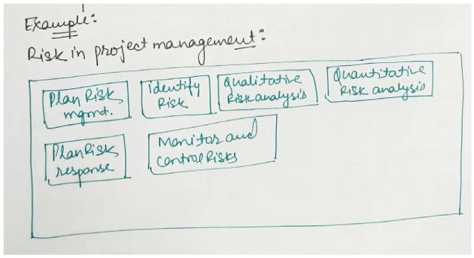
☑ Risk Management Activities Structure



• Risk Assessment – Evaluating risks to understand likelihood and impact
  o Risk Identification – Finding and documenting potential risks

- Risk Analysis – Assessing probability, impact, and root causes
- Risk Prioritization – Ranking risks based on severity and urgency

• Risk Control – Taking actions to manage and respond to risks
- Risk Planning – Developing strategies to avoid, mitigate, transfer, or accept risks
- Risk Monitoring – Tracking risks and checking if responses are effective
- Risk Resolution – Implementing actions to eliminate or reduce risks to acceptable levels



✅ 1. Risk Identification

The project manager must identify risks as early as possible to reduce their potential impact through proper planning.
Common techniques include:

- **Brainstorming:** Generating a wide range of risk ideas collaboratively with the team without judging or evaluating them
- **SWOT Analysis:** Identifying risks by evaluating strengths, weaknesses, opportunities, and threats.
- **Causal Mapping:** Identifying root causes and relationships between risks.
- **Flowchart Methods:** Analyzing process steps and workflows to uncover potential risk points.

Types of Risks

1. **Technology Risk:** Issues related to using new, immature, or complex technologies.
2. **People Risk:** Risks tied to personnel availability, skills, or performance.
3. **Organizational Risk:** Internal issues such as changes in strategy, restructuring, or project environment.
4. **Tools Risk:** Problems with tools, platforms, or support systems required for development.
5. **Requirements Risk:** Changing, unclear, or unstable customer requirements.
6. **Estimation Risk:** Incorrect estimation of time, cost, effort, or required resources.

✅ 2. Risk Analysis and Prioritization

In the analysis phase, the project team:

1. Identifies the cause of each risk
2. Determines the likelihood of the risk occurring
3. Evaluates the impact if it occurs

This step may include:

- **Qualitative risk analysis**
- **Quantitative risk analysis**

Probability Categories

- **Very Low (0–10%)**: Tolerable risk (minimal or no harm)
- **Low (10–25%)**: Minor effect
- **Moderate (25–50%)**: Medium risk (some impact on time/cost)
- **High (50–75%)**: Major impact on schedule or budget
- **Very High (75–100%)**: Intolerable risk (serious damage to time, cost, quality, performance)

✅ **3. Risk Planning**

Risk planning defines how to respond to significant risks.

Main Risk Response Strategies

1. **Avoidance**
   Change requirements, reduce project scope, or adjust processes to eliminate the risk.
2. **Transfer**
   Shift the risk to a third party (insurance, outsourcing, purchasing components).
3. **Mitigation / Reduction**
   Take actions to reduce likelihood or impact.
   Example: planning for backup resources if key staff may leave.
4. **Acceptance**

   Acceptance means **acknowledging the risk but not taking immediate action**, usually because the risk is low or the cost of response is too high.
   Two types:
   - **Passive Acceptance:** Do nothing; deal with the risk if it occurs.
   - **Active Acceptance:** Create a small reserve (time, money) to handle potential impacts.
5. **Contingency planning**

   Contingency planning establishes **predefined backup plans** to follow if a risk occurs.

✅ **4. Risk Monitoring**

Risk monitoring is a continuous process throughout the project.
It involves:

- Tracking identified risks
- Detecting new risks
- Evaluating whether mitigation actions are working
- Updating assumptions and taking corrective action

✅ **5. Risk Resolution**

Risk resolution ensures that risks are handled within acceptable limits.
Success depends on:

- Accurate identification
- Effective analysis
- Strong mitigation planning
- Quick response to issues as they arise

It focuses on keeping the project on track by resolving risks early and efficiently.

RMMM Plan (Risk Mitigation, Monitoring, and Management)

1️⃣ **RMMM**

- Stands for **Risk Mitigation, Monitoring, and Management**.
- It's the **concept or framework** used to handle risks in a project.
- Think of it as **"what you do"** when dealing with risks: mitigate, monitor, and manage.

2️⃣ **RMMM Plan**

- This is the **documented version of RMMM**.
- It explains **how the RMMM process will be applied** in a specific project.
- Includes details like:
    - Identified risks
    - Mitigation strategies
    - Monitoring procedures
    - Contingency plans
    - Risk register
- Think of it as **"the blueprint or roadmap"** for implementing RMMM in a project.

The **RMMM plan** is a key component of **software project management**. It helps project managers handle potential risks by:

1. **Identifying risks**
2. **Mitigating them**
3. **Monitoring their progress**
4. **Managing risks if they occur**

In some teams, risks are documented using a **Risk Information Sheet (RIS)** and stored in a database for easy tracking, prioritization, and analysis.

1️⃣ Risk Mitigation

Purpose: Avoid risks before they happen (**Risk Avoidance**).
Steps:

1. Identify potential risks.
2. Remove causes that could create risks.
3. Update documents regularly.
4. Conduct timely reviews to speed up work.

Example (High Staff Turnover):

- Meet staff to understand causes of turnover (low pay, poor working conditions, competitive job market).
- Mitigate controllable causes before the project starts.
- Organize project teams so knowledge is spread widely.
- Set documentation standards and timely documentation.
- Assign backup staff for critical roles.

2️⃣ Risk Monitoring

Purpose: Track risks during the project.
Objectives:

- Check if predicted risks are occurring.
- Ensure risk mitigation steps are being applied.
- Collect data for future risk analysis.
- Determine which problems are caused by which risks.

Example (High Staff Turnover):

- Monitor team morale and interpersonal relationships.
- Track potential compensation issues.
- Observe job availability internally and externally.

3️⃣ Risk Management (Contingency Planning)

Purpose: Respond when mitigation fails and risks occur.
Steps:

- Implement backup plans and reassign resources.
- Adjust project schedules if needed.
- Ensure continuity by leveraging documented information and dispersed knowledge.

Example (High Staff Turnover):

- If employees leave, use backups and documented knowledge.

- Refocus resources on fully staffed areas.
- Integrate new team members efficiently.

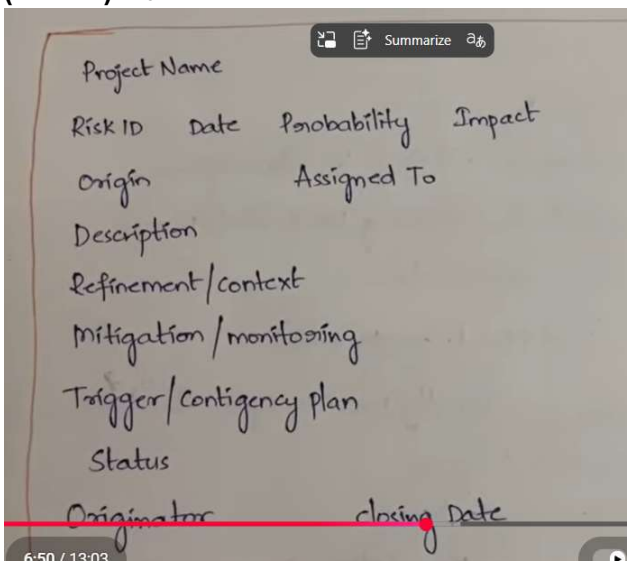
Drawbacks of RMMM

- Increases project costs.
- Takes extra time.
- Can be tedious for large projects.
- Does **not guarantee a risk-free project**; new risks may appear after delivery.

✅ **What is RIS in RMMM?**

RIS = Risk Information Sheet, a structured document used to record detailed information about each identified risk.

RIS is commonly included as part of the **Risk Mitigation, Monitoring, and Management (RMMM) Plan**.



- Project Name: Name of the project where risks are being logged.
- Risk ID: Unique identifier for each risk.
- Date: The date the risk is identified or updated.
- Probability: Likelihood of the risk occurring, typically rated as High/Medium/Low or numerically (e.g., 0.2).
- Impact: Severity/consequence if the risk occurs (often a scale).
- Origin: Source or cause of the risk.
- Assigned To: The team member responsible for addressing or monitoring the risk.
- Description: Brief explanation of the risk.
- Refinement/Context: Additional details, background, or circumstances relevant to the risk.
- Mitigation/Monitoring: Steps taken to reduce likelihood or impact and how it's being monitored.
- Trigger/Contingency Plan: Events or signals indicating risk realization, and backup plans if risk materializes.
- Status: Current condition (e.g., active, resolved, closed).

- Originator: Person who reported or logged the risk.
- Closing Date: Date when the risk is declared closed/resolved.

Software Project Risk Management (Identification + Projection + Refinement)

---

1 Risk Identification

Risk Identification means **finding all possible risks** that may affect the project.

A. Types of Risk Identification

1. Generic Risks

- Common to all software projects.
- Do not depend on technology or team.
- Examples: requirement changes, schedule delays, staffing issues.

2. Product-Specific Risks

- Depend on the **specific project**, such as:
    - Technology stack
    - Team skill levels
    - Development tools
    - Unique system requirements

---

2 Steps in risk identification

Preparation of Risk Item Checklist

A checklist helps ensure all types of risks are considered.

Checklist Categories:

1. **Product Size** – software size, complexity, LOC, features.
2. **Business Impact** – market effect, penalties, customer importance.
3. **Customer Characteristics** – requirement clarity, involvement, domain knowledge.
4. **Process Definition** – maturity of development process.
5. **Development Environment** – tool availability, hardware/software support.
6. **Technology to Be Built** – newness, uncertainty, complexity.
7. **Staff Size & Experience** – skill levels, availability, team stability.

✓ Helps identify **generic** and **product-specific** risks.

Creation of Risk Components & Drivers List
Risk Components (4):

1. **Performance Risk** – product may not meet requirements.
2. **Cost Risk** – project may exceed budget.
3. **Support Risk** – maintenance may be difficult.
4. **Schedule Risk** – project may miss deadlines.

Risk Drivers (Causes):

- Requirement changes
- New/unproven technology
- Staff inexperience
- Poor documentation
- Schedule pressure
- Lack of tools
- High complexity

✓ Components tell **which area is impacted**.
✓ Drivers tell **what causes the risk**.

---

2 Risk Projection **(Risk Estimation)**

Attempts to evaluate each risk in two ways:

- **Probability** (likelihood of the risk happening)
- **Impact** (damage if the risk happens)

Activities in Risk Projection

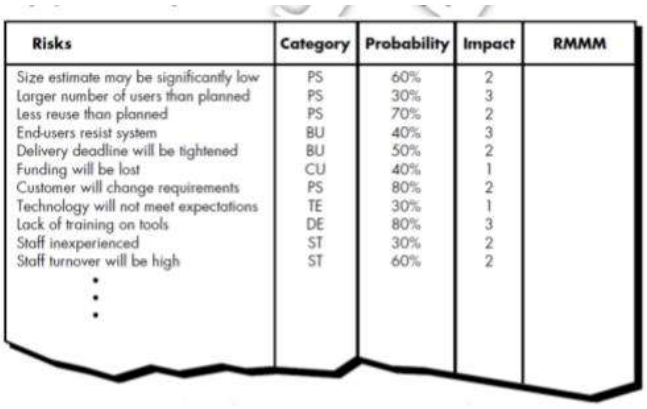
1. Establish a scale for probability
2. Estimate probability of each risk
3. Estimate impact of each risk
4. Determine accuracy of the estimates

Risk Table

A table that organizes and prioritizes risks.

Columns in Risk Table

1. **Risk** – Description of the risk
2. **Category** – Type of risk
    - PS = Project Size



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 than 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 requirements	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	
•				
•				
•				

    - BU = Business
    - ST = Staff
    - DE = Development Environment
    - GS = Group Skill
3. **Probability** – Chance of occurrence (e.g., 60%, 40%)
4. **Impact** – Severity (1–4 scale)
    - 1 = Catastrophic
    - 2 = Critical
    - 3 = Marginal
    - 4 = Negligible

Purpose

- Sort risks by **probability + impact**
- High-probability & high-impact risks go to the **top**
- Low-probability & low-impact risks go to the **bottom**
  → **First-order risk prioritization**

---

3 Risk Refinement

Used when a risk is stated too generally in early planning.
As more information is known, the risk is **refined** into more detailed, specific risks.

CTC Format (Condition–Transition–Consequence)

Used to rewrite and clarify a risk.

Format:
Given that , then there is concern that (possibly) .

(Refinement Example)

General Risk

“Only 70% of reusable components may be integrated.”

Refined CTC Statement

Given that all reusable components must meet design standards and some do not,
there is concern that only 70% will integrate, and the remaining 30% must be custom-developed.

Subconditions (Further Refinement)

Break down the general condition into smaller, specific causes:

Subcondition 1

Some reusable components were developed by third parties and do not follow internal design standards.

Subcondition 2

The design standards for component interfaces are not finalized and may not match existing reusable components.
