Software Engineering

Course Code: OEC-AIML 801C Semester: VIII

UNIT 1 — Introduction to Software Engineering

Syllabus

- Overview of System Analysis & Design
- Business System Concept
- System Development Life Cycle (SDLC)
- Waterfall Model
- Spiral Model
- Feasibility Analysis
- Technical Feasibility
- Cost-Benefit Analysis
- COCOMO Model

1. Software Engineering

Definition : Software Engineering is the **systematic, disciplined, and measurable approach** to the development, operation, maintenance, and evolution of software.

According to **IEEE**: "Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software."

Why Software Engineering is Needed

Earlier software was small.

Example: Calculator, Snake Game, Notepad

One programmer could build everything.

Today's software includes : Banking Systems, Google, Facebook, Hospital Management, AI Systems, Amazon, Online Examination Systems etc. These systems contain millions of lines of code.

Without proper engineering,

- deadlines fail
- bugs increase
- maintenance becomes impossible
- cost increases

Hence Software Engineering was introduced.

Objectives of Software Engineering

- Produce quality software
- Reduce development cost
- Finish within schedule
- Improve maintainability
- Improve reliability
- Increase customer satisfaction
- Reduce risk
- Make software reusable

Characteristics of Good Software

A good software system should have:

Characteristic	Meaning
Correctness	Produces expected output
Reliability	Works continuously without failure
Efficiency	Uses less CPU, RAM, storage
Maintainability	Easy to modify
Portability	Runs on different platforms
Usability	Easy to use
Security	Protects data
Scalability	Can support more users

2. System

Definition : A **System** is a collection of interconnected components working together to achieve a common objective.

Example: College Management System

Components:

- Students
- Teachers
- Accounts
- Examination
- Library

All communicate with each other.

Elements of a System

Inputs -> Processing -> Outputs -> Feedback -> Control

1. Input

Raw data entering the system.

Example : Student marks.

2. Processing

Converts input into meaningful information.

Example : Calculate total marks.

3. Output

Final result.

Example : Marksheet.

4. Feedback

Checks if objectives are achieved.

Example : Error reports.

5. Control

Maintains proper functioning.

Example : Administrator permissions.

3. Business System Concept

Definition : A Business System is an organized arrangement of people, processes, technology, and resources that work together to achieve business objectives.

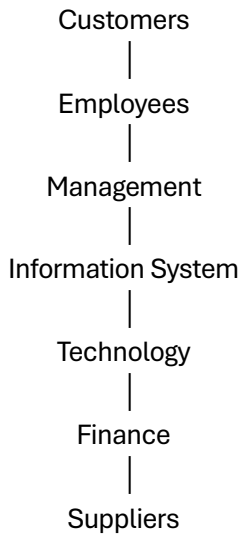
Example : Amazon

Resources:

- Customers
- Warehouses
- Payment Gateway
- Delivery Network
- Inventory
- Website

Together they form one business system.

Components of Business System :



Characteristics :

- ✓ Goal-oriented
- ✓ Organized
- ✓ Dynamic
- ✓ Open system
- ✓ Resource sharing
- ✓ Information flow

Types of Business Systems :

1. Manual System

Uses paper files.

Example : Old library records.

2. Automated System

Uses computers.

Example : Online Banking.

3. Hybrid System

Uses both manual and computerized methods.

Example : Many government offices.

Advantages :

- Faster operations
- Better decision making
- Less paperwork
- Improved communication
- Higher productivity
- Better customer satisfaction

4. System Analysis :

Definition : System Analysis is the process of studying an existing system to identify problems, understand user requirements, and propose improvements.

It answers: "What should the system do?"

Objectives :

- Identify problems
- Gather requirements
- Improve efficiency
- Reduce cost
- Understand user needs

Responsibilities of System Analyst :

A System Analyst acts as a bridge between users and developers.

- Requirement gathering
- Feasibility study
- Preparing SRS
- Communication with users
- Risk identification
- Documentation
- Coordinating development

Skills Required :

Technical Skills

- Programming knowledge
- Database knowledge
- Networking
- SDLC understanding

Communication Skills

- Speaking
- Listening
- Writing

Management Skills

- Team handling
- Planning
- Decision making

Analytical Skills

- Problem solving
- Logical thinking

5. System Design

After analysis, System Design decides "How will the system work?"

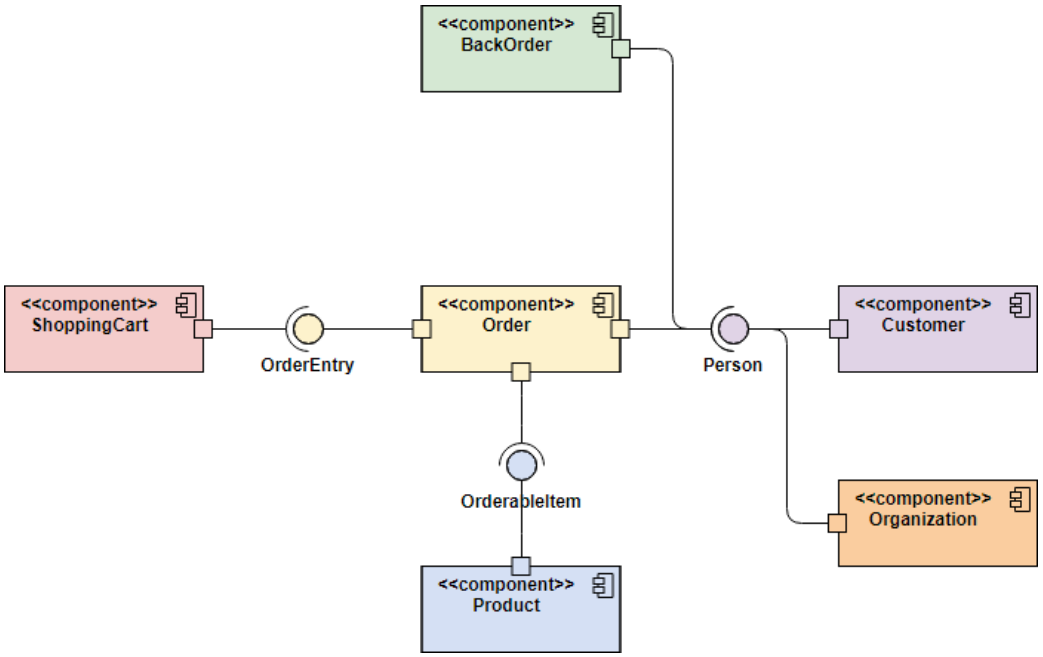
Types of Designs :

High-Level Design (HLD)

- Architecture
- Modules
- Database
- Interfaces

Low-Level Design (LLD)

- Algorithms
- Classes
- Functions
- Tables



Design Objectives

- Simplicity
- Reliability
- Reusability
- Flexibility
- Efficiency

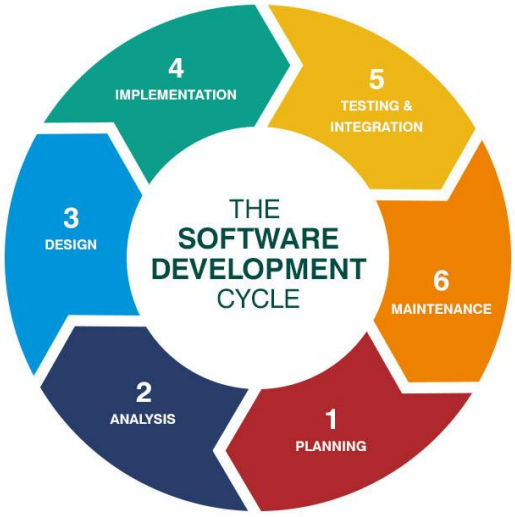
Difference Between Analysis and Design

System Analysis	System Design
Problem identification	Problem solution
What to build	How to build
Requirement focused	Implementation focused
Done first	Done after analysis
Prepared by analyst	Prepared by designer

6. SDLC (System Development Life Cycle)

Definition : SDLC is a structured process followed for developing software from idea to maintenance.

Planning
↓
Requirement Analysis
↓
System Design
↓
Coding
↓
Testing
↓
Deployment
↓
Maintenance



Phase 1 — Planning

- Identify problem
- Define objectives
- Estimate budget
- Estimate time
- Resource planning

Output : Project Plan

Phase 2 — Requirement Analysis

- Interview users
- Questionnaires
- Observation
- Study existing system

Output : Software Requirement Specification (SRS)

Phase 3 — Design

- Database design
- UI design
- Architecture
- Module design

Output : Design Document

Phase 4 — Coding

Programmers convert design into source code.

Languages : Java, Python, C#, C++, PHP

Output : Working software modules

Phase 5 — Testing

Checks software quality.

Types of Testing :

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

Output : Bug reports and validated software

Phase 6 — Deployment

Software is installed for users.

Deployment types

- Direct
- Parallel
- Pilot
- Phased

Phase 7 — Maintenance

Software is updated after release.

Types

- Corrective
- Adaptive
- Perfective
- Preventive

Advantages of SDLC :

- Better planning
- Cost control
- Improved quality
- Better documentation
- Reduced risk
- Easier maintenance

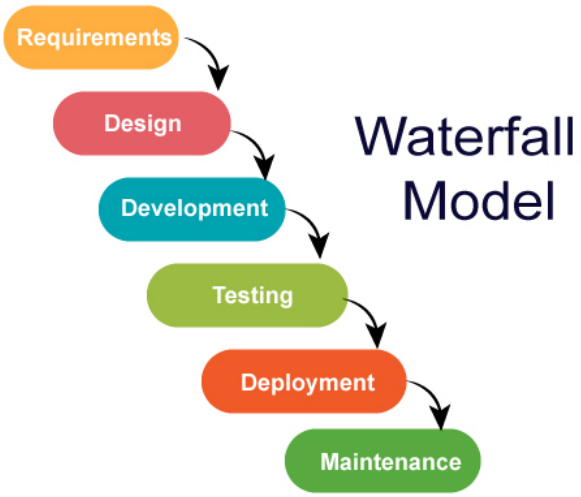
Disadvantages

- Time consuming
- More documentation
- Difficult to handle changing requirements in rigid models

7. Waterfall Model

Definition : The Waterfall Model is the **first and simplest SDLC model**, where each phase is completed before the next begins. It is a **linear sequential** model.

Requirements
↓
Design
↓
Coding
↓
Testing
↓
Deployment
↓
Maintenance



Characteristics :

- Linear approach
- No overlap between phases
- Heavy documentation
- Suitable for stable requirements

Advantages

- Easy to understand
- Easy management
- Clear milestones
- Good documentation
- Suitable for small projects with fixed requirements

Disadvantages

- Difficult to accommodate changes
- Testing occurs late
- High risk if requirements are incorrect
- Customer feedback comes late

Suitable Applications

- Payroll System
- Library Management
- Government Projects with fixed requirements
- Academic Projects

Waterfall vs Spiral Model

Waterfall	**Spiral**
Linear	Iterative
Less flexible	Highly flexible
Low risk handling	Strong risk handling
Customer involved mainly at start/end	Continuous customer involvement
Best for stable requirements	Best for changing requirements

8. Spiral Model

Definition : The Spiral Model, proposed by Barry Boehm, combines iterative development with systematic risk analysis. Development proceeds through repeated cycles (spirals), each reducing uncertainty and adding functionality.

Spiral Diagram :

Iteration 1

↺

Iteration 2

↺

Iteration 3

↺

Iteration 4

↺

Final Product : Each loop has four activities.

Four Quadrants :

1. Planning

- Gather objectives
- Define requirements

↓

2. Risk Analysis

- Identify risks
- Evaluate alternatives
- Build prototypes if required

↓

3. Engineering

- Design
- Coding
- Testing

↓

4. Evaluation

- Customer review
- Plan next iteration

Advantages

- Excellent risk management
- Continuous customer feedback
- Supports requirement changes
- Suitable for large and complex projects

Disadvantages

- Expensive
- Complex to manage
- Requires risk analysis expertise
- Not suitable for small projects

Applications

- Banking software
- Defence systems
- Space research
- AI platforms
- Enterprise software

Difference: Waterfall vs Spiral

Feature	Waterfall	Spiral
Development	Sequential	Iterative
Risk Handling	Minimal	Extensive
Cost	Lower	Higher
Flexibility	Low	High
Customer Feedback	Limited	Frequent
Best For	Small, stable projects	Large, high-risk projects

9. Feasibility Analysis

Definition : Feasibility Analysis determines whether a proposed software project is practical, economically viable, technically possible, and worth undertaking before significant resources are committed.

It answers: "Should we build this software?"

Objectives

- Reduce project failure
- Estimate cost
- Identify risks
- Improve planning
- Support decision-making

Types of Feasibility

1. Technical Feasibility

Checks whether the required technology, infrastructure, and expertise are available.

Questions include:

- Is the required hardware available?
- Is the software platform suitable?
- Does the team have the necessary skills?
- Can the system meet performance requirements?

Example: Building an AI-powered application without access to GPUs or machine learning expertise may not be technically feasible.

2. Economic Feasibility (Cost-Benefit Analysis)

Determines whether expected benefits justify the project's costs.

3. Operational Feasibility

Evaluates whether users and the organization will accept and effectively use the new system.

4. Legal Feasibility

Ensures compliance with laws, regulations, licenses, and contracts.

Examples:

- Data protection laws
- Software licensing
- Copyright compliance

5. Schedule Feasibility

Checks whether the project can realistically be completed within the required time.

10. Technical Feasibility (Detailed)

Technical feasibility examines whether the organization has:

- Suitable hardware
- Required software
- Skilled developers
- Reliable network infrastructure
- Adequate storage
- Security capabilities
- Scalability for future growth

Factors Affecting Technical Feasibility

- Availability of technology
- Team expertise
- System complexity
- Integration with existing systems
- Performance requirements
- Security requirements
- Maintenance capability

Advantages

- Reduces technical risk
- Improves planning
- Avoids technology failures
- Ensures realistic project execution

11. Cost-Benefit Analysis (CBA)

Definition : Cost-Benefit Analysis compares the **total expected costs** of a project with its **expected benefits** to determine whether it is financially worthwhile.

Types of Costs :

Development Costs

- Software licenses
- Hardware
- Developer salaries
- Testing
- Documentation

Operational Costs

- Maintenance
- Electricity
- Internet
- Hosting
- Cloud services
- Support staff

Indirect Costs

- User training
- Downtime
- Productivity loss during transition

Types of Benefits :

1. Tangible Benefits : Can be measured in money.

Examples:

- Increased sales
- Reduced labor cost
- Lower maintenance cost

2. Intangible Benefits : Hard to measure directly.

Examples:

- Better customer satisfaction
- Improved reputation
- Faster decision-making
- Better employee morale

Simple Cost-Benefit Formula

$$\text{Net Benefit} = \text{Total Benefits} - \text{Total Costs}$$

If Net Benefit > 0, the project is generally considered financially beneficial.

Benefit-Cost Ratio (BCR)

$$\text{BCR} = \frac{\text{Benefits}}{\text{Costs}}$$

- BCR > 1 → Accept
- BCR = 1 → Break-even
- BCR < 1 → Reject

Example

Project Cost = ₹10,00,000

Expected Benefit = ₹15,00,000

Net Benefit = ₹5,00,000

BCR = 15,00,000 / 10,00,000 = **1.5**, indicating the project is economically feasible.

12. COCOMO Model

Full Form : COnstructive **CO**st **MO**del

Developed by Barry Boehm in 1981.

It estimates:

- Development effort
- Cost
- Time
- Team size

based primarily on the estimated size of the software.

KLOC

KLOC = Thousand Lines of Code

Example:

50,000 lines of code = **50 KLOC**

Basic COCOMO Formula

Effort (Person-Months):

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

Development Time (Months):

$$T = c \times (E)^d$$

Where:

- **E** = Effort
- **T** = Development time
- **a, b, c, d** = Constants based on project type

Types of COCOMO :

1. Organic Mode

Characteristics

- Small teams
- Experienced developers
- Familiar problem domain
- Simple projects

Examples

- Library Management System
- Attendance Management System

Typical constants:

- a = 2.4
- b = 1.05
- c = 2.5
- d = 0.38

2. Semi-Detached Mode

Characteristics

- Medium-sized teams
- Mixed experience
- Moderate complexity

Examples

- Inventory Management
- ERP modules

Typical constants:

- a = 3.0
- b = 1.12
- c = 2.5
- d = 0.35

3. Embedded Mode

Characteristics

- Strict hardware/software constraints
- High reliability requirements
- Complex systems

Examples

- Aircraft control systems
- Medical devices
- Spacecraft software

Typical constants:

- a = 3.6
- b = 1.20
- c = 2.5
- d = 0.32

Advantages

- Simple estimation method
- Helps estimate budget and schedule
- Useful in project planning
- Widely studied in software engineering

Limitations

- Depends on accurate size estimation
- Less suitable for Agile development
- Does not fully capture modern software practices
- Original model is based on historical project data

Unit 1 Quick Revision Sheet

- **Software Engineering:** Systematic approach to software development.
- **System Analysis:** Determines **what** the system should do.
- **System Design:** Determines **how** the system will do it.
- **Business System:** People + processes + technology working toward business goals.
- **SDLC Phases:** Planning → Requirements → Design → Coding → Testing → Deployment → Maintenance.
- **Waterfall:** Linear, sequential, best for fixed requirements.
- **Spiral:** Iterative, risk-driven, ideal for large/high-risk projects.
- **Feasibility Types:** Technical, Economic, Operational, Legal, Schedule.
- **Cost-Benefit Analysis:** Compares project costs against expected benefits using metrics like Net Benefit and BCR.
- **COCOMO:** Estimation model based on KLOC; project modes are **Organic**, **Semi-Detached**, and **Embedded**.

UNIT 2 — System Design

Syllabus

- System Design
- Context Diagram
- Data Flow Diagram (DFD)
- Problem Partitioning
- Top-Down Design
- Bottom-Up Design
- Decision Tree
- Decision Table
- Structured English
- Functional vs Object-Oriented Approach

1. System Design

Definition : System Design is the process of converting the requirements identified during **System Analysis** into a blueprint for implementing the software system.

Simply put : **Analysis tells us WHAT to build, Design tells us HOW to build it.**

A good system design ensures that the software is efficient, reliable, maintainable, scalable, and easy to understand.

Objectives of System Design

- Convert requirements into implementable modules.
- Reduce system complexity.
- Improve maintainability.
- Ensure high performance.
- Increase reliability.
- Support future expansion.
- Improve software quality.

Importance of System Design

Without proper design:

- Code becomes difficult to maintain.
- Bugs increase.
- Team collaboration becomes difficult.
- Performance decreases.
- Future modifications become expensive.

Good design reduces overall development cost.

Types of System Design :

1. Logical Design

Focuses on **what the system should do**, without considering specific technologies.

Includes:

- Process flow
- Data flow
- Inputs
- Outputs
- Database relationships

Example: Designing an Online Library System without deciding whether Java or Python will be used.

2. Physical Design

Focuses on **how the system will actually be implemented**.

Includes:

- Programming language
- Database selection
- Hardware
- Network architecture
- File organization

Example: Choosing MySQL, Java Spring Boot, and AWS for deployment.

Characteristics of a Good Design

A good software design should be:

Characteristic	**Meaning**
Simple	Easy to understand
Efficient	Uses minimum resources
Modular	Divided into independent modules
Reusable	Components can be reused
Flexible	Easy to modify
Reliable	Produces consistent output
Secure	Protects data
Maintainable	Easy to update

2. Context Diagram

Definition

A **Context Diagram** is the highest-level Data Flow Diagram (Level 0) that represents the **entire system as a single process** and shows only its interactions with external entities.

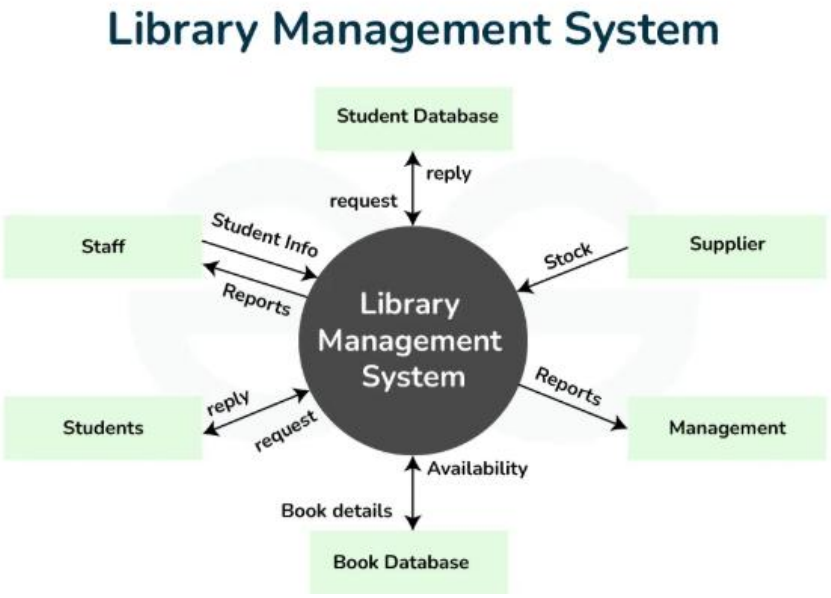
It answers: **Who communicates with the system?**

It does **not** show internal processing.

Components :

A Context Diagram contains only three elements:

1. External Entities
2. The System (single process)
3. Data Flows



Symbols

Rectangle → External Entity
Circle/Oval → System
Arrow → Data Flow

Example: Library Management System

```
+----------------+
|    Student     |
+----------------+
        |
  Borrow Request
        |
        V
   (Library System)
        ^
        |
    Book Details
        |
+----------------+
|    Librarian   |
+----------------+
```

External Entities:

- Student
- Librarian

System:

- Library System

Data Flow:

- Borrow Request
- Book Details

Advantages

- Very easy to understand.
- Gives an overall system view.
- Useful for client discussions.
- Helps define system boundaries.

Limitations

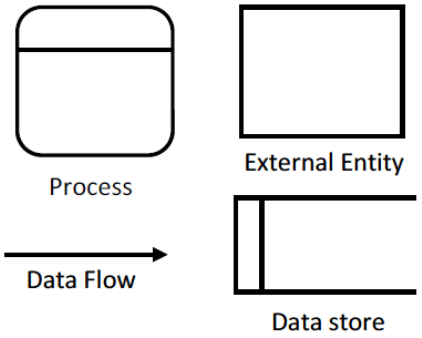
- No internal details.
- Cannot describe complex logic.
- Not suitable for implementation.

3. Data Flow Diagram (DFD)

Definition : A **Data Flow Diagram (DFD)** graphically represents how data moves through a system.

It illustrates:

- Input
- Processing
- Output
- Data Storage
- Data Movement



Unlike a flowchart, a DFD focuses on **data**, not program control.

Components of DFD

1. External Entity : Source or destination of data.

Symbol:

```
+-----------+
| Customer|
+-----------+
```

2. Process : Transforms input into output.

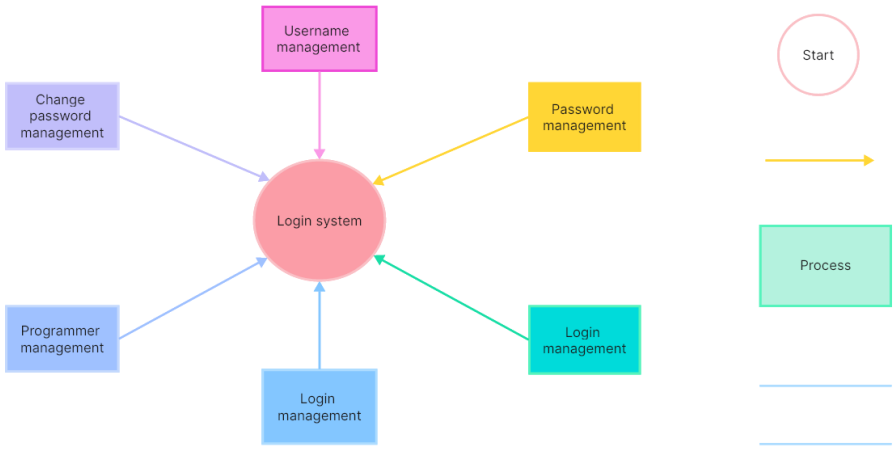
Symbol:

(Process)

3. Data Store : Stores information.

Symbol:

```
===========
Database
===========
```



4. Data Flow : Represents movement of information.

Symbol:

---------->

Levels of DFD :

Level 0 (Context Diagram)

Entire system shown as one process.

Customer
  |
  V
(Order System)
  |
  V
Supplier

Level 1 DFD

System divided into major processes.

Customer
  |
  V
Receive Order
  |
  V
Verify Order
  |
  V
Generate Bill
  |
  V
Database

Level 2 DFD

Each Level 1 process is further divided into sub-processes.

Example :

Verify Order

↓

- Check Customer
- Check Product
- Check Stock
- Approve Order

Advantages of DFD

- Easy to understand.
- Shows complete data movement.
- Helps identify missing processes.
- Improves communication.
- Supports modular design.

Disadvantages

- Cannot represent timing.
- Cannot show decision logic.
- Becomes complex for very large systems.

Context Diagram vs DFD

Context Diagram	**DFD**
Single process	Multiple processes
Highest level	Detailed levels
No internal details	Shows internal processing
Shows system boundary	Shows complete data flow
Simpler	More detailed

4. Problem Partitioning

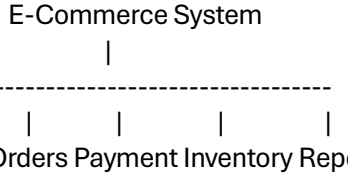
Definition : Problem Partitioning is the process of breaking a large software problem into smaller, manageable modules.

This follows the **Divide and Conquer** principle.

Why Partition Problems?

Suppose you're building an E-commerce website.

Instead of writing everything in one file, divide it into:



Each module can be developed independently.

Advantages

- Easier development.
- Better testing.
- Easier debugging.
- Parallel development.
- Better maintenance.
- Code reuse.

Types of Partitioning :

Horizontal Partitioning : Divides based on major functions.

Example : Bank System (Accounts, Loans, ATM, Internet Banking)

Vertical Partitioning : Divides based on control levels.

Example :

User Interface
↓
Business Logic
↓
Database

5. Top-Down Design

Definition

Top-Down Design starts with the overall system and gradually divides it into smaller modules.

Think of it as:

Whole
↓
Subsystem
↓
Module
↓
Function
↓
Code

Example :

College Management :

Admission
Library
Examination
Hostel
Accounts

Each module is divided further.

Advantages

- Clear architecture.
- Better planning.
- Easier maintenance.
- Modular development.

Disadvantages

- Low-level details may be overlooked initially.
- Requires complete understanding of requirements.

6. Bottom-Up Design

Definition

Bottom-Up Design begins by developing small modules first and then integrating them into larger systems.

Functions
↓
Modules
↓
Subsystems
↓
Complete System

Example

Build first:

- Login Module
- Payment Module
- Database Module

Then integrate them.

Advantages

- Reusable modules.
- Easier testing.
- Lower-level components become reliable.
- Good for component-based development.

Disadvantages

- Overall architecture may emerge late.
- Integration can be challenging.

Top-Down vs Bottom-Up

Top-Down	Bottom-Up
Starts from system	Starts from modules
Architecture first	Components first
Planning driven	Implementation driven
Easy to visualize system	Easy to reuse components
Less reuse initially	Higher reuse

7. Decision Tree

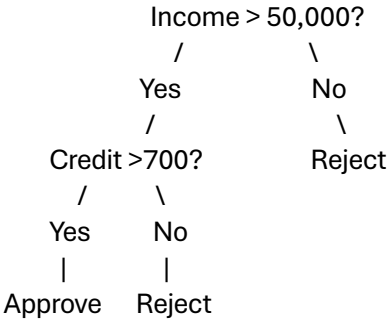
Definition : A Decision Tree is a graphical representation of decision-making using branches to show possible conditions and resulting actions.

It is useful when multiple conditions determine different outcomes.

Components

- Decision Node
- Condition
- Branch
- Action

Example: Loan Approval



Advantages

- Easy to understand.
- Visual representation.
- Suitable for complex decisions.
- Helps identify all possible outcomes.

Disadvantages

- Large trees become difficult to manage.
- Repeated conditions increase complexity.

8. Decision Table

Definition : A Decision Table represents complex business rules in a tabular format.

It contains:

- Conditions
- Possible values
- Actions

Structure

Example: Loan Approval

Conditions	Rule1	Rule2	Rule3	Rule4
Income >50K	Y	Y	N	N
Credit >700	Y	N	Y	N
Approve	✓	-	-	-
Reject	-	✓	✓	✓

Components :

Condition Stub : Lists all conditions.

Action Stub : Lists all actions.

Rule Columns : Each column represents one possible combination.

Advantages

- Eliminates ambiguity.
- Covers all combinations.
- Easy to verify business rules.
- Excellent documentation.

Disadvantages

- Large tables become difficult to maintain.
- Number of rules increases exponentially with conditions.

Decision Tree vs Decision Table

Decision Tree	Decision Table
Graphical	Tabular
Easy to visualize	Easy to analyze rules
Good for sequential decisions	Good for many combinations
Harder to modify when large	Easier to update systematically

9. Structured English

Definition : Structured English is a technique used to describe the logic of a process using simple English combined with programming constructs.

It avoids programming syntax while remaining precise.

Features

- Uses IF, THEN, ELSE.
- Uses WHILE, DO.
- Uses indentation.
- Uses simple English.
- No programming language required.

Example

```
READ Marks
IF Marks >= 40
   PRINT "PASS"
ELSE
   PRINT "FAIL"
ENDIF
```

Another Example :

```
READ Username
READ Password
IF Username and Password are correct
   Allow Login
ELSE
   Display Error
ENDIF
```

Advantages

- Easy for both users and developers.
- Improves communication.
- Independent of programming language.
- Useful during system design.

Disadvantages

- Not executable.
- May become lengthy for complex systems.
- Interpretation may vary if not written carefully.

Decision Tree vs Decision Table vs Structured English

Decision Tree	Decision Table	Structured English
Graphical	Tabular	Textual
Shows decision paths	Shows rule combinations	Describes logic in English
Easy to visualize	Easy to verify	Easy to write
Best for branching	Best for complex business rules	Best for algorithms and process descriptions

10. Functional Approach

Definition : The Functional Approach organizes software around **functions or procedures**.

Focus: **What functions does the system perform?**

Programs are divided into functions.

Example:

```
        Main
          |
--------------------------
Login()
Payment()
Search()
Logout()
```

Characteristics

- Function-oriented.
- Data is often shared.
- Procedural programming.
- Sequential execution.

Advantages

- Simple.
- Easy for small projects.
- Faster initial development.

Disadvantages

- Poor reusability.
- Weak data security.
- Difficult maintenance in large projects.
- High coupling.

Languages

- C
- Pascal
- FORTRAN

11. Object-Oriented Approach (OOA)

Definition : The Object-Oriented Approach organizes software around **objects**, where each object combines data and behavior.

Focus: **What objects exist, and how do they interact?**

Basic Concepts :

1.Class : Blueprint of objects.

Example: Car

2. Object : Instance of a class.

Example: BMW, Audi, Tesla

3. Encapsulation : Combining data and methods into one unit while restricting direct access to internal data.

4. Inheritance : A child class acquires properties from a parent class.

Example:

Vehicle
|
Car
|
Electric Car

5.Polymorphism : The same interface behaves differently depending on the object.

Example: draw() for Circle and Rectangle.

6.Abstraction : Showing only essential details while hiding implementation complexity.

Advantages

- High code reusability.
- Better security through encapsulation.
- Easier maintenance.
- Better scalability.
- Suitable for large projects.
- Promotes modular design.

Disadvantages

- More complex to learn.
- Slightly higher initial design effort.
- Can introduce overhead for very small projects.

Languages

- Java
- C++
- C#
- Python
- Kotlin

Functional vs Object-Oriented Approach

Functional Approach	Object-Oriented Approach
Function-centered	Object-centered
Procedures manipulate data	Objects encapsulate data and behavior
Lower reusability	Higher reusability
Data often shared globally	Data hidden through encapsulation
Suitable for small applications	Suitable for medium and large applications
Examples: C, Pascal	Examples: Java, C++, C#, Python

Unit 2 Quick Revision Sheet

- **System Design:** Converts requirements into an implementation blueprint.
- **Context Diagram:** Highest-level DFD showing the entire system as one process interacting with external entities.
- **DFD:** Models the flow of data using external entities, processes, data stores, and data flows; common levels are 0, 1, and 2.
- **Problem Partitioning:** Breaks a complex problem into smaller modules using horizontal or vertical partitioning.
- **Top-Down Design:** Starts from the overall system and decomposes into smaller modules.
- **Bottom-Up Design:** Builds small reusable modules first and integrates them into the complete system.
- **Decision Tree:** Graphical representation of decisions and their possible outcomes.
- **Decision Table:** Tabular representation of business rules covering all condition combinations.
- **Structured English:** Uses structured control statements (IF, ELSE, WHILE, etc.) with simple English to describe logic.
- **Functional Approach:** Organizes software around procedures/functions.
- **Object-Oriented Approach:** Organizes software around objects using concepts such as classes, encapsulation, inheritance, polymorphism, and abstraction.

UNIT 3 – Coding, Documentation & Testing

Syllabus

Part A – Coding & Documentation (4L)

- Structured Programming
- Object-Oriented Programming
- Information Hiding
- Reuse (Software Reusability)
- System Documentation

Part B – Testing (8L)

- Levels of Testing
- Integration Testing
- Test Case Specification
- Reliability Assessment
- Validation & Verification
- Metrics
- Monitoring & Control

PART A – CODING & DOCUMENTATION

1. Coding

Definition : Coding is the process of converting the software design into source code using a programming language.

It is the implementation phase of the SDLC.

Example:
Design says: Calculate Student Percentage
Coding: percentage = (marks/500)*100;

Objectives of Coding

- Convert design into executable software
- Produce efficient code
- Minimize errors
- Improve readability
- Support future maintenance

Characteristics of Good Code

Characteristic	Meaning
Readable	Easy to understand
Simple	Avoid unnecessary complexity
Efficient	Uses fewer resources
Reusable	Can be used elsewhere
Maintainable	Easy to modify
Reliable	Produces correct output
Portable	Runs on multiple platforms
Secure	Protects against vulnerabilities

Coding Standards : Coding standards improve consistency and collaboration.

Examples:

Naming Convention

Good :
studentName
calculateSalary()
totalMarks

Bad :
x
abc
temp123

Proper Indentation

Good:
```
if(condition)
{
   statement;
}
```

Bad :
```
if(condition){
statement;
}
```

Comments

// Calculate total salary

Avoid excessive or outdated comments.

Avoid Code Duplication

Instead of writing the same logic repeatedly:

Create

calculateInterest()

and call it wherever needed.

Advantages of Good Coding Practices

- Easy debugging
- Better maintenance
- Faster development
- Higher reliability
- Easier teamwork

2. Structured Programming

Definition : Structured Programming is a programming methodology that develops programs using **well-defined control structures** while avoiding unnecessary jumps (such as excessive goto statements).

It emphasizes:

- Sequence
- Selection
- Iteration

Three Basic Control Structures

1. Sequence : Statements execute one after another.

```
Read A
Read B
Sum = A+B
Print Sum
```

2. Selection : Decision making.

```
IF Marks >=40
   Pass
ELSE
   Fail
```

3. Iteration : Repeating instructions.

```
FOR i =1 to 10
   Print i
```

Features

- Modular programming
- Easy debugging
- Clear logic
- Less complexity
- Better maintenance

Advantages

- Easy to understand
- Reduced bugs
- Easier testing
- Improved reliability
- Better documentation

Disadvantages

- Less suitable for very large systems
- Data security is limited
- Lower code reuse than OOP

Languages

- C
- Pascal
- BASIC

3. Object-Oriented Programming (OOP)

Definition : Object-Oriented Programming is a programming methodology that organizes software into **objects**, where each object contains **data and methods**.

Basic Concepts :

Class : Blueprint of an object.

Example: Student

Object : Instance of a class.

Rahul, Priya, Ankit

Encapsulation : Combining data and functions into one unit while restricting direct access to internal data.

Inheritance : Allows one class to inherit features of another.

Vehicle
|
Car
|
Electric Car

Polymorphism : One interface, many implementations.

Example :

draw()
↓
Circle
Rectangle
Triangle

Abstraction : Showing only necessary details while hiding implementation complexity.

Advantages

- High code reuse
- Better maintenance
- Data security
- Modular design
- Easy scalability

Disadvantages

- Higher initial complexity
- More memory usage
- Longer design phase

Languages

- Java
- C++
- Python
- C#
- Kotlin

Structured Programming vs OOP

Structured Programming	Object-Oriented Programming
Function-oriented	Object-oriented
Data is separate	Data + methods together
Less reusable	Highly reusable
Less secure	More secure
Suitable for small projects	Suitable for large systems

4. Information Hiding

Definition : Information Hiding is the principle of hiding the internal implementation details of a module or class while exposing only the necessary interface.

Users know **what** a component does, not **how** it works.

Example : ATM Machine

You insert a card and enter a PIN.

You do **not** know how:

- PIN verification works
- Database queries execute
- Cash dispenser is controlled

Those implementation details are hidden.

Example in OOP :

class BankAccount
private balance
public deposit()
public withdraw()

The variable balance is hidden from direct access.

Advantages

- Better security
- Easier maintenance
- Reduced dependency
- Supports modularity
- Improves reliability

Information Hiding vs Encapsulation

Information Hiding	**Encapsulation**
Hides implementation details	Bundles data and methods
Focuses on access restriction	Focuses on organization
Security principle	Programming mechanism

5. Software Reuse (Reusability)

Definition : Software Reuse means using existing software components to build new applications instead of creating everything from scratch.

Examples

- Login module
- Payment gateway
- Authentication service
- Notification library
- Database access layer

Types of Reuse

Code Reuse : Reusing functions/classes.

Component Reuse : Using ready-made modules.

Library Reuse : Using libraries.

Example : NumPy, TensorFlow, Spring Boot

Framework Reuse : Using complete frameworks.

Examples: Django, React, ASP.NET

Advantages

- Faster development
- Lower cost
- Better reliability
- Less testing effort
- Higher productivity

Disadvantages

- Compatibility issues
- Dependency management
- Learning existing code
- Licensing concerns

6. System Documentation

Definition : System Documentation is the written description of software that explains its design, implementation, operation, maintenance, and usage.

Importance : Documentation helps:

- Developers
- Testers
- Users
- Maintenance engineers
- Future developers

Types :

User Documentation : Prepared for end users.

Contains :

- Installation guide
- User manual
- FAQ
- Help files

Technical Documentation

Prepared for developers.

Contains :

- Design documents
- Architecture
- Source code documentation
- API documentation
- Database schema

Operational Documentation

Prepared for system administrators.

Contains :

- Backup procedures
- Deployment guide
- Disaster recovery
- Server configuration

Advantages

- Easy maintenance
- Better communication
- Easier training
- Faster troubleshooting
- Knowledge preservation

PART B – SOFTWARE TESTING

7. Software Testing

Definition : Software Testing is the process of evaluating software to identify defects and verify that it meets specified requirements.

Testing answers: "Does the software work correctly?"

Objectives

- Find defects
- Verify requirements
- Improve quality
- Increase reliability
- Reduce maintenance cost

Principles of Testing

1. Testing shows the presence of defects, not their absence.
2. Exhaustive testing is impossible.
3. Start testing early.
4. Defects cluster together.
5. Tests should be repeatable.
6. Different software requires different testing strategies.

8. Levels of Testing

Software testing is performed in stages.

Unit Testing
↓
Integration Testing
↓
System Testing
↓
Acceptance Testing

1. Unit Testing : Testing individual functions or modules.

Performed by: Developers

Example :Test Login()

Advantages

- Finds bugs early
- Easy debugging
- Low cost

2. Integration Testing : Testing interaction between modules.

Example :

Login Module
↓
Database Module
↓
Dashboard

Types of Integration Testing :

Big Bang : All modules integrated together.

Advantages : Simple.

Disadvantages : Bug isolation is difficult.

Top-Down Integration : Starts from top modules; Uses **Stubs** for lower modules.

Main
↓
Login
↓
Database (Stub)

Bottom-Up Integration : Starts from lower modules; Uses **Drivers** for upper modules.

Driver
↓
Database
↓
Login
↓
Main

Sandwich (Hybrid) : Combination of Top-Down and Bottom-Up.

3. System Testing : Entire software tested as one system.

Checks:

- Performance
- Security
- Reliability
- Functionality

4. Acceptance Testing: Performed by customers.

Purpose: Verify software meets business needs.

Types:
Alpha Testing: Performed at developer's site by internal users.
Beta Testing: Performed by real users in real environments before full release.

Levels of Testing Summary

Testing Level	Performed By	Purpose
Unit	Developer	Test individual module
Integration	Developer/Tester	Test interaction between modules
System	Testing Team	Test complete system
Acceptance	Customer/User	Validate business requirements

9. Test Case Specification

Definition: A Test Case is a documented set of inputs, execution conditions, expected results, and pass/fail criteria used to verify software behavior.

Components

- Test Case ID
- Objective
- Preconditions
- Test Steps
- Input Data
- Expected Output
- Actual Output
- Status (Pass/Fail)

Example

Field	Value
ID	TC001
Module	Login
Input	Correct username & password
Expected Result	Login successful
Status	Pass

Characteristics of Good Test Cases

- Clear
- Repeatable
- Independent
- Traceable to requirements
- Covers positive and negative scenarios

10. Reliability Assessment

Definition: Reliability is the probability that software performs its required functions correctly **without failure** for a specified period under specified conditions.

Factors Affecting Reliability

- Number of defects
- Code quality
- Testing quality
- Environment
- Maintenance

Improving Reliability

- Code reviews
- Automated testing
- Continuous integration
- Preventive maintenance
- Fault tolerance

Reliability vs Availability

Reliability	Availability
Probability of failure-free operation	Probability the system is operational when needed
Focuses on correctness over time	Focuses on uptime

11. Verification and Validation

Verification

Definition : Verification checks whether the software is being built **according to specifications**.

Question: **Are we building the product right?**

Verification activities:

- Reviews
- Inspections
- Walkthroughs
- Static analysis

No program execution is required.

Validation

Definition: Validation checks whether the final software satisfies user needs.

Question: **Are we building the right product?**

Validation involves executing the software.

Examples:

- Functional testing
- User acceptance testing
- System testing

Verification vs Validation

Verification	Validation
Static process	Dynamic process
No execution required	Execution required
Checks specifications	Checks user requirements
Performed during development	Performed after implementation/testing
Finds design/document defects	Finds functional defects

12. Software Metrics

Definition : Software Metrics are quantitative measures used to evaluate software products, processes, and projects.

Objectives

- Measure quality
- Improve productivity
- Estimate cost
- Estimate effort
- Track project progress

Types of Metrics

Product Metrics: Measure software itself.

Examples:

- LOC (Lines of Code)
- Cyclomatic Complexity
- Defect Density

Process Metrics: Measure development activities.

Examples:

- Defect Removal Efficiency
- Testing Efficiency

Project Metrics: Measure project management.

Examples:

- Cost
- Schedule
- Productivity
- Team size

Common Metrics:

LOC (Lines of Code): Measures program size.

Defect Density: KLOC (thousand lines of code)
Lower values indicate better quality.

Productivity: LOC / Person-Month

Measures development efficiency.

Code Coverage: Percentage of code executed during testing.

Higher coverage generally indicates more thorough testing, though it does not guarantee the absence of defects.

Advantages of Metrics

- Better estimation
- Quality improvement
- Performance measurement
- Early problem detection
- Data-driven decisions

13. Monitoring & Control

Definition: Monitoring is the continuous observation of project progress.

Control involves taking corrective actions when deviations occur.

Monitoring Activities

- Track progress
- Review milestones
- Monitor budget
- Monitor quality
- Monitor risks
- Monitor resources

Control Activities

- Adjust schedule
- Allocate resources
- Correct defects
- Update plans
- Manage risks

Monitoring Tools

- Gantt Chart
- Burndown Chart
- Dashboards
- Earned Value Management (EVM)
- Issue/Bug Tracking Systems

Benefits

- Better quality
- Reduced delays
- Cost control
- Improved customer satisfaction
- Early detection of issues
-

Unit 3 Quick Revision Sheet

Coding & Documentation

- **Structured Programming:** Uses sequence, selection, and iteration; function-oriented.
- **Object-Oriented Programming:** Uses classes and objects; supports encapsulation, inheritance, polymorphism, and abstraction.
- **Information Hiding:** Conceals internal implementation details while exposing only necessary interfaces.
- **Software Reuse:** Reusing existing code, components, libraries, or frameworks to reduce cost and development time.
- **System Documentation:** Includes user, technical, and operational documentation.

Testing

- **Levels of Testing:** Unit → Integration → System → Acceptance.
- **Integration Testing Types:** Big Bang, Top-Down (uses stubs), Bottom-Up (uses drivers), Sandwich.
- **Test Case:** Defines inputs, execution steps, expected outputs, and pass/fail criteria.
- **Reliability:** Probability of failure-free operation over a specified time.
- **Verification:** "Are we building the product right?" (static, specification-focused).
- **Validation:** "Are we building the right product?" (dynamic, user-focused).
- **Software Metrics:** Product, Process, and Project metrics; examples include LOC, Defect Density, Productivity, and Code Coverage.
- **Monitoring & Control:** Tracks project progress and applies corrective actions to keep scope, cost, schedule, and quality under control.

UNIT 4 – Software Project Management

Syllabus

- Software Project Management
- Project Scheduling
- Staffing
- Software Configuration Management (SCM)
- Software Quality Assurance (SQA)
- Project Monitoring

1. Software Project Management (SPM)

Definition: Software Project Management (SPM) is the process of **planning, organizing, directing, coordinating, and controlling** software projects to ensure they are completed:

- On time
- Within budget
- According to specifications
- With the desired quality

In simple words, Software Project Management is the art of managing people, resources, time, cost, and quality to successfully complete a software project.

Objectives of Software Project Management

- Complete the project within the deadline.
- Minimize development cost.
- Deliver high-quality software.
- Efficiently utilize resources.
- Manage project risks.
- Improve customer satisfaction.
- Coordinate team members.
- Monitor project progress.

Importance of Project Management

Without proper management:

- Deadlines are missed.
- Budget exceeds estimates.
- Quality decreases.
- Team communication suffers.
- Customer requirements may not be met.

A good project manager ensures the project stays under control throughout its lifecycle.

Responsibilities of a Software Project Manager

The Project Manager is responsible for:

- Project planning
- Requirement understanding
- Cost estimation
- Scheduling
- Resource allocation
- Team management
- Risk management
- Client communication
- Quality assurance
- Progress monitoring
- Final project delivery

Skills Required for a Project Manager

Technical Skills

- Software development knowledge
- Database concepts
- Networking
- SDLC understanding
- Testing methodologies

Management Skills

- Leadership
- Planning
- Decision making
- Risk management
- Conflict resolution

Communication Skills

- Presentation
- Negotiation
- Documentation
- Team coordination

Personal Skills

- Problem solving
- Time management
- Adaptability
- Motivation

Project Management Activities

Project Planning
↓
Cost Estimation
↓
Scheduling
↓
Staffing
↓
Development
↓
Quality Assurance
↓
Monitoring
↓
Delivery

2. Project Scheduling

Definition: Project Scheduling is the process of deciding **when each project activity will start, how long it will take, and when it will finish**.

It helps ensure that work is completed in the correct order and within the project deadline.

Objectives

- Complete project on time.
- Avoid delays.
- Allocate resources efficiently.
- Monitor progress.
- Improve coordination.

Scheduling Steps

Step 1

Identify project tasks.

↓

Step 2

Estimate time for each task.

↓

Step 3

Identify task dependencies.

↓

Step 4

Assign resources.

↓

Step 5

Prepare project schedule.

↓

Step 6

Monitor progress.

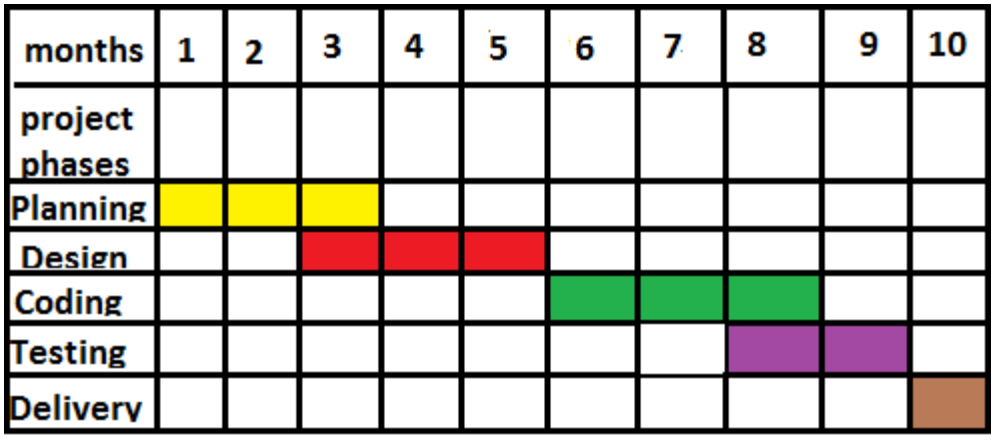
Scheduling Techniques

1. Gantt Chart

A Gantt Chart is a bar chart used to represent project activities over time.

Example

Task	Week
Requirement	▮▮▮
Design	▮▮▮
Coding	▮▮▮▮▮
Testing	▮▮▮▮
Deployment	▮▮



Advantages

- Easy to understand.
- Shows project timeline.
- Tracks progress.
- Useful for small and medium projects.

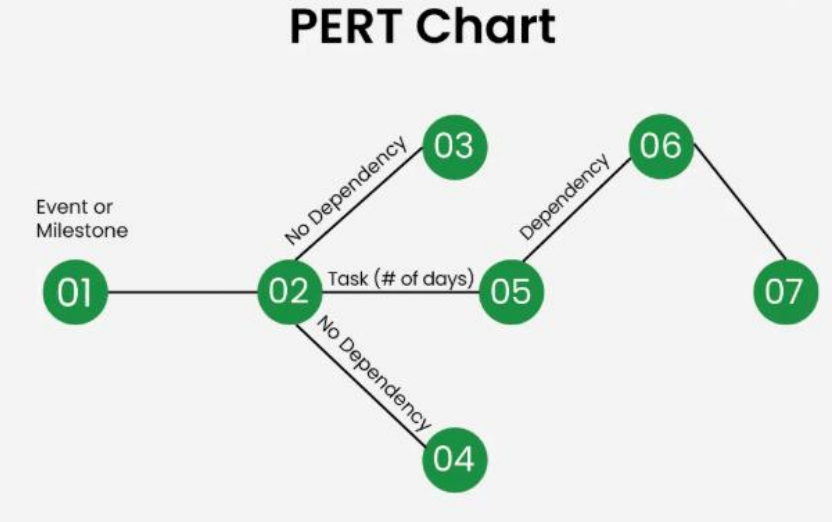
Disadvantages

- Complex for large projects.
- Dependencies are not clearly visible.

2. PERT Chart : PERT = **Program Evaluation and Review Technique**

A PERT chart represents tasks as a network showing dependencies and the sequence of activities.

```
Start
  |
  A
  |
  B ------ C
   \       /
     D
     |
   Finish
```



Advantages

- Shows task dependencies.
- Identifies the critical path.
- Useful for large projects.

Difference Between Gantt and PERT

Gantt Chart	**PERT Chart**
Bar chart	Network diagram
Shows timeline	Shows task dependencies
Simple	Complex
Easy to read	Better for planning large projects

3. Staffing : Staffing is the process of selecting, assigning, training, and managing people required to complete a software project.
Simply, "Putting the right person in the right job at the right time."

Staffing Activities

- Recruitment
- Selection
- Training
- Role assignment
- Performance evaluation
- Motivation
- Team development

Project Team Members:

- **Project Manager:** Plans and controls the project.
- **System Analyst:** Collects requirements and prepares system specifications.
- **Software Developer:** Writes source code.
- **Tester (QA Engineer):** Tests software and identifies defects.
- **Database Administrator (DBA):** Manages databases.
- **UI/UX Designer:** Designs user interfaces and user experience.
- **DevOps Engineer:** Handles deployment, automation, and infrastructure.

Factors Affecting Staffing

- Project size
- Budget
- Required skills
- Project duration
- Technology used
- Team experience

Advantages of Good Staffing

- Better productivity
- Improved quality
- Faster completion
- Better communication
- Higher customer satisfaction

4. Software Configuration Management (SCM)

Definition: Software Configuration Management (SCM) is the process of **identifying, organizing, controlling, and tracking changes** made to software throughout its lifecycle.

It ensures that changes are managed systematically without affecting software quality.

Why SCM is Needed

Imagine five developers modifying the same project.

Without SCM:

- Files may be overwritten.
- Wrong versions may be deployed.
- Bugs may reappear.
- Code conflicts increase.

SCM solves these problems.

Objectives

- Manage software changes.
- Maintain version history.
- Prevent unauthorized modifications.
- Support teamwork.
- Improve software quality.

SCM Activities

1. Configuration Identification

Identify all configuration items.

Examples:

- Source code
- Documents
- Database scripts
- Test cases
- Configuration files

2. Version Control

Maintains different versions of software.

Example

Version 1.0
↓
Version 1.1
↓
Version 2.0

3. Change Control

Every change should be:

Requested

↓

Reviewed

↓

Approved

↓

Implemented

↓

Verified

4. Configuration Status Accounting

Maintains records of:

- Current version
- Previous versions
- Approved changes
- Pending changes

5. Configuration Audit

Checks whether:

- Approved changes were implemented.
- Documentation matches software.
- No unauthorized changes exist.

SCM Tools

Examples include:

- Git
- GitHub
- GitLab
- Bitbucket
- Apache Subversion (SVN)

Advantages

- Better teamwork
- Easy rollback
- Version tracking
- Reduced conflicts
- Improved maintenance

5. Software Quality Assurance (SQA)

Definition: Software Quality Assurance (SQA) is a planned and systematic set of activities performed to ensure that software processes and products conform to specified quality standards.

It focuses on **preventing defects**, not just finding them.

Objectives

- Improve software quality.
- Prevent defects.
- Ensure standards compliance.
- Increase customer satisfaction.
- Reduce maintenance cost.

Difference Between QA and QC

Quality Assurance (QA)	**Quality Control (QC)**
Process-oriented	Product-oriented
Prevents defects	Detects defects
Proactive	Reactive
Focuses on process improvement	Focuses on testing and inspection

SQA Activities

- Process definition
- Standards enforcement
- Code reviews
- Design reviews
- Audits
- Testing support
- Documentation review
- Defect prevention

Benefits

- Higher software quality
- Lower maintenance cost
- Better customer satisfaction
- Reduced rework
- Increased reliability

Software Quality Factors

Important quality attributes include:

Quality Factor	Meaning
Correctness	Produces expected results
Reliability	Operates without failure
Efficiency	Uses resources effectively
Integrity	Protects against unauthorized access
Usability	Easy to learn and use
Maintainability	Easy to modify
Flexibility	Easy to adapt to new requirements
Portability	Runs on different platforms
Reusability	Components can be reused
Interoperability	Works with other systems

6. Project Monitoring

Definition: Project Monitoring is the continuous process of measuring project progress and comparing it with the project plan.

It helps identify problems early so corrective actions can be taken.

Objectives

- Track project progress.
- Compare actual vs planned work.
- Detect delays.
- Control project cost.
- Improve quality.
- Manage risks.

Monitoring Activities

- Progress tracking
- Budget monitoring
- Schedule monitoring
- Quality monitoring
- Resource monitoring
- Risk monitoring
- Defect tracking

Monitoring Process

Project Plan
↓
Collect Progress Data
↓
Compare with Plan
↓
Identify Deviations
↓
Take Corrective Action
↓
Continue Monitoring

Key Performance Indicators (KPIs)

Common indicators include:

- Percentage of work completed
- Number of defects
- Budget utilization
- Schedule variance
- Team productivity
- Customer satisfaction

Monitoring Tools

1. Gantt Chart: Tracks activity completion over time.

2. Dashboards: Displays project status using charts and graphs.

3. Bug Tracking Systems: Jira , Bugzilla , Redmine

4. Earned Value Management (EVM): Measures project performance by comparing:

- Planned work
- Completed work
- Actual cost

Benefits

- Early problem detection
- Better cost control
- Better schedule management
- Improved quality
- Better decision making

Project Monitoring vs Project Control

Project Monitoring	Project Control
Observes progress	Takes corrective action
Identifies deviations	Removes deviations
Collects information	Implements solutions
Continuous activity	Performed when needed

SQA vs Testing

Software Quality Assurance	Software Testing
Prevents defects	Finds defects
Process-oriented	Product-oriented
Performed throughout development	Performed mainly after coding
Focuses on improving processes	Focuses on verifying software behavior

Version Control vs Change Control

Version Control	Change Control
Maintains software versions	Approves and manages requested changes
Focuses on version history	Focuses on decision-making before implementation
Examples: Git commits and branches	Examples: Change requests and approval workflows

Unit 4 Quick Revision Sheet

Software Project Management

- Planning, organizing, directing, and controlling software projects.
- Objectives: complete on time, within budget, and with high quality.

Project Scheduling

- Plans project activities over time.
- Techniques: **Gantt Chart** and **PERT Chart**.

Staffing

- Recruiting, assigning, training, and managing project personnel.
- Common roles: Project Manager, System Analyst, Developer, Tester, DBA, UI/UX Designer, DevOps Engineer.

Software Configuration Management (SCM)

- Manages and controls software changes.
- Activities: Configuration Identification, Version Control, Change Control, Status Accounting, Configuration Audit.

Software Quality Assurance (SQA)

- Process-oriented approach to prevent defects.
- Activities: Reviews, audits, standards enforcement, process improvement, documentation review.

Project Monitoring

- Tracks project progress, cost, schedule, quality, resources, and risks.
- Uses KPIs and tools such as Gantt Charts, dashboards, bug trackers, and EVM.

UNIT 5 – UML (Unified Modeling Language)

Syllabus

- Static and Dynamic Models
- Why Modeling
- UML (Unified Modeling Language)
- Class Diagram
- Interaction Diagram
    - Collaboration Diagram
    - Sequence Diagram
- State Chart Diagram
- Activity Diagram
- Implementation Diagram

1. Software Modeling

Definition: Software Modeling is the process of creating a **visual representation (blueprint)** of a software system before actual coding begins.

Instead of directly writing code, developers first prepare diagrams that describe:

- Structure
- Behavior
- Relationships
- Data flow
- Object interaction

Just as an architect prepares a building blueprint before construction, software engineers prepare models before software development.

Objectives of Modeling

- Understand system requirements.
- Reduce complexity.
- Improve communication.
- Detect design errors early.
- Simplify implementation.
- Improve documentation.
- Support maintenance.

Advantages of Modeling

- Better planning.
- Easy communication among stakeholders.
- Improved software quality.
- Faster development.
- Easier debugging.
- Better documentation.
- Lower development cost.

2. Why Modeling?

Modeling is important because software systems are often too complex to understand directly from code.

Without models:

- Requirements may be misunderstood.
- Design errors increase.
- Team communication becomes difficult.
- Maintenance becomes expensive.

Modeling provides a clear and organized view of the system.

Importance of Modeling

- Visualizes the system.
- Simplifies complex systems.
- Improves customer understanding.
- Helps developers.
- Helps testers.
- Helps project managers.
- Supports future modifications.

3. UML (Unified Modeling Language)

Definition: Unified Modeling Language (UML) is a **standard graphical language** used to specify, visualize, construct, and document software systems.

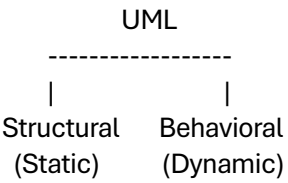
UML is **not a programming language**. It is used only for designing software.

Features of UML

- Standard notation.
- Object-oriented.
- Platform independent.
- Graphical.
- Easy communication.
- Widely accepted.

UML Categories

UML diagrams are broadly divided into:



4. Static Model

Definition: A Static Model describes the **structure** of the software.

It answers: **"What components exist?"**

It does **not** describe system behavior over time.

Characteristics

- Represents classes.
- Represents objects.
- Represents relationships.
- Represents attributes.
- Represents methods.

Examples

- Class Diagram
- Object Diagram
- Component Diagram
- Deployment Diagram

5. Dynamic Model

Definition: A Dynamic Model describes the **behavior** of software during execution.

It answers: **"How does the system behave?"**

Characteristics

- Shows activities.
- Shows interactions.
- Shows message flow.
- Shows state changes.
- Shows execution sequence.

Examples

- Sequence Diagram
- Collaboration Diagram
- Activity Diagram
- State Chart Diagram

Static Model vs Dynamic Model

Static Model	**Dynamic Model**
Shows structure	Shows behavior
Describes components	Describes execution
Does not change with time	Changes with time
Easy to design	Represents runtime behavior
Example: Class Diagram	Example: Sequence Diagram

6. UML Class Diagram

Definition: A Class Diagram represents the **static structure** of an object-oriented system.

It shows:

- Classes
- Attributes
- Methods
- Relationships

Structure: A class contains three sections.

```
---------------------
    Student
---------------------
RollNo
Name
Marks
---------------------
calculateResult()
display()
---------------------
```

Components

Class: Blueprint of objects.

Example: Student

Attributes: Data members.

Example: Name , Age , RollNo

Methods: Functions

Example: Login(), Save(), Calculate()

Relationships: Association

Objects communicate.

Teacher -------- Student

Inheritance

Vehicle
↑
Car

Car inherits Vehicle.

Aggregation: Weak "has-a" relationship.

Example: Department → Teachers

Teachers can exist independently.

Composition: Strong "has-a" relationship.

Example: House → Rooms

If the house is destroyed, its rooms no longer exist as part of that house.

Advantages

- Easy visualization.
- Better system design.
- Supports OOP.
- Improves documentation.

7. Interaction Diagram

Definition: Interaction Diagrams show **how objects communicate** with each other to perform a task.

There are two important interaction diagrams:

- Sequence Diagram
- Collaboration Diagram

8. Sequence Diagram

Definition: A Sequence Diagram shows the **order of messages exchanged between objects over time**.

It focuses on **time sequence**.

Components

- Objects
- Lifelines
- Messages
- Activation bars

Example: ATM Withdrawal

Customer
|
ATM
|
Bank Server
|
Cash Dispenser

Customer → ATM
Insert Card
ATM → Bank
Verify PIN
Bank → ATM
Verified
ATM → Cash Dispenser
Dispense Cash
Cash Dispenser → Customer
Cash

Advantages

- Easy to understand.
- Shows execution order.
- Useful during design.
- Helps debugging.

9. Collaboration Diagram

Definition: A Collaboration Diagram (also called a **Communication Diagram** in UML 2.x) shows **how objects interact** to complete a task, emphasizing their relationships rather than the timing of messages.

Example

Customer
|
ATM
|
Bank
|
Cash Dispenser

Messages are usually numbered.

Example:
1 Insert Card
2 Verify PIN
3 Dispense Cash

Advantages

- Shows object relationships.
- Simple.
- Compact.
- Good for communication analysis.

Sequence Diagram vs Collaboration Diagram

Sequence Diagram	**Collaboration Diagram**
Shows time sequence	Shows object relationships
Vertical layout	Network layout
Easy to follow execution order	Easy to understand interactions
Focus on message order	Focus on communication links

10. State Chart Diagram

Definition: A State Chart Diagram represents the **different states** of an object during its lifetime and the events that cause transitions between those states.

Components

- Initial state
- States
- Transitions
- Final state

Example: Online Order

Start
↓
Order Placed
↓
Payment Done
↓
Packed
↓
Shipped
↓
Delivered
↓
End

Applications

- ATM
- Traffic lights
- Online shopping
- Mobile applications
- Banking systems

Advantages

- Shows complete lifecycle.
- Easy to understand state transitions.
- Useful for event-driven systems.

11. Activity Diagram

Definition: An Activity Diagram represents the **workflow** of activities in a system.

It is similar to a flowchart.

Example: Online Shopping

Start
↓
Login
↓
Select Product
↓
Add to Cart
↓
Payment
↓
Order Confirmation
↓
End

Decision Example

Payment Successful?
| |
Yes No
↓ ↓
Order Confirmed Retry Payment

Applications

- Business process modeling
- Workflow analysis
- Algorithm representation
- Requirement analysis

Advantages

- Easy to understand.
- Shows parallel activities.
- Represents business processes effectively.

12. Implementation Diagram

Implementation Diagrams describe the physical aspects of a software system.They show how software components are organized and deployed.
There are two important implementation diagrams:

- Component Diagram
- Deployment Diagram

13. Component Diagram

Definition: A Component Diagram shows the organization and dependencies among software components.

Example:

Web Application
|

|
Database Module
|
Payment Module
|
Login Module
|
Report Module

Advantages

- Modular design.
- Easy maintenance.
- Supports reuse.
- Better understanding of system architecture.

14. Deployment Diagram

Definition: A Deployment Diagram shows how software components are deployed on hardware devices.

Example:

Client PC
|
Internet
|
Web Server
|
Application Server
|
Database Server

Applications

- Cloud computing
- Client-server architecture
- Distributed systems
- Enterprise software

Advantages

- Shows physical architecture.
- Helps deployment planning.
- Improves network understanding.

UML Diagram Summary

Diagram	Purpose
Class Diagram	Shows classes and relationships
Sequence Diagram	Shows message sequence over time
Collaboration Diagram	Shows communication between objects
State Chart Diagram	Shows object state transitions
Activity Diagram	Shows workflow and business processes
Component Diagram	Shows software components
Deployment Diagram	Shows hardware deployment

Static Model vs Dynamic Model

Static Model	Dynamic Model
Represents structure	Represents behavior
No time dependency	Time-dependent behavior
Classes, components, deployment	Interactions, activities, states
Used during system design	Used to model runtime behavior

Class Diagram vs Object Diagram

Class Diagram	Object Diagram
Represents classes	Represents instances (objects)
Shows design	Shows runtime snapshot
Static	Static snapshot of objects

Sequence Diagram vs Collaboration Diagram

Sequence Diagram	Collaboration Diagram
Emphasizes message timing	Emphasizes object relationships
Vertical timeline	Network-style connections
Easy to understand execution flow	Easy to visualize communication links

Activity Diagram vs Flowchart

Activity Diagram	Flowchart
UML-based	General algorithm representation
Can model concurrent activities	Typically models sequential control flow
Used in object-oriented design	Used for general programming logic

Component Diagram vs Deployment Diagram

Component Diagram	Deployment Diagram
Shows software components	Shows hardware nodes and deployment
Focuses on software architecture	Focuses on physical architecture
Logical organization	Physical implementation

Unit 5 Quick Revision Sheet

Software Modeling

- Creates a visual blueprint before coding.
- Simplifies design, communication, and maintenance.

UML (Unified Modeling Language)

- Standard graphical language for software modeling.
- Not a programming language.

Static Models

- Describe the structure of the system.
- Examples: Class, Component, Deployment diagrams.

Dynamic Models

- Describe the behavior of the system.
- Examples: Sequence, Collaboration, State Chart, Activity diagrams.

UML Diagrams

- **Class Diagram:** Classes, attributes, methods, and relationships.
- **Sequence Diagram:** Time-ordered interaction between objects.
- **Collaboration Diagram:** Object communication and relationships.
- **State Chart Diagram:** States and transitions of an object.
- **Activity Diagram:** Workflow of activities and decision points.
- **Component Diagram:** Organization and dependencies of software components.
- **Deployment Diagram:** Physical deployment of software onto hardware.