

|                                     |                                   |                           |
|-------------------------------------|-----------------------------------|---------------------------|
| Year: 2 <sup>nd</sup>               | Semester: 4 <sup>th</sup>         | Academic Session: 2024-25 |
| Course Code: ECS404<br>[B.Tech CSE] | Course Name: Software Engineering |                           |

## UNIT 1

### Evaluation and Impact of Software Engineering

#### *Understanding the Impact of Software Engineering*

Software engineering plays a **critical role** in enabling the technological systems that power modern society. Here's a structured view of how it's evaluated and its wide-ranging impact:

---

#### 1. Evaluation of Software Engineering

Evaluation in software engineering involves **assessing processes, practices, and outcomes** to ensure quality, efficiency, and alignment with user needs. Key areas of evaluation include:

- **Process Assessment and Improvement:** Techniques like **CMMI (Capability Maturity Model Integration)** and **ISO standards** are used to evaluate the maturity and quality of software processes.
- **Metrics and Measurement:** Software engineering uses **product and process metrics** to evaluate performance, productivity, and software quality. These include measures like defect density, code coverage, and productivity rates.
- **Testing and Quality Assurance:** Through rigorous testing strategies (unit, integration, system, acceptance), the functionality and reliability of software are validated.

Here is a clear explanation of three important **software engineering metrics: Defect Density, Code Coverage, and Productivity Rates.**

---

## Defect Density

### Definition:

Defect Density = Total Defects Found / Size of Work Product

**Unit:** Defects per KLOC (Thousand Lines of Code), UML diagram, or page

### Purpose:

- Measures software quality.
- Helps predict error rates in future projects.
- Lower defect density = Higher quality software.

### Example:

If a 32-page requirements document has 22 defects:

- Defect Density =  $22 / 32 = 0.68$  errors per page
- 

## Code Coverage

### Definition:

A metric that indicates **how much of the source code is executed** during testing.

### Types:

- **Statement Coverage** – Measures if each statement is executed.
- **Branch Coverage** – Measures if all branches (e.g., if-else) are tested.
- **Path Coverage** – Checks all paths through the code.

### Purpose:

- Identifies untested parts of code.

- Improves test quality.
- Essential for safety-critical systems.

**Tools:** McCabeTest, TestWorks, e-TEST Suite

---

## Productivity Rates

### Definition:

Estimates the amount of development work completed over time.

### Expressed As:

- **LOC/month (Lines of Code per month)**
- **FP/month (Function Points per month)**

### Purpose:

- Measures development team efficiency.
- Helps in project cost estimation and scheduling.

## 2. Impact of Software Engineering

The **impact** of software engineering spans multiple dimensions:

### *a. Economic Impact*

- Software engineering drives the **global software economy**, from small-scale app developers to multinational tech corporations.
- Reduces costs of failures by improving quality and reducing time-to-market.

### *b. Social Impact*

- Powers critical sectors: **healthcare, transportation, education, communication**, and more.

- Systems like **health management platforms**, **air traffic control**, and **weather forecasting** rely on dependable software.

#### *c. Environmental and Global Challenges*

- Enables solutions for climate modeling, resource optimization, and renewable energy systems.
- Plays a key role in addressing 21st-century challenges like **climate change**, **sustainable development**, and **population management**.

#### *d. Technological Advancement*

- Made possible the **Internet**, **mobile applications**, **cloud computing**, and **AI-powered systems**.
- Software engineering has transformed **industries**, **communication**, and **human life** overall.

#### *e. Cultural Impact*

- Shaped modern digital culture via social media, gaming, and mobile apps.
- Democratized access to technology, allowing more participation in the digital economy.

---

## Evaluation and Impact of Software Engineering

Software engineering is a **systematic, disciplined, and quantifiable approach** to the development, operation, and maintenance of software systems. Its evaluation and impact can be understood through various dimensions:

---

### 1. Evaluation of Software Engineering

Evaluating software engineering practices involves assessing both the processes and the products to ensure they meet desired quality standards and effectively address user requirements.

### a. Process Assessment and Improvement

- **Capability Maturity Model Integration (CMMI):** A framework that assesses the maturity of software development processes, guiding organizations in process improvement.

### b. Metrics and Measurement

- **Performance Metrics:** Evaluating aspects like response time and throughput to ensure software meets performance expectations.
- **Quality Metrics:** Assessing defect density, code complexity, and maintainability to gauge software quality.
- **Process Metrics:** Measuring efficiency indicators such as development time and resource utilization.

### c. Testing and Quality Assurance

- **Unit Testing:** Verifying individual components for correct functionality.
  - **Integration Testing:** Ensuring combined components work together as intended.
  - **System Testing:** Validating the complete system against specified requirements.
  - **Acceptance Testing:** Confirming the system meets business needs and is ready for deployment.
- 

## 2. Impact of Software Engineering

The influence of software engineering extends across various sectors, driving innovation and efficiency.

### a. Economic Impact

- **Industry Growth:** Software engineering propels the global software industry, fostering economic development and creating employment opportunities.

- **Cost Efficiency:** Implementing structured engineering practices reduces development costs and mitigates the expenses associated with software failures.

#### b. Social Impact

- **Enhanced Communication:** Software solutions have revolutionized communication, enabling instant connectivity across the globe.
- **Healthcare Advancements:** Software applications support diagnostic tools, patient management systems, and telemedicine, improving healthcare delivery.
- **Educational Tools:** E-learning platforms and educational software have democratized access to knowledge.

#### c. Environmental Impact

- **Energy Efficiency:** Software optimizes energy consumption in various systems, contributing to environmental sustainability.
- **Smart Cities:** Software engineering underpins the development of smart grids and intelligent transportation systems, promoting efficient urban living.

#### d. Technological Advancement

- **Innovation Enablement:** Software engineering is foundational to emerging technologies like artificial intelligence, blockchain, and the Internet of Things (IoT).
- **Automation:** Software drives automation in industries, enhancing productivity and precision.

## Software Development Life Cycle Models:

### 1. Waterfall Model (Classic Life Cycle)

#### Concept:

- A **linear and sequential** model where each phase must be completed before the next begins.

#### Phases:

1. **Requirements** Gathering
2. **System Design**
3. **Implementation (Coding)**
4. **Integration & Testing**
5. **Deployment**
6. **Maintenance**

**Best For:**

- Projects with **clearly defined and fixed requirements**.

**Drawback:**

- Difficult to go back; changes are expensive and late feedback is not handled well.
- 

## 2. Prototype Model

**Concept:**

- Build a **quick, rough version** of the system, get feedback, and refine.

**Steps:**

1. Collect basic requirements
2. Create a quick design
3. Develop a prototype
4. User feedback

5. Refine and build final product

**Best For:**

- When **requirements are unclear or exploratory**.

### 3. Spiral Model

**Concept:**

- Combines iterative development and systematic aspects of waterfall. It's **risk-driven**.

**Phases (in each spiral loop):**

1. **Planning**
2. **Risk Analysis**
3. **Engineering**
4. **Evaluation**

**Best For:**

- **Large, high-risk projects** where continuous risk analysis is important.

### 4. Agility and Agile Process Model

**Concept:**

- Emphasizes **flexibility, customer collaboration, and embracing change**.

### Core Principles:

- Individuals over processes
- Working software over documentation
- Customer collaboration
- Responding to change

## 5. Extreme Programming (XP)

### Concept:

- Agile model focusing on **customer satisfaction, simplicity, and frequent delivery.**

### Practices:

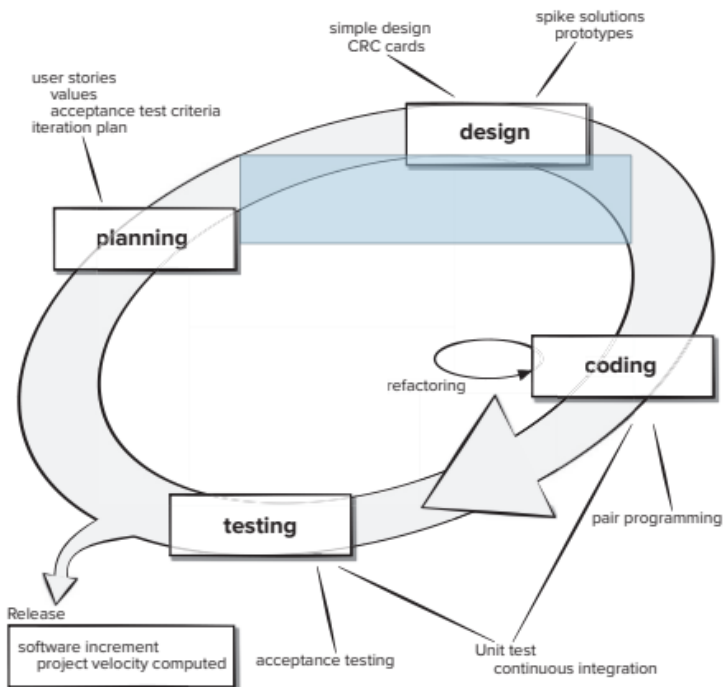
- Pair programming
- Test-first development
- Small releases
- Continuous integration
- Refactoring

### Best For:

- **Fast-paced development** with active user involvement.

**FIGURE 3.3**

**The Extreme Programming process**

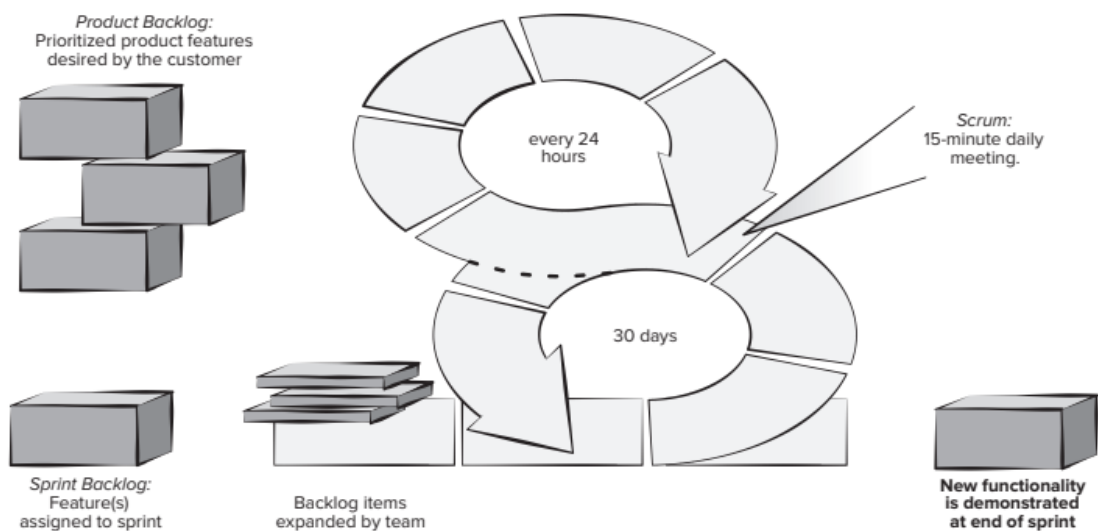


## 6. Other Agile Process Models

### a. Scrum:

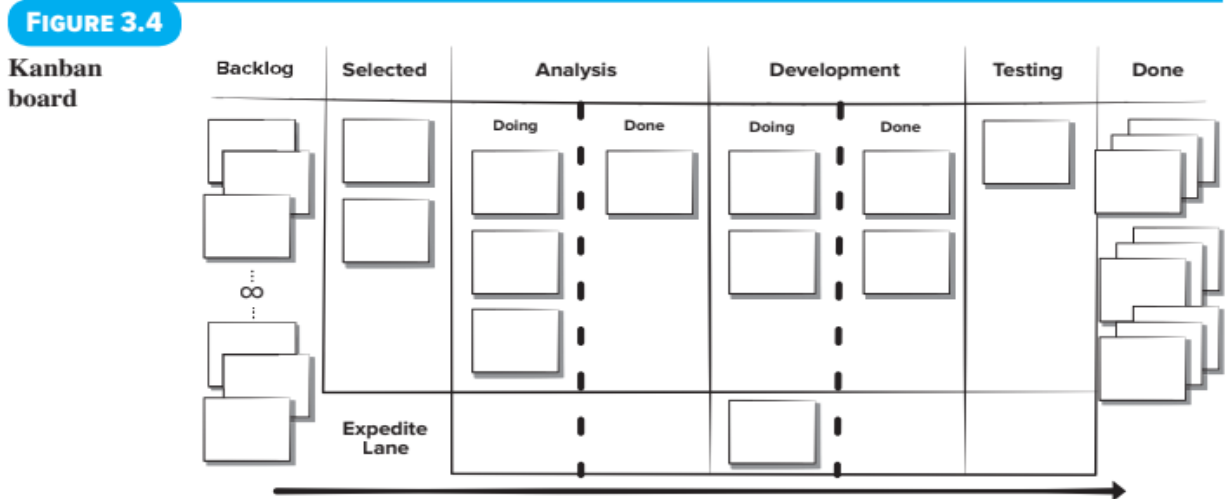
- Time-boxed iterations and **sprints**, daily stand-ups, roles like **Product Owner** and **Scrum Master**.

**FIGURE 3.2** Scrum process flow



## b. Kanban:

- Visualize workflow, limit work-in-progress, continuous delivery.



## 7. Layered Approach

### Concept:

- The SDLC models are supported by **process layers** like:
  - **Communication** (requirements)
  - **Planning**
  - **Modeling** (design)
  - **Construction** (coding)
  - **Deployment**

It shows that regardless of the model, these layers are **present and essential**.

# Software Requirement Analysis And Specification:

## What is Software Requirement Analysis?

**Software Requirement Analysis** is the process of understanding **what the customer needs** from a software system. It involves:

- Studying the problem
- Understanding the needs of stakeholders
- Identifying conflicts or inconsistencies
- Deriving **functional** and **non-functional** requirements

This step is **crucial** because any mistake here can lead to costly rework later in the software life cycle.

---

## Types of Requirements

### *1. Functional Requirements (FRs)*

These describe **what the system should do**.

Example:

“The system shall allow users to log in using a username and password.”

They define:

- Services
- Reactions to inputs
- Behavior in certain situations

## 2. Non-Functional Requirements (NFRs)

These describe **how the system performs** its tasks or constraints on the system.

Examples:

- **Performance** (e.g., system response time should be under 2 seconds)
- **Security** (e.g., only authorized users can access patient data)
- **Reliability** (e.g., 99.99% uptime)

They are **often more critical** than functional requirements because failing them might make the system unusable.

---

## What is Requirements Specification?

**Requirements Specification** is the process of **formally documenting** user and system requirements. It results in the **Software Requirements Specification (SRS)** document.

The SRS should:

- Be understandable to both developers and clients
- Be complete, consistent, and verifiable
- Describe both **user requirements** (in natural language) and **system requirements** (more detailed and technical)

**Example:**

**User Requirement:**

The user should be able to reset their password.

### System Requirement:

If a user forgets their password, the system shall send a reset link to their registered email within 30 seconds of request.

### Key Activities in Requirements Engineering

1. **Elicitation** – Interacting with stakeholders to gather requirements
2. **Analysis** – Understanding and resolving conflicts or ambiguities
3. **Specification** – Documenting requirements clearly
4. **Validation** – Ensuring requirements meet stakeholder needs
5. **Management** – Handling changes in requirements over time

## What is a Feasibility Study?

A **Feasibility Study** is a **short, focused assessment** conducted early in the **requirements engineering process** to determine whether a proposed software project is **viable** and **worth pursuing**.

---

### Purpose of a Feasibility Study

The goal is to **evaluate the practicality** of the software idea. It answers the following **three key questions**:

1. **Organizational Feasibility:**  
*Does the system support the business goals and objectives of the organization?*

2. **Technical Feasibility:**

*Can the system be implemented using existing technology, tools, and expertise?*

3. **Economic Feasibility:**

*Can the system be built within budget and time constraints?*

If the answer to any of these is “**No**”, then the project **should not proceed**.

### **Additional Dimensions of Feasibility**

From Pressman’s perspective, a feasibility study also involves evaluating:

- **Scope and Constraints** (functions, performance, environment)
- **Risk Evaluation** (cost, tech, schedule, team)
- **Business Needs** (will users benefit from this product?)

### **Summary Table**

| Type of Feasibility | Description                      |
|---------------------|----------------------------------|
| Organizational      | Aligns with strategic goals      |
| Technical           | Technology & skill availability  |
| Economic            | Cost-benefit and budget concerns |

**Legal & Regulatory**

Compliance with laws/standards

**Operational**

Usability and stakeholder acceptance

## Requirements Gathering:

**Requirements gathering** is the first and one of the most crucial steps in **Requirements Engineering**. It focuses on **discovering and understanding** what stakeholders need from the system.

---

### Definition:

**Requirements elicitation** is the process of collecting requirements from stakeholders through **interviews, observations, meetings**, and other collaborative techniques.

### Collaborative Requirements Gathering

A widely used approach involves structured, team-oriented sessions with the following **guidelines**:

1. Meetings include both software engineers and stakeholders.
2. Preparation and participation rules are set.
3. A balanced agenda is followed.

4. A **facilitator** leads the meeting.
5. Tools like worksheets, flipcharts, or virtual boards are used.

---

## Techniques for Requirements Gathering

, common techniques include:

- **Interviews:**
  - *Closed*: Predefined questions
  - *Open*: Exploratory discussions
- **Observation:**
  - Watch users in their natural environment
- **Workshops & Brainstorming**
- **Prototyping**
- **Scenarios & Use Cases**: Describe how the system is expected to be used

---

## Challenges in Requirements Gathering

- Stakeholders may not know exactly what they want.
- Miscommunication due to technical vs. business language.
- Conflicting requirements among different stakeholders.

- Requirements can evolve or change over time.

---

## **Outputs of Requirements Gathering**

Work products include:

1. Statement of need & feasibility
2. System scope
3. Stakeholder list
4. Technical environment
5. Functional & domain requirements
6. Use scenarios or use cases
7. Prototypes (if applicable)

## **Requirement Analysis and Specification using DFD (Data Flow Diagrams)**

---

**What is a DFD?**

A **Data Flow Diagram (DFD)** is a **graphical tool** used in **requirement analysis** to visually represent the flow of data within a system. It is **simple, intuitive**, and helps in **specifying requirements functionally** by showing how input data is transformed into output through a system.

### Why Use DFDs in Requirement Analysis?

- Clarifies system processes and data transformation
  - Easily understood by stakeholders
  - Helps in functional requirement specification
  - Foundation for system design
- 

### Components of a DFD

| Symbol          | Description                                          |
|-----------------|------------------------------------------------------|
| Process         | Function or transformation applied to data           |
| Data Store      | Where data is stored for later use                   |
| Data Flow       | Data movement between processes, stores, or entities |
| External Entity | Source or destination of data outside the system     |

---

## DFD Levels

| Level                     | Description                                                                           |
|---------------------------|---------------------------------------------------------------------------------------|
| Context Diagram (Level 0) | Shows the whole system as a single process and its interaction with external entities |
| Level 1 DFD               | Breaks down the main process into sub-processes                                       |
| Level 2+ DFDs             | Further decomposes complex subprocesses                                               |

---

## Example Use Case: Library Management System

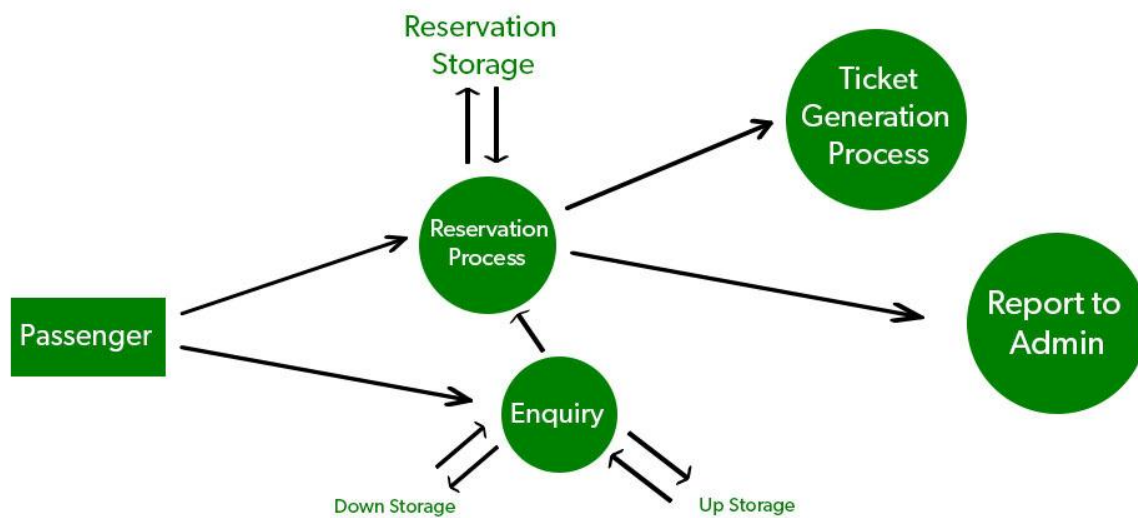
### *Context Diagram (Level 0)*

- Entities: Passenger, Admin
- Process: Railway Ticket Reservation System
- Data Flows: Cancellation, Reservation, Ticket info etc.



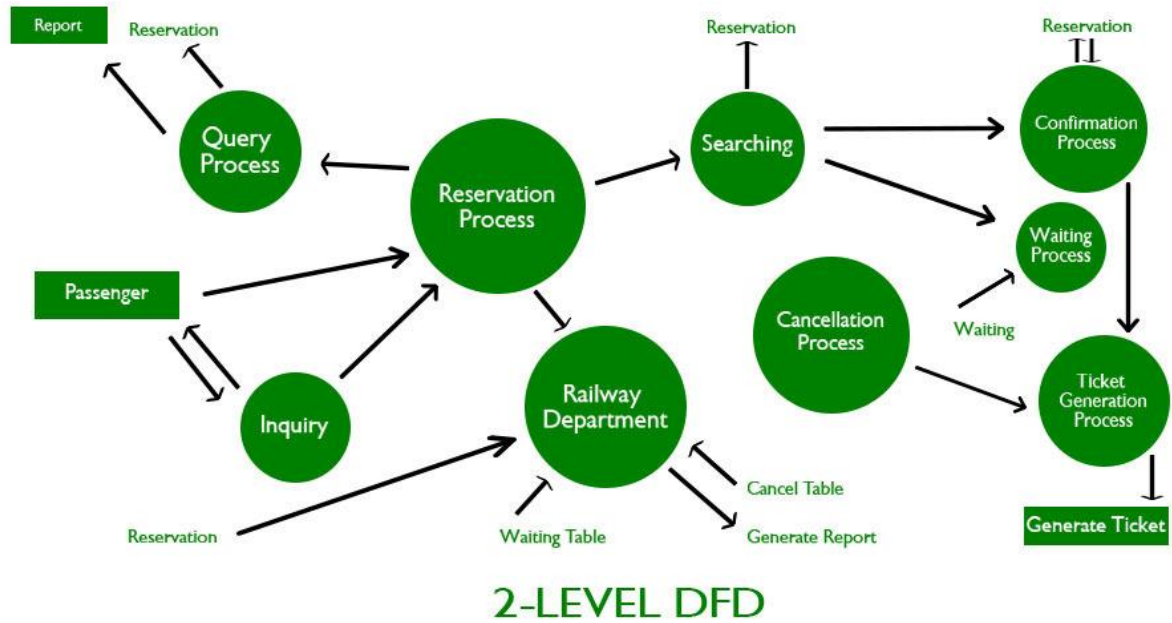
**O-LEVEL DFD**

**Level 1 Breakdown:**



**1-LEVEL DFD**

**Level 2 Breakdown:**



## Data Dictionary

A **Data Dictionary** is a **repository** that stores detailed information about the data elements used in a system. It describes:

- Data name
- Description
- Type (integer, string, etc.)
- Format
- Source and destination
- Usage and constraints

**Purpose:**

- Acts as a **reference for developers and analysts**
- Helps ensure **consistency** across the system
- Facilitates **communication** among team members

---

## ER Diagram (Entity-Relationship Diagram)

An **ER Diagram** is used to **model the data structure** of a system at a high level. It shows how **entities (things)** relate to each other.

---

### Key Components

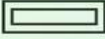
| Component    | Description                                      |
|--------------|--------------------------------------------------|
| Entity       | A thing or object (e.g., Student, Course)        |
| Attribute    | Properties of an entity (e.g., Name, ID)         |
| Relationship | How entities are connected (e.g., "enrolled in") |

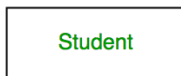
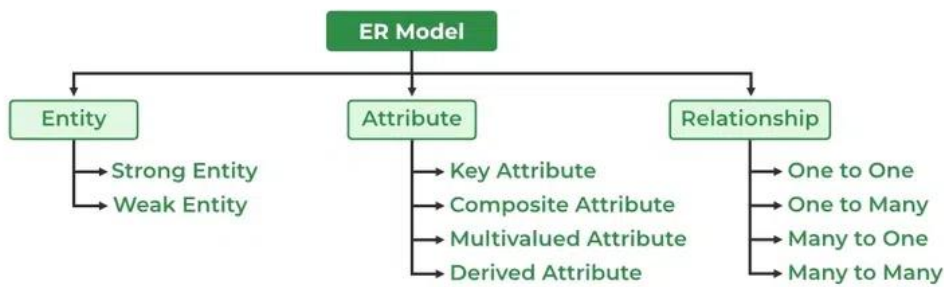
---

## Types of Relationships

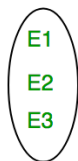
- **One-to-One (1:1):** A person has one passport.
  - **One-to-Many (1:N):** A teacher teaches many students.
  - **Many-to-Many (M:N):** Students enroll in multiple courses and each course has multiple students.
- 

## ER Diagram :

| Figures          | Symbols                                                                             | Represents                                                           |
|------------------|-------------------------------------------------------------------------------------|----------------------------------------------------------------------|
| Rectangle        |  | Entities in ER Model                                                 |
| Ellipse          |  | Attributes in ER Model                                               |
| Diamond          |  | Relationships among Entities                                         |
| Line             |  | Attributes to Entities and Entity Sets with Other Relationship Types |
| Double Ellipse   |  | Multi-Valued Attributes                                              |
| Double Rectangle |  | Weak Entity                                                          |



Entity Type



Entity Set

## Types of Entity

There are two types of entity:

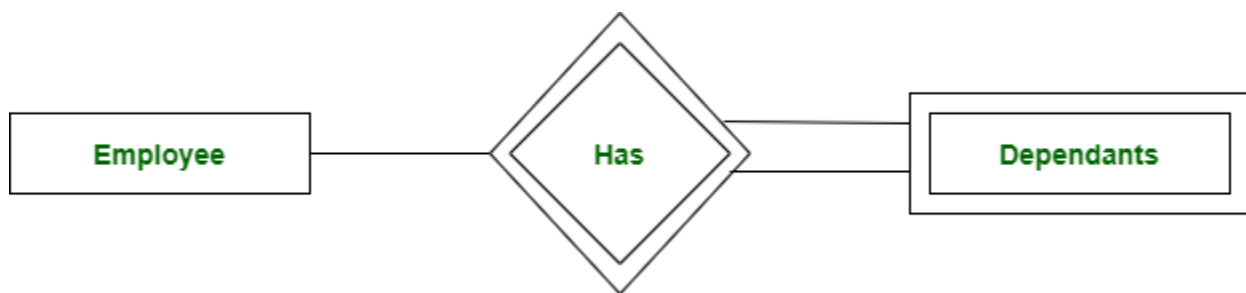
### 1. Strong Entity

A [Strong Entity](#) is a type of entity that has a key Attribute. Strong Entity does not depend on other Entity in the Schema. It has a primary key, that helps in identifying it uniquely, and it is represented by a rectangle. These are called Strong Entity Types.

### 2. Weak Entity

An Entity type has a key attribute that uniquely identifies each entity in the entity set. But some entity type exists for which key attributes can't be defined. These are called [Weak Entity types](#).

**For Example,** A company may store the information of dependents (Parents, Children, Spouse) of an Employee. But the dependents can't exist without the employee. So Dependent will be a **Weak Entity Type** and Employee will be Identifying Entity type for Dependent, which means it is **Strong Entity Type**.



## Types of Attributes

### 1. Key Attribute

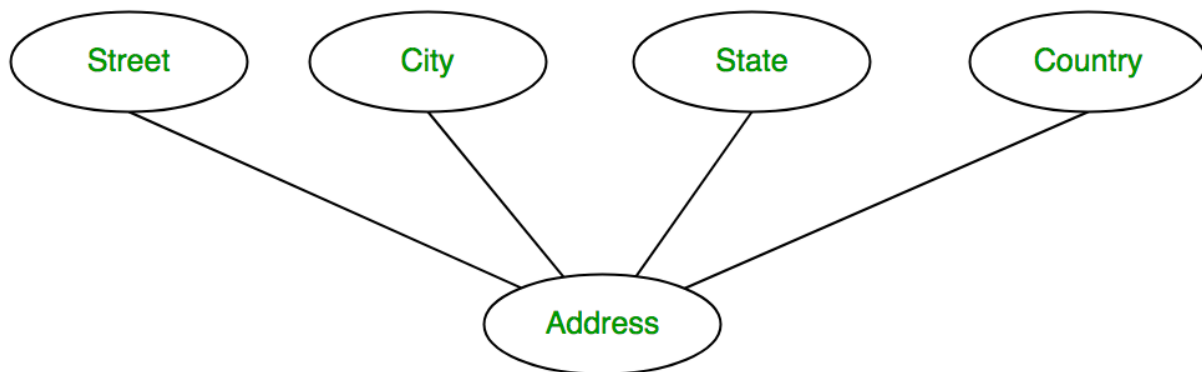
The attribute which **uniquely identifies each entity** in the entity set is called the key attribute. For example, Roll\_No will be unique for each student. In the ER diagram, the key attribute is represented by an oval with underlying lines.



*Key Attribute*

## 2. Composite Attribute

An attribute **composed of many other attributes** is called a composite attribute. For example, the Address attribute of the student Entity type consists of Street, City, State, and Country. In the ER diagram, the composite attribute is represented by an oval consisting of ovals.



*Composite Attribute*

## 3. Multivalued Attribute

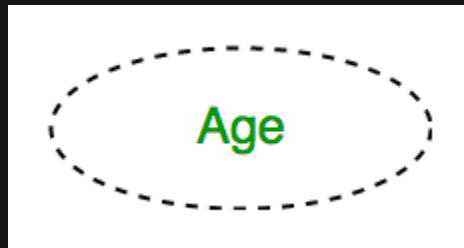
An attribute consisting of more than one value for a given entity. For example, Phone\_No (can be more than one for a given student). In the ER diagram, a multivalued attribute is represented by a double oval.



*Multivalued Attribute*

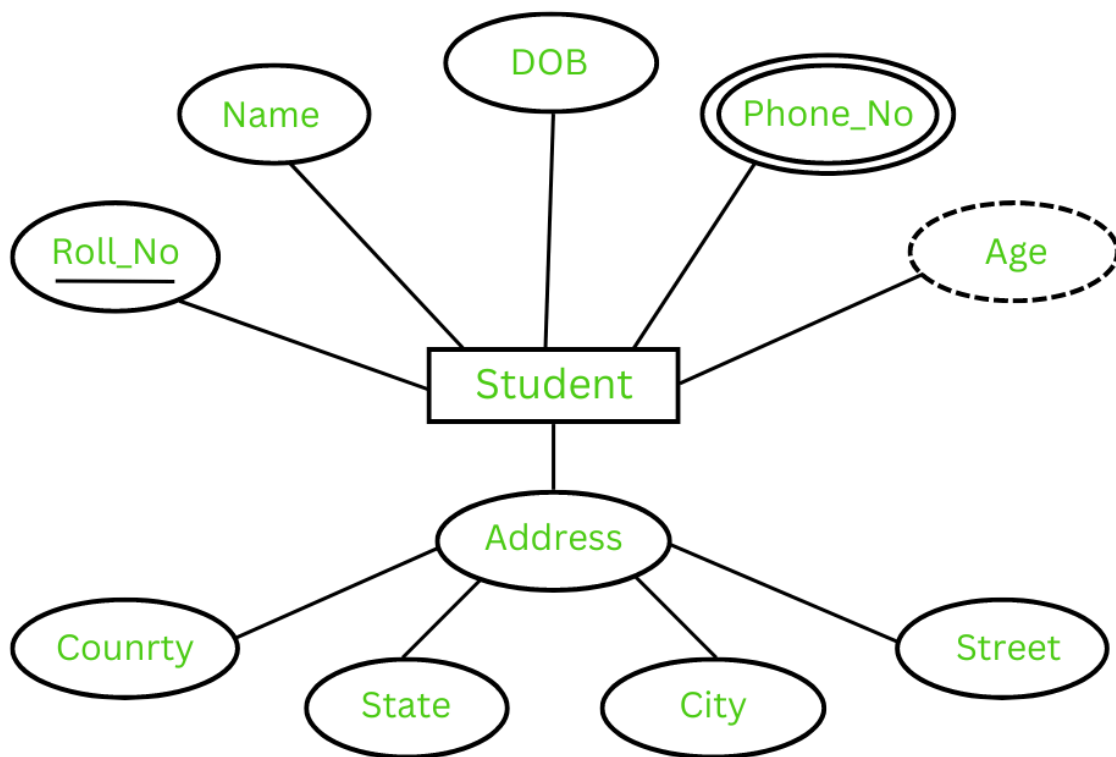
#### 4. Derived Attribute

An attribute that can be derived from other attributes of the entity type is known as a derived attribute. e.g.; Age (can be derived from DOB). In the ER diagram, the derived attribute is represented by a dashed oval.



*Derived Attribute*

The Complete Entity Type Student with its Attributes can be represented as:



*Entity and Attributes*

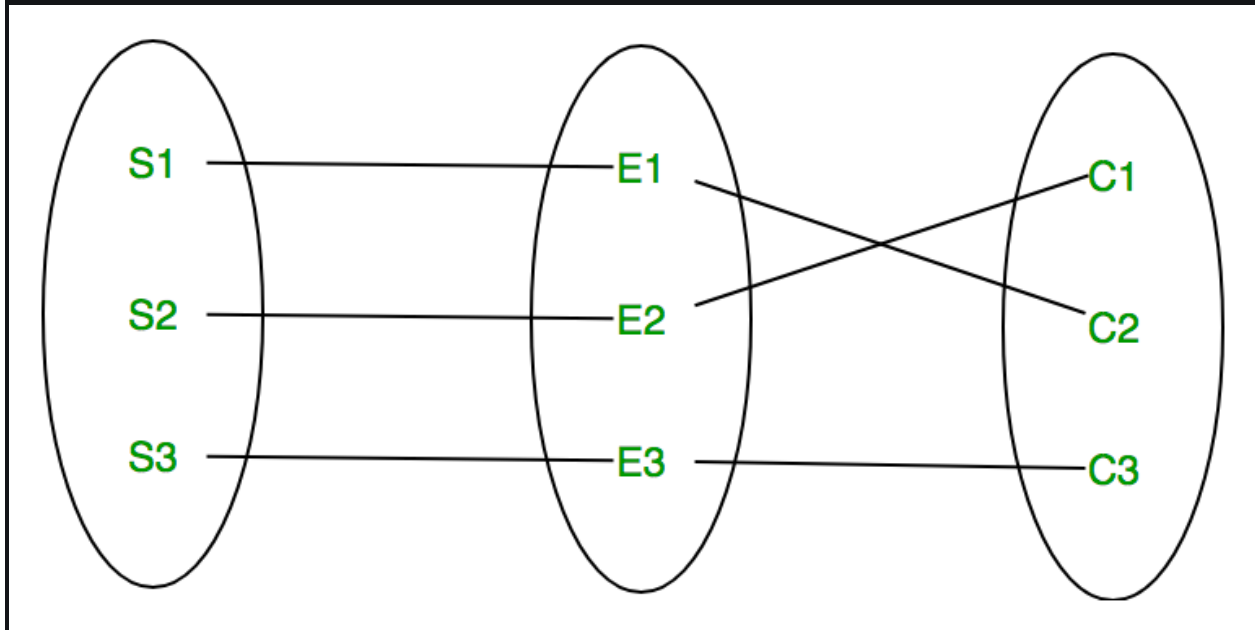
## Relationship Type and Relationship Set

A Relationship Type represents the association between entity types. For example, 'Enrolled in' is a relationship type that exists between entity type Student and Course. In the ER diagram, the relationship type is represented by a diamond and connects the entities with lines.



*Entity-Relationship Set*

A set of relationships of the same type is known as a relationship set. The following relationship set depicts S1 as enrolled in C2, S2 as enrolled in C1, and S3 as registered in C3.

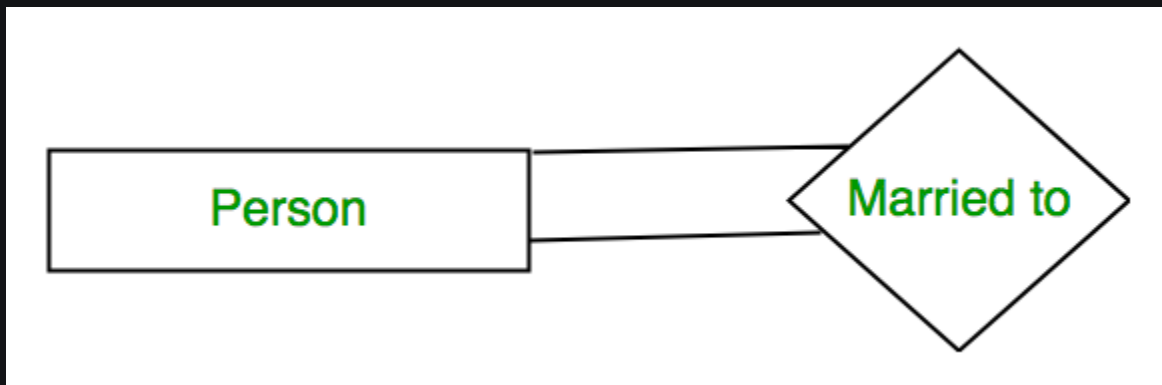


*Relationship Set*

## Degree of a Relationship Set

The number of different entity sets participating in a relationship set is called the [degree of a relationship set](#).

**1. Unary Relationship:** When there is only ONE entity set participating in a relation, the relationship is called a unary relationship. For example, one person is married to only one person.



#### Unary Relationship

**2. Binary Relationship:** When there are TWO entities participating in a relationship, the relationship is called a binary relationship. For example, a Student is enrolled in a Course.



#### Binary Relationship

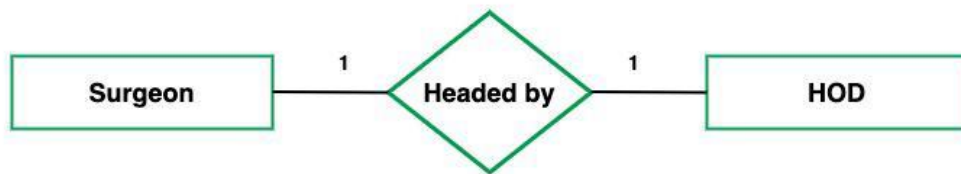
**3. Ternary Relationship:** When there are three entity sets participating in a relationship, the relationship is called a ternary relationship.

**4. N-ary Relationship:** When there are n entities set participating in a relationship, the relationship is called an n-ary relationship.

## What is Cardinality?

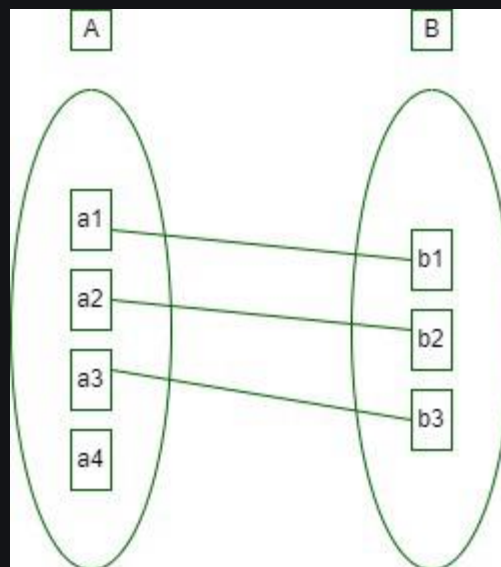
The number of times an entity of an entity set participates in a relationship set is known as [cardinality](#). Cardinality can be of different types:

**1. One-to-One:** When each entity in each entity set can take part only once in the relationship, the cardinality is one-to-one. Let us assume that a male can marry one female and a female can marry one male. So the relationship will be one-to-one. The total number of tables that can be used in this is 2.



*one to one cardinality*

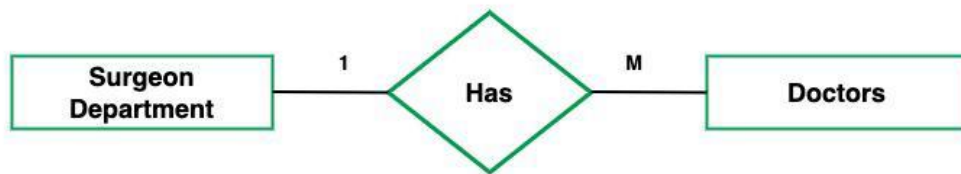
Using Sets, it can be represented as:



*Set Representation of One-to-One*

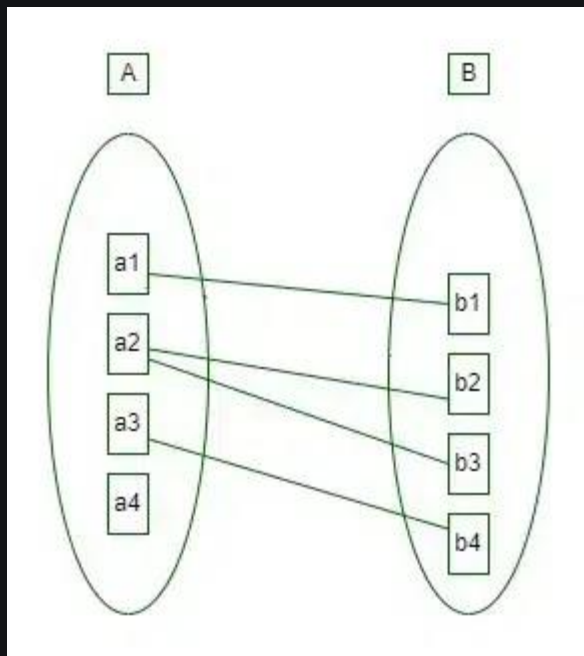
**2. One-to-Many:** In one-to-many mapping as well where each entity can be related to more than one entity and the total number of tables that can be used in this is 2. Let us assume that one surgeon department can accommodate many doctors. So the Cardinality will be 1 to M. It means one department has many Doctors.

the total number of tables that can be used is 3.



*one to many cardinality*

Using sets, one-to-many cardinality can be represented as:



*Set Representation of One-to-Many*

**3. Many-to-One:** When entities in one entity set can take part only once in the relationship set and entities in other entity sets can take part more than once in the relationship set, cardinality is many to one. Let us assume that a student can take only one course but one course can be taken by many students. So

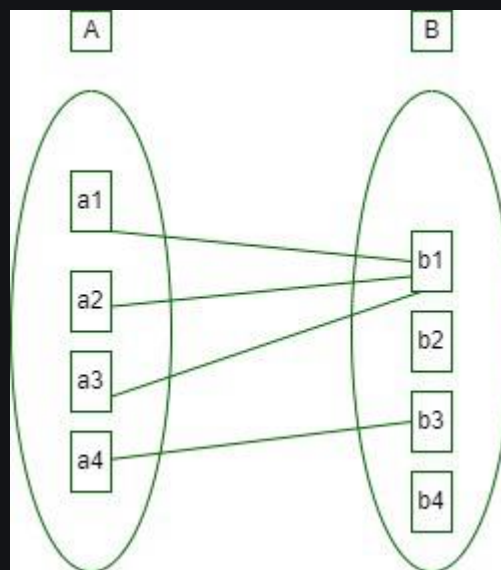
the cardinality will be n to 1. It means that for one course there can be n students but for one student, there will be only one course.

The total number of tables that can be used in this is 3.



*many to one cardinality*

Using Sets, it can be represented as:



*Set Representation of Many-to-One*

In this case, each student is taking only 1 course but 1 course has been taken by many students.

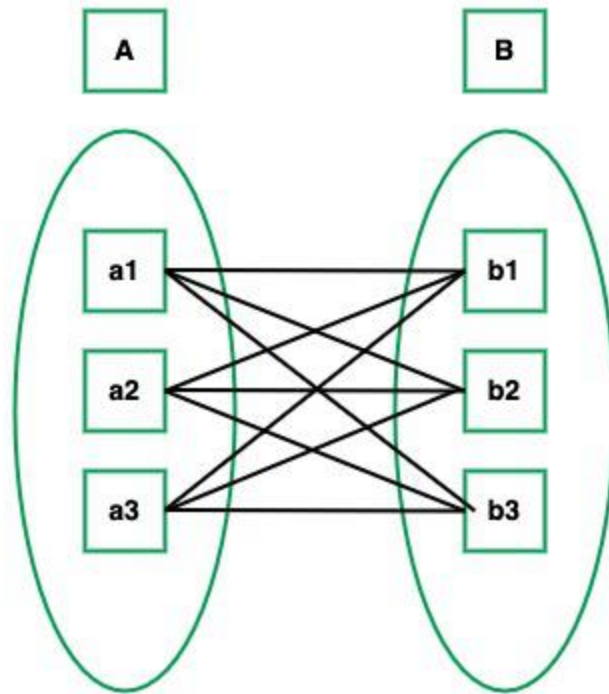
**4. Many-to-Many:** When entities in all entity sets can take part more than once in the relationship cardinality is many to many. Let us assume that a student can take more than one course and one course can be taken by many students. So the relationship will be many to many.

The total number of tables that can be used in this is 3.



*many to many cardinality*

Using Sets, it can be represented as:



*Many-to-Many Set Representation*

In this example, student S1 is enrolled in C1 and C3 and Course C3 is enrolled by S1, S3, and S4. So it is many-to-many relationships.

## Participation Constraint

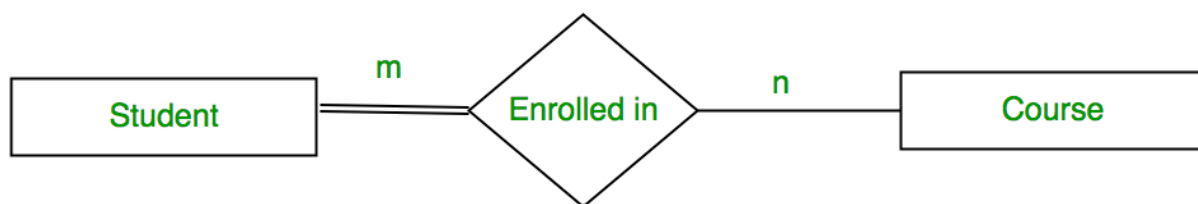
[Participation Constraint](#) is applied to the entity participating in the relationship set.

**1. Total Participation** – Each entity in the entity set must participate in the relationship. If each student must enroll in a course, the participation of

students will be total. Total participation is shown by a double line in the ER diagram.

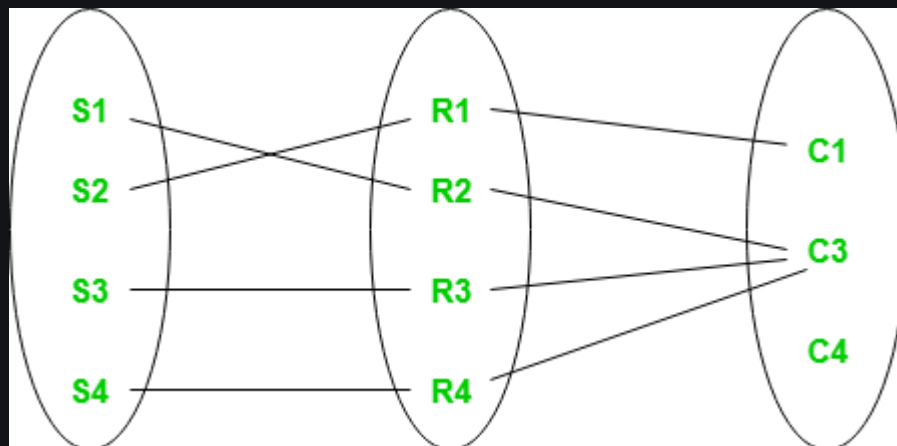
**2. Partial Participation** – The entity in the entity set may or may NOT participate in the relationship. If some courses are not enrolled by any of the students, the participation in the course will be partial.

The diagram depicts the 'Enrolled in' relationship set with Student Entity set having total participation and Course Entity set having partial participation.



*Total Participation and Partial Participation*

Using Set, it can be represented as,



*Set representation of Total Participation and Partial Participation*

Every student in the Student Entity set participates in a relationship but there exists a course C4 that is not taking part in the relationship.

# UNIT 2

---

## 1. Problem Partitioning

### Definition:

**Problem partitioning** refers to breaking down a large and complex software problem into smaller, manageable, and independent sub-problems or modules.

### Importance:

- Simplifies software design and implementation.
- Enables **parallel development**.
- Facilitates **testing and debugging**.
- Improves **maintainability** and **scalability**.

### Techniques:

- **Functional Decomposition**: Break the system into functions (e.g., `login()`, `checkout()`).
- **Layered Architecture**: Divide into UI, business logic, and data access layers.

### Example: Online Shopping System

Break it into modules:

- **User Module** – registration, login
- **Product Module** – browse, search, filter
- **Cart Module** – add/remove products

- **Payment Module** – process payments
- **Order Module** – track orders

Each of these modules can be developed and tested independently.

### **Analogy:**

Like building a car – different teams work on the engine, body, interior, etc., and assemble them at the end.

---

## Layered Approach in Software Design

### **Definition:**

The **Layered Approach** is a design method where a system is **structured into distinct layers**, each with a specific role, and **each layer interacts only with its immediate neighboring layers**.

### **Common Layers:**

1. **Presentation Layer (UI)**
    - Handles user interface and interaction.
  2. **Business Logic Layer**
    - Contains rules and logic of the application.
  3. **Data Access Layer**
    - Connects and manages communication with databases.
  4. **Database Layer**
    - Stores actual data.
-

## Why Use It?

- Improves **modularity**.
  - Enhances **maintainability** (you can change one layer without affecting others).
  - Supports **scalability** and **testing**.
- 

## Real-World Example: Library Management System

| Layer              | Description                                                                                     |
|--------------------|-------------------------------------------------------------------------------------------------|
| Presentation Layer | User sees options: login, search book, borrow book                                              |
| Business Logic     | Applies rules: Is the user active? Is the book available?                                       |
| Data Access        | Sends queries like <code>SELECT * FROM books WHERE available = true</code>                      |
| Database Layer     | Actual database tables like <code>users</code> , <code>books</code> , <code>transactions</code> |

---

## Analogy:

Think of it like making a sandwich :

- **Top Bread (UI)** – What the user sees.
- **Filling (Logic)** – The actual taste.

- **Bottom Bread (Database)** – Holds everything together.
- 

## 2. Abstraction

### Definition:

**Abstraction** is a design principle that focuses on the **essential features** of a system while hiding the **complex implementation details**.

### Types:

1. **Data Abstraction** – Hiding internal data structure and showing only essential attributes.
2. **Control Abstraction** – Hiding the flow of control using functions or procedures.

### Importance:

- Reduces complexity.
- Enhances reusability.
- Promotes modularity.

### Examples:

#### ***Data Abstraction:***

```
class BankAccount:
```

```
    def __init__(self, balance):
```

```
        self.__balance = balance
```

```
    def deposit(self, amount):
```

```
        self.__balance += amount
```

```
def get_balance(self):  
    return self.__balance
```

We don't need to know *how* deposit works internally—just that it exists and works.

***Control Abstraction:***

```
def calculate_area(radius):  
    return 3.14 * radius ** 2
```

You don't see the math every time; just call the function.

---

### 3. Top-down and Bottom-up Design

**Definition:**

These are two **software design strategies** used to plan the structure of a software system.

---

#### **Top-Down Design**

***Concept:***

- Start from the **highest-level functionality** (system overview).
- Break it into **smaller sub-functions** or modules.
- Keep breaking down until you reach **implementable units**.

***Advantages:***

- Clear system-level perspective first.
- Easy to trace how each module fits into the overall system.

**Example:**

Designing an **Online Shopping App**:

1. System: **Online Shopping System**
  - Module: **User Interface**
  - Module: **Product Management**
  - Module: **Payment Processing**
    - Submodule: **Card Payment**
    - Submodule: **UPI Payment**

You focus on the big picture first, then drill down.

---

## Bottom-Up Design

**Concept:**

- Start by designing **low-level modules** (e.g., utility functions, libraries).
- Then **integrate** them into larger systems.

**Advantages:**

- Strong **reusable components**.
- Detailed understanding of core modules early on.

**Example:**

Designing a **Banking System** from the bottom:

1. Build low-level modules:
    - `DatabaseConnector`
    - `Encryptor`
    - `TransactionLogger`
  2. Combine them into:
    - `Account Module`
    - `Loan Module`
  3. Then integrate everything into the `Banking Portal`
- 

## Real-World Analogy

- **Top-down:** Designing a city by drawing a map and planning districts, then roads, then buildings.
  - **Bottom-up:** Starting by building homes and roads, then connecting them into neighborhoods, then cities.
- 

## Meet-in-the-Middle Design

In practice, modern systems use a **hybrid approach**:

- High-level architecture is planned (top-down).
  - Reusable components are built and integrated (bottom-up).
-

## Comparison Table:

| Feature     | Top-Down Design             | Bottom-Up Design               |
|-------------|-----------------------------|--------------------------------|
| Approach    | System to Subsystems        | Components to System           |
| Focus       | Functionality/requirements  | Reusability of components      |
| Control     | High-level design decisions | Low-level utility development  |
| Suitability | Complex, large systems      | Modular and reusable libraries |

---

## Detailed Breakdown (Top-Down Design)

### Main System: Online Shopping System

---

#### 1. User Interface Module

This is the layer the user directly interacts with. We now divide it into smaller components:

##### ► *Submodules of User Interface:*

##### 1. Home Page UI

- Display product categories
- Promotional banners

## 2. **Login/Register Page**

- User login form
- Registration form
- Password reset

## 3. **Product Listing Page**

- Search bar
- Filters (price, category, brand)
- Pagination controls

## 4. **Product Detail Page**

- Product image & description
- "Add to Cart" and "Buy Now" buttons

## 5. **Cart UI**

- List of selected items
- Quantity adjustments
- Remove item button

## 6. **Checkout UI**

- Address entry
- Payment method selection

## 7. **Order Confirmation Page**

- Display order summary and tracking link
-

## 2. Product Management Module

This module handles all product-related backend functionalities.

### ► *Submodules of Product Management:*

#### 1. Product Catalog Service

- Add/Edit/Delete products
- Manage product attributes (title, price, stock)

#### 2. Category Management

- Create/manage categories and subcategories
- Assign products to categories

#### 3. Inventory Management

- Track stock availability
- Alert on low stock

#### 4. Search & Filter Engine

- Enable keyword search
- Filter products by attributes

#### 5. Product Recommendation

- Suggest similar items
- Frequently bought together

---

**Why this breakdown matters in Top-Down Design:**

- You start with big blocks like **User Interface** and **Product Management**.
  - Each block is further divided until you reach **modules simple enough to code directly**.
  - This makes **design clear, structured, and traceable** from requirement to implementation.
- 

## 3. Payment Processing Module

The **Payment Processing Module** is responsible for securely collecting, validating, and processing customer payments. It connects with external payment gateways and handles various payment types.

---

### Submodules of Payment Processing:

---

#### 1. Payment Gateway Integration

Responsible for handling the actual communication with external services like Razorpay, Stripe, PayPal, etc.

##### Tasks:

- Connect to APIs of payment gateways
  - Send payment request
  - Receive success/failure response
  - Handle API errors and retry logic
- 

#### 2. Payment Method Manager

Handles different payment types a user can choose from.

**Subtypes:**

- **Card Payments**
    - Validate card number (Luhn algorithm)
    - Handle OTP/3D secure
  - **UPI Payments**
    - Redirect to UPI apps
    - Confirm real-time status
  - **Net Banking**
    - Redirect to bank login portals
    - Track transaction status
  - **Wallets (e.g., Amazon Pay)**
    - Verify wallet balance
    - Deduct amount
- 

### **3. Transaction Validator**

Ensures every transaction is:

- Unique
- Secure (uses encryption)
- Linked to a valid user and order
- Verified against fraud (e.g., exceeding daily limits)

---

## 4. Order-Payment Linker

Links payment confirmation to a specific order and triggers fulfillment.

### Functions:

- Update order status (from "Pending" to "Paid")
- Notify other systems (e.g., Inventory, Shipping)
- Send payment confirmation email/SMS

---

## 5. Refund and Cancellation Processor

Handles payment reversal scenarios.

### Tasks:

- Initiate refund via gateway
- Track refund status
- Update customer wallet or bank
- Log refund for audit

---

## 6. Payment Logs and Audit Trail

Maintains detailed logs of all payment attempts.

### Purpose:

- Troubleshooting failed payments

- Compliance and auditing
  - Reporting and analytics
- 

### Real-World Example:

Suppose a user selects "UPI" and initiates payment:

- **Payment Method Manager** activates UPI handler
  - UPI request sent to gateway via **Payment Gateway Integration**
  - Response received → success
  - **Order-Payment Linker** updates order as "Paid"
  - Email is sent, and order moves to shipping
  - All details logged by **Audit Trail**
- 

This layered approach ensures: **Security, Scalability, Maintainability**

---

## 4. Structured Approach

### Definition:

The **Structured Approach** is a systematic method of software design and programming that emphasizes a clear, logical structure using well-defined **control constructs** like:

- **Sequence** – step-by-step execution
- **Selection** – decisions using if-else
- **Iteration** – loops (for, while)

It encourages breaking down a large problem into **modules**, which are easier to code, test, and maintain.

---

### Key Concepts:

| Concept                | Description                                               |
|------------------------|-----------------------------------------------------------|
| Top-down Design        | Start from the overall problem and break into sub-modules |
| Modularity             | Each task/function in a separate module                   |
| Structured Programming | Use of clean constructs: if, while, functions             |

---

### Tools & Techniques:

1. **Flowcharts**  
Graphical representation of logic.
2. **Pseudocode**  
Human-readable version of algorithm without syntax.
3. **Structure Charts**  
Shows relationships between modules.

---

## Example: ATM Withdrawal (Using Pseudocode)

START

INPUT PIN

IF PIN is correct THEN

INPUT Amount

IF Amount  $\leq$  Balance THEN

Deduct Amount

Dispense Cash

Print Receipt

ELSE

Display "Insufficient Balance"

END IF

ELSE

Display "Invalid PIN"

END IF

END

Here we use:

- **Sequence:** input  $\rightarrow$  validate  $\rightarrow$  process
  - **Selection:** check if PIN is valid
  - **Iteration** (optional): retry on failure
-

### Benefits of Structured Approach:

- Clear and easy-to-follow logic
  - Reduces bugs due to modularity
  - Easier to test and debug
  - Code is reusable and maintainable
- 

### Real-World Analogy:

Think of **structured approach** like a **recipe**:

- Do steps in order (sequence)
  - Choose paths (vegetarian or non-veg – selection)
  - Stir for 5 mins (iteration)
- 

## 5. Functional vs Object-Oriented Approach

Both are fundamental **programming paradigms**, and understanding their differences helps in choosing the right approach for a project.

---

### Functional Approach

#### Concept:

- Focuses on **functions** that take input and return output.
- Functions are **pure** (no side effects).

- **Data and logic are separate.**

### Characteristics:

- Stateless (no data is retained between calls)
- No modification of variables (immutability)
- Emphasizes recursion over loops

### Example:

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

```
    return a + b
```

```
print(add(5, 3)) # Output: 8
```

Here, **add** is a pure function — it does not change any external state.

---

## Object-Oriented Approach (OOP)

### Concept:

- Focuses on **objects**, which combine **data** and **behavior**.
- Uses **classes** to create reusable code templates.
- Promotes **encapsulation**, **inheritance**, and **polymorphism**.

### Characteristics:

- Objects maintain their own state.

- Easier to model real-world entities.
- Supports reusability and scalability.

### Example:

```
class Calculator:
```

```
    def __init__(self):
```

```
        self.result = 0
```

```
    def add(self, x, y):
```

```
        return x + y
```

```
calc = Calculator()
```

```
print(calc.add(5, 3)) # Output: 8
```

Here, behavior (**add**) and data (**result**) are part of the object **calc**.

---

### Comparison Table:

| Feature   | Functional Programming | Object-Oriented Programming |
|-----------|------------------------|-----------------------------|
| Main unit | Function               | Object/Class                |
| State     | Stateless              | Stateful                    |

|                   |                                   |                                  |
|-------------------|-----------------------------------|----------------------------------|
| Reusability       | Through functions                 | Through inheritance/polymorphism |
| Data & Logic      | Separate                          | Combined                         |
| Popular Languages | Haskell, C, JavaScript (FP style) | Java, Python, C++                |

---

## When to Use Which?

| Use Case                        | Preferred Approach      |
|---------------------------------|-------------------------|
| Simple mathematical operations  | Functional              |
| Real-world modeling (Bank, ATM) | Object-Oriented         |
| Stateless microservices         | Functional (serverless) |
| Enterprise web apps             | Object-Oriented         |

---

## Real-World Analogy:

- **Functional:** Like a calculator — you input numbers, get results, no memory of the past.
  - **Object-Oriented:** Like a customer account — stores data, history, and offers actions (withdraw, deposit, etc.).
- 

## 6. UML (Unified Modeling Language)

### Definition:

**UML (Unified Modeling Language)** is a **standardized modeling language** used to **visualize, specify, construct, and document** the structure and behavior of a software system.

It's like a **blueprint** for developers, testers, and stakeholders to understand how a system is designed — **before any code is written**.

---

### Why Use UML?

- Clarifies complex systems.
  - Helps in team communication.
  - Makes documentation formal and standardized.
  - Helps in verifying system design early.
- 

### Categories of UML Diagrams:

UML has **14 standard diagrams**, grouped into:

***Structure Diagrams (Show what the system consists of)***

1. **Class Diagram**
2. **Object Diagram**

3. **Component Diagram**
4. **Deployment Diagram**
5. **Package Diagram**
6. **Composite Structure Diagram**

***Behavior Diagrams (Show how the system behaves)***

1. **Use Case Diagram**
2. **Activity Diagram**
3. **Sequence Diagram**
4. **State Diagram**
5. **Communication Diagram**
6. **Interaction Overview Diagram**
7. **Timing Diagram**

Let's focus on the 4 most commonly used diagrams for beginners:

---

## **1. Use Case Diagram**

- Shows interactions between **users (actors)** and the **system**.
- Helps identify main functionalities (use cases).

**Example – Library System:**

- **Actors:** Student, Librarian
- **Use Cases:** Search Book, Issue Book, Return Book

Implement the diagram

---

## 2. Class Diagram

- Shows **classes**, **attributes**, **methods**, and **relationships** between classes.
- Used in object-oriented design.

### Example – Online Store:

- Class: **Product**
  - Attributes: id, name, price
  - Methods: addToCart(), getDetails()

Visual Structure:

```
+-----+
|  Product  |
+-----+
| - id: int   |
| - name: string |
| - price: float |
+-----+
| +addToCart() |
| +getDetails() |
+-----+
```

Implement the other classes.

---

### 3. Sequence Diagram

- Shows how **objects interact over time** using **messages**.
- Useful for tracking system flow for a specific feature.

#### Example – Login Process:

User → LoginPage → AuthService → DB

---

### 4. Activity Diagram

- Like a **flowchart** — shows workflows, decisions, parallel paths.

#### Example – Online Payment Flow:

- Start → Choose Method → [Card or UPI] → Verify → Confirm → End

---

#### Tools for UML:

- StarUML, Lucidchart, Draw.io, Visual Paradigm, PlantUML

---

#### Real-World Analogy:

UML is like an **architect's building plan**. Before building, you visualize the rooms, structure, and flows.

---

## 7. Design Specifications and Verification

## Definition:

- **Design Specification** is a detailed **blueprint of what the software system should do and how it should behave**.
  - **Design Verification** ensures that the design:
    - Matches the **requirements**
    - Is **correct, complete, and consistent**
    - Is ready for implementation
- 

## Key Components of a Design Specification:

1. **Functional Specification**
    - Describes what the system should do (features, use cases)
  2. **Interface Specification**
    - Defines interaction between modules and with external systems
  3. **Data Specification**
    - Details of data formats, structures, input/output
  4. **Constraints and Assumptions**
    - Hardware/software constraints, system boundaries, assumptions made
  5. **Diagrams**
    - UML diagrams (class, sequence, use-case), flowcharts
-

## Example – Login Feature Specification

### Functional Spec:

- The system shall allow registered users to log in using email and password.

### Interface Spec:

- POST `/api/login` with payload `{email, password}`

### Data Spec:

- Email: String (max 50 chars)
- Password: Encrypted string (min 8 chars)

### Constraint:

- Block user after 3 failed attempts in 10 minutes
- 

## Design Verification

Verifying the design is like reviewing a plan before construction starts.

### Techniques:

| Method       | Description                        |
|--------------|------------------------------------|
| Walkthroughs | Informal peer review of the design |

**Inspections**              Formal review by a team, checking for completeness

**Prototyping**            Building a quick model to validate user needs

**Model Checking**       Mathematical method to verify properties of design

---

### **Example – Verifying ATM Design**

**Requirement:** Show an error if withdrawal amount exceeds balance.

**Verification Checklist:**

- Is the balance checked before transaction?
- Is the error message shown?
- Is the user prevented from continuing?

If yes to all → verified!

---

### **Benefits of Design Specification & Verification:**

- Prevents expensive bugs later in development
  - Ensures alignment with customer requirements
  - Facilitates easier coding and testing
  - Improves quality and reliability
- 

**Analogy:**

Think of **design specification** as the **blueprint of a house**, and **verification** as the architect reviewing it before building begins.

---

## 8. Cohesion and Coupling

These two principles **determine how well your software modules are organized**. Good software design tries to **maximize cohesion** and **minimize coupling**.

---

### Cohesion – "How related tasks are inside a module"

- Refers to **how closely the functions in a module are related to one another**.
- **High cohesion** = all tasks in the module are **strongly related** and serve **one clear purpose**.
- Promotes **clarity, reusability, and easier maintenance**.

### Example of High Cohesion:

```
class InvoiceProcessor:
```

```
    def calculate_total(self): ...
```

```
    def apply_discount(self): ...
```

```
    def generate_invoice_pdf(self): ...
```

All methods are related to **processing an invoice**.

---

### Example of Low Cohesion:

```
class Utility:
```

```
    def calculate_tax(self): ...
```

```
    def send_email(self): ...
```

```
def export_csv(self): ...
```

These methods do **unrelated tasks**, making the class confusing and hard to maintain.

---

## Coupling – "How much one module depends on another"

- Refers to the **degree of interdependence between software modules**.
  - **Tight coupling** means modules are heavily dependent on each other.
  - **Loose coupling** means modules are **independent**, interact via **well-defined interfaces**.
- 

## Types of Coupling (From best to worst):

| Type             | Description                                          |
|------------------|------------------------------------------------------|
| Data Coupling    | Modules share data via parameters only               |
| Stamp Coupling   | Share part of a data structure (not all data needed) |
| Control Coupling | One module controls flow of another (e.g., flags)    |

**Common Coupling**      Shared global variables

**Content Coupling**      One module directly modifies another's code (worst)

---

### Example of Low Coupling (Good):

```
class OrderService:

    def process_order(self, order):

        payment_service = PaymentService()

        payment_service.make_payment(order.total)
```

Here, **OrderService** depends on **PaymentService** only through an interface.

---

### Example of High Coupling (Bad):

```
class OrderService:

    def process_order(self, order):

        payment_service = PaymentService()

        payment_service.card_number = order.card_number

        payment_service.expiry_date = order.expiry_date

        payment_service.process_directly(order)
```

Here, **OrderService** is too tightly connected to the **inner workings** of **PaymentService**.

---

## Why It Matters:

| Cohesion                    | Coupling                          |
|-----------------------------|-----------------------------------|
| Should be <b>high</b>       | Should be <b>low</b>              |
| Makes code <b>focused</b>   | Makes code <b>modular</b>         |
| Easier to <b>understand</b> | Easier to <b>reuse &amp; test</b> |

---

## Analogy:

- **High Cohesion:** A coffee machine that only makes coffee
  - **Low Coupling:** That machine works without knowing how the power grid functions .
- 

## Types of Cohesion (from worst to best)

Cohesion refers to **how well the elements within a module work together**. The more focused and unified the module is, the higher the cohesion — which is what we want.

---

### 1. Coincidental Cohesion (Worst)

- Elements in the module are **completely unrelated**.
- Put together just for convenience or laziness.

## Example:

```
class Miscellaneous:
```

```
    def send_email(): ...
```

```
    def calculate_tax(): ...
```

```
    def export_file(): ...
```

Problems:

- No clear purpose
  - Hard to maintain or reuse
- 

## 2. Logical Cohesion

- Functions are **logically categorized**, but do different tasks.
- A control variable is used to decide which task to perform.

**Example:**

```
def process(task_type):
```

```
    if task_type == 'print':
```

```
        ...
```

```
    elif task_type == 'save':
```

```
        ...
```

Issues:

- Control flags make code harder to follow
-

### 3. Temporal Cohesion

- Elements are grouped because they are executed at the **same time** (e.g., on startup or shutdown).

#### Example:

```
def on_startup():
```

```
    load_config()
```

```
    open_log_file()
```

```
    show_splash_screen()
```

Problems:

- Grouped by timing, not purpose
- 

### 4. Procedural Cohesion

- Elements are grouped because they follow a specific sequence.

#### Example:

```
def process_invoice():
```

```
    validate_data()
```

```
    calculate_total()
```

```
    save_invoice()
```

Weakness:

- Sequence-based but may involve **unrelated data**
- 

## 5. Communicational Cohesion

- Functions in the module **operate on the same data**.

**Example:**

```
def generate_report(customer):
```

```
    get_data(customer)
```

```
    format_data(customer)
```

```
    print_report(customer)
```

Getting better! Shared data gives some cohesion.

---

## 6. Sequential Cohesion

- The output of one part is the **input to another**.

**Example:**

```
def process_order():
```

```
    calculate_price()
```

```
    apply_discount()
```

```
    generate_invoice()
```

Clear flow and logical connection between operations.

---

## 7. Functional Cohesion (Best)

- All parts of the module work together to **achieve a single, well-defined task**.

### Example:

```
def validate_user_credentials():  
    check_username_exists()  
    check_password_matches()
```

Clean, focused, and highly reusable.

---

## Summary Table

| Cohesion Type | Description                 | Quality |
|---------------|-----------------------------|---------|
| Coincidental  | Unrelated functions grouped | Worst   |
| Logical       | Grouped by logic type       | Weak    |

|                 |                              |           |
|-----------------|------------------------------|-----------|
| Temporal        | Grouped by time of execution | Weak      |
| Procedural      | Grouped by control flow      | Moderate  |
| Communicational | Operate on same data         | Good      |
| Sequential      | Output-input chaining        | Very Good |
| Functional      | Single, well-defined purpose | Best      |

---

## 9. Distributed Software Design

### Definition:

**Distributed Software Design** is about building software systems where **components are located on different networked computers**, but work together as a unified system.

It's the backbone of **modern web apps, cloud platforms, IoT systems, and microservices**.

---

### Key Characteristics of Distributed Systems:

| Characteristic          | Description                                        |
|-------------------------|----------------------------------------------------|
| <b>Resource Sharing</b> | Different parts share data, services, and hardware |

|                        |                                                         |
|------------------------|---------------------------------------------------------|
| <b>Concurrency</b>     | Multiple clients can use the system at the same time    |
| <b>Scalability</b>     | System can grow by adding more machines                 |
| <b>Fault Tolerance</b> | Continues to work even if one component fails           |
| <b>Transparency</b>    | Users see one unified system, not individual components |

---

## Design Considerations:

### 1. Communication Protocols

- REST, gRPC, WebSockets, Message Queues
- Must handle delays, retries, and network failures

### 2. Data Consistency

- Use of **distributed databases** (like MongoDB, Cassandra)
- Eventual vs Strong consistency

### 3. Service Discovery

- Dynamic lookup of services (e.g., via DNS, Consul)

### 4. Load Balancing

- Spreading traffic across servers

### 5. Security

- Secure APIs, encrypted channels, authentication

---

## Real-World Example: Ride-Sharing App (e.g., Uber)

| Module               | Runs Where                         |
|----------------------|------------------------------------|
| User App             | Mobile device                      |
| Driver App           | Mobile device                      |
| Backend Server       | Cloud server (Node.js, Java, etc.) |
| Database             | Hosted on cloud (MySQL, DynamoDB)  |
| Notification Service | Separate microservice (SMS, Email) |

Each component communicates via **APIs** and **events**.

---

## Common Architectures:

### 1. Client-Server

- A centralized server serves multiple clients.
- Classic websites or file servers.

### 2. Peer-to-Peer (P2P)

- No central server, all nodes are equal (e.g., BitTorrent).

### 3. Microservices

- System is divided into independent, self-contained services.

**4. Cloud-Native / Serverless**

- Uses cloud functions (like AWS Lambda) that scale automatically.

---

**Technologies & Tools:**

| Layer         | Tools / Technologies                |
|---------------|-------------------------------------|
| Frontend      | React, Vue.js                       |
| Backend APIs  | Express.js, Spring Boot, Django     |
| Communication | REST, gRPC, GraphQL                 |
| Databases     | MongoDB, PostgreSQL, Redis          |
| Deployment    | Docker, Kubernetes, AWS, Azure, GCP |
| Messaging     | Kafka, RabbitMQ, MQTT               |

---

**Benefits of Distributed Design:**

- High availability (no single point of failure)
- Better performance under load

- Easy to scale horizontally
  - More resilient and flexible
- 

### **Challenges:**

- Debugging is harder
  - Network latency and failures
  - Data consistency issues
  - Complex deployment and monitoring
- 

### **Real-World Analogy:**

Think of it like **a team of remote employees**:

- Each handles a task (service).
  - They communicate over Zoom/Slack (APIs).
  - If one is sick (down), others continue their work (resilience).
-

# UNIT 3

## Software Testing Fundamentals

---

### What is Software Testing?

**Software Testing** is the process of executing a program with the **intent of finding errors**. It helps validate whether the software system meets its **requirements** and works as expected for the customer or end-user.

---

### Objectives of Software Testing

1. **To Uncover Errors**  
Testing is meant to find defects like incorrect computations, unwanted interactions, or system crashes.
  2. **Verification**  
➤ *"Are we building the product right?"*  
Ensures the system conforms to specifications.
  3. **Validation**  
➤ *"Are we building the right product?"*  
Ensures the system meets customer expectations
- 

### Types of Testing

| Type                | Description                                       |
|---------------------|---------------------------------------------------|
| Unit Testing        | Testing individual modules or functions           |
| Integration Testing | Testing interaction between integrated components |

|                           |                                                   |
|---------------------------|---------------------------------------------------|
| <b>System Testing</b>     | Testing the complete system as a whole            |
| <b>Acceptance Testing</b> | Done by customer to verify requirements are met   |
| <b>Regression Testing</b> | Ensures new changes don't break existing features |

---

## Testing Principles

1. Testing is done to **find errors**.
  2. A good test case has a **high probability of finding errors**.
  3. A successful test is one that **uncovers an undiscovered error**
- 

## Software Testability

**Testability** is the ease with which software can be tested. Key characteristics include:

- **Operability:** Software works well
  - **Observability:** Outputs are visible and traceable
  - **Controllability:** Can control inputs and states
  - **Simplicity & Stability:** Less complexity means easier testing
  - **Understandability:** Clear architecture and documentation help testers
- 

## SDLC Testing – Testing Across Software Development Life Cycle

**Software testing is not just a phase**—it’s integrated across the entire **Software Development Life Cycle (SDLC)**. The **V-Model** or **Verification and Validation Model** shows how testing aligns with each SDLC phase.

---

## V-Model: Testing Mapped to SDLC

The **V-Model** is an extension of the **Waterfall model** that emphasizes **early test planning**. For every **development activity** on the left, there's a **testing activity** on the right.

**V-Model Structure:**

| Development Phase      | Corresponding Test Phase |
|------------------------|--------------------------|
| Requirements Gathering | Acceptance Testing       |
| System Design          | System Testing           |
| Architectural Design   | Integration Testing      |
| Component Design       | Unit Testing             |
| Code Implementation    | Executable Software      |

---

## Testing Phases in SDLC

### 1. Unit Testing

- Focus: Individual functions or methods
- Who: Developers
- Tools: JUnit, NUnit

## 2. Integration Testing

- Focus: Interfaces between components
- Who: Developers or dedicated testers
- Types: Top-down, bottom-up, sandwich

## 3. System Testing

- Focus: Entire integrated system
- Verifies both functional and non-functional requirements

## 4. Acceptance Testing

- Focus: Verify the system against user needs
- Done by: End users or client
- Types: Alpha & Beta Testing

## 5. Regression Testing?

**Regression Testing** is the process of **re-testing existing software functionality** after code changes to ensure that **new updates have not broken** previously working features.

---

## Why Is It Important?

- Every time a bug is fixed or a new feature is added, there's a risk of unintentionally introducing new bugs in existing areas.
  - Regression testing ensures **software stability** as it evolves.
- 

## Smoke Testing:

**Smoke Testing** is a **preliminary integration test** performed on software builds to ensure that the **critical functionalities are working** correctly. It's sometimes referred to as "**build verification testing**".

---

### Activities Involved in Smoke Testing

1. Code is integrated into a **build** (includes components, data files, libraries).
  2. A **set of test cases** is run to check **basic operations**.
  3. The build is then **integrated** into larger systems and re-tested **daily**.
- 

### Objective of Smoke Testing

- Quickly **validate stability** of the build.
  - **Expose "show-stopper" bugs** early in integration.
  - Decide whether the build is **ready for more rigorous testing**.
- 

### Benefits of Smoke Testing

- **Reduces integration risk**
  - **Improves quality** by catching early architectural/functional flaws
  - **Simplifies debugging** (errors are linked to recent changes)
  - Provides a **realistic measure of progress**
  - Encourages **daily builds** (heartbeat of the project)
- 

## Security Testing:

**Security Testing** is the process of identifying and evaluating software vulnerabilities to ensure the **confidentiality, integrity, and availability** of system assets are not compromised.

It is a **core part** of software quality assurance aimed at identifying potential **security loopholes** before attackers do.

---

### Objectives of Security Testing

1. **Prevent unauthorized access**
  2. **Ensure data protection**
  3. **Resist malicious attacks**
  4. **Ensure system integrity and availability**
  5. Evaluate resilience to **internal and external threats**
- 

### Why is Security Testing Challenging?

- Security requirements are often **“negative”**: they describe what **should not happen**.
  - Attackers are **intelligent and creative** – they don’t follow usage norms.
  - It's **impossible to test** all possible attacks, especially unknown ones.
- 

## Key Techniques in Security Testing

| Technique                | Description                                                                         |
|--------------------------|-------------------------------------------------------------------------------------|
| Experience-Based Testing | Use known vulnerabilities (e.g., SQL injection, buffer overflow) to test the system |
| Penetration Testing      | Simulate real-world attacks to test how the system defends itself                   |
| Tool-Based Analysis      | Use static code analyzers and vulnerability scanners to detect flaws                |
| Formal Verification      | Use mathematical proof to ensure compliance with security specs                     |

---

## What Should Be Tested?

Security tests should target vulnerabilities across:

- **Client-side** (e.g., browser flaws, input forms)
  - **Network** (e.g., interception, DoS)
  - **Server-side** (e.g., authentication, authorization, database access)
- 

## Best Practices for Secure Software

1. Limit data visibility
  2. Validate all inputs
  3. Handle exceptions properly
  4. Avoid error-prone constructs
  5. Use encryption and timeouts
  6. Check array bounds
- 

### **Security Testing Tools (Examples)**

- **Netsparker** – WebApp scanner for SQL Injection
  - **Fortify** – Source code vulnerability scanner
  - **RATS** – Rough Auditing Tool for Security
- 

## **Performance Testing:**

### **Definition**

Performance testing is a type of software testing that focuses on assessing how well a system or component performs under specific workloads. It aims to ensure the software behaves acceptably under expected or stress conditions, focusing on speed, responsiveness, and stability.

---

## Key Objectives of Performance Testing

1. **Check Response Time** – How quickly does the system respond under load?
  2. **Measure System Capacity** – How many users or transactions can it handle?
  3. **Identify Bottlenecks** – Which parts slow down performance?
  4. **Validate Reliability** – Is it stable under increasing user loads?
  5. **Evaluate Resource Usage** – CPU, memory, disk, and network consumption.
- 

## Types of Performance Testing

| Type                | Purpose                                                                                                        |
|---------------------|----------------------------------------------------------------------------------------------------------------|
| Load Testing        | Tests system performance under expected usage load. Helps determine behavior under normal and peak conditions. |
| Stress Testing      | Pushes system beyond its limits to see how it breaks. Helps identify breaking points and failure handling.     |
| Spike Testing       | Sudden large increases in load to see how the system recovers.                                                 |
| Endurance Testing   | Checks how the system performs over an extended period of time (like overnight).                               |
| Scalability Testing | Measures the system's ability to grow (e.g., handle more users).                                               |
| Volume Testing      | Tests performance with large amounts of data.                                                                  |

---

## Real-World Testing Flow

1. **Simulate Users** – Virtual users mimic real ones.

2. **Gradually Increase Load** – Monitor how the system handles increasing demand.
3. **Record Metrics** – Response times, errors, throughput, CPU usage, etc.
4. **Analyze Results** – Identify when and where performance degrades.

## Example Scenario

Imagine a sports news website:

- 20,000 users
- Each sends a request every 2 minutes
- Each request downloads a 3 KB article

Throughput =  $20,000 \times 0.5 \times 3 \text{ KB} = 500 \text{ KB/sec}$

This tells you the required server capacity and helps in testing the response under real loads.

---

## Stress Testing:

**Stress Testing** is a type of software testing that evaluates how a system behaves **under extreme or beyond-normal conditions**.

Its goal is to determine the **breaking point** of a system by **forcing it to operate beyond its designed capacity** — in terms of users, data, or transactions.

---

### Objectives of Stress Testing

1. Assess **system robustness** beyond expected limits.
2. Discover how the system **fails** — does it crash, freeze, or degrade gracefully?
3. Identify **data loss or corruption** under high loads.

4. Ensure the system recovers well after stress conditions return to normal.

---

## Parameters Often Pushed in Stress Testing

| Parameter | Description                       |
|-----------|-----------------------------------|
| N         | Number of concurrent users        |
| T         | Number of transactions per second |
| D         | Data volume per transaction       |

Stress testing goes **beyond operational limits** for N, T, and D to evaluate system behavior.

---

## What Do We Test During Stress Testing?

- Does the system degrade **gracefully** or crash completely?
- Are **users notified** if the system is unavailable?
- Are **queued requests** managed properly once the system recovers?
- Is **data integrity maintained** during and after overload?
- How long does it take to **recover from failure**?

## Variants: Spike / Bounce Testing

- **Spike Testing:** Sudden burst of high load → observe response
- **Bounce Testing:** Rapidly shift between high and low load to test adaptability and resource recovery

---

## Stress Testing Example

Let's say a web application is designed for **500 users**. During stress testing:

- We simulate **2,000 users** performing data-intensive operations.
- Observe how **response times** change and whether the system **fails safely**.

## Functional Testing / Black-Box Testing (Same Concept)

---

### Definition

**Black-box testing**, also referred to as **functional testing** or **behavioral testing**, evaluates the software based on its **functional requirements** without any knowledge of the internal code or structure. The system is treated as a "black box" where testers only care about the **input-output behavior**.

---

### Primary Objective

To verify:

- That **all specified functions** perform as intended.
  - The system reacts correctly to a wide variety of **inputs**, including **valid**, **invalid**, and **boundary** cases.
- 

### What It Focuses On

Black-box testing aims to identify:

1. Incorrect or missing functions

2. Interface errors
  3. Data structure or database access errors
  4. Behavior or performance issues
  5. Initialization and termination problems
- 

## Key Testing Questions

- Does the system validate **correct functionality**?
  - How does it respond to **unexpected inputs**?
  - Are the **performance limits** acceptable?
  - Does the system behave correctly across **data partitions**?
  - How do combinations of data inputs affect the system?
- 

## Techniques Used

1. **Equivalence Partitioning**: Divides input data into valid and invalid partitions to test representative values.
  2. **Boundary Value Analysis**: Focuses on values at the **edge** of input domains.
  3. **Orthogonal Array Testing**: Efficient testing for multiple parameter combinations.
  4. **Model-Based Testing**: Uses behavioral models like **state diagrams** to create test cases.
-

## What is Boundary Value Analysis?

**Boundary Value Analysis (BVA)** is a software testing technique where you test the **edges or limits** of input values rather than the values in the middle. Why? Because bugs are most likely to happen at these **boundaries**.

---

### Think of it Like a Doorway:

Imagine there's a doorway, and the rule is: only people between **18 and 60 years old** can enter.

- If someone is **17** (just below the limit) → should **not** be allowed.
- If someone is **18** (on the edge) → should **be allowed**.
- If someone is **60** (on the edge) → should **be allowed**.
- If someone is **61** (just above the limit) → should **not** be allowed.

These edge points (17, 18, 60, 61) are what we test in **Boundary Value Analysis**.

---

### Why Use BVA?

- **Most errors happen at the edges.**
  - Saves time: You test fewer cases but still catch a lot of bugs.
  - It's easy to apply and very effective.
- 

### Real-World Example:

Let's say we have a function that accepts **marks between 0 and 100**.

```
def grade_student(score):
```

```
if 0 <= score <= 100:  
    return "Valid score"  
  
else:  
    return "Invalid score"
```

***With BVA, we test:***

- Minimum boundary: **-1, 0, 1**
- Maximum boundary: **99, 100, 101**

These help check:

- Is the system correctly handling the **lowest valid** value?
- Is it catching the **slightly invalid** ones near the limits?

---

## What is Alpha Testing?

**Alpha Testing** is the **first stage of testing** done **by the internal team** (developers or testers) **before the product is released to the outside world**.

Think of it like this:

You've baked a cake, and before giving it to your guests (users), you and your family (your team) taste it to make sure it's good. That's Alpha Testing.

---

## Where and How Does Alpha Testing Happen?

- It's done **in-house**, inside the development environment.
- It's performed by **developers** and **quality assurance (QA)** teams.

- It usually happens **after unit testing and integration testing**, but **before Beta Testing**.
- 

## What is Tested?

- **Functionality:** Does the app/software work as expected?
  - **Usability:** Is it easy to use?
  - **Bugs and crashes:** Does it break anywhere?
  - **Performance:** Is it fast and smooth?
- 

## Real-Life Example:

Let's say your team is building a mobile app for online shopping.

- Before letting real users try it (Beta Testing), your team installs the app on internal devices.
- They check if:
  - You can log in/register.
  - Items can be added to the cart.
  - Payment process works.
  - App doesn't crash when switching screens.

This is **Alpha Testing** in action.

---

## Key Characteristics of Alpha Testing:

| Feature      | Description                                    |
|--------------|------------------------------------------------|
| Who does it? | Developers and internal testers                |
| Where?       | Inside the organization (lab/test environment) |
| Purpose      | Catch bugs early before going to real users    |
| Access       | Limited – only to internal team                |
| Stage        | First phase of acceptance testing              |

---

## What Happens After Alpha Testing?

Once Alpha Testing is complete and major bugs are fixed, the product moves to **Beta Testing** – where **real users** outside the company try it and give feedback.

---

## What is Beta Testing?

**Beta Testing** is when the **software is tested by real users outside** your company, in a real-world environment, **before the final release**.

Think of it like this:

You've baked that cake, and after your family (Alpha Testers) said it's good, you now let your friends (Beta Testers) taste it. You want to know what **real people** think — is it tasty? Too sweet? Easy to cut?

---

## Why Do Beta Testing?

Because real users:

- Use your software **differently** than you expected.
- Spot bugs you didn't find internally.
- Give **valuable feedback** to improve the product.

---

## Who Does Beta Testing?

- **External users** (customers, early adopters, or volunteers).
- They aren't part of your development team.
- Usually, they use the product **freely in their natural environment** (their own devices, offices, etc.).

---

## Real-Life Example:

Let's say your team made a new social media app.

- You release it to 500 selected users in a **closed beta**.
- They install it, post content, invite friends, and give feedback.
- Some report bugs: “Notifications don’t work on Android.”
- Some share suggestions: “Add dark mode!”

You fix those issues before launching publicly.

---

### Types of Beta Testing:

| Type                  | What It Means                                    |
|-----------------------|--------------------------------------------------|
| <b>Closed Beta</b>    | Limited users, invite-only. Controlled feedback. |
| <b>Open Beta</b>      | Open to everyone. More users, more feedback.     |
| <b>Technical Beta</b> | Focused on tech-savvy users or testers.          |
| <b>Public Beta</b>    | Available to the general public for testing.     |

---

## Alpha vs. Beta Testing Comparison:

| Feature     | Alpha Testing                    | Beta Testing                   |
|-------------|----------------------------------|--------------------------------|
| Tested by   | Internal team                    | External real users            |
| Environment | Controlled (lab/dev environment) | Real-world (user environment)  |
| Purpose     | Find bugs early                  | Get real-world feedback & bugs |
| Stage       | Before Beta                      | Before final launch            |

---

## What is Acceptance Testing?

**Acceptance Testing** is the final level of testing to check if the software meets **business requirements** and is **ready to be delivered to the customer**.

You can think of it like this:

You've baked the cake, your family liked it (Alpha), your friends liked it (Beta), and now the **customer who ordered it** is going to try it. If they're happy — you get paid!

---

## Who Performs It?

- Usually performed by the **client, end-user, or product owner**.
  - Sometimes assisted by QA teams or business analysts.
-

## What Is It Trying to Verify?

Acceptance testing answers the question:

“Does this software do what the **user expects** it to do?”

It focuses on:

- Business rules and needs
  - User workflows
  - Legal or compliance requirements
  - "Done" criteria from user stories (in Agile)
- 

## Example Scenario:

Imagine you built an **e-commerce website** for a client.

During Acceptance Testing, they might check:

- Can users sign up and log in?
- Can they search and buy products?
- Does the payment gateway work correctly?
- Do they get order confirmation emails?

If everything works as expected → software is **accepted**

---

## Types of Acceptance Testing:

| Type                              | Who Performs It       | What It Focuses On                   |
|-----------------------------------|-----------------------|--------------------------------------|
| User Acceptance Testing (UAT)     | End-users / clients   | Does it meet user needs?             |
| Business Acceptance Testing (BAT) | Business team         | Does it meet business goals?         |
| Contract Acceptance Testing       | Client vs Vendor      | Does it meet what's in the contract? |
| Regulation Acceptance Testing     | Legal/compliance team | Does it follow laws or regulations?  |

---

### Real World Analogy:

Imagine building a custom house:

- You show the blueprint to the client — that's like requirement gathering.
- You build the house — that's development.
- You walk the client through it, checking if everything they asked for is there — that's **Acceptance Testing**.
- If they approve it, they move in — that's the software going **live**.

---

### Final Step Before Launch:

Acceptance Testing is the **last step before the final release**:

- If passed → product is accepted and deployed.
  - If failed → developers fix issues, and testing is repeated.
- 

## What is Structural (White Box) Testing?

**White Box Testing** is a testing method where the tester looks **inside the code** and tests the **internal logic and structure** of the program.

You can think of it like this:

Imagine testing a car. White Box Testing means you open the hood and check the **engine, wires, fuel system** — not just how it drives. You want to know *how* it works inside, not just *if* it works.

---

## Why "Structural"?

Because you're checking the **structure** of the code:

- Logic flow (if-else, loops)
  - Conditions
  - Function calls
  - Paths and branches
- 

## Why is it called “White Box”?

Imagine a **transparent box** (white box):

- You can **see inside** and understand the internal workings.
  - Unlike **Black Box Testing**, where you just give inputs and check outputs without knowing what happens inside.
- 

## Who Performs White Box Testing?

Usually done by:

- **Developers**
  - **Technical testers** Because it requires **knowledge of the code**.
- 

## What is Tested?

| Element             | Description                                   |
|---------------------|-----------------------------------------------|
| <b>Control flow</b> | Are all branches and conditions tested?       |
| <b>Loops</b>        | Do loops work correctly and terminate?        |
| <b>Statements</b>   | Are all lines of code executed at least once? |

**Conditions**      Do all logical conditions return true/false?

---

### Example:

Let's say we have this function:

```
def is_eligible(age):
```

```
    if age >= 18:
```

```
        return True
```

```
    else:
```

```
        return False
```

***A White Box Test would check:***

- What happens if **age = 17** (false path)?
- What happens if **age = 18** (true path)?
- Are both the **if** and **else** branches covered?

This ensures all **logic paths** are working correctly.

---

### Types of White Box Testing Techniques:

| Technique          | What It Means                         |
|--------------------|---------------------------------------|
| Statement Coverage | Test every line of code at least once |

**Branch Coverage**      Test every possible branch (true/false conditions)

**Path Coverage**      Test all possible paths through the code

**Condition Coverage**      Test all individual conditions in decision-making

---

### Key Characteristics:

| Feature         | White Box Testing                    |
|-----------------|--------------------------------------|
| Focus           | Internal logic and structure of code |
| Access to Code? | Yes                                  |
| Who tests?      | Developers or technical testers      |
| Stage           | Unit Testing, Integration Testing    |

---

### Limitations:

- Can't catch missing features (only checks what's there).
- Doesn't test how the user interacts with the system.
- Can be time-consuming for large codebases.

---

## White Box vs Black Box Testing:

| Feature           | White Box Testing | Black Box Testing |
|-------------------|-------------------|-------------------|
| Knowledge of Code | Required          | Not Required      |
| Focus             | How it works      | What it does      |
| Performed by      | Developers        | Testers, QA       |

### 1. Basis Path Testing

#### What it is:

It's a method that ensures all **independent paths** in a program are tested at least once.

#### Think of it like:

Making sure you've tried every possible route through a maze.

#### Why it matters:

It gives **maximum coverage** with a **minimal number of test cases**.

#### How it works:

- Build a control flow graph (nodes = statements, edges = flow).
- Calculate **Cyclomatic Complexity** (explained next).
- Create test cases for each independent path.

---

### 2. DD Paths (Decision-to-Decision Paths)

**What it is:**

DD paths are the **paths between decisions** (like `if`, `while`, etc.) in your control flow graph.

**Why it matters:**

They help **break down complex logic** into smaller testable parts.

**Think of it like:**

You're traveling from one city (decision point) to another, and you want to explore **all the roads between them**.

---

### 3. Cyclomatic Complexity

**What it is:**

A metric that tells you **how complex your code is** — basically, how many **independent paths** it has.

**Formula:**

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

(E = edges, N = nodes, P = connected components like functions or classes)

**Simplified version:**

Just count:

- Number of `if`, `while`, `for`, `case`... each adds complexity.

**Why it matters:**

- High complexity = hard to test and maintain.
  - Low complexity = clean, manageable code.
-

## 4. Data Flow Testing

### What it is:

It focuses on the **flow of data** through the code — how variables are **defined, used, and killed** (no longer needed).

### Think of it like:

Tracking how a package moves from sender to receiver — making sure it doesn't get lost or misused.

### Key elements:

- **def** (definition): when a variable is given a value.
- **use**: when it's used in a calculation or condition.
- **kill**: when it's overwritten or goes out of scope.

### Why it matters:

It catches bugs like:

- Using a variable before it's assigned
  - Unused variables
- 

## 5. Mutation Testing

### What it is:

You **intentionally introduce small errors** (mutants) in the code to see if your test cases can catch them.

**Example:** Change **a + b** to **a - b**, or **>** to **<**.

### Goal:

If your tests detect the mutant = good.

If not = your test coverage may be weak.

### Think of it like:

Testing the security of a lock by **trying to pick it** on purpose — if it fails, you know it's solid.

---

## Summary Table:

| Technique             | Focus Area              | Goal                                     |
|-----------------------|-------------------------|------------------------------------------|
| Basis Path Testing    | Logic paths             | Ensure all paths are tested              |
| DD Paths              | Between decision points | Fine-tune logic testing                  |
| Cyclomatic Complexity | Code complexity         | Measure how difficult code is to test    |
| Data Flow Testing     | Variable usage          | Catch issues with data usage             |
| Mutation Testing      | Test strength           | Check if tests detect intentional errors |

# UNIT 4

## Test Cycle:

Think of it like planning a vacation :

You don't just jump on a plane — you plan the trip, pack your bags, book tickets, travel, enjoy, and then come back and review how it went. The **Test Cycle** works the same way for testing a software product.

---

### What is a Test Cycle?

A **Test Cycle** is the **end-to-end process** of planning, preparing, executing, and reporting software tests — repeated as needed across development phases.

---

### Phases of a Test Cycle

| Phase                            | What Happens                                  | Real-World Analogy                     |
|----------------------------------|-----------------------------------------------|----------------------------------------|
| 1. <b>Test Planning</b>          | Define what, how, and when to test            | Planning your vacation                 |
| 2. <b>Test Case Design</b>       | Write detailed test cases/scenarios           | Making your packing list               |
| 3. <b>Test Environment Setup</b> | Prepare tools, hardware, and data             | Booking tickets and hotels             |
| 4. <b>Test Execution</b>         | Run the tests, manually or automatically      | Actually traveling and exploring       |
| 5. <b>Defect Reporting</b>       | Log bugs/issues found during testing          | Reporting problems to the travel agent |
| 6. <b>Test Reporting</b>         | Summarize what was tested, what passed/failed | Writing a travel blog or review        |

|                                       |                                                   |                                         |
|---------------------------------------|---------------------------------------------------|-----------------------------------------|
| 7. <b>Re-testing &amp; Regression</b> | Re-test fixed issues; ensure nothing else broke   | Visiting again to check improvements    |
| 8. <b>Test Closure</b>                | Finalize and document everything; lessons learned | Wrapping up the trip, storing souvenirs |

---

## Repeat for Each Release or Sprint

In Agile or DevOps, testing is **not a one-time thing** — test cycles are repeated for each:

- Sprint
  - Feature release
  - Bug fix
  - Major update
- 

## Why Is It Important?

- Ensures **structured and complete testing**.
  - Helps find issues **early and efficiently**.
  - Makes it easier to track **progress and quality**.
  - Supports continuous improvement.
- 

## Real Example:

Imagine you're working on a **food delivery app**.

1. Plan to test user login, order placing, and delivery tracking.
  2. Write test cases for each.
  3. Set up a test environment with fake user accounts and orders.
  4. Run your tests.
  5. Find a bug — delivery ETA is wrong.
  6. Report it, devs fix it.
  7. Re-test and confirm it's resolved.
  8. Close the cycle and move to the next sprint.
- 

## What is Test Estimation?

**Test Estimation** is the process of **calculating the time, resources, and effort** needed to complete testing tasks in a project.

Think of it like this:

You're organizing a wedding — you need to estimate how many guests are coming, how much food is needed, and how long it'll all take to prepare. In testing, it's the same — you estimate how many test cases, how long to execute them, and what resources you need.

---

## Why Is Test Estimation Important?

- Helps create realistic project **timelines**
- Ensures **resource planning**
- Avoids **under-testing** or **delays**
- Builds **trust** with stakeholders and clients

---

## Common Techniques for Test Estimation

| Technique                      | How It Works                                                                |
|--------------------------------|-----------------------------------------------------------------------------|
| Experience-Based               | Use your past experience to guess how long it will take                     |
| Work Breakdown Structure (WBS) | Break down testing tasks into small parts, estimate each, and add them up   |
| Three-Point Estimation         | Use Optimistic, Pessimistic, and Most Likely estimates to calculate average |
| Function Point/LOC-Based       | Estimate based on the size of the code or number of functionalities         |
| Delphi Method                  | Ask a group of experts and average out their estimates                      |

---

### Example: Three-Point Estimation

Let's say you're estimating the time to test the login feature:

- **Optimistic (O):** 2 hours
- **Most Likely (M):** 4 hours
- **Pessimistic (P):** 6 hours

Formula:

Estimated Time (E) = (O + M + P) / 3

= (2 + 4 + 6) / 3

= 12 / 3 = 4 hours

---

### What to Include in Estimation?

- Test planning
  - Test case writing
  - Test environment setup
  - Test execution
  - Defect reporting & verification
  - Regression testing
  - Reviews and meetings
  - Buffer time (for unexpected issues)
-

## Risks If Estimation Is Wrong

- Project delays
  - Incomplete testing
  - Team burnout
  - Budget overruns
  - Missed deadlines
- 

## Tips for Accurate Estimation:

- Always **add buffer time** (20–30% is common).
  - Consider **team experience and skill level**.
  - Use **historical data** from past projects.
  - Revisit and **adjust your estimates** during the project.
- 

## What is a Test Case?

A **Test Case** is a **set of actions, inputs, and expected results** designed to verify that a particular feature or function in your software works as intended.

---

## Think of It Like This:

A Test Case is like a **recipe** :

- You have ingredients (inputs)
- A method (steps)

- And you expect a tasty pizza at the end (expected result) If anything goes wrong, you know where to check.

---

## Structure of a Test Case

Here's a simple format:

| Field           | Description                                                      |
|-----------------|------------------------------------------------------------------|
| Test Case ID    | Unique ID (e.g., TC001)                                          |
| Title           | What feature you are testing (e.g., Login Functionality)         |
| Preconditions   | What needs to be set up first (e.g., User account exists)        |
| Steps           | What actions to perform (e.g., Enter username, click login)      |
| Test Data       | Specific values used (e.g., username = "john", password = "123") |
| Expected Result | What should happen (e.g., User is logged in)                     |
| Actual Result   | What actually happened (used during execution)                   |
| Status          | Pass/Fail                                                        |
| Remarks         | Notes, bugs, screenshots, etc.                                   |

---

## Simple Test Case Example

**Feature:** Login Function

| Field           | Example                                                            |
|-----------------|--------------------------------------------------------------------|
| Test Case ID    | TC001                                                              |
| Title           | Valid Login Test                                                   |
| Preconditions   | User is registered                                                 |
| Steps           | 1. Open login page2. Enter valid username & password3. Click login |
| Test Data       | Username: user123, Password: pass123                               |
| Expected Result | User is redirected to dashboard                                    |
| Actual Result   | (Filled after test execution)                                      |
| Status          | Pass / Fail                                                        |
| Remarks         | --                                                                 |

---

## Types of Test Cases

| Type           | Description                               |
|----------------|-------------------------------------------|
| Positive       | Testing valid inputs (expected to work)   |
| Negative       | Testing invalid inputs (expected to fail) |
| Boundary       | Testing edge cases (e.g., min/max values) |
| Functional     | Checking what the software does           |
| Non-Functional | Testing speed, security, usability, etc.  |
| Regression     | Rechecking old features after changes     |

---

## Why Are Test Cases Important?

- Provide **clarity** on what to test
  - Help catch **bugs** early
  - Make **testing repeatable** and consistent
  - Form the basis for **automation scripts**
  - Help new testers understand the system
- 

## Best Practices

- Keep it **simple and clear**
- Write from a **user's perspective**
- Include **both positive and negative** test cases
- Use **unique and traceable IDs**
- Keep test data **realistic**

## What is a Test Scenario?

A **Test Scenario** is a **high-level description** of what to test — it outlines **what functionality** or **business flow** needs to be verified.

You can think of it like this:

**Test Scenario = What to test**

**Test Case = How to test it**

---

## Real-World Analogy:

Imagine you're testing a **coffee machine**.

- **Test Scenario:** Verify the user can make a coffee.
- **Test Cases:**
  - Insert coffee pod and press "Start".
  - Choose "Espresso" and check if hot coffee is dispensed.
  - Try starting without water to see if an error is shown.

---

## Characteristics of Test Scenarios

| Feature              | Description                                              |
|----------------------|----------------------------------------------------------|
| High-level           | Doesn't go into step-by-step details                     |
| Covers functionality | Represents a feature, use case, or user journey          |
| Easy to understand   | Useful for communicating with non-technical stakeholders |

---

### Example: E-commerce Website

| Module         | Test Scenario                                    |
|----------------|--------------------------------------------------|
| Login          | Verify user can log in with valid credentials    |
| Login          | Verify user cannot log in with wrong password    |
| Search         | Verify search returns relevant products          |
| Cart           | Verify user can add and remove items             |
| Checkout       | Verify successful payment with valid card        |
| Order Tracking | Verify user can track their order after purchase |

---

## Relationship: Test Scenarios vs Test Cases

| Test Scenario                    | Related Test Cases                                    |
|----------------------------------|-------------------------------------------------------|
| Verify user can log in           | - Valid login- Invalid password- Blank fields         |
| Verify user can add item to cart | - Add one item- Add multiple items- Add from wishlist |

---

### When to Use Test Scenarios?

- Early in test planning to **identify coverage**
  - Useful in **Agile** and **Exploratory Testing**
  - Helpful when you have limited time — focus on **what matters most**
- 

### Best Practices

- Keep it user-focused and functional
  - Cover **end-to-end business flows**
  - Use **clear, simple language**
  - Derive **test cases** from each scenario
- 

## What Are Static and Dynamic Testing Tools?

These two types of tools help testers find different types of issues **at different stages** of the development cycle.

## Static Testing Tools (No code execution)

Static = “Still”

These tools **analyze the code without running it.**

Think of it like reading an essay for spelling mistakes without actually reading it out loud.

### ***What They Do:***

- Check code structure, syntax, and standards
- Catch bugs, vulnerabilities, and style issues early
- Improve code quality before runtime

### ***Examples of Static Testing Tools:***

| Tool              | Purpose                             |
|-------------------|-------------------------------------|
| SonarQube         | Code quality and security scanning  |
| ESLint            | JavaScript linting (style & errors) |
| PMD               | Java static code analysis           |
| FindBugs/SpotBugs | Java bug patterns detection         |

|                 |                       |
|-----------------|-----------------------|
| <b>Cppcheck</b> | C/C++ static analysis |
|-----------------|-----------------------|

|                           |                          |
|---------------------------|--------------------------|
| <b>Checkmarx, Fortify</b> | Security vulnerabilities |
|---------------------------|--------------------------|

---

## Dynamic Testing Tools (Code execution required)

Dynamic = “Moving”

These tools **test the software while it’s running**.

Think of it like actually running that essay through a text-to-speech tool to hear if it sounds right.

### ***What They Do:***

- Find runtime issues like crashes, incorrect outputs, or slow performance
- Validate functionality, UI behavior, security, and memory leaks

### ***Examples of Dynamic Testing Tools:***

| Tool                | Purpose                    |
|---------------------|----------------------------|
| <b>Selenium</b>     | Web UI automation          |
| <b>Postman</b>      | API testing                |
| <b>JUnit, NUnit</b> | Unit testing for Java/.NET |
| <b>Appium</b>       | Mobile app testing         |

|                   |                                        |
|-------------------|----------------------------------------|
| <b>JMeter</b>     | Performance and load testing           |
| <b>Burp Suite</b> | Dynamic security testing (pen testing) |

---

## Static vs Dynamic: Quick Comparison

| Feature        | Static Testing Tools           | Dynamic Testing Tools            |
|----------------|--------------------------------|----------------------------------|
| Code Execution | Not needed                     | Needed                           |
| When Used      | Early in development           | After software is built          |
| Detects        | Syntax errors, vulnerabilities | Runtime bugs, crashes, UI issues |
| Examples       | SonarQube, ESLint, PMD         | Selenium, Postman, JUnit         |

---

## Why Use Both?

- **Static tools** = Prevent bugs early → cheaper to fix
- **Dynamic tools** = Catch real-world problems → better user experience

Together, they create a **strong, full testing strategy**

---

# Characteristics of Modern Testing Tools & Automation

Here's a breakdown of what **modern tools** bring to the table:

---

## 1. Cross-Platform & Cross-Browser Support

Modern tools can test across:

- Different operating systems (Windows, macOS, Linux, mobile)
- Different browsers (Chrome, Firefox, Safari, Edge)

### Example:

Tools like **Selenium**, **Playwright**, and **Appium** support testing across devices and browsers.

---

## 2. CI/CD Integration

They seamlessly integrate with **Continuous Integration/Deployment pipelines**, enabling **automated testing after every code change**.

### Example:

Integrating **JUnit** or **Cypress** with **Jenkins**, **GitHub Actions**, or **GitLab CI**.

---

## 3. Intelligent & AI-Based Testing

Some tools use **AI/ML** to:

- Predict flaky tests
- Auto-generate tests
- Optimize test coverage
- Identify the most impacted areas from recent code changes

**Example:**

Tools like **Testim**, **Applitools**, and **Mabl** use AI for visual and smart testing.

---

## **4. Data-Driven & Keyword-Driven Testing**

Modern frameworks allow:

- **Data-driven testing:** Same test logic with multiple input datasets
- **Keyword-driven testing:** Non-coders can write test cases using predefined keywords

**Example:**

**TestNG**, **Katalon Studio**, and **Robot Framework**

---

## **5. Real-Time Reporting & Dashboards**

Modern tools provide:

- Visual dashboards
- Real-time pass/fail metrics
- Test history and trend analysis

**Example:**

**Allure Reports**, **ExtentReports**, and dashboards in **TestRail** or **QMetry**

---

## **6. Security and Compliance Ready**

They support:

- Secure testing environments
- Role-based access control

- Audit trails for compliance (e.g., GDPR, HIPAA)

**Example:**

**Fortify, Checkmarx**, and cloud-based tools with enterprise-grade security

---

## 7. Collaboration Friendly

Modern tools allow:

- Shared test repositories
- Comments, reviews, and feedback loops
- Integration with Jira, Slack, Trello, etc.

**Example:**

**Zephyr, Qase, Testmo** — team-friendly test management tools

---

## 8. Scalability and Cloud Support

Test automation can run:

- On local machines or
- On the cloud (for faster parallel execution)

**Example:**

**BrowserStack, Sauce Labs, LambdaTest** let you test on 100s of devices instantly.

---

## 9. Easy to Learn, Low-Code/No-Code Options

Modern tools are becoming more accessible with **no-code or low-code interfaces**, ideal for non-technical testers.

**Example:**  
**Katalon Studio, Leapwork, TestProject**

---

**10. Maintenance and Reusability**

- Support for reusable components
- Easy to maintain test scripts when code changes
- Modular test design

**Example:**  
**Page Object Model (POM) frameworks in Selenium/Cypress**

---

**Summary Table:**

| Feature                      | Why It Matters                    |
|------------------------------|-----------------------------------|
| Cross-Platform Support       | Test everywhere your users are    |
| CI/CD Integration            | Enables fast, continuous delivery |
| AI-Based Capabilities        | Smarter, faster testing           |
| Real-Time Reporting          | Instant feedback on test results  |
| Collaboration & DevOps Ready | Fits into modern agile workflows  |

Scalable & Cloud-Friendly

Test faster across more environments

Easy to Use (Low/No Code)

Empowers non-technical team members

Reusable & Maintainable

Reduces effort as the project grows

---

## UNIT 5

### Unit 5: Software Maintenance – Study Notes

#### 1. Updates, Upgrades, Patches, and Versions

- **Update:** A small change to fix bugs or improve performance.
  - *Example:* A security update to fix a vulnerability in Windows 10.
- **Upgrade:** A major improvement with new features.
  - *Example:* Moving from Windows 10 to Windows 11.
- **Patch:** A piece of code designed to fix a specific problem or bug.
  - *Example:* A patch that fixes a memory leak in a game.
- **Version:** A way to label different stages of software.
  - *Example:* Version 2.3.1 — major.minor.patch versioning.

**Best Practice:** Always document changes in a *changelog* for users and developers.

---

## 2. Error Reporting

- Reporting bugs or problems by users or automated systems.
- **Manual Reporting:** Users send screenshots or logs.
- **Automated Reporting:** The system sends crash data to developers.

*Example:* When your browser crashes, it shows "Send crash report to developers."

---

## 3. Customer Support

- Helps users with installation, usage, and bug issues.
- Channels: Email, live chat, forums, FAQs, help desk software.

*Goal:* Provide quick, helpful, and friendly responses.

---

## 4. Maintenance Process

The maintenance process includes:

1. **Problem identification** – Bug found by a user.
2. **Analysis** – Developers analyze logs and code.
3. **Design** – Plan how to fix or improve.

4. **Implementation** – Apply the fix.
5. **Testing** – Ensure the fix works and doesn't break other parts.
6. **Delivery** – Release the patch or update.

*Continuous process* — especially for products in production.

---

## 5. Software Reliability

### *Importance*

- Ensures the software performs well under expected conditions over time.
- Impacts customer trust and product success.

### *Hardware vs Software Reliability*

- **Hardware:** Physical components may fail due to wear/tear.
- **Software:** Fails due to bugs, logic errors, or unexpected inputs. No physical degradation.

### *Failure and Faults*

- **Fault:** A defect in the code (bug).
- **Failure:** When the software behaves incorrectly (caused by a fault).

*Example:* A divide-by-zero fault causes a crash — the crash is the failure.

### *Reliability Models*

- **Mean Time Between Failures (MTBF):** Average time between system failures.
- **Fault Tree Analysis (FTA):** Diagram to analyze causes of failures.
- **Reliability Block Diagram (RBD):** Graphical representation of system reliability.

---

## 6. Software Reuse

- Using existing software components in new applications.
- Reduces development time and cost.
- Promotes standardization and reduces bugs.

*Example:* Reusing a login authentication module in multiple apps.

---

## 7. CASE (Computer-Aided Software Engineering)

### *Scope*

- Tools that support software development activities: design, coding, testing, etc.

### *Architecture*

- CASE tools may include:
  - **Front-end tools** (e.g., diagram editors)
  - **Back-end tools** (e.g., code generators)
  - **Database:** Stores project info (designs, code, test cases)

### *CASE Support in SDLC*

- **Requirement Analysis:** Use CASE to capture user needs.
- **Design:** Create UML diagrams, database schemas.
- **Implementation:** Generate code templates.
- **Testing:** Automated testing tools.

- **Maintenance:** Track changes and issues.

 *Example:* Rational Rose, StarUML, Visual Paradigm.

#### **Documentation**

- CASE tools help auto-generate and update documentation.
- Improves consistency and saves time.

#### **Reverse Software Engineering**

- Analyzing software to identify components and relationships.
- Useful when source code is unavailable or undocumented.

*Example:* Decompiling an old Java program to understand how it works.

---

## Summary Table

| Concept | Description          | Example                     |
|---------|----------------------|-----------------------------|
| Update  | Small changes        | Fixing a bug in a web app   |
| Upgrade | Major changes        | Android 12 → Android 13     |
| Patch   | Specific fix         | Security patch for browsers |
| Version | Software stage label | v1.2.0                      |
| Failure | Runtime error        | App crash                   |
| Fault   | Code defect          | Null pointer                |

CASE Tool

Automation in development

StarUML

Reverse Engineering

Understanding existing software

Decompiling legacy code