SCNR Government Degree College, Proddatur,
Department of Computer Science
II BSc-IV Sem
Object Oriented Software Engineering

Syllabus: UNIT-I : Introduction to Object-Oriented Programming: Overview of software engineering, Introduction to Object-Oriented Programming (OOP) concepts (classes, objects, inheritance, polymorphism), Unified Modelling Language (UML) basics, Introduction to software development process and software development life cycle (SDLC).

Overview of Software Engineering

Software Engineering is the application of engineering principles to the design, development, testing, and maintenance of software. It focuses on building high-quality, reliable, and cost-effective software systems in a systematic way.

It involves activities such as requirements analysis, design, implementation, testing, deployment, and maintenance, usually organized through the Software Development Life Cycle (SDLC). Software engineering helps manage complexity, improve software quality, reduce cost, and ensure that software meets user requirements.

Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around **objects**, rather than functions or logic. An object represents a real-world entity and contains both **data** (attributes) and **methods** (functions) that operate on the data. OOP helps in building modular, reusable, secure, and easy-to-maintain software systems. The core concepts of OOP include **classes, objects, inheritance, and polymorphism**.

1. Class

A **class** is a blueprint or template for creating objects. It defines the properties (variables) and behaviors (methods) that the objects created from the class will have. A class does not occupy memory until an object is created from it.

Characteristics of a Class:

- Defines data members and member functions
- Acts as a logical structure
- Supports code reusability and abstraction

Example:
A class Car may define attributes such as color, model, and speed, and methods like start() and stop().

2. Object

An **object** is an instance of a class. It represents a real-world entity and occupies memory. Objects interact with each other by invoking methods.

Characteristics of an Object:

- Has identity, state, and behavior
- Created using a class
- Uses memory when instantiated

Example:
If Car is a class, then myCar and yourCar are objects of the class Car, each having different values for the same attributes.

3. Inheritance

Inheritance is a mechanism in OOP by which one class (called the **subclass** or child class) acquires the properties and behaviors of another class (called the **superclass** or parent class). It promotes **code reusability** and establishes a relationship between classes.

Types of Inheritance:

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance (supported through interfaces in some languages)

Advantages of Inheritance:

- Reduces code duplication
- Improves code maintainability
- Supports extensibility

Example:
A class Vehicle can have a method move(). A class Car can inherit from Vehicle and use the move() method without redefining it.

4. Polymorphism

Polymorphism means "many forms." It allows the same method or operation to behave differently based on the object that invokes it. Polymorphism improves flexibility and scalability of the program.

Types of Polymorphism:

1. **Compile-time Polymorphism (Method Overloading):**
   - Same method name with different parameter lists
2. **Run-time Polymorphism (Method Overriding):**
   - Subclass provides a specific implementation of a method already defined in the parent class

Advantages of Polymorphism:

- Enhances flexibility
- Simplifies code readability
- Supports dynamic behavior

Example:
A method draw() may behave differently for objects like Circle, Rectangle, and Triangle.

Advantages of OOP

- Modularity and structured programming
- Code reusability through inheritance
- Improved security through data hiding
- Easy maintenance and scalability
- Real-world problem modeling

Unified Modeling Language (UML) – Basics

Unified Modeling Language (UML) is a standardized visual modeling language used in software engineering to **analyze, design, visualize, and document** software systems. It is mainly used in **object-oriented development**, but it can model any type of system.

Purpose of UML

- To **visualize** the structure and behavior of a system before coding
- To **specify** system components and their relationships
- To **document** system design for future reference
- To **improve communication** among developers, designers, and stakeholders

UML Building Blocks
1. Things (Basic Elements)

- **Structural Things**: Represent static parts of the system
  - Class, Object, Interface
- **Behavioral Things**: Represent dynamic behavior
  - Interaction, State
- **Grouping Things**:
  - Package (groups related elements)
- **Annotational Things**:
  - Notes (explain or comment on elements)

2. Relationships

- **Association** – general connection between classes
- **Aggregation** – weak "has-a" relationship
- **Composition** – strong "has-a" relationship
- **Generalization (Inheritance)** – "is-a" relationship
- **Dependency** – one element depends on another

UML Diagrams

Structural Diagrams

- **Class Diagram** – shows classes, attributes, methods, and relationships
- **Object Diagram** – shows instances of classes
- **Component Diagram** – shows software components
- **Deployment Diagram** – shows hardware and software mapping

Behavioral Diagrams

- **Use Case Diagram** – shows system functions from user view
- **Sequence Diagram** – shows object interactions in time order
- **Activity Diagram** – shows workflow of activities
- **State Machine Diagram** – shows state changes of an object

Advantages of UML

- Makes system design **clear and understandable**
- Reduces errors by identifying issues early
- Helps in **maintaining and extending** software
- Acts as a common language for all stakeholders

Introduction to Software Development Process and Software Development Life Cycle (SDLC)

Introduction to Software Development Process: The **software development process** refers to a structured set of activities used to design, develop, test, deploy, and maintain software systems. It provides a systematic approach to transforming user requirements into a reliable and efficient software product. A well-defined process helps ensure software quality, timely delivery, cost control, and effective management of complex projects. To manage these activities effectively, organizations follow a formal framework known as the **Software Development Life Cycle (SDLC)**.

Software Development Life Cycle (SDLC):

The **Software Development Life Cycle (SDLC)** is a step-by-step process that defines the phases involved in developing software from initial idea to final deployment and maintenance. It serves as a roadmap for developers, project managers, and stakeholders to ensure that software is built systematically, meets user requirements, and maintains high quality.

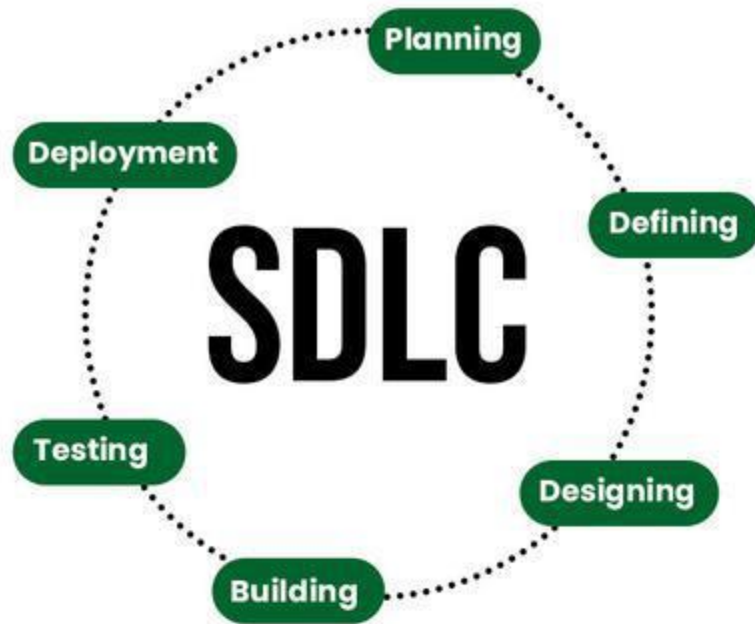
Phases of SDLC

1. **Requirement Analysis**
   This is the first and most critical phase of SDLC. In this phase, system analysts gather and analyze user requirements through interviews, questionnaires, observations, and documentation. The goal is to clearly understand what the user needs and to prepare a **Software Requirement Specification (SRS)** document, which acts as a reference throughout the development process.

2. **System Design**
   Based on the requirements, the system design phase defines the overall architecture of the software. It includes designing data structures, system interfaces, database design, hardware requirements, and software modules. The design phase is usually divided into:
   - **High-Level Design (HLD)** – overall system architecture
   - **Low-Level Design (LLD)** – detailed module-level design
3. **Implementation (Coding)**
   In this phase, developers write the actual source code using appropriate programming languages, tools, and frameworks. Each module is developed according to the design specifications. Coding standards and best practices are followed to ensure code quality, readability, and maintainability.
4. **Testing**
   Testing is carried out to identify and fix defects in the software. Different levels of testing are performed, such as unit testing, integration testing, system testing, and acceptance testing. The objective is to ensure that the software functions correctly, meets requirements, and is free from errors before deployment.
5. **Deployment (Installation)**
   After successful testing, the software is deployed to the user environment. This may involve installation, configuration, data migration, and user training. Deployment can be done in phases or all at once, depending on project requirements.
6. **Maintenance**
   Maintenance is an ongoing phase where the software is monitored and updated after deployment. It includes fixing bugs, improving performance, adding new features, and adapting the software to changes in the operating environment or user needs. Maintenance ensures the software remains useful and efficient over time.

Importance of SDLC

- Provides a structured and disciplined approach to software development
- Improves software quality and reliability
- Helps manage time, cost, and resources effectively
- Reduces development risks and errors
- Ensures user satisfaction by meeting requirements



SDLC

SCNR Government Degree College, Proddatur,
Department of Computer Science
II BSc-IV Sem
Object Oriented Software Engineering

Syllabus: UNIT-II : Requirements Analysis and Design: Requirements analysis and specification, Use cases and scenarios, Object-oriented analysis and design (OOAD), Design patterns, UML modelling techniques (class diagrams, sequence diagrams, state machine diagrams, activity diagrams) .

Requirements Analysis and Specification

Requirements Analysis and Specification is the process of identifying, understanding, and clearly documenting what a software system must do and the constraints under which it must operate. It is a crucial phase of software engineering because errors here can lead to costly rework later.

Key activities include:

- **Requirement gathering** from stakeholders (users, customers, managers)
- **Analysis** to remove ambiguity, conflicts, and incompleteness
- **Classification** of requirements into:
    - **Functional requirements** – what the system should do
    - **Non-functional requirements** – performance, security, usability, reliability, etc.
- **Specification** of requirements in a clear, structured document called the **Software Requirements Specification (SRS)**

Importance:

- Provides a clear understanding of system goals
- Acts as a contract between developers and customers
- Serves as a basis for design, development, and testing

Use Cases and Scenarios

Use Cases describe how users (actors) interact with the system to achieve a specific goal. They focus on **functional behavior** from the user's point of view.

Components of a Use Case:

- Actor (user or external system)
- Use case name
- Preconditions
- Main flow of events
- Alternate or exception flows
- Postconditions

Scenarios are specific, detailed sequences of steps within a use case. They show **one possible path** through the system, such as a normal case or an error case.

Example:

- **Use Case**: “User Login”
- **Scenario**: User enters valid username and password → system verifies → user is logged in successfully.

Object-Oriented Analysis and Design (OOAD)

Object-Oriented Analysis and Design (OOAD) is a software development approach that models a system as a collection of **interacting objects**. Each object represents a real-world entity and combines **data (attributes)** with **behavior (methods)**. OOAD helps in building systems that are modular, reusable, and easy to maintain.

Object-Oriented Analysis (OOA)

- Focuses on **understanding the problem domain**
- Identifies **objects, classes, and relationships**
- Uses models such as **use case diagrams and class diagrams**
- Answers *what the system should do*

Object-Oriented Design (OOD)

- Focuses on **how the system will be built**
- Defines **class structures, interfaces, and interactions**
- Specifies algorithms and data structures
- Prepares the system for implementation

Benefits of OOAD

- Better **modularity and reusability**
- Easier **maintenance and scalability**
- Improved **code organization**
- Natural mapping to real-world systems

Design Patterns

Design Patterns are **reusable, proven solutions** to commonly occurring problems in software design. They are not code, but **templates or best practices** that guide developers in designing flexible and efficient systems.

Types of Design Patterns

1. **Creational Patterns** – deal with object creation
    - Example: *Singleton, Factory, Abstract Factory*
2. **Structural Patterns** – deal with class and object composition
    - Example: *Adapter, Composite, Decorator*
3. **Behavioral Patterns** – deal with object interaction and responsibility
    - Example: *Observer, Strategy, Command*

Advantages of Design Patterns

- Promote **best design practices**
- Improve **code flexibility and readability**
- Reduce development time
- Make systems easier to modify and extend

UML Modelling Techniques

UML modelling techniques use different **diagrams** to represent the **structure and behavior** of a software system. Each diagram provides a specific view that helps in analysis, design, and communication.

1.Class Diagrams (UML)

Class diagrams represent the **static structure** of a system.

- Show **classes**, their **attributes**, and **methods**
- Display **relationships** such as association, inheritance, aggregation, and composition
- Used mainly during **object-oriented design**

Uses of Class Diagrams

- Designing system architecture
- Visualizing object-oriented structure
- Supporting code generation
- Helping in system maintenance

a).Structure of a Class: A class is represented by a rectangle divided into **three sections**:

1. **Class Name** **2.Attributes** (data members) 3.**Methods** (functions/operations)

b).Relationships in Class Diagrams:

1.Association – general relationship between classes (1-1,1-M,M-1,M-M)
Symbol: ClassA ———— ClassB
2.Aggregation – One class contains another, but both can exist independently

- weak “has-a” relationship

Symbol: ClassA ◇———— ClassB
3.Composition – Part cannot exist without the whole.

- strong “has-a” relationship

Symbol: ClassA ◆——— ClassB
4.Inheritance (Generalization) – One class inherits properties and methods of another.

- “is-a” relationship

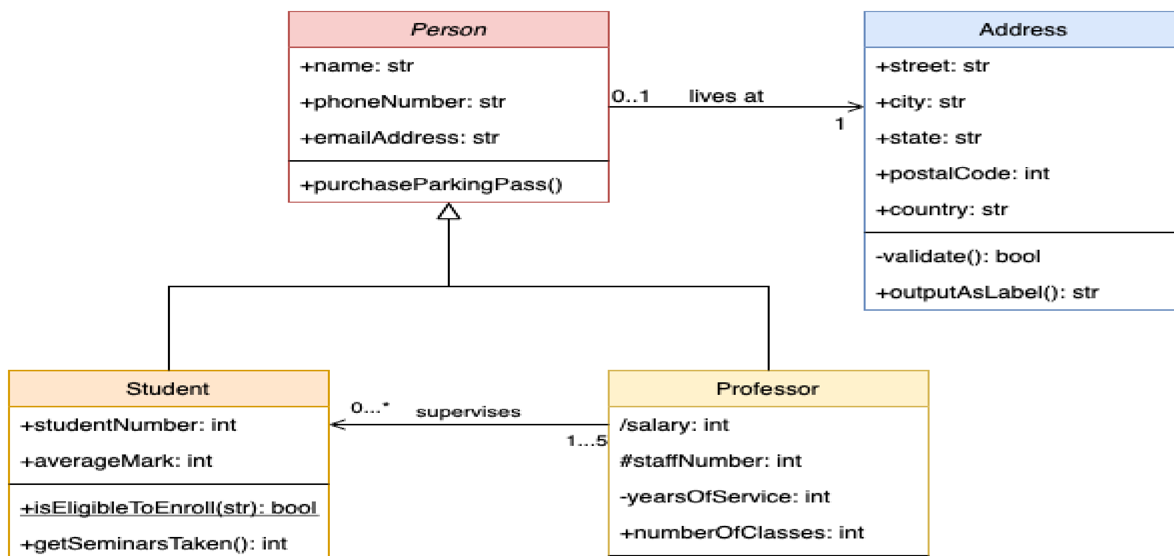
Symbol: Subclass ———▷ Superclass
5.Dependency – One class depends on another temporarily.

- a **“uses-a” relationship**

Symbol: ClassA - - - -▷ ClassB

c.)Visibility Symbols: visibility symbols indicate the **access level** of class members (attributes and methods).

Symbol	Meaning	Access Level
+	**Public**	Accessible from **anywhere** (inside or outside the class)
-	**Private**	Accessible **only within the class** itself
#	**Protected**	Accessible within the **class and its subclasses** (inheritance)
~	**Package / Default**	Accessible within the **same package/module** (in some languages like Java)



Example for Class Diagram

2. Sequence Diagrams

Sequence diagrams illustrate how **objects interact over time**.

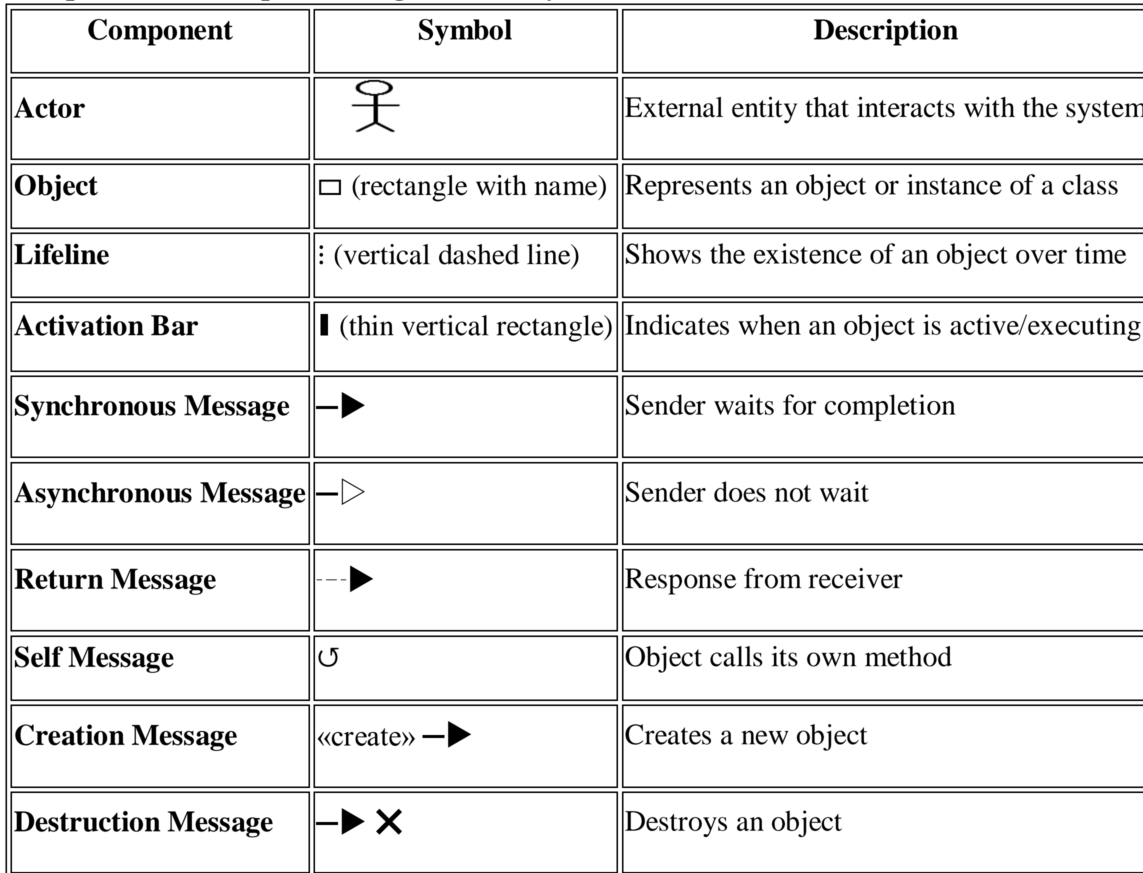
- Show objects as lifelines
- Messages are shown in the order they are sent
- Focus on **method calls and responses**

Purpose: To understand the **flow of control** and interaction logic for a use case.

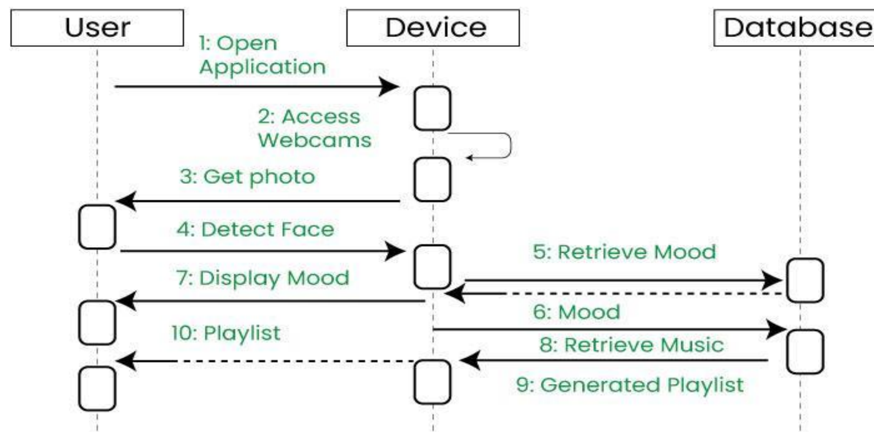
Advantages:

- Easy to understand interaction flow
- Helps detect design errors early

Components of a Sequence Diagram with Symbols



Component	Symbol	Description
Actor	웃	External entity that interacts with the system
Object	▭ (rectangle with name)	Represents an object or instance of a class
Lifeline	⋮ (vertical dashed line)	Shows the existence of an object over time
Activation Bar	▌ (thin vertical rectangle)	Indicates when an object is active/executing
Synchronous Message	—▶	Sender waits for completion
Asynchronous Message	—▷	Sender does not wait
Return Message	---▶	Response from receiver
Self Message	↺	Object calls its own method
Creation Message	«create» —▶	Creates a new object
Destruction Message	—▶ ✕	Destroys an object



Sequence Diagram for Playlist generation depends on the user mood

3. State Machine Diagrams:

State machine diagrams show the **states of an object** and transitions between them.

- Represent events that cause state changes
- Useful for systems with **complex behavior**

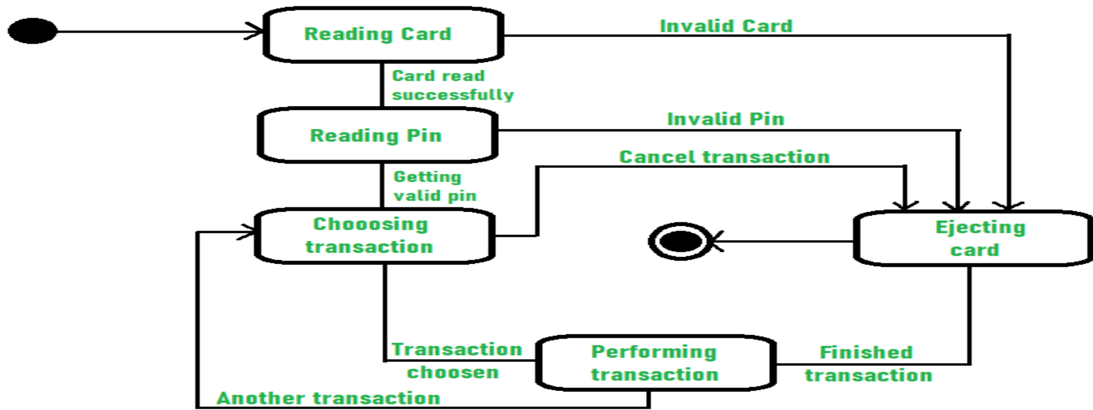
Purpose: To model **object behavior** in response to events.

Advantages

- Easy to understand object life cycle
- Clearly shows valid and invalid states

Components of a State Machine Diagram with Symbols

Component	Symbol	Description
Initial State	●	Starting point of the state machine
State	◯ / ▭ (rounded rectangle)	Represents a condition or situation of an object
Transition	→	Movement from one state to another
Event	event / trigger	Causes a transition
Action	/ action	Activity performed during transition
Final State	◎	Indicates end of the state machine
Self Transition	↺	Transition to the same state
Guard Condition	[condition]	Condition required for transition



4. Activity Diagrams:

Activity diagrams represent the **workflow or flow of activities**.

- Similar to flowcharts
- Show decisions, parallel activities, and synchronization
- Used to model **business processes** or algorithms

Purpose: To visualize the **step-by-step execution** of a process.

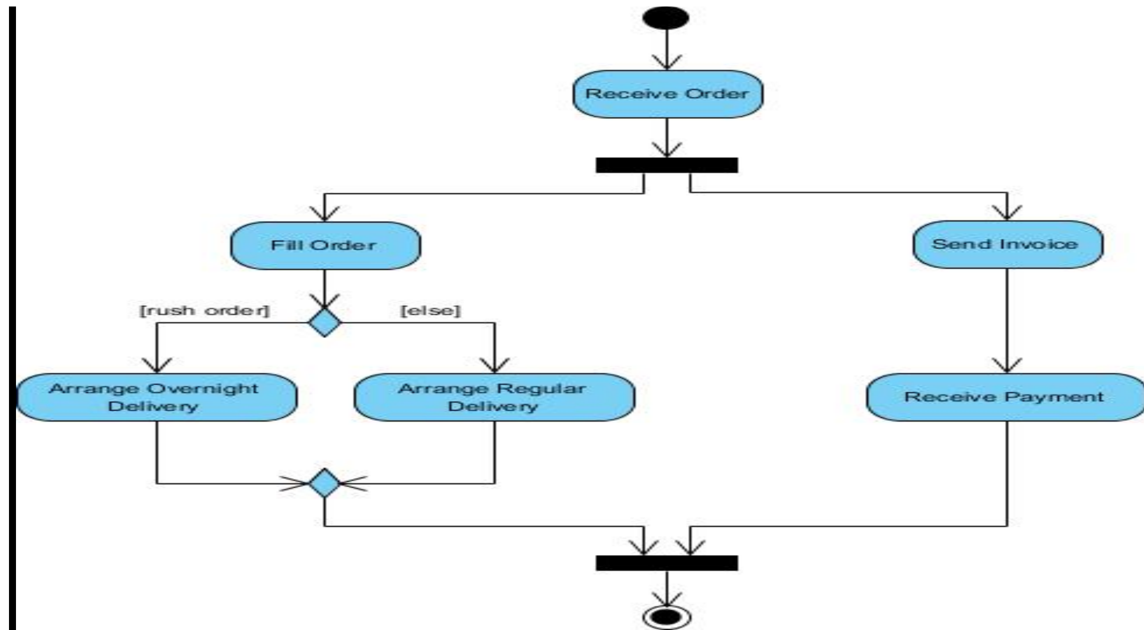
Advantages

- Simple and easy to understand
- Useful for both technical and non-technical users

Components of an Activity Diagram with Symbols

Component	Symbol	Description
Initial Node	●	Starting point of the activity
Activity / Action	▭ (rounded rectangle)	Represents an operation or step
Control Flow	→	Shows sequence of actions
Decision Node	◇	Represents branching (if/else)
Merge Node	◇	Merges alternate paths
Fork Node	━━	Splits flow into parallel activities

Component	Symbol	Description
Join Node	━━	Joins parallel flows
Final Node	◎	End of the activity
Swimlane	| |	Shows responsibility of activities



Activity Diagram Example - Process Order

SCNR Government Degree College, Proddatur
Department of Computer Science
II B.Sc-IV Sem-Software Engineer

UNIT-III :Software Construction and Testing: Software construction basics, Object-oriented design principles, Object-oriented programming languages (Java, C++, Python), Software testing basics (unit testing, integration testing, system testing), Test-driven development (TDD) .

Software Construction Basics:

Software construction is the process of writing, testing, debugging, and integrating code to build software according to design specifications.

Key Basics:

- **Coding**: Converting design into source code
- **Programming Languages**: Using appropriate languages and tools
- **Coding Standards**: Following proper naming and formatting rules
- **Debugging**: Identifying and fixing errors
- **Unit Testing**: Testing individual components
- **Code Reuse**: Using libraries and components

Object-Oriented Design Principles:

Object-oriented design principles help in creating flexible, reusable, and maintainable software**.**

Key Principles:

- **Encapsulation** – Binding data and methods together and hiding internal details
- **Abstraction** – Showing essential features while hiding complexity
- **Inheritance** – Reusing properties and behavior of existing classes
- **Polymorphism** – One interface, multiple implementations

Object-Oriented Programming Languages: Java, C++, Python: Object-Oriented Programming (OOP) languages support concepts like **classes, objects, inheritance, polymorphism, encapsulation, and abstraction**.

1. Java:

Java is a **pure object-oriented**, high-level programming language widely used for enterprise and web applications.

Key Features

- **Platform independent** (Write Once, Run Anywhere – WORA)
- Uses **JVM (Java Virtual Machine)**
- Strong **type checking**
- Automatic **garbage collection**
- Supports **multithreading**

OOP Support

- Everything (except primitives) is an object
- Supports inheritance using extends
- Supports interfaces for multiple inheritance
- Encapsulation using access modifiers (private, protected, public)

Example: class Student {

```
    int rollNo;
    String name;

    void display() {
        System.out.println(rollNo + " " + name);
    } }
```

Applications: Web applications, Android apps, Enterprise systems

2. C++:

C++ is a **powerful, hybrid language** supporting both **procedural and object-oriented programming**.

Key Features

- High performance
- Supports **low-level memory management**
- Uses **pointers**
- Supports **operator overloading**

OOP Support

- Classes and objects
- Inheritance (single, multiple, multilevel, hierarchical)
- Polymorphism (function overloading & overriding)
- Encapsulation using access specifiers

Example

```
class Student {
public:
    int rollNo;
    string name;

    void display() {
        cout << rollNo << " " << name;
    }
};
```

Applications: System software, Game development, Embedded systems

3. Python

Python is a **high-level, interpreted language** with simple and readable syntax.

Key Features

- Easy to learn and use
- Dynamically typed
- Large standard library
- Supports multiple paradigms (OOP, procedural, functional)

OOP Support

- Everything is an object
- Supports inheritance and polymorphism
- Encapsulation using naming conventions (_, __)
- Supports abstract classes using abc module

Example

```
class Student:
    def __init__(self, rollNo, name):
        self.rollNo = rollNo
        self.name = name

    def display(self):
        print(self.rollNo, self.name)
```

Applications

- Web development
- Data science & AI
- Automation and scripting

Software Testing Basics

Software testing is the process of **evaluating a software application** to ensure it meets specified requirements and works correctly. It helps in **finding defects, improving quality, and ensuring reliability** of software.

1. Unit Testing:

Unit testing focuses on **individual units or components** of software such as functions, methods, or classes.

Key Points

- Smallest level of testing
- Performed by **developers**

- Done before integration
- Usually automated

- Verify that each unit works correctly
- Detect bugs early in development

Example: Testing a function that calculates total marks:

```
int add(int a, int b) {
   return a + b;
}
```

Tools: JUnit (Java), NUnit (.NET), PyTest (Python)

Advantages:

- Finds errors early
- Improves code quality
- Easy debugging

Disadvantages:

- Time-consuming
- Does not test the complete system

2. Integration Testing:

Integration testing checks the **interaction between integrated modules** to ensure they work together properly.

Objectives of Integration Testing

- Detect **interface defects** between modules
- Verify **data flow** between integrated components
- Ensure combined modules function as expected
- Identify issues not found during unit testing
- It can be automated or manual.

Types of Integration Testing

1. Top-Down Integration Testing

- Testing starts from **top-level modules** and moves down
- **Stubs** are used to simulate lower-level modules

Example: Main menu → Login → Database

2. Bottom-Up Integration Testing

- Testing starts from **lower-level modules** and moves upward

- **Drivers** are used to simulate higher-level modules

Example: Database → Business logic → User interface

3. Big Bang Integration Testing

- All modules are integrated at once
- Entire system is tested together

4. Sandwich (Hybrid) Integration Testing

- Combination of **top-down and bottom-up**
- Used for large systems

Advantages of Integration Testing

- Identifies interface and communication issues
- Ensures modules work together correctly
- Reduces risk of system failure

Disadvantages of Integration Testing

- Requires test drivers and stubs
- Defect isolation can be difficult
- Time-consuming for large projects

Example: In an online banking system:

- Login module
- Account module
- Transaction module

Integration testing checks whether:

- Login correctly retrieves account details
- Transactions update balances properly

3. System Testing:

System testing evaluates the **complete and fully integrated software system**. It is performed **after integration testing and before acceptance testing**.

Objectives of System Testing

- Validate the **entire system behavior**
- Ensure the system meets **user and business requirements**
- Verify both **functional and non-functional** aspects
- Identify defects in real-world usage scenarios

Characteristics of System Testing

- Performed in a **production-like environment**
- Treated as **black-box testing**
- Conducted by **independent testers**
- Covers end-to-end system functionality
- **Example** Testing a full online shopping application including: Login, Product selection, Payment, Order confirmation

Types of System Testing

1. Functional System Testing

- Tests all system features and functions
- Ensures correct input/output behavior

Example: Checking login, search, payment, and logout in an e-commerce application

2. Non-Functional System Testing: Focuses on system performance and quality attributes:

- **Performance Testing** – speed, scalability, load
- **Security Testing** – data protection, authentication
- **Usability Testing** – user-friendliness
- **Reliability Testing** – stability over time
- **Compatibility Testing** – different devices, OS, browsers

Advantages of System Testing

- Validates the complete system
- Detects issues missed in earlier testing stages
- Ensures high software quality

Limitations of System Testing

- Time-consuming
- Requires complete system availability
- Difficult to isolate the root cause of defects

Comparison Table

Testing Level	Focus Area	Performed By	When Done
Unit Testing	Individual modules	Developers	During coding
Integration Testing	Module interaction	Developers/Testers	After unit testing
System Testing	Entire system	Testers	After integration

Test-Driven Development (TDD)

Test-Driven Development (TDD) is a software development approach in which tests are written **before** the actual code. The goal is to ensure correctness, improve design, and reduce defects by continuously validating the software through automated tests.

Core Idea: *"Write a test first, then write code to pass the test."*

TDD Cycle (Red–Green–Refactor)

1. **Red – Write a Test**
   - Write a small test for a new feature or function.
   - The test fails because the functionality is not yet implemented.
2. **Green – Write Code**
   - Write the minimum code required to pass the test.
   - Focus only on making the test succeed.
3. **Refactor – Improve Code**
   - Clean up the code (remove duplication, improve structure).
   - Ensure all tests still pass after refactoring.

This cycle is repeated for every new feature.

Example (Simple Illustration)

Requirement: Add two numbers.

1. **Write Test (Red)**
   - Test that add(2, 3) returns 5.
2. **Write Code (Green)**
   - Implement add(a, b) to return a + b.
3. **Refactor**
   - Improve naming or structure if needed (without changing behavior).

Advantages of TDD

- **✓Early bug detection**
- **✓High code quality**
- **✓Better software design**
- **✓Confidence in code changes (safe refactoring)**
- **✓Acts as documentation through tests**

Disadvantages of TDD

- ✕Initial development may be slower
- ✕Requires strong testing skills
- ✕Not always suitable for UI-heavy or experimental projects

When to Use TDD

- In **agile development**
- For **business logic and core functionality**
- When **maintainability and reliability** are critical

SCNR Government Degree College, Proddatur
Department of Computer Science
II BSc-IV Sem -Software Engineer

UNIT-IV Software Maintenance and Evolution: Software maintenance basics, refactoring techniques Software version control, Code review and inspection, Software evolution and reengineering

Software Maintenance – Basics

Software maintenance is the process of **modifying and updating software after delivery** to correct faults, improve performance, or adapt to a changed environment

Need for Software Maintenance

- User requirements change over time
- Errors (bugs) are discovered after deployment
- Hardware, OS, or technology changes
- Performance and security need improvement
- Software must remain useful for a long period

Types of Software Maintenance:

1. **Corrective Maintenance**
   This type focuses on fixing defects or errors found during software operation.
   Example: Correcting a calculation error in a banking application.
2. **Adaptive Maintenance**
   It modifies the software to adapt to changes in the environment such as hardware or operating systems.
   Example: Updating an application to work on Windows 11.
3. **Perfective Maintenance**
   Enhances performance or adds new features based on user requests.
   Example: Adding a dark mode feature in a mobile app.
4. **Preventive Maintenance**
   Improves code quality to prevent future problems.
   Example: Refactoring complex code to reduce future bugs.

Importance:

- Extends software lifespan
- Improves reliability
- Reduces long-term cost

Refactoring Techniques

Refactoring is the process of **improving the internal structure of software code without changing its external behavior**. It makes the software easier to understand and maintain.

Common Refactoring Techniques

1. **Extract Method**
    - Breaks large methods into smaller ones
    - Improves readability and reuse
2. **Rename Variable / Method**
    - Uses meaningful names
    - Makes code easier to understand
3. **Remove Duplicate Code**
    - Common logic moved to a single method
    - Reduces errors and maintenance cost
4. **Inline Method**
    - Removes unnecessary small methods
    - Simplifies code structure
5. **Replace Magic Numbers with Constants**
    - Improves clarity and reduces errors
6. **Simplify Conditional Expressions**
    - Makes decision logic clearer

Example:

A 200-line function is split into smaller reusable methods, making the code easier to debug and modify.

Benefits of Refactoring

- Cleaner and efficient code
- Easier debugging
- Better software quality
- Lower long-term maintenance cost

Software Version Control

Software version control is a system that **records changes to source code and allows developers to manage multiple versions of software**.

Key Features:

- Tracks changes
- Supports teamwork
- Enables rollback
- Manages branches and merges

Types of Version Control Systems

1. **Local Version Control**
    - Stores versions on a local machine
2. **Centralized Version Control (CVCS)**
    - Single central repository
    - Example: SVN
3. **Distributed Version Control (DVCS)**
    - Each user has a full copy
    - Example: Git

Popular Version Control Systems:

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

Example: Using Git, a developer can create a new branch to add features without affecting the main codebase.

Advantages:

- Prevents data loss
- Improves collaboration
- Ensures code stability

Code Review

Definition:
Code review is a **systematic examination of source code** by developers to find errors, improve quality, and ensure coding standards.

Example:
A teacher or senior student checks your C program before final submission.

Key points:

- Informal or semi-formal process
- Done by peers or senior developers
- Can be manual or tool-assisted
- Focuses on logic errors, readability, performance, and standards

Advantages:

- Detects bugs early
- Improves code quality and maintainability
- Knowledge sharing among team

Disadvantages:

- Time-consuming
- Depends on reviewer’s skill

Code Inspection

Code inspection is a **formal and structured review technique** to detect defects **without executing the program**.

Example:
A group of teachers checks a banking software code using a checklist.

Key points:

- Formal process with defined roles (Author, Moderator, Reviewer, Recorder)
- Uses checklists and documented procedures
- No code execution involved
- Emphasizes defect detection, not fixing

Advantages:

- Highly effective in finding defects
- Reduces testing and maintenance cost
- Improves software reliability

Disadvantages:

- Requires more time and preparation
- Needs trained personnel

Software Evolution

Software evolution refers to the **continuous modification of software to meet changing user needs and environments**.

Reasons:

- New user requirements
- Technological advancements
- Market competition

Types of software evolution:

1. **Corrective** – fixing defects
2. **Adaptive** – adapting to new environment (OS, hardware)
3. **Perfective** – improving performance or usability
4. **Preventive** – improving maintainability

Example: An e-commerce website evolves by adding AI-based product recommendations.

Advantages:

- Extends software life
- Keeps system relevant
- Improves user satisfaction

Disadvantages:

- Increased complexity over time
- Higher maintenance cost

Software Reengineering

Definition:
Software reengineering is the **process of analyzing and modifying existing software** to improve its structure, maintainability, or performance **without changing its functionality**.

Key activities:

- Reverse engineering
- Code restructuring
- Data restructuring
- Documentation improvement

Example:

Migrating a legacy payroll system from COBOL to Java.

Advantages:

- Improves software quality
- Reduces maintenance effort
- Extends system lifespan

Disadvantages:

- Initial cost is high
- Risk of introducing new defects

UNIT-V Advanced Topics in Object-Oriented Software Engineering: Model-driven engineering (MDE), Aspect-oriented programming (AOP), Component-based software engineering (CBSE), Service oriented architecture (SOA), Agile software development and Scrum methodologies.

Model-Driven Engineering (MDE)

Definition: Model-Driven Engineering (MDE) is a software development approach where **models are the primary focus**, and source code is automatically generated from these models.

Basic Concept: Instead of writing code directly, developers first create **UML models** that describe system structure and behavior.

Flow: Requirements → Model Creation → Model Transformation → Code Generation

Types of Models

1. **CIM (Computation Independent Model)**
   - Describes requirements
   - Example: Use Case Diagram
2. **PIM (Platform Independent Model)**
   - Describes system design without platform details
   - Example: Class Diagram
3. **PSM (Platform Specific Model)**
   - Includes technology details (Java, .NET, etc.)

Example: For a **Bank Management System**:

- Create UML class diagram (Account, Customer, Transaction)
- Use an MDE tool to generate Java classes
- Deploy on a specific platform

5. Advantages

- Reduces manual coding
- Faster development
- Fewer errors
- Easy maintenance
- Better documentation

6. Disadvantages

- Requires special tools
- High initial cost
- Not suitable for very small projects

Aspect-Oriented Programming (AOP)

Aspect-Oriented Programming (AOP) is a programming paradigm that separates **cross-cutting concerns** (such as logging, security, and transaction management) from the main business logic of the program.

Need for AOP: In Object-Oriented Programming (OOP), some functions like logging and security are required in many classes.
This leads to:

- Code duplication
- Scattered code
- Difficult maintenance

AOP helps to organize such common functionalities separately.

Key Concepts of AOP

1. **Aspect** – A module that contains cross-cutting concern (e.g., Logging Aspect).
2. **Join Point** – A point during execution (method call, exception handling).
3. **Advice** – Code that runs at a join point (Before, After, Around).
4. **Pointcut** – Expression that selects join points where advice should run.

Working of AOP:

Flow: Business Logic + Aspect → AOP Framework → Final Executable Program

The aspect code is automatically inserted at required places.

Example: In an **Online Banking System**:

- Methods: transfer(), withdraw()

Without AOP: Every method includes login and logging code.

With AOP:

- Business logic → transfer(), deposit()

- Logging aspect automatically runs before and after these methods.
- Tools: **Spring AOP, AspectJ**

Advantages

- Reduces code duplication
- Improves modularity
- Easy maintenance
- Better separation of concerns

Disadvantages

- Complex for beginners
- Debugging is difficult
- Needs framework support

Component-Based Software Engineering (CBSE)

Component-Based Software Engineering (CBSE) is a software development approach in which applications are developed by **assembling reusable, pre-built software components** instead of coding everything from scratch.

Concept of Component: A **component** is a self-contained software module that:

- Performs a specific function
- Has a well-defined interface
- Can be reused in different applications
- Can be replaced independently

Basic Idea of CBSE:

Software is built like assembling **LEGO blocks**.
Each component provides a specific service and interacts through interfaces.

Example: In an **E-commerce System**:

- Use a ready-made **payment gateway component** (PayPal/Razorpay)
- Use an existing **login/authentication module**
- Use a pre-built **shopping cart component**

These components are integrated to build the complete system.

Advantages

- Reduces development time
- Lower development cost
- Improves reliability (components are pre-tested)
- Easy maintenance and upgrade

Disadvantages

- Compatibility issues between components
- Limited customization
- Security risks
- Dependency on third-party vendors

<u>Service-Oriented Architecture (SOA)</u>

Service-Oriented Architecture (SOA) is a software architecture style in which applications are built using **independent, reusable services** that communicate over a network.

Basic Concept

- Application is divided into small **services**.
- Each service performs a specific function.
- Services communicate using standard protocols like **HTTP, SOAP, REST**.
- Services are loosely coupled (independent).

Main Components of SOA

1. **Service Provider** – Creates and maintains the service.
2. **Service Consumer** – Uses the service.
3. **Service Registry** – Stores service details so consumers can find them.

Working of SOA

Service Provider → Registers service in Registry →
Service Consumer → Searches and invokes service →
Communication through messages over network.

Example: In an **Online Banking System**:

- Account Service
- Payment Service
- Loan Service
- Customer Service

Each service works independently but communicates to complete banking operations.

Advantages

- Reusable services
- Easy integration of different systems
- Platform independent
- Scalable and flexible

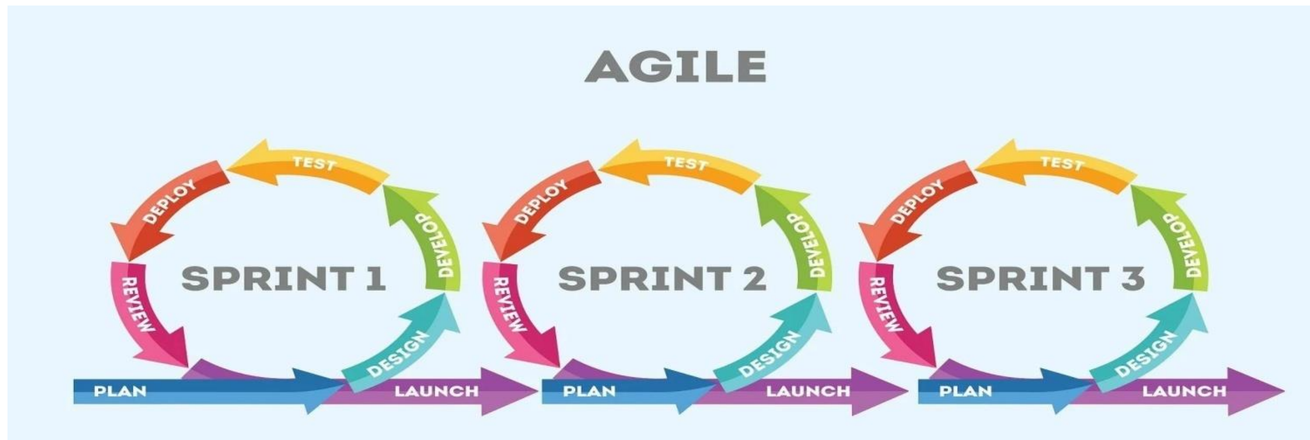
Disadvantages

- Network dependency
- Security concerns
- Complex service management

Agile Software Development

Definition: Agile Software Development is an iterative and incremental approach to software development that focuses on flexibility, customer collaboration, continuous improvement, and quick delivery of working software.

It follows the principles defined in the **Agile Manifesto (2001)**.



Agile Software Development

Agile Manifesto Values: Agile is based on four core values

1. **Individuals and interactions** over processes and tools
2. **Working software** over comprehensive documentation
3. **Customer collaboration** over contract negotiation
4. **Responding to change** over following a plan

Key Characteristics of Agile

- Development in **small iterations (sprints)**
- Continuous customer feedback
- Frequent releases
- Adaptive planning
- Self-organizing teams
- Continuous testing and integration

Agile Process Flow

1. Requirements collected as **User Stories**
2. Stories prioritized in **Product Backlog**
3. Work divided into **Sprints (2–4 weeks)**
4. Development + Testing done within sprint
5. Sprint Review and Retrospective
6. Repeat cycle until product completion

Popular Agile Methodologies:

- **Scrum**
- **Extreme Programming (XP)**
- **Kanban**
- **Lean Development**
- **Crystal Method**

Example: For a Library Management System**:**

- Sprint 1: User login and registration
- Sprint 2: Book search and issue
- Sprint 3: Return and fine calculation

Each module is delivered step-by-step with feedback.

Advantages of Agile

- Faster delivery of software
- Easy to accommodate requirement changes
- High customer satisfaction
- Improved product quality
- Better team collaboration

Disadvantages of Agile

- Less documentation
- Difficult in large, complex projects
- Requires experienced team members
- Scope creep risk

Scrum

Scrum is a popular Agile framework used to manage and control iterative software development.

Key Roles in Scrum

1. **Product Owner** – Defines requirements
2. **Scrum Master** – Manages process
3. **Development Team** – Builds the product

Scrum Process

- Work is divided into **Sprints (2–4 weeks)**
- Daily short meeting called **Daily Scrum**
- At the end of sprint → Review and feedback

Advantages

- Faster delivery
- Flexible to changes
- Better customer satisfaction
- Improved teamwork

Example

In an E-commerce Website:

- Sprint 1: Login & Registration
- Sprint 2: Product listing
- Sprint 3: Cart & Checkout
- Sprint 4: Payment integration