SRI KRISHNA COLLEGE OF ENGINEERING AND TECHNOLOGY

An Autonomous Institution | Approved by AICTE | Affiliated to Anna University | Accredited by NAAC with A++ Grade

Kuniamuthur, Coimbatore – 641008

Phone : (0422)-2678001 (7 Lines) | Email : info@skcet.ac.in | Website : www.skcet.ac.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

22CS604 – OBJECT ORIENTED ANALYSIS AND DESIGN LABORATORY

RECORD NOTE

Submitted by

Name `	**:**
Register No.	**:**
Degree & Branch	**: B.E Computer Science and Engineering**
Class	**: III CSE - A**

SRI KRISHNA COLLEGE OF ENGINEERING AND TECHNOLOGY

An Autonomous Institution | Approved by AICTE | Affiliated to Anna University | Accredited by NAAC with A++ Grade

Kuniamuthur, Coimbatore – 641008

Phone : (0422)-2678001 (7 Lines) | Email : info@skcet.ac.in | Website : www.skcet.ac.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

22CS604– OBJECT ORIENTED ANALYSIS AND DESIGN LABORATORY

Practical record submitted by

Name :	**Register No. :**
Class : III CSE A	**Degree & Branch : B.E CSE**

BONAFIDE CERTIFICATE

This is to certify that this is a bonafide record of work done by ______________________

(Register No.: ________________) during the academic year 2024 – 2025.

Faculty In-charge **Head of the Department**

Submitted for the University practical examination held on _______________________

INTERNAL EXAMINER **EXTERNAL EXAMINER**

INDEX

SRI KRISHNA COLLEGE OF ENGINEERING AND TECHNOLOGY

An Autonomous Institution | Approved by AICTE | Affiliated to Anna University | Accredited by NAAC with A++ Grade

Kuniamuthur, Coimbatore – 641008

Phone : (0422)-2678001 (7 Lines) | Email : info@skcet.ac.in | Website : www.skcet.ac.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

22CS604 – OBJECT ORIENTED ANALYSIS AND DESIGN LABORATORY

Record of laboratory work

EVEN SEMESTER - 2024-2025

CONTINUOUS EVALUATION SHEET

REFERENCES RUBRICS TABLE

Criteria	Range of Marks			
	Excellent	Good	Average	Below Average
Aim & Algorithm (20)	18-20	14-17	10-13	0-9
Coding (30)	27-30	21-26	15-20	0-14
Compilation and Debugging (30)	27-30	21-26	15-20	0-14
Execution and Result (10)	9-10	7-8	5-6	0-4
Documentation (10)	9-10	7-8	5-6	0-4
Overall Marks	90-100	70-85	50-68	0-45

SRI KRISHNA COLLEGE OF ENGINEERING AND TECHNOLOGY

An Autonomous Institution | Approved by AICTE | Affiliated to Anna University | Accredited by NAAC with A++ Grade

Kuniamuthur, Coimbatore – 641008

Phone : (0422)-2678001 (7 Lines) | Email : info@skcet.ac.in | Website : www.skcet.ac.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

22CS604 – OBJECT ORIENTED ANALYSIS AND DESIGN LABORATORY

Record of laboratory work

EVEN SEMESTER - 2024-2025

CONTINUOUS EVALUATION SHEET

Criteria	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6	Exp 7	Exp 8	Exp 9	Exp 10
Aim & Algorithm (20)										
Coding (30)										
Compilation and Debugging (30)										
Execution and Result (10)										
Documentation (10)										
Total										

Faculty Signature

Exercise No: 1

Identify a Software System and document the Software Requirement Specification for the identified system.

Date:

Pre-Lab Exercise:

1. What is the significance of documenting requirements in software engineering?

   - **Clarity and Understanding** – Ensures all stakeholders have a clear and shared vision of the system’s functionalities and objectives.
   - **Better Project Management** – Helps in planning, resource allocation, and tracking progress against defined goals.
   - **Reduced Miscommunication** – Serves as a reference to avoid misunderstandings between developers, clients, and project managers.
   - **Facilitates Testing and Validation** – Provides a basis for creating test cases to verify that the software meets user needs and expectations

2. List the key sections of an SRS document.

   - **Introduction** – Overview, purpose, scope, and definitions related to the project.
   - **Overall Description** – System context, user characteristics, and key constraints.
   - **Functional Requirements** – Detailed list of features and functionalities the system must provide.
   - **Non-Functional Requirements** – Performance, security, scalability, and other system qualities.
   - **External Interfaces** – Describes interactions with hardware, software, or other systems.
   - **System Models** – Diagrams and representations, such as UML models, to visualize system structure.
   - **Appendices & References** – Additional supporting information and references for further reading.

3. How can ambiguous requirements impact a project?

- **Misinterpretation** – Different stakeholders may interpret requirements differently, leading to inconsistencies in development.
- **Scope Creep** – Vague requirements can result in frequent changes, increasing project time and cost.
- **Poor Quality Assurance** – Testers may struggle to create proper test cases, leading to defective software.
- **Increased Development Effort** – Developers may need multiple revisions due to unclear expectations, delaying project completion.

4. Describe at least two methods used to gather software requirements.

- **Interviews** – Direct discussions with stakeholders to understand their needs, expectations, and pain points. Helps clarify requirements but can be time-consuming.
- **Surveys & Questionnaires** – Collects structured feedback from a large group of users, making it useful for identifying common trends and preferences.
- **Observation** – Analysts observe users interacting with existing systems to identify pain points and improvement areas. Provides real-world insights but may miss undocumented needs.
- **Prototyping** – Developing a mock version of the system to gather user feedback and refine requirements before full development.

In-Lab: Case Study and Problem Statement

Case Study:

You are tasked with developing a Task and Project Management System for an organization. The system should:

- Allow employees to create, assign, and track tasks.
- Provide project progress visualization and reporting.
- Enable notifications for overdue tasks or project milestones.
- Support multiple user roles (e.g., Manager, Team Member).

Steps to Perform:

1. **Stakeholder Identification:**
    - Identify key stakeholders such as managers, team members, and administrators.
2. **Requirement Gathering:**
    - Use brainstorming or interviews to elicit system requirements.
    - Document both functional and non-functional requirements.
3. **Draft the SRS Document:**
    - Write the SRS document using the following structure:
        - Introduction: Define the purpose, scope, and stakeholders.
        - System Overview: Describe the key functionalities and constraints.
        - Functional Requirements: Detail user actions like task creation and project tracking.
        - Non-Functional Requirements: Include system uptime, scalability, and security.
4. **Review and Refine:**
    - Conduct a peer review to identify and address gaps or ambiguities.

EX.NO:01 DATE:	BANKING MANAGEMENT SYSTEM

AIM

To create an automated system to manage bank operations, including account management, transaction processing, and report generation.

PROBLEM STATEMENT

The **Banking Management System** is designed to automate essential banking processes, ensuring efficiency, accuracy, and security in handling transactions and customer records. The system enables online registration, fund transfers, loan management, and real-time balance inquiries. By integrating customer and administrator functionalities into a single platform, the system minimizes manual efforts and streamlines banking operations.

(I) SOFTWARE REQUIREMENT SPECIFICATION

INTRODUCTION

The **Banking Management System (BMS)** provides an interface between customers and the bank, focusing on improving the speed, security, and ease of banking operations.

PURPOSE

The BMS offers a faster, more user-friendly banking experience. It minimizes manual intervention and helps banks scale operations efficiently as customer numbers grow.

SCOPE

- Online interface for customers to manage accounts and transactions.
- Secure, real-time updates for users and administrators.
- Integration with payment systems for seamless processing.
- Automation of reporting and auditing processes.

DEFINITIONS, ACRONYMS AND THE ABBREVIATIONS

- **Admin**: Bank staff managing system operations.
- **Customer**: An individual holding an account.
- **BMS**: Bank Management System.
- **HTTP**: Hypertext Transfer Protocol.
- **TCP/IP**: Transmission Control Protocol/Internet Protocol.

REFERENCES

- IEEE Software Requirement Specification format.
- Banking Compliance Guidelines.

TECHNOLOGIES TO BE USED

- **Frontend**: HTML, CSS, JavaScript.
- **Backend**: Java (Spring Framework).
- **Database**: MySQL.

TOOLS TO BE USED

- Eclipse IDE (Integrated Development Environment)
- Apache Tomcat Server.

OVERVIEW

The **SRS** includes two sections: overall description and specific requirements.

1. **Overall Description**: Explains the components and interconnections of the system.

2. **Specific Requirements**: Details the roles and functions of actors.

(II) OVERALL DESCRIPTION

PRODUCT PERSPECTIVE

The BMS is an integrated solution for both customers and bank staff. It enables secure transactions, efficient operations, and reliable data handling.

SOFTWARE INTERFACE

- **Frontend**: HTML and JSP for customer and admin interfaces.
- **Backend**: Java for business logic.
- **Database**: MySQL for storing account and transaction data.

HARDWARE INTERFACE

Server requirements: Minimum 16GB RAM, 500GB SSD.

SYSTEM FUNCTIONS

- Secure account registration and management.
- Processing transactions in real time.
- Notifications via SMS or email.
- Generation of detailed reports.

USER CHARACTERISTICS

- **Customer**: Interacts via a simple interface for account and transaction management.
- **Admin**: Manages system operations and monitors reports.

CONSTRAINTS

- Requires stable internet connectivity.
- Must comply with PCI DSS for data security.

ASSUMPTIONS AND DEPENDENCIES

- Users need basic computer knowledge.
- External payment gateway integration is required.

(III) USECASE DIAGRAM

The **Banking Management System** use cases are:

1. Login
2. Account Registration
3. Balance Inquiry
4. Fund Transfer
5. Loan Application
6. Approve Loan
7. Transaction History

ACTORS INVOLVED:

1. Customer
2. Administrator
3. External Payment Gateway

USE-CASE NAME

1. LOGIN

Description: Customers and administrators log in to access their respective functionalities.

2. ACCOUNT REGISTRATION

Description: A new customer registers by entering personal and account details.

3. BALANCE INQUIRY

Description: Customers check their current account balance in real-time.

4. FUND TRANSFER

Description: Customers transfer funds between accounts securely.

5. LOAN APPLICATION

Description: Customers apply for loans, which are reviewed by the administrator.

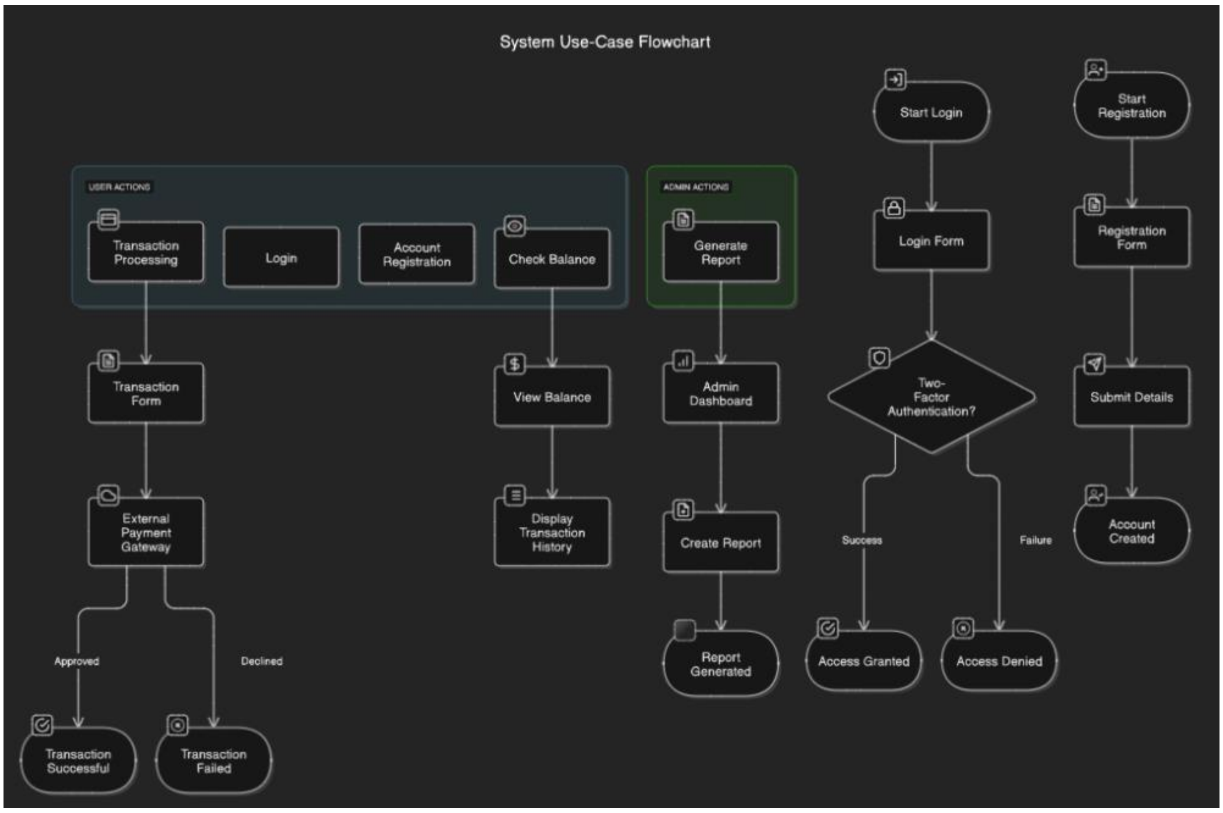
6. APPROVE LOAN

Description: Administrators review and approve/reject loan applications.

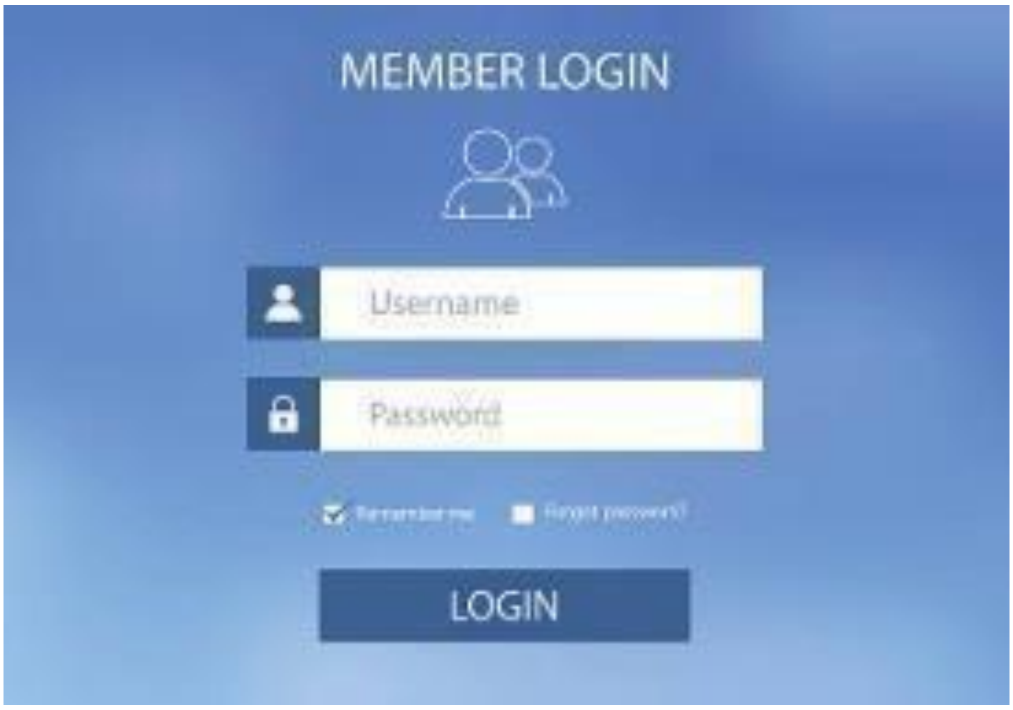
7. TRANSACTION HISTORY

Description: Customers view a detailed history of their past transactions.

USE CASE DIAGRAM



(IV) IMPLEMENTATION OF USER INTERFACE LAYER



RESULT

The Bank Management System simplifies banking operations, enhancing efficiency and user satisfaction. Its modules are ready for deployment and implementation.

Post-Lab: Inference and Analysis Questions

Inference Questions:

1. What challenges did you face while gathering requirements for the system?

   - **Unclear or Vague Expectations** – Stakeholders may provide ambiguous or incomplete requirements, leading to misunderstandings.
   - **Changing Requirements** – Business needs may evolve during development, causing scope creep and delays.
   - **Conflicting Stakeholder Interests** – Different users may have contradicting priorities, making it difficult to finalize requirements.
   - **Difficulty in Prioritization** – Determining which features are most critical can be challenging, especially with limited resources.

2. How did you ensure the requirements were comprehensive and unambiguous?

   - **Stakeholder Collaboration** – Conducted detailed interviews, surveys, and workshops to clarify expectations and gather diverse perspectives.
   - **Clear and Specific Documentation** – Used precise language, structured templates, and SRS documentation to avoid vagueness.
   - **Prototyping and Feedback** – Created mockups or prototypes to validate requirements with stakeholders before finalizing them.
   - **Requirement Validation** – Reviewed and refined requirements with developers, testers, and business analysts to ensure feasibility and completeness.

3. What additional functionalities could enhance the task and project management system?

- **Automated Task Scheduling** – Uses AI to prioritize tasks and set deadlines efficiently.
- **Time Tracking & Reports** – Logs work hours and generates productivity insights.
- **Integration with Third-Party Tools** – Connects with Google Calendar, Slack, and Trello for seamless workflow.
- **Custom Notifications & Reminders** – Sends alerts for deadlines, updates, and team mentions.

Analysis Questions:

1. Compare your SRS document with a peer's. Identify similarities and differences in structure and content.

- **Standard Structure** – Both follow the IEEE SRS format, including sections like Functional & Non-Functional Requirements.
- **Use of Diagrams** – Both include UML diagrams to visualize system processes and interactions
- **Level of Detail** – One may provide more detailed use cases, while the other focuses on high-level descriptions.
- **Requirement Prioritization** – Some documents rank features by priority, while others list them without categorization.

2. How would you prioritize the requirements if the organization has a limited budget?

- **Core Functionalities First** – Focus on essential features like task management, user authentication, and project tracking.
- **High-Impact, Low-Cost Features** – Prioritize features that add value but require minimal resources, such as notifications and basic reporting.
- **Scalability & Future Expansion** – Design the system so additional features (e.g., AI automation, integrations) can be added later.
- **User-Centric Approach** – Prioritize features most critical to end users, ensuring usability and efficiency within budget constraints.

3. If the organization scales to multiple branches, how would the system's requirements change?

- **Multi-Branch User Management** – Implement role-based access control (RBAC) to manage users across different branches.
- **Centralized Data Management** – Ensure secure cloud-based storage for real-time data synchronization across locations.
- **Enhanced Performance & Scalability** – Optimize the system for higher traffic, load balancing, and distributed processing.
- **Branch-Specific Customization** – Allow configurable workflows and reporting tailored to individual branch needs.

Result:

By completing this experiment, I have:

1. Gained hands-on experience in creating an SRS document.
2. Developed skills to analyze and document software requirements effectively.
3. Understood the significance of clear and detailed requirements for successful project execution.

Exercise No: 2

Sketch the class diagrams to identify and describe key concepts and their relationships.

Date:

Pre-Lab Exercise:

1. What is the purpose of a class diagram in software design?

   - **Visualizing System Structure** – Represents classes, attributes, methods, and relationships, providing a clear system blueprint.
   - **Defining Object Relationships** – Shows associations, inheritance, and dependencies between classes for better organization.
   - **Supporting Code Implementation** – Acts as a guide for developers, ensuring consistency in system architecture.
   - **Enhancing Maintainability** – Helps in understanding, modifying, and scaling the system efficiently.

2. List the key components of a class diagram and explain each briefly.

   - **Classes** – Represent blueprints for objects, containing attributes (variables) and methods (functions).
   - **Attributes** – Define the characteristics or properties of a class (e.g., name, age in a User class).
   - **Methods** – Specify the functions or behaviors that a class can perform (e.g., login(), register() in a User class).
   - **Relationships** – Show how classes interact with each other, including association, inheritance, composition, and aggregation.

3. How do relationships in class diagrams reflect real-world interactions?

- **Association** – Represents a general relationship (e.g., a student enrolls in a course → `Student` ↔ `Course`).
- **Inheritance (Generalization)** – Shows a hierarchy where one class derives from another (e.g., Car is a type of Vehicle → `Car` ⬅ `Vehicle`).
- **Composition** – Indicates a strong "whole-part" relationship where one object cannot exist independently (e.g., Engine is part of a Car → `Car` ◆ `Engine`).
- **Aggregation** – Represents a "whole-part" relationship where parts can exist independently (e.g., **Library has Books**, but books can exist elsewhere → `Library` ◇ `Book`).

4. Describe the role of abstraction in identifying key classes for a system.

- **Simplifies Complexity** – Focuses on essential attributes and behaviors, ignoring unnecessary details to create clear class structures.
- **Identifies Core Entities** – Helps define key objects in a system (e.g., User, Order, Product in an e-commerce system).
- **Promotes Reusability** – Groups common features into base classes, allowing subclasses to inherit them (e.g., Vehicle as a parent for Car and Bike).
- **Enhances Maintainability** – Ensures a flexible and scalable design by structuring classes logically, reducing duplication and complexity.

In-Lab: Case Study and Problem Statement

Case Study:

You are tasked with creating a reservation and booking system for a variety of clients. The system should:

- Allow customers to search for available services (e.g., hotels, flights, event bookings).
- Enable customers to reserve and pay for services.
- Provide administrators with tools to manage bookings and customer data.

Steps to Perform:

1. **Identify Key Concepts:**
    - Determine the main entities in the system, such as Customer, Reservation, Service, and Payment.
2. **Define Attributes and Operations:**
    - For each class, list attributes (e.g., Customer Name, Reservation ID) and operations (e.g., makeReservation(), cancelReservation()).
3. **Establish Relationships:**
    - Define associations between classes. For example, a Customer can have multiple Reservations, and a Reservation is linked to a specific Service.
4. **Draw the Class Diagram:**
    - Use UML modeling tools to create a class diagram that includes:
        - Classes with attributes and operations.
        - Relationships such as associations, generalizations, and aggregations.
5. **Validate the Diagram:**
    - Ensure that the class diagram accurately represents the system’s requirements and functionality.

Ex No: 2	**Class Diagram and Association**
Date:	

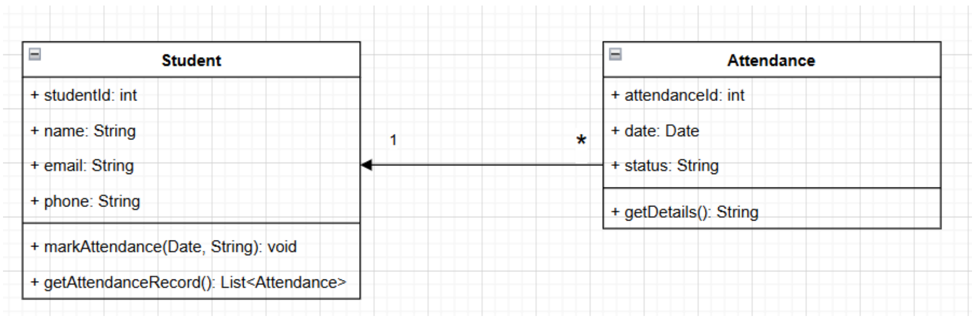
AIM:

To demonstrate a class Diagram with different types of associations or relations.

Association:

An association diagram, commonly referred to as a class diagram, is a visual tool used in software engineering and system design to illustrate the connections among different classes within an object-oriented system.

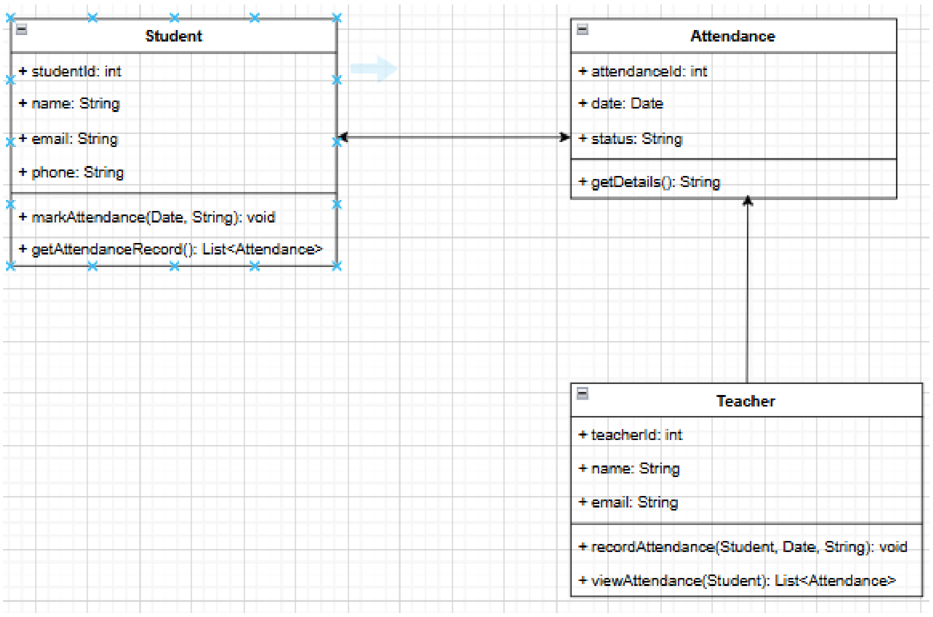
Ex: In the Student Attendance Management System, the Student class and Attendance class are associated together.



Directed Association:

Directed association relationships are associations with navigability in a single direction. They signify that control moves from one classifier to another, such as from a teacher to the attendance records.

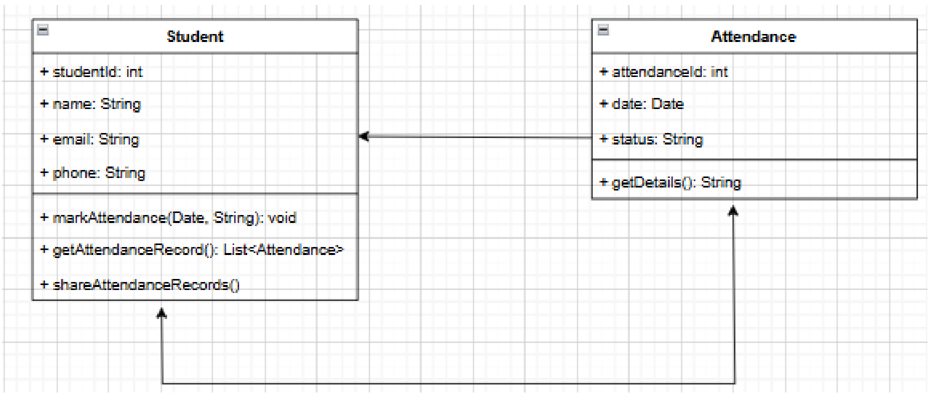
Ex: In the Student Attendance Management System, the Teacher class is directed toward the Attendance class.



Reflexive Association:

A reflexive association, or self-association, emerges when a class establishes a connection with itself. It is useful to represent relationships where an object engages with other objects of the same kind.

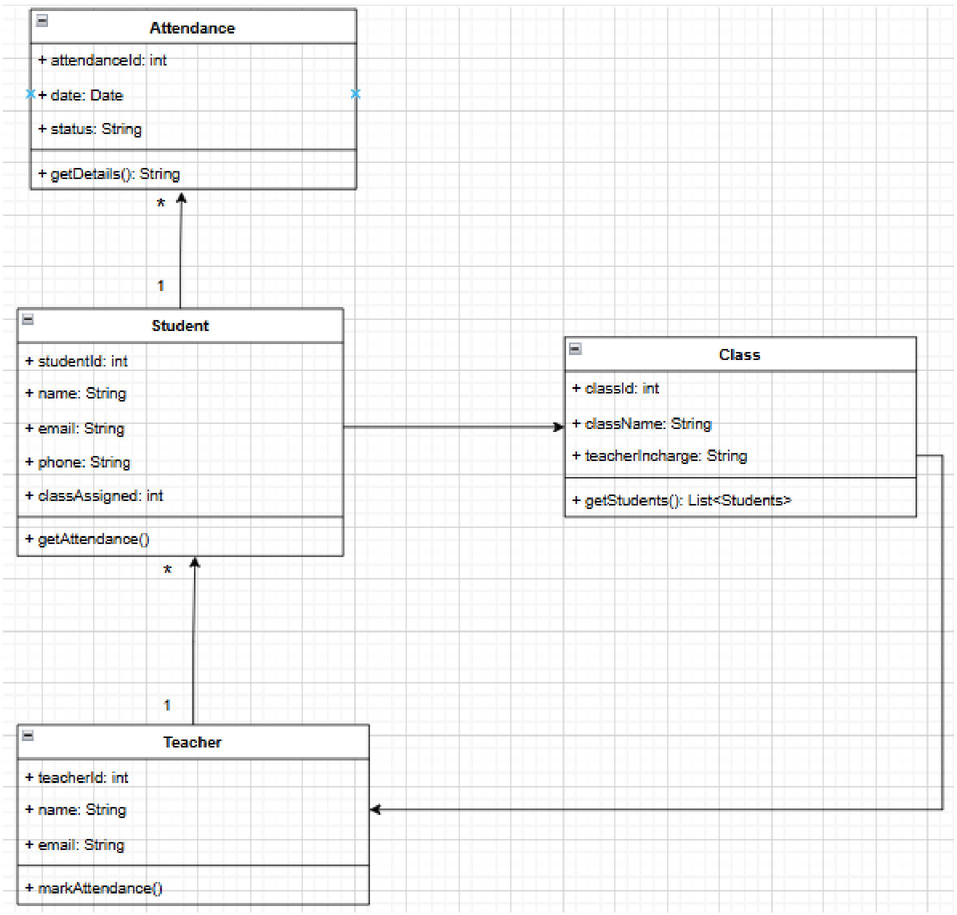
Ex: In the Student Attendance Management System, the Student class can interact with other Student objects, such as sharing group attendance records.



Multiplicity:

Multiplicity defines the numerical extent of a relationship connecting two classes. It shows how many instances of one class relate to instances of another.

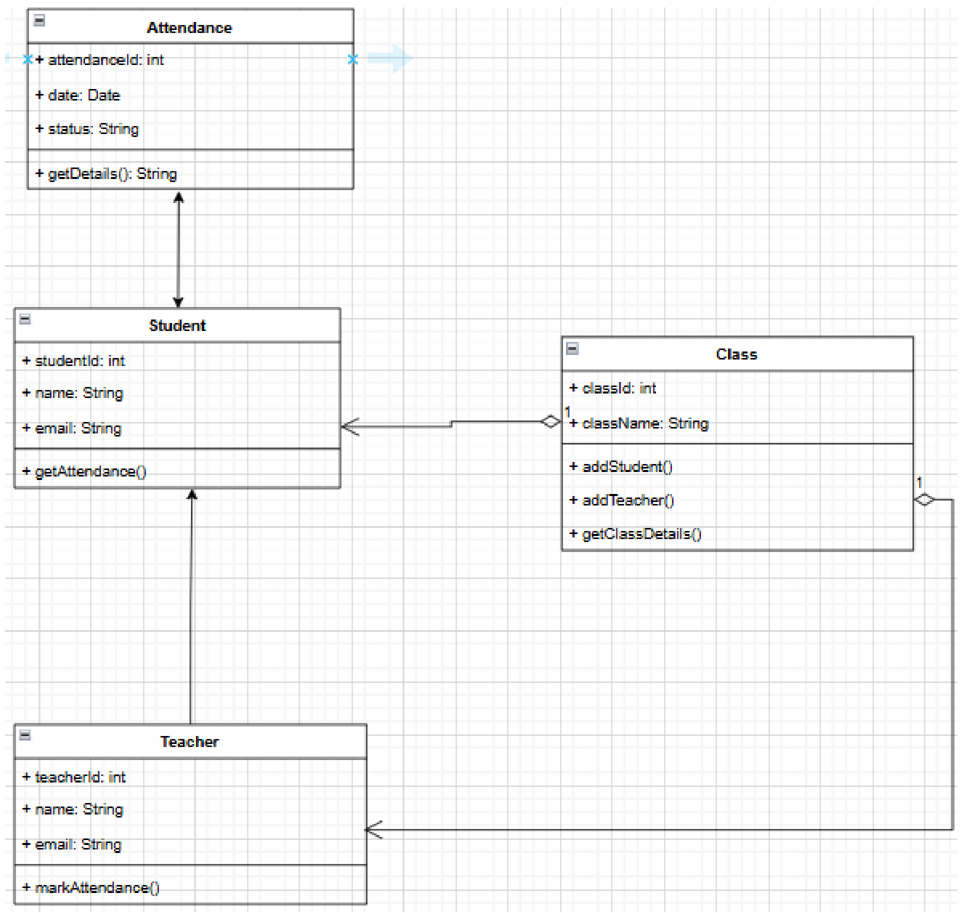
Ex: One Teacher can mark attendance for multiple Students, and each Student belongs to one specific Class.



Aggregation:

Aggregation symbolizes a connection where a part-whole relationship exists, but the child element is independent of the parent.

Ex: In the Student Attendance Management System, the Class aggregates Students and Teachers.



Composition:

Composition illustrates a whole-part relationship where the parts cannot exist independently of the whole.

Ex: In the Student Attendance Management System, Attendance Records cannot exist without the Class they are associated with.\

Generalization:

Generalization occurs when one entity is the parent, and another is the child, inheriting its properties and methods.

Ex: The Person class acts as a parent for the Teacher and Student classes, inheriting common attributes like Name and ID.

Classes, Methods, and Attributes:

1. Student:

A student is an individual associated with the attendance system.

- **Methods:**
  - markAttendance(): Allows the student to mark their attendance.
  - viewAttendance(): Allows the student to check their attendance.
- **Attributes:**
  - Name, ID, Class, Section.

2. Teacher:

A teacher is responsible for managing attendance records.

- **Methods:**
  - markAttendanceForClass(): Marks attendance for all students in a class.
  - viewClassAttendance(): Views attendance records for a class.
- **Attributes:**
  - Name, ID, Department.

3.Attendance:

Represents the attendance records for students.

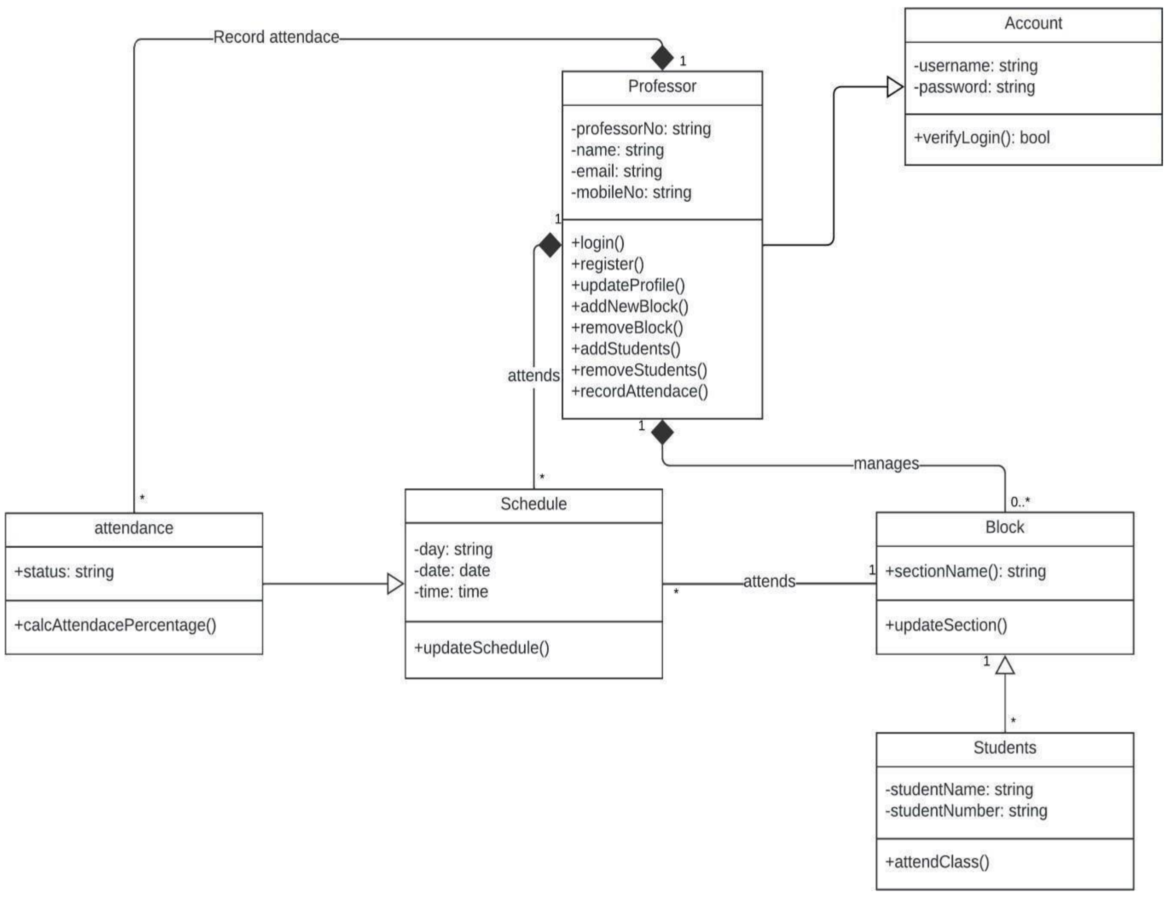
- **Methods:**
  - addAttendance(): Adds a new attendance record.
  - getAttendanceByDate(): Retrieves attendance for a specific date.
- **Attributes:**
  - Date, Student ID, Status (Present/Absent).

3.Class:

Represents a group of students and their associated teacher.

- **Methods:**
  - assignTeacher(): Assigns a teacher to the class.
  - getStudents(): Retrieves the list of students in the class.
- **Attributes:**
  - Class Name, Teacher, Students.

Class Diagram



RESULT:

Hence, the Student Attendance Management System class diagram and relationships between the classes were successfully demonstrated

Post-Lab: Inference and Analysis Questions

Inference Questions:

1. What were the key challenges in identifying the classes and their relationships for the reservation system?

    - **Defining Core Classes** – Determining essential entities like Customer, Reservation, Payment, and Room while avoiding unnecessary complexity.
    - **Establishing Relationships** – Deciding how classes interact, such as one-to-many between `Customer` and `Reservation` or composition between `Reservation` and `Payment`.
    - **Handling Dynamic Data** – Managing real-time updates (e.g., room availability, cancellations) while maintaining consistency.
    - **Integrating Business Rules** – Ensuring the system supports cancellation policies, discounts, and overbooking prevention logically within class structures.

2. How did you ensure that the class diagram captures all necessary system functionalities?

    - **Requirement Analysis** – Mapped functional and non-functional requirements to identify essential classes (e.g., Customer, Reservation, Payment).
    - **Use Case Alignment** – Ensured that all user interactions, such as booking, cancellation, and payment, were represented in class relationships.
    - **Defining Relationships Clearly** – Established associations (e.g., one-to-many between Customer and Reservation) to reflect real-world interactions.
    - **Stakeholder Review & Iteration** – Regularly reviewed the diagram with developers and business analysts to refine missing functionalities.

3. What additional features could be incorporated into the reservation system to improve user experience?

   - **Real-Time Availability Updates** – Display live room or seat availability to prevent double bookings.
   - **Automated Notifications** – Send booking confirmations, reminders, and cancellation alerts via email or SMS.
   - **Flexible Payment Options** – Support multiple payment methods, including credit cards, digital wallets, and pay-later options.
   - **User Reviews & Ratings** – Allow customers to rate their experience and provide feedback for better service improvement.

Analysis Questions:

1. Compare your class diagram with a peer's. What are the differences in structure and representation?

   - **Level of Detail** – One diagram may have more attributes and methods per class, while another keeps it high-level.
   - **Relationships Representation** – Differences in using association, composition, or aggregation for class connections (e.g., Reservation and Payment).
   - **Use of Abstract Classes** – Some diagrams may introduce abstract classes for reusability, while others define all entities explicitly.
   - **Additional Functionalities** – One diagram might include extra features like notifications or loyalty programs, making it more comprehensive.

2. How would the class diagram change if the system needed to support multiple payment methods (e.g., credit card, digital wallets)?

    - **Introduce an Abstract Payment Class** – Create a **base class** (Payment) with common attributes like amount and transactionID.
    - **Create Subclasses for Payment Methods** – Add specific classes like CreditCardPayment, DigitalWalletPayment, and BankTransferPayment that inherit from Payment.
    - **Modify Reservation Association** – Instead of linking directly to a single Payment class, associate Reservation with **multiple payment types** via polymorphism.
    - **Enhance Payment Processing** – Include additional attributes (e.g., cardNumber for CreditCardPayment, walletProvider for DigitalWalletPayment) to support different payment details.

3. Discuss how inheritance could be applied to represent different types of services in the reservation system.

    - **Create a Base Service Class** – Define a parent class Service with shared attributes like serviceName, price, and availability.
    - **Define Specialized Subclasses** – Extend Service into specific services like RoomBooking, CarRental, and EventReservation, inheriting common properties.
    - **Use Method Overriding** – Implement methods like calculateCost() in Service and **override them** in subclasses to customize pricing logic.
    - **Enhance System Scalability** – New services (e.g., SpaService, TourPackage) can be **easily added** by extending Service, improving flexibility.

Result:

By completing this experiment, I have:

1. Developed the ability to create detailed class diagrams for complex systems.
2. Gained insight into identifying key concepts and relationships in a software system.
3. Understood how UML diagrams aid in visualizing and designing software architecture.

Exercise No: 3

Identify Use Cases and develop the Use Case model.

Date:

Pre-Lab Exercise:

1. What is the purpose of a Use Case Diagram in software design?

   - **Visualizing System Interactions** – Represents how users (actors) interact with the system's functionalities.
   - **Defining System Scope** – Clearly outlines what the system should do by identifying key use cases.
   - **Improving Requirement Clarity** – Helps stakeholders understand system behavior without technical details.
   - **Supporting Development & Testing** – Provides a foundation for designing system architecture and test cases.

2. List the main components of a Use Case Diagram and explain each briefly.

   - **Actors** – Represent users or external systems interacting with the system.
   - **Use Cases** – Define system functionalities or actions performed by actors.
   - **Relationships** – Show interactions between actors and use cases (e.g., association, include, extend).
   - **System Boundary** – Defines the scope of the system and what functionalities it covers.

3. How do you identify actors in a software system?

- **Analyze User Roles** – Identify **who interacts** with the system (e.g., Customer, Admin, Support Staff).
- **Consider External Systems** – Include **third-party services or APIs** that communicate with the system (e.g., Payment Gateway).
- **Define Responsibilities** – Determine what **each actor needs to accomplish**, such as Customer booking a reservation.
- **Group Similar Roles** – Combine users with **identical system interactions** to avoid redundancy (e.g., Guest and Registered User as Customer).

4. Explain the difference between include and extend relationships in a Use Case Diagram.

- **Include Relationship** – Represents mandatory reuse of a use case by another. The included use case always executes as part of the base use case.

  Example: Process Payment includes Validate Payment Details (validation is always required).
- **Extend Relationship** – Represents optional or conditional behavior that occurs only in specific scenarios.

  Example: Make Reservation extends Apply Discount (discount is applied only if applicable).

In-Lab: Case Study and Problem Statement

Case Study:

You are tasked with developing an inventory management system to streamline operations in a warehouse. The system should:

- Allow administrators to add, update, and delete inventory items.
- Enable warehouse staff to view inventory details and stock levels.
- Generate low-stock alerts and reports for administrators.
- Support the creation of purchase orders for replenishment.

Steps to Perform:

1. **Identify Actors:**
    - Determine the main actors interacting with the system, such as Administrator, Warehouse Staff, and Supplier.
2. **List Use Cases:**
    - Identify functionalities such as Add Inventory, Update Inventory, View Inventory, Generate Reports, and Create Purchase Orders.
3. **Define Relationships:**
    - Establish connections between actors and use cases.
    - Use include relationships for shared functionalities (e.g., Validate User in Add Inventory and Update Inventory).
4. **Draw the Use Case Diagram:**
    - Use UML modeling tools to create a Use Case Diagram showing all actors, use cases, and their relationships.
5. **Validate the Diagram:**
    - Ensure the Use Case Diagram reflects all functional requirements of the inventory management system.

EX.NO:03 **DATE:**	**IDENTIFY USECASES AND DEVELOP THE USECASE MODEL**

AIM

To identify the UseCases and develop the UseCase Model for Payroll Management System.

PROCEDURE

Step 1: Identify the System and Boundaries

Define the Payroll Management System for which the use case modeling is to be developed.

Step 2: Identify Actors

Identify all the actors (users, external systems, etc.) interacting with the system.

Step 3: List Use Cases

Identify the main tasks or interactions that the actors have with the system. Each task becomes a use case.

Step 4: Describe Use Cases

Write a brief description for each use case explaining its purpose and steps involved.

Step 5: Create Use Case Diagram

Visualize the relationships between actors and use cases using a use case diagram.

USECASE MODEL

A use case model is a visual representation and description of the functional requirements of the Payroll Management System from the perspective of its users, known as actors. It is used to capture and communicate interactions between users and the system.

- **Employee** – Can view salary details and request leave.
- **HR Manager** – Manages employee records, salaries, and approvals.
- **Admin** – Responsible for payroll calculations and system maintenance.
- **Bank System** – Handles salary disbursements.

Include Relationship:

- Payroll processing includes salary calculation.
- Salary disbursement includes tax deduction.

Extends Relationship:

- Leave approval extends from employee request.
- Payroll correction extends from salary disbursement.

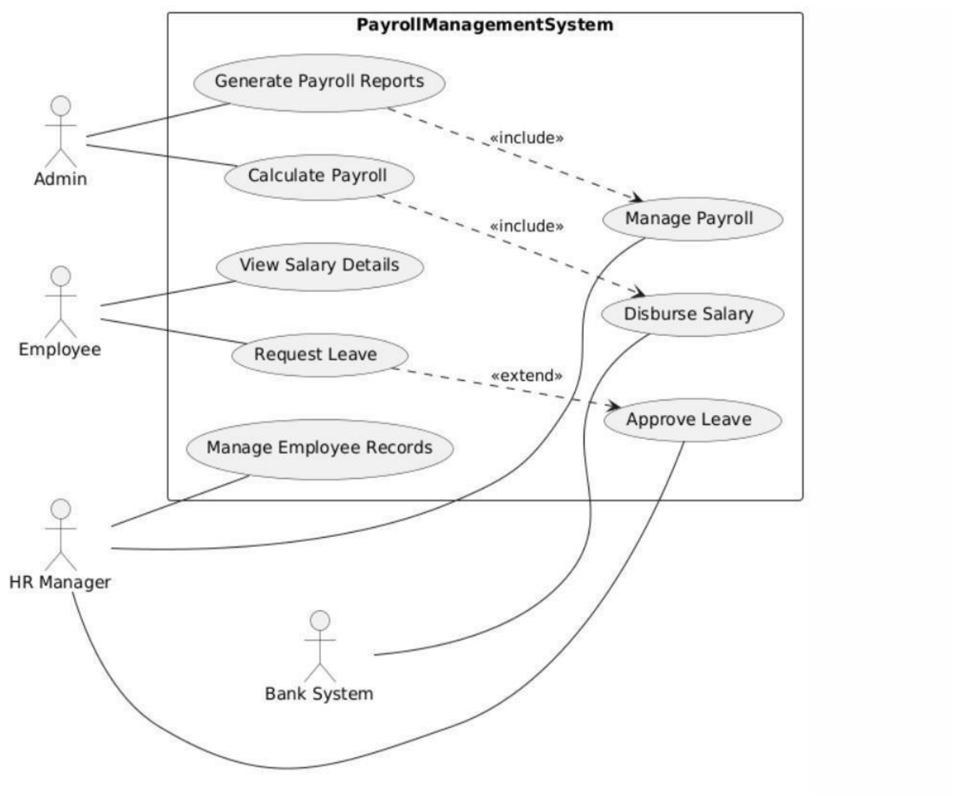
3. USECASE SCENARIO FOR PAYROLL MANAGEMENT SYSTEM

1. **Employee Payroll Processing:**
    - The system computes employee salaries based on working hours, overtime, and deductions.
    - Tax and provident fund deductions are applied before final salary computation.
2. **Salary Disbursement:**
    - After processing, salaries are transferred to employees' bank accounts.
    - The system generates salary slips and notifies employees.
3. **Leave and Attendance Management:**
    - Employees can request leave through the system.
    - HR managers review and approve/reject requests.
4. **Employee Record Management:**
    - HR managers can add, update, or remove employee records.
    - The system maintains employment history and payroll details.

- The system calculates tax deductions and generates reports.
- Employees can view their tax details.

5. **Payroll Reports Generation:**
    - The system generates reports on payroll, taxes, and expenses.
    - Admins can export reports for auditing purposes

USECASE DIAGRAM FOR PAYROLL MANAGEMENT SYSTEM



RESULT

Thus, the Use Cases are identified and the UseCase Model is developed for the Payroll Management .

Post-Lab: Inference and Analysis Questions

Inference Questions:

1. What were the key challenges in identifying the use cases for the inventory management system?

    - **Defining System Scope** – Determining which functionalities to include (e.g., inventory tracking, reporting, supplier management).
    - **Identifying Actors Clearly** – Distinguishing between internal users (e.g., warehouse staff, managers) and external systems (e.g., suppliers, sales platforms).
    - **Handling Complex Workflows** – Managing conditional processes like low-stock alerts, restocking, and multi-location inventory updates.
    - **Avoiding Redundant Use Cases** – Ensuring that similar actions (e.g., Add Stock and Update Stock) are not duplicated unnecessarily.

2. How did you ensure that all functional requirements were captured in the Use Case Diagram?

    - **Requirement Mapping** – Matched each functional requirement to a corresponding use case (e.g., "manage stock" → Update Inventory).
    - **Stakeholder Review** – Verified the diagram with end-users, managers, and developers to ensure completeness.
    - **Use Case Refinement** – Used include and extend relationships to cover common and optional functionalities.
    - **Traceability Check** – Ensured every listed requirement was represented in at least one use case to prevent omissions.

3. What additional use cases could be included to enhance the system's functionality?

- **Generate Inventory Reports** – Allows users to view stock trends, shortages, and restocking needs.
- **Set Low-Stock Alerts –** Automatically notifies managers when stock levels drop below a threshold.
- **Supplier Management** – Tracks supplier details, order history, and automated restocking.
- **User Access Control** – Restricts functionalities based on user roles (e.g., admin, warehouse staff).

Analysis Questions:

1. Compare your Use Case Diagram with a peer's. What are the differences in representation and scope?

- **Level of Detail** – One diagram may include more specific use cases (e.g., Generate Sales Report vs. a general View Reports).
- **Use of Relationships** – Differences in Include and Extend usage, where one diagram explicitly reuses common use cases while another keeps them separate.
- **Actor Identification** – Some diagrams may group similar users under one actor (e.g., Employee), while others differentiate roles (Manager, Warehouse Staff).
- **System Scope** – Variations in covered functionalities, such as one diagram focusing on inventory tracking, while another includes supplier management.

2. How would the system requirements change if the inventory system had to support multiple warehouses?

   - **Warehouse-Specific Inventory Tracking** – Each warehouse must have separate stock levels, requiring a Warehouse entity in the system.
   - **Inter-Warehouse Transfers** – Introduce functionality to track and manage stock movement between warehouses.
   - **Role-Based Access Control** – Users should only access data for their assigned warehouse, requiring permission settings.
   - **Centralized Reporting** – Generate individual and consolidated reports for stock levels, sales, and restocking needs across multiple warehouses.

3. Discuss how the use of include and extend relationships improves the clarity of the Use Case Diagram.

   - **Avoids Redundancy (Include)** – Common functionality (e.g., Validate Payment) is reused across multiple use cases (e.g., Process Online Payment, Process Cash Payment), making the diagram cleaner and more efficient.
   - **Clarifies Optional Behavior (Extend**) – Represents conditional actions (e.g., Apply Discount extends Make Purchase only if a coupon is used), improving readability.
   - **Enhances Modularity** – Breaking down large use cases into smaller, reusable ones ensures better organization and easier updates.
   - **Improves Maintainability** – Any changes to an included use case (e.g., Generate Invoice) automatically apply wherever it is used, reducing duplication and maintenance effort.

Result:

By completing this experiment, I have:

1. Developed the ability to identify and model use cases for complex systems.
2. Gained insight into the importance of clearly defining system interactions and boundaries.
3. Understood how UML Use Case Diagrams facilitate communication between stakeholders and developers.

Exercise No: 4

Identify the conceptual classes and develop a Domain Model with Class Diagrams.

Date:

Pre-Lab Exercise:

1. What is the purpose of a Domain Model in object-oriented design?

   - **Represents Real-World Entities** – Captures key objects, their attributes, and relationships within the system.
   - **Defines Object Interactions** – Establishes how different objects communicate and interact with each other.
   - **Serves as a Blueprint** – Provides a foundation for database design, class diagrams, and system architecture.
   - **Enhances Communication** – Acts as a common reference for stakeholders, developers, and designers

2. List the key components of a Class Diagram.

   - **Classes** – Represent objects in the system, defined by their name, attributes, and methods.
   - **Relationships** – Show associations between classes, including association, aggregation, composition, and inheritance.
   - **Multiplicity** – Defines how many instances of one class relate to another (e.g., 1..*, 0..1, * etc.).
   - **Visibility Modifiers** – Control access to class members using public (+), private (-), protected (#), and package (~).

3. Explain how conceptual classes are identified from user requirements.

- **Identify Nouns in the Requirements** – Extract key nouns from the description that represent entities (e.g., Customer, Order).
- **Check for Relevance and Uniqueness** – Ensure the selected nouns are meaningful and not duplicates or implementation-specific.
- **Define Attributes and Responsibilities** – Determine what data each class holds and what actions it performs.
- **Establish Relationships Between Classes** – Identify how classes are related (e.g., Customer places Order).

4. Differentiate between association, aggregation, and composition in a Class Diagram.

- **Association** – A general relationship where one class interacts with another (e.g., Student enrolls in Course).
- **Aggregation** – A weak "has-a" relationship, where parts can exist independently (e.g., Library has Books).
- **Composition** – A strong "has-a" relationship, where parts cannot exist without the whole (e.g., Car has an Engine).
- **UML Symbols** – Association (solid line), Aggregation (hollow diamond), Composition (solid diamond).

In-Lab: Case Study and Problem Statement

Case Study:

You are tasked with designing an online marketplace platform. The system should:

- Allow customers to browse products, add items to a cart, and complete purchases.
- Enable vendors to list products and manage inventory.
- Provide administrators with tools to oversee transactions and manage user accounts.

Steps to Perform:

1. **Identify Conceptual Classes:**
    - Determine key classes such as Customer, Vendor, Product, Cart, Order, and Administrator.
2. **Define Attributes and Operations:**
    - For example, the Product class may include attributes like ProductID, Name, Price, and Stock.
    - Define operations like addProduct(), removeProduct(), and updateStock().
3. **Establish Relationships:**
    - Identify relationships between classes, such as:
        - A Customer can place multiple Orders.
        - A Vendor manages multiple Products.
4. **Draw the Domain Model:**
    - Use UML modeling tools to create a Domain Model that includes all conceptual classes and their relationships.
5. **Validate the Model:**
    - Ensure the Domain Model reflects all user requirements and system functionalities.

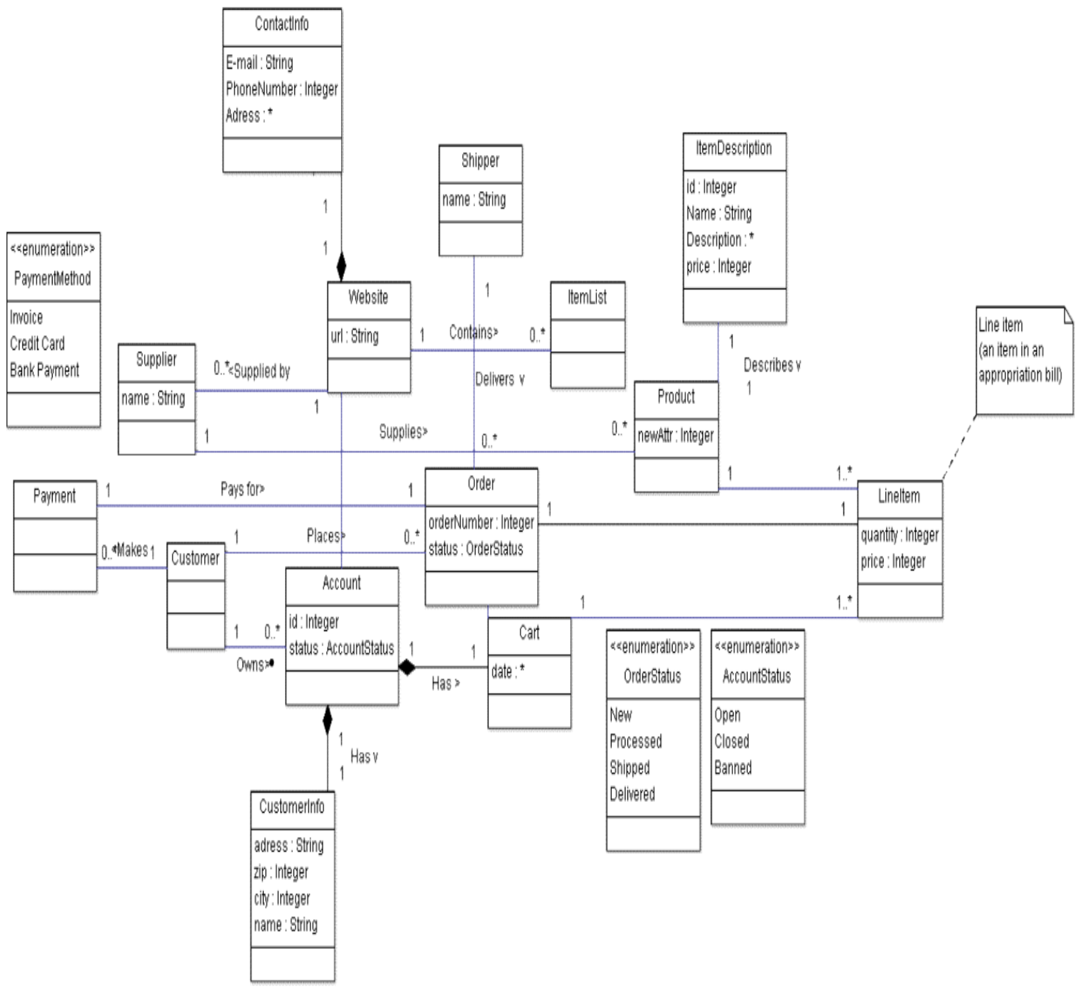
EX.NO:04 **DATE:**	**DOMAIN MODEL WITH DIAGRAM**

OBJECTIVE:

To design a new online marketplace platform and identify the conceptual classes of the platform and create a comprehensive domain model with class diagram.

Problem Statement:

Each online store has an admin who controls one or many items in the website. Customers can place orders and make payments. Each order has shipping information.



Result:

Thus the UML domain model with class diagram was created for online market place application

Post-Lab: Inference and Analysis Questions

Inference Questions:

1. What challenges did you face while identifying conceptual classes for the marketplace platform?

   - **Handling Ambiguous Requirements** – User requirements may be unclear or vague, making it difficult to identify relevant conceptual classes.
   - **Distinguishing Between Classes and Attributes** – Some concepts (e.g., `Price`) may seem like separate classes but are better represented as attributes of another class (`Product`).
   - **Managing Complex Relationships** – Defining associations between entities like `Seller`, `Buyer`, `Product`, and `Order` can be challenging.
   - **Avoiding Redundant or Overlapping Classes** – Ensuring that similar concepts (e.g., `User` vs. `Customer` vs. `Seller`) are correctly structured without unnecessary duplication.

2. How did you ensure that the Domain Model accurately represents the system requirements?

   - **Requirement Analysis** – Carefully reviewed user requirements and extracted key entities, relationships, and behaviors.
   - **Stakeholder Validation** – Discussed the domain model with stakeholders (users, business analysts, developers) to confirm accuracy.
   - **Iterative Refinement** – Continuously updated the model based on feedback and new insights from system analysis.
   - **Mapping to Use Cases** – Ensured that all use cases (e.g., buying a product, placing an order) were properly supported by the domain model.

3. What additional classes or relationships could enhance the functionality of the platform?

- **Review & Rating Class** – Allows buyers to leave feedback and ratings for products and sellers, improving trust.
- **Payment Class** – Manages different payment methods, transactions, and order processing.
- **Shipping Class** – Tracks delivery status, shipping options, and estimated arrival times.
- **Wishlist Class** – Enables users to save favorite products for future purchase.

Analysis Questions:

1. Compare your Domain Model with a peer's. Identify differences in structure and representation.

- **Class Representation** – Differences in the number of conceptual classes (e.g., one model may have separate Buyer and Seller classes, while another combines them into User).
- **Relationships** – Variations in associations (e.g., one model may use **composition** for Order and Payment, while another uses **aggregation**).
- **Attributes and Methods** – Differences in attribute placement (e.g., Price as an attribute of Product vs. a separate Pricing class).
- **Level of Abstraction** – Some models may be more detailed with additional classes (e.g., Cart, Wishlist), while others keep it minimal.

2. How would the Domain Model change if the platform introduced a subscription-based feature?

   - **New Subscription Class** – Stores subscription details like plan type, price, and duration.
   - **Relationship Between User and Subscription** – A user can have one or multiple active subscriptions.
   - **Payment Integration** – Extend the Payment class to handle recurring payments and billing cycles.
   - **Access Control Mechanism** – Modify existing classes (Product, Services) to restrict access based on subscription level.

3. Discuss the importance of clearly defining relationships in a Domain Model.

   - **Ensures Data Integrity** – Prevents inconsistencies by correctly structuring associations (e.g., an Order must always have a Customer).
   - **Improves System Understanding** – Helps developers, designers, and stakeholders easily grasp how different entities interact.
   - **Supports Accurate Implementation** – Guides database design and object-oriented programming, ensuring smooth functionality.
   - **Enhances Maintainability and Scalability** – Well-defined relationships make it easier to modify or extend the system without breaking existing functionality.

Result:

By completing this experiment, I have:

1. Developed the ability to identify and model conceptual classes for complex systems.
2. Gained insight into the importance of creating comprehensive Domain Models.
3. Understood how Class Diagrams facilitate system design and stakeholder communication.

Exercise No: 5

Using the identified scenarios, find the interaction between objects and represent them using UML Sequence and Collaboration Diagrams.

Date:

Pre-Lab Questions:

1. What is the purpose of a Sequence Diagram in software design?

   - **Visualizes Object Interactions** – Shows how objects communicate over time by exchanging messages.
   - **Defines System Workflow** – Represents the sequence of operations for a specific use case or functionality.
   - **Helps in Debugging & Optimization** – Identifies inefficiencies, redundant calls, or missing interactions in the system.
   - **Aids in Implementation** – Serves as a blueprint for developers to understand message flow and method calls.

2. How do Collaboration Diagrams complement Sequence Diagrams?

   - **Focus on Object Relationships** – Collaboration diagrams emphasize structural relationships between objects, while sequence diagrams focus on message flow over time.
   - **Highlight Message Exchange** – Both diagrams show object interactions, but collaboration diagrams visually depict how objects are linked and interact.
   - **Aid in System Understanding** – Sequence diagrams are better for understanding the order of execution, while collaboration diagrams help in understanding object dependencies.
   - **Provide a Complete View** – Together, they offer a detailed behavioral and structural representation of the system’s functionality.

3. List the components of a Sequence Diagram.

- **Actors & Objects** – Represent users or system components interacting in the sequence (e.g., User, Order, Payment).
- **Lifelines** – Dashed vertical lines showing an object’s existence during the interaction.
- **Messages** – Arrows representing communication between objects, including **synchronous, asynchronous, and return messages**.
- **Activation Bars** – Rectangles on lifelines indicating when an object is active and performing a task.

4. Explain how to identify objects and interactions for a given scenario.

- **Analyze the Scenario Description** – Identify key nouns (potential objects) and verbs (possible interactions or messages).
- **Identify Primary Objects** – Determine the main entities involved in the interaction (e.g., User, Order, Payment for an e-commerce checkout).
- **Define Interactions & Messages** – Establish how objects communicate (e.g., User sends a placeOrder() request to Order).
- **Map the Sequence of Events** – Arrange interactions in the correct order using a Sequence Diagram, ensuring logical flow.

In-Lab: Case Study and Problem Statement

Case Study:

You are tasked with building an online banking application. The system should:

- Allow customers to log in securely and view their account details.
- Enable fund transfers between accounts.
- Provide transaction history and balance details.
- Notify users of successful transactions.

Steps to Perform:

1. **Identify Objects:**
   - Determine key objects such as Customer, Account, Transaction, Bank, and Notification.
2. **Define Interactions:**
   - Example interactions include:
     - Customer logs in using credentials.
     - System verifies credentials and retrieves account details.
     - Customer initiates a fund transfer, and the system updates the accounts.
     - Notification is sent to confirm the transaction.
3. **Draw Sequence Diagram:**
   - Represent the order of messages exchanged between objects for scenarios like login, fund transfer, and viewing transaction history.
4. **Draw Collaboration Diagram:**
   - Depict the relationships between objects and the flow of messages for the same scenarios.

5. **Validate the Diagrams:**
    - Ensure the diagrams align with the functional requirements of the banking application.

EX NO.5 **DATE:**	**Using the identified scenarios, find the interaction between objects and represent them using UML Sequence and Collaboration Diagrams**

AIM:

To draw the diagrams [use case, activity, sequence, collaboration, class] for the E-ticketing system.

HARDWARE REQUIREMENTS:

Intel Pentium Processor 3

SOFTWARE REQUIREMENTS:

Rational rose / Visual Basic

PROJECT DESCRIPTION:

This software is designed for supporting the computerized e-ticketing. This is widely used by the passenger for reserving the tickets for their travel. This E-ticketing is organized by the central system. The information is provided from the railway reservation system

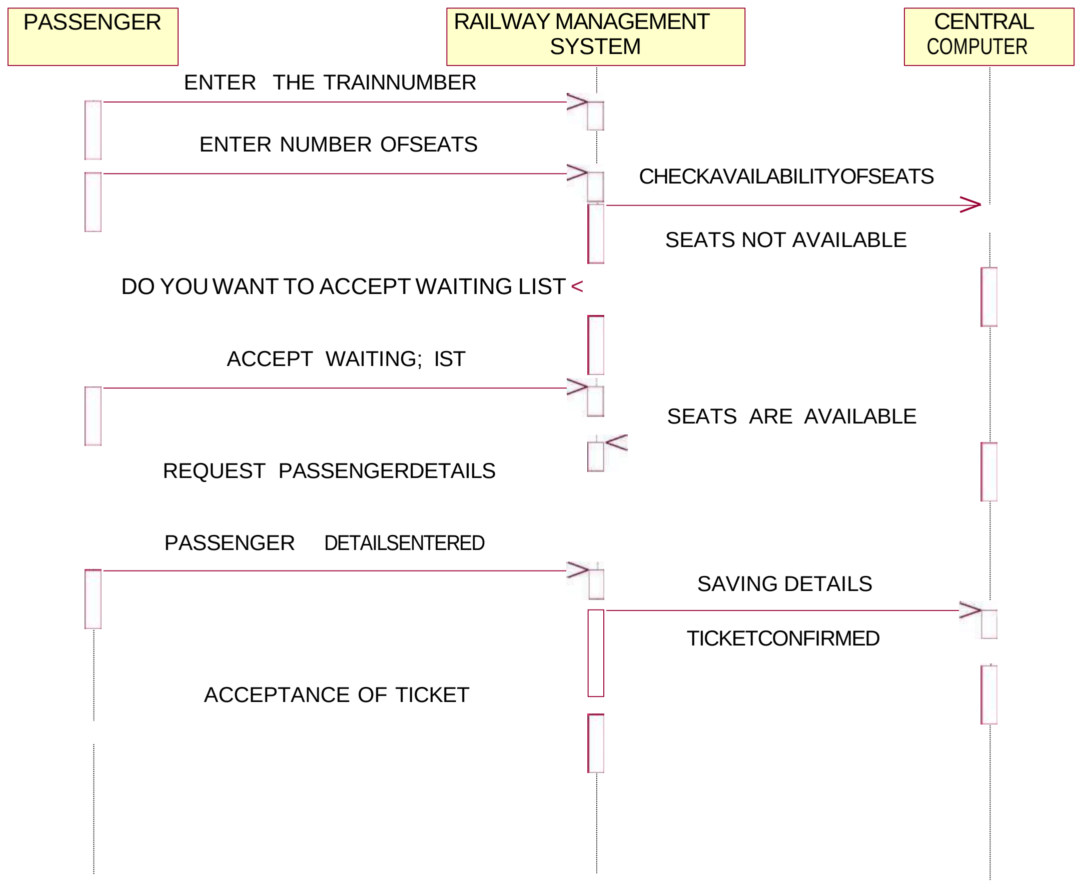
SEQUENCE DIAGRAM:

This diagram consists of the objects, messages and return messages.
Object: Passenger, Railway reservation system, Central computer

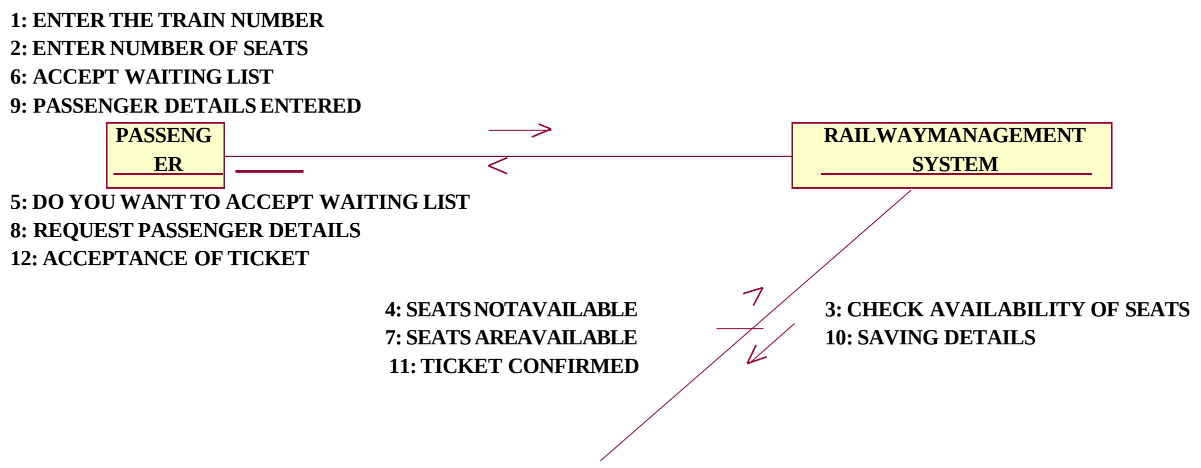
COLLABORATION DIAGRAM:

This diagram contains the objects and actors. This will be obtained by the completion of the sequence diagram and pressing the F5 key.

SEQUENCE DIAGRAM:



COLLABORATION DIAGRAM:



RESULT:

Thus the diagrams [sequence, collaboration] for the E-ticketing has been designed, executed and output is verified.

Post-Lab: Inference and Analysis Questions

Inference Questions:

1. What were the key challenges in identifying objects and their interactions for the banking application?

   - **Complex Business Rules** – Defining objects like Account, Transaction, and Customer while ensuring compliance with banking regulations.
   - **Handling Multiple Interactions** – Managing diverse operations like deposits, withdrawals, transfers, and account management across different objects.
   - **Security & Access Control** – Identifying interactions while ensuring secure authentication and authorization between users and the system.
   - **Real-Time Processing & Dependencies** – Ensuring accurate transaction sequencing, avoiding inconsistencies in funds transfer, and managing system dependencies.

2. How did you ensure that the Sequence Diagram accurately represents the flow of operations?

   - **Requirement Validation** – Cross-checked the sequence diagram against system requirements and use cases to ensure correctness.
   - **Step-by-Step Verification** – Ensured that each interaction follows a logical order, from user actions to system responses.
   - **Stakeholder Review** – Collaborated with developers, analysts, and stakeholders to confirm that the diagram reflects real-world banking processes.
   - **Testing & Refinement** – Simulated transactions (e.g., deposits, withdrawals) and adjusted the diagram to align with actual system behavior.

3. What additional scenarios could be modeled to improve the system's documentation?

   - **Account Registration & Login** – Models user authentication, including multi-factor authentication (MFA) and password recovery.
   - **Fund Transfers Between Accounts** – Captures interactions for internal and external bank transfers, including verification and processing.
   - **Loan Application & Approval** – Represents the workflow from loan request submission to approval, including credit checks.
   - **Fraud Detection & Security Alerts** – Illustrates how the system detects suspicious transactions and notifies users or administrators.

Analysis Questions:

1. Compare your Sequence Diagram with a peer's. What differences do you observe in the structure or detail?

   - **Level of Detail** – One diagram may include more granular interactions (e.g., database queries, validation steps), while another keeps it high-level.
   - **Message Representation** – Differences in using synchronous vs. asynchronous messages, affecting the execution flow.
   - **Object Interactions** – Variations in how objects communicate (e.g., some may use a BankingSystem class as an intermediary, while others have direct interactions).
   - **Error Handling & Alternate Flows** – Some diagrams may include failure scenarios (e.g., insufficient funds), while others focus only on the happy path.

2. How would the diagrams change if the system introduced two-factor authentication?

- **Extra Steps in Sequence Diagram** – New steps like sending and verifying an OTP would be added before granting access.
- **New Components in Collaboration Diagram** – Entities like an Authentication Server or Notification Service would be included.
- **Longer Interaction Flow** – The Sequence Diagram would now show extra interactions for OTP generation and validation.
- **More Object Connections** – The Collaboration Diagram would reflect new relationships between the User, Authentication System, and Notification Service.

3. Discuss the advantages of using both Sequence and Collaboration Diagrams for interaction modeling.

- **Improved System Design and Debugging** – Sequence diagrams highlight potential timing issues and process sequences, while collaboration diagrams reveal object dependencies, helping developers identify design flaws early.
- **Better Communication Among Teams** – Developers, designers, and business analysts benefit from both views, as sequence diagrams assist in defining processes, and collaboration diagrams support structural analysis, leading to better team coordination.
- **Comprehensive Understanding** – Sequence diagrams show the time-based flow of interactions, while collaboration diagrams highlight object relationships.
- **Different Perspectives** – Sequence diagrams focus on execution order, whereas collaboration diagrams emphasize structural connections.

Result:

By completing this experiment, I have:

1. Developed the ability to model object interactions using Sequence and Collaboration Diagrams.
2. Understood the importance of accurately representing system interactions in design documentation.
3. Gained insight into the use of UML diagrams to communicate design details to stakeholders.

Exercise No: 6

Sketch the Activity and State Diagrams for an identified application.

Date:

Pre-Lab Questions:

1. What is the purpose of an Activity Diagram in software design?

   - **Visualizing Workflow** – It graphically represents the flow of activities, decisions, and parallel processes in a system.
   - **Understanding Business Logic** – Helps stakeholders analyze and improve the logic behind a process or system functionality.
   - **Identifying Bottlenecks** – Highlights inefficiencies, loops, and decision points that may affect performance.
   - **Aiding Development and Testing** – Guides developers in implementation and helps testers understand expected behavior.

2. How does a State Diagram complement an Activity Diagram?

   - **Focus on Object States** – While an Activity Diagram shows the flow of actions, a State Diagram focuses on how an object transitions between different states.
   - **Better Understanding of System Behavior** – The State Diagram helps model dynamic changes in an object’s lifecycle, while the Activity Diagram visualizes process execution.
   - **Clarifying Event-Driven Changes** – State diagrams show how external events trigger state changes, complementing activity diagrams that represent process flow.
   - **Improved System Design** – Together, they provide a complete picture of both process flow (Activity Diagram) and state-dependent behavior (State Diagram) for better system modeling.

3. List the components of an Activity Diagram and a State Diagram.

   - **Flow Representation Elements** – Includes the Initial Node (starting point), Final Node (ending point), and Activity/Action Nodes (specific tasks or operations).
   - **Decision and Control Elements** – Includes Decision Nodes (for conditional branching) and flow connectors that manage process transition.
   - **State Representation Elements** – Includes the Initial State (starting state), Final State (ending state), and States (different conditions an object can be in).
   - **Transition Elements** – Includes Transitions (arrows showing movement between states based on events) and triggers that cause state changes.

4. Explain how to identify states and transitions for a given scenario.

   - **Identify Key Objects** – Determine the main entity that undergoes state changes (e.g., "Order" in an e-commerce system).
   - **List Possible States** – Define the different conditions the object can have (e.g., "Pending," "Shipped," "Delivered").
   - **Determine Transitions** – Identify events that trigger state changes (e.g., "Payment Received" moves an order from "Pending" to "Processing").
   - **Map the Flow** – Represent states and transitions in a **State Diagram** to visualize how the object progresses through different conditions.

In-Lab: Case Study and Problem Statement

Case Study:

You are tasked with designing an e-commerce website. The system should:

- Allow users to browse products by category or search for specific items.
- Enable users to add products to a shopping cart and proceed to checkout.
- Support a secure payment gateway for completing purchases.
- Provide an order summary and confirmation to users upon successful payment.

Steps to Perform:

1. **Identify Key Activities:**
    - Activities include browsing products, adding items to the cart, checking out, making a payment, and confirming the order.
2. **Define Workflow:**
    - Map the sequence of actions users perform to complete a purchase.
    - Identify decision points (e.g., payment success/failure).
3. **Draw Activity Diagram:**
    - Represent the flow of actions and decisions in the system.
    - Include swimlanes to separate user actions from system actions.
4. **Identify States and Transitions:**
    - States include Browsing, Adding to Cart, Checkout, Payment Processing, and Order Confirmation.
    - Define events that trigger state changes (e.g., clicking "Add to Cart," completing payment).
5. **Draw State Diagram:**
    - Represent the states of the system and transitions based on user actions and system responses.

6. **Validate the Diagrams:**
    - Ensure the diagrams align with the functional requirements of the e-commerce system.

EX NO.6 **DATE:**	**Draw the Activity and State Diagrams for given application.** **a) Stock MaintenanceSystem.** **b) Passport Automationsystem.**

PASSPORT AUTOMATION SYSTEM

AIM:

To draw the diagrams [activity, state] for the Passport automation system.

HARDWARE REQUIREMENTS:

Intel Pentium Processor 3

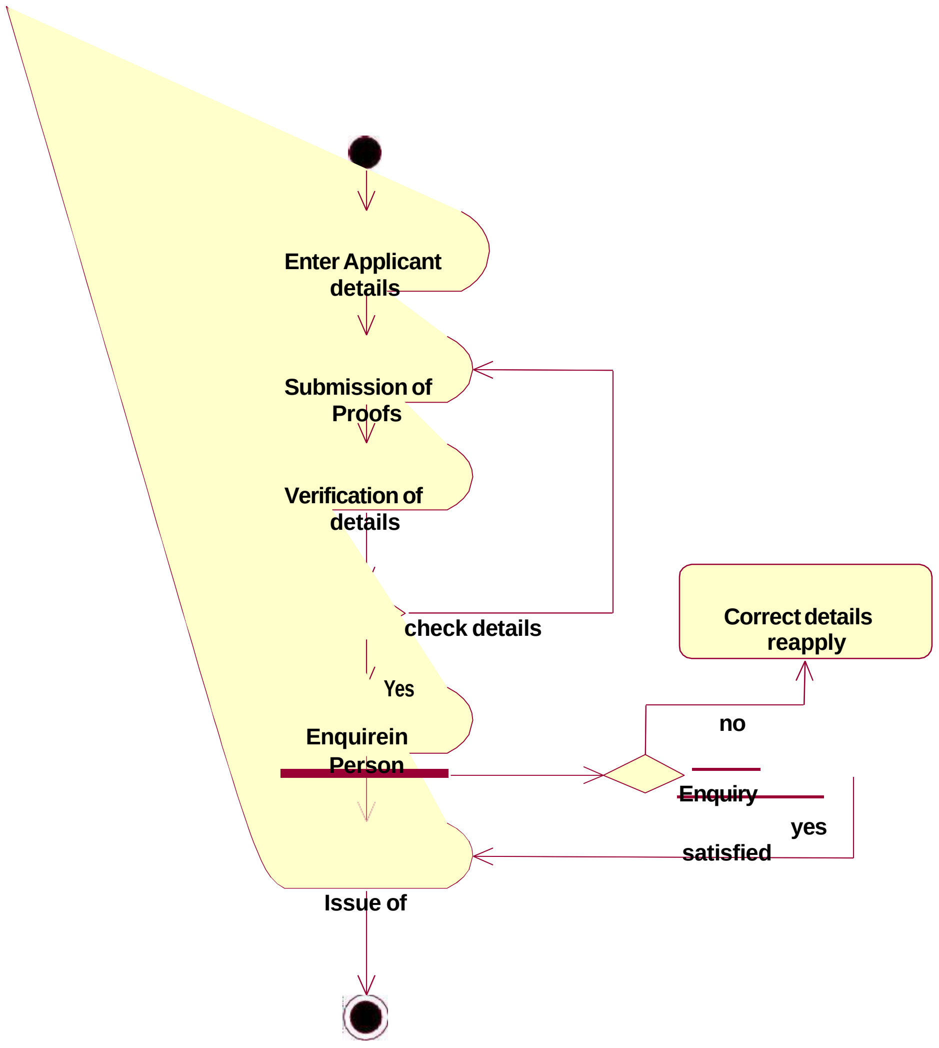
SOFTWARE REQUIREMENTS:

AGROUML

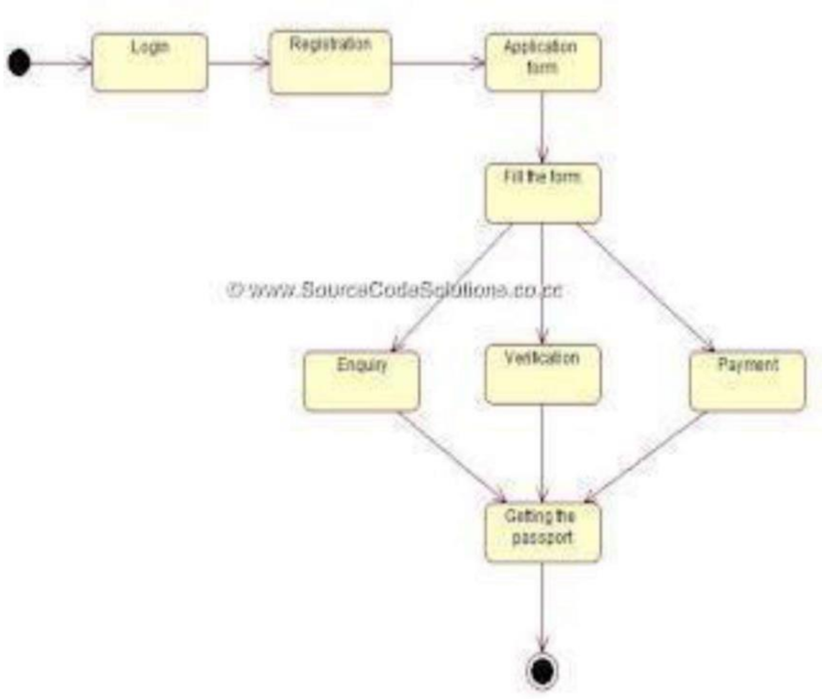
PROJECT DESCRIPTION:

This software is designed for the verification of the passport details of the applicant bythe central computer. The details regarding the passport will be provided to the central computer and the computer will verify the details of applicant and provide approval to the office. Then the passport will issue from the office to theapplicant.

ACTIVITY DIAGRAM:



STATE DIAGRAM:



RESULT:

Thus the diagrams [Use case, class, activity, sequence, collaboration] for the passport automation system has been designed, executed and output is verified.

Post-Lab: Inference and Analysis Questions

Inference Questions:

1. What challenges did you face while identifying the activities and states for the e-commerce system?

    - **Complexity of User Interactions** – Defining clear activities for different user roles (e.g., customers, admins, delivery personnel) can be challenging, as each has distinct workflows.
    - **Handling Multiple States** – An order, for example, can have various states like "Pending," "Processed," "Shipped," and "Delivered." Identifying all possible states and their valid transitions requires careful analysis.
    - **Conditional and Exception Handling** – Managing different scenarios, such as order cancellations, payment failures, or returns, adds complexity to defining states and transitions.
    - **Integration with External Systems** – Activities like payment processing, stock updates, and shipment tracking depend on third-party systems, making it difficult to define clear states and transitions.

2. How did you ensure that the Activity Diagram captures all possible workflows?

    - **Identifying All User Roles** – Considered different users (e.g., customers, admins, and delivery personnel) to ensure all relevant activities are included.
    - **Mapping All Possible Scenarios** – Included normal workflows (e.g., order placement, payment, and delivery) as well as exceptions (e.g., order cancellation, payment failure, and returns).
    - **Using Decision Nodes for Branching** – Added decision points for different outcomes (e.g., successful vs. failed payment) to cover all workflow possibilities.
    - **Validating with Stakeholders** – Reviewed the diagram with developers, business analysts, and end-users to ensure completeness and correctness.

3. What additional scenarios could be modeled to improve the system's documentation?

- **Order Modification** – Allowing users to update or cancel orders before shipment, capturing states like "Order Updated" or "Order Cancelled."
- **User Account Management** – Modeling account registration, login, password reset, and account deactivation for better security and user experience.
- **Customer Support Workflow** – Handling queries, refund
- s, and dispute resolution, including states like "Support Requested" and "Issue Resolved."
- **Inventory Management** – Tracking stock levels, restocking alerts, and supplier interactions to prevent order failures due to insufficient stock.

Analysis Questions:

1. Compare your Activity Diagram with a peer’s. What differences do you observe in representation or detail?

- **Level of Detail** – Some diagrams may include more granular steps, such as email notifications after an order is placed, while others may generalize these actions.
- **Decision Node Usage** – One diagram might use explicit decision nodes (e.g., "Payment Successful?" → Yes/No), while another may imply conditions within activity nodes.
- **Parallel vs. Sequential Flow** – Some diagrams may represent parallel processes (e.g., order confirmation and inventory update happening simultaneously), whereas others may show them sequentially.
- **Exception Handling** – Differences in handling failures (e.g., payment failure, stock unavailability) where one diagram may include alternative flows, while another may focus only on the standard process.

2. How would the diagrams change if the system introduced user accounts with order history?

    - **New Authentication Steps** – The Activity Diagram would include login/signup activities before placing an order, ensuring users have an account.
    - **Order History Retrieval** – A new workflow for viewing past orders would be added, with actions like "View Order History" and "Reorder" in both Activity and State Diagrams.
    - **State Updates for Users** – The State Diagram would now track user-related states such as "Logged In," "Browsing," "Order Placed," and "Order History Viewed."
    - **Persistent Data Management** – The system would need additional interactions in both diagrams to fetch and update user order history from the databases.

3. Discuss how Activity and State Diagrams improve the clarity of the system's design.

    - **Visualizing Process Flow** – Activity Diagrams provide a step-by-step representation of workflows, making it easier to understand how users interact with the system.
    - **Defining Object Behavior** – State Diagrams clearly show how objects (e.g., orders, users) transition between different states based on events, improving system comprehension.
    - **Identifying Decision Points** – Activity diagrams highlight key decision nodes (e.g., payment success or failure), helping developers anticipate different scenarios.
    - **Enhancing Communication** – Both diagrams serve as visual documentation, making it easier for stakeholders, developers, and testers to collaborate and refine the system.

Result:

By completing this experiment, I have:

1. Developed the ability to model workflows using Activity Diagrams.
2. Understood the importance of State Diagrams in representing object behavior.
3. Gained insight into using UML diagrams to communicate design details effectively.

Exercise No: 7

Sketch the UML package diagram to show the User Interface, Domain objects and Technical services.

Date:

Pre-Lab Questions:

1. What are the four core principles of object-oriented programming, and how are they applied in software design?

   - **Encapsulation** – Restricts direct access to data using private attributes and getter/setter methods for security.
   - **Abstraction** – Hides complex details using interfaces and abstract classes, simplifying interactions.
   - **Inheritance** – Allows code reuse by enabling a class to inherit properties from another.
   - **Polymorphism** – Enables method overloading and overriding for flexible and maintainable code.

2. What is the significance of domain objects in a CRM system?

   - **Represent Real-World Entities** – Domain objects like Customer, Order, and Invoice model real-world CRM elements.
   - **Encapsulate Business Logic** – They contain rules and validations, ensuring data consistency within the system.
   - **Enable Data Persistence** – Stored in databases, they help in retrieving and managing customer-related information efficiently.
   - **Improve Code Maintainability** – Centralizing business logic in domain objects makes the system modular and scalable.

3. How do UML Package Diagrams help in visualizing the system structure?

- **Organize System Components** – Groups related classes into packages, improving modularity and structure.
- **Show Dependencies** – Illustrates relationships between packages, making interactions clearer.
- **Enhance Maintainability** – Identifies independent and dependent modules, simplifying system updates.
- **Improve Collaboration** – Helps teams understand the system architecture for better development.
- **Reduce Complexity** – Breaks down large systems into manageable units, improving clarity.
- **Promote Reusability** – Encourages reuse of common functionality across different modules.
- **Improve Scalability** – Makes it easier to expand the system by adding new packages without disrupting existing ones.
- **Aid in Access Control** – Defines visibility and access restrictions between packages to ensure security and proper encapsulation.

In-Lab: Case Studies and Problem Statements

Case Study:

You are tasked with designing a CRM system for a medium-sized business. The business requires the system to:

- Allow employees to manage customer data (e.g., names, contact information, transaction history).
- Support services like data validation, searching, and analytics.
- Provide a user-friendly interface for employees to interact with the system.

Problem Statement:

Develop a comprehensive CRM system that includes:

1. **User Interface (UI)**: Design the layout and components for employee interaction.
2. **Domain Objects**: Identify and create classes for customer-related data.
3. **Technical Services**: Develop services for processing and managing customer data.
4. **UML Package Diagram**: Create a diagram to visually organize the components of the CRM system.

Steps to Perform:

1. **Requirement Analysis:**
    - List the features required for the CRM system.
    - Identify the domain objects needed (e.g., Customer, Order, Transaction).
2. **System Design:**
    - Create classes for domain objects and define their attributes and methods.
    - Design the technical services required (e.g., DataValidationService, AnalyticsService).
3. **UML Package Diagram:**
    - Divide the system into logical packages, such as:

        - UI Package
        - Domain Objects Package
        - Services Package
        - Data Access Package
    - Show the relationships and dependencies between packages.

Implementation Example:

Sample Packages:

- UI Package
    - Classes: EmployeeDashboard, CustomerForm
- Domain Objects Package
    - Classes: Customer, Order, Transaction
- Services Package
    - Classes: DataValidationService, AnalyticsService
- Data Access Package
    - Classes: DatabaseConnection, DataRepository

EX NO.7 **DATE:**	**Draw the UML package diagram to show the User Interface, Domain objects and Technical services.**

AIM: To draw the UML Package Diagram for Ticket Reservation System and Bank Management System

HARDWARE REQUIREMENTS:

Intel Pentium Processor 3

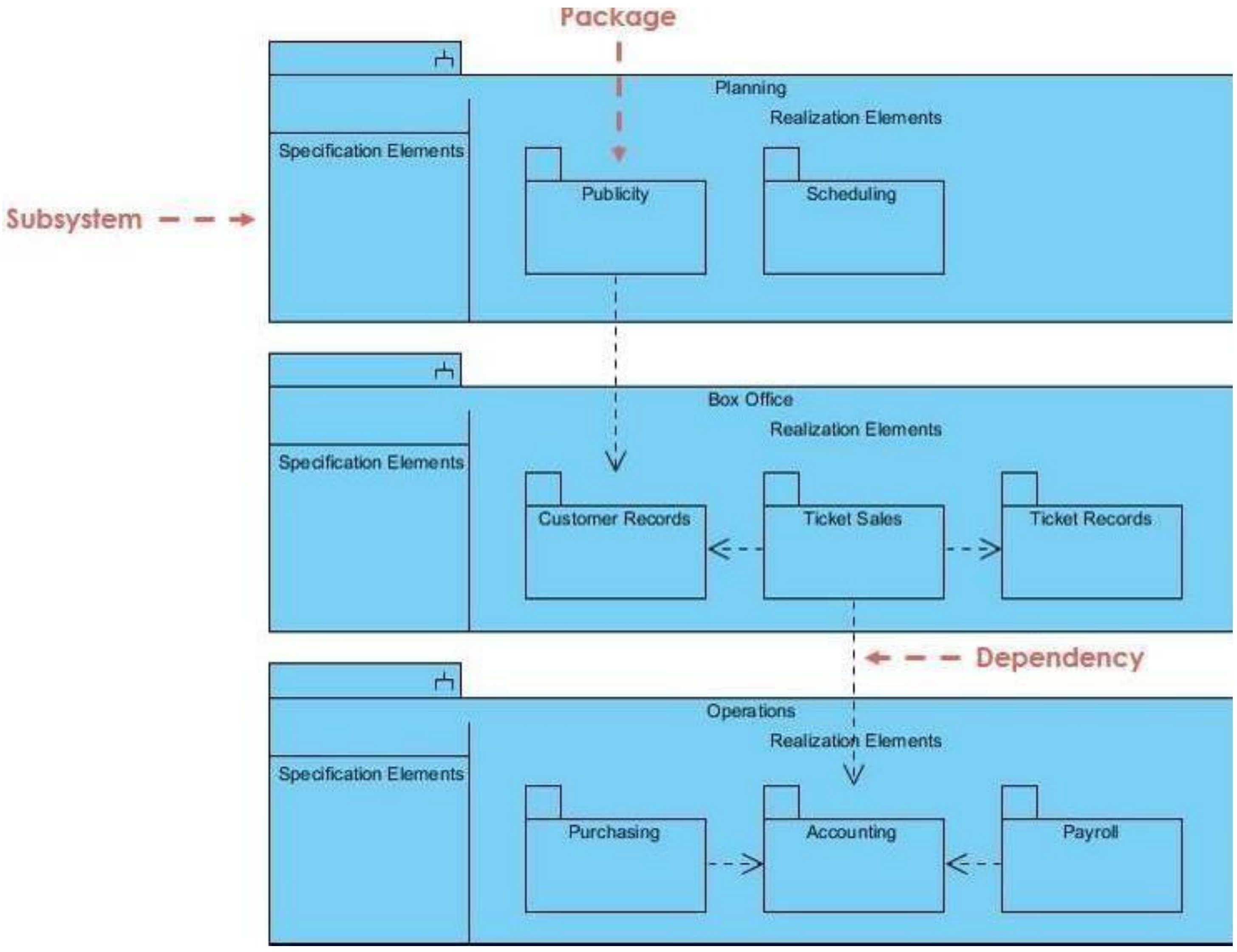
SOFTWARE REQUIREMENTS:

AGROUML

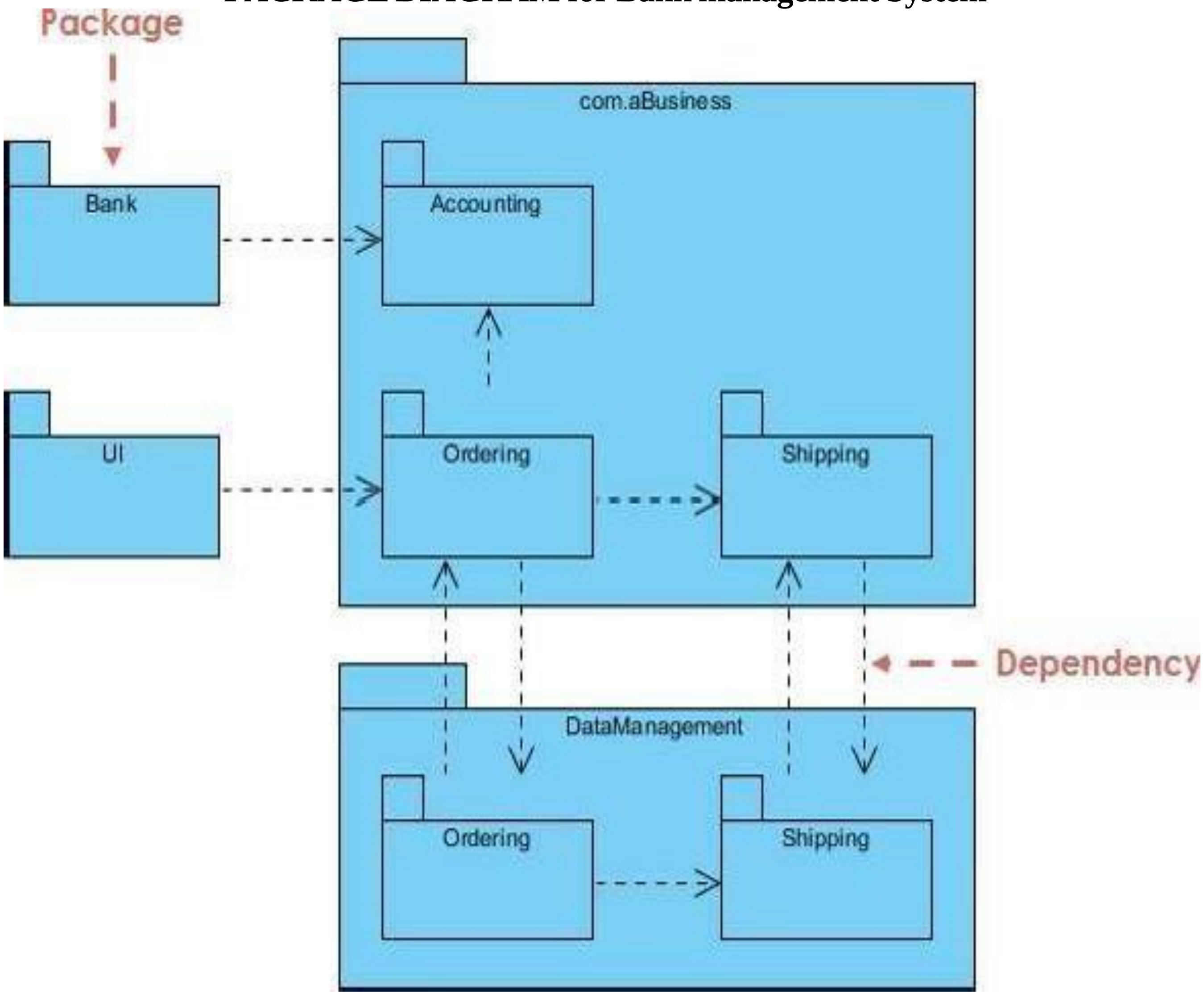
PROJECT DESCRIPTION:

A package diagram is represented as a folder shown as a large rectangle with a top attached to its upper left corner. A package may contain both sub ordinate package and ordinary
model elements. All uml models and diagrams are organized into package. A package diagram in unified modeling language that depicts the dependencies between the packages that make up a model. A Package Diagram (PD) shows a grouping of elements in the OO model, and
is a Cradle extension to UML. PDs can be used to show groups of classes in Class Diagrams (CDs), groups of components or processes in Component Diagrams (CPDs), or groups of processors
in Deployment Diagrams (DPDs). There are three types of layer. They are a. User interface layer b. Domain layer c. Technical services layer

PACKAGE DIAGRAM for Ticket Reservation System



PACKAGE DIAGRAM for Bank management System



RESULT:

Thus the UML Package diagrams for the Ticket Reservation system and Bank management system has been designed, executed and output is verified

Post-Lab: Inference and Analysis

Analysis Questions:

1. How does the UML Package Diagram improve the modularity of the CRM system?

   - **Groups Related Components** – Organizes CRM elements like Customer Management, Sales, and Support into separate packages, improving structure.
   - **Reduces System Complexity** – Breaks down the system into manageable units, making it easier to develop and maintain.
   - **Enhances Reusability** – Allows shared modules like Authentication or Reporting to be reused across different parts of the system.
   - **Improves Dependency Management** – Clearly defines interactions between packages, minimizing tight coupling and enhancing flexibility.

2. What are the advantages of separating domain objects and technical services into distinct packages?

   - **Better Modularity** – Keeps business logic (domain objects) separate from technical concerns (services), making the system well-structured.
   - **Easier Maintenance** – Changes in technical services (e.g., database, APIs) do not affect core business logic, simplifying updates.
   - **Improved Reusability** – Domain objects can be reused across different technical services without modification.
   - **Enhanced Scalability** – New features or services can be added without disrupting existing domain logic, making the system more adaptable.

3. Were there any challenges in identifying the relationships between packages? If so, how did you resolve them?
    - **Complex Dependencies** – Some packages had tight coupling, making it difficult to separate concerns. Solution: Used dependency inversion to reduce direct dependencies.
    - **Overlapping Responsibilities** – Some features seemed to belong to multiple packages. Solution: Clearly defined each package's responsibility to avoid duplication.
    - **Unclear Interactions** – Understanding how domain objects interact with technical services was challenging. Solution: Created UML diagrams to visualize data flow and dependencies.
    - **Scalability Concerns** – Some packages had too many dependencies, limiting future expansion. Solution: Refactored dependencies to follow a layered architecture for better scalability.

Inference:

- Reflect on the use of object-oriented principles in structuring the CRM system.
- Summarize the benefits of using UML diagrams in system design.

Result:

By completing this experiment, I have:

1. Drawn UML Package Diagram for the CRM system.
2. Learned the Documentation of identified domain objects and their attributes/methods.
3. Answered to the post-lab questions.

Exercise No: 8

Sketch the component diagram assuming that you will build your system by reusing existing components along with few new components.

Date:

Pre-Lab Questions:

1. What is component-based software engineering, and how does it differ from traditional development approaches?

- **Reusability** – CBSE uses **prebuilt components**, while traditional development builds everything **from scratch**.
- **Modularity** – CBSE promotes **loosely coupled modules**, whereas traditional methods often result in **tightly integrated systems**.
- **Faster Development** – CBSE reduces coding time by **reusing components**, while traditional approaches require **more effort**.
- **Scalability** – CBSE systems are **easier to expand and maintain**, while traditional development needs **major modifications** for growth.

2. What are the key elements of a UML Component Diagram?

    - **Components** – Represent modular parts of a system, such as services, databases, or UI elements.
    - **Interfaces** – Define how components interact, specifying provided and required services.
    - **Dependencies** – Show relationships between components, indicating communication or reliance.
    - **Connectors** – Represent links like associations, dependencies, or communications between components.

3. How does component reuse benefit system design?

- **Faster Development** – Prebuilt components reduce coding time and speed up development.
- **Cost Efficiency** – Reusing existing components lowers development and maintenance costs.
- **Improved Reliability** – Well-tested components reduce the risk of errors and improve system stability.
- **Enhanced Maintainability** – Modular components make it easier to update or replace specific parts without affecting the entire system.
- **Scalability** – Reusable components enable easy expansion of the system without major rewrites.
- **Consistency** – Using standardized components ensures uniform functionality across different parts of the system.
- **Better Collaboration** – Teams can work independently on different components, improving productivity.
- **Technology Adaptability** – Reusable components can be adapted to different platforms or technologies with minimal changes.

In-Lab: Case Studies and Problem Statements

Case Study:

You are tasked with designing a new Learning Management System (LMS) for a university. The LMS must:

- Manage user roles such as students, instructors, and administrators.
- Support features like course enrollment, content delivery, and performance tracking.
- Integrate existing modules for user authentication and payment processing.
- Include new components for online assessments, discussion forums, and gradebook management.

Problem Statement:

Create a UML Component Diagram that illustrates the integration of reused and new components in the LMS. Your design should:

1. Clearly separate existing components (e.g., Authentication Module, Payment Gateway) from newly developed ones (e.g., Assessment Module, Forum Module).
2. Show dependencies and interfaces between components.
3. Highlight how the components interact to deliver LMS functionality.

Steps to Perform:

1. **Requirement Analysis:**
    - List the functional requirements for the LMS.
    - Identify existing components to be reused and new components to be developed.
2. **Component Identification:**
    - Define the components required, such as:
        - Reused: AuthenticationModule, PaymentGateway
        - New: AssessmentModule, ForumModule, GradebookModule

3. **UML Component Diagram Design:**
    - Model the physical and logical components.
    - Use appropriate UML notations for components, interfaces, and dependencies.

Implementation Example:

Sample Components:

- **Reused Components**:
    - AuthenticationModule
    - PaymentGateway
- **New Components**:
    - AssessmentModule
    - ForumModule
    - GradebookModule

Example Diagram Elements:

- Interfaces: Define connections between components (e.g., AuthenticationModule provides LoginService).
- Dependencies: Illustrate how new components rely on reused ones.

EX NO.8 DATE:	**Draw component diagram assuming that you will build your system by reusing existing components along with few new components.**

AIM:

To draw the component diagrams for Ticket Reservation System and Bank Management System

HARDWARE REQUIREMENTS:

Intel Pentium Processor 3

SOFTWARE REQUIREMENTS:

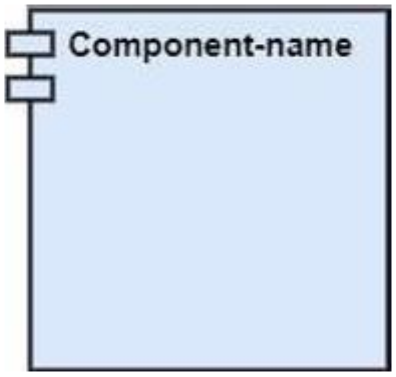
AGROUML

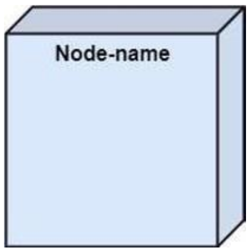
PROJECT DESCRIPTION:

A component diagram is used to break down a large object-oriented system into the smaller components, so as to make them more manageable. It models the physical view of a system such as executables, files, libraries, etc. that resides within the node. It visualizes the relationships as well as the organization between the components present in the system. It helps in forming an executable system. A component is a single unit of the system, which is replaceable and executable. The implementation details of a component are hidden, and it necessitates an interface to execute a function. It is like a black box whose behavior is explained by the provided and required interfaces.

Notation of a Component Diagram

a) A component





Purpose of a Component Diagram

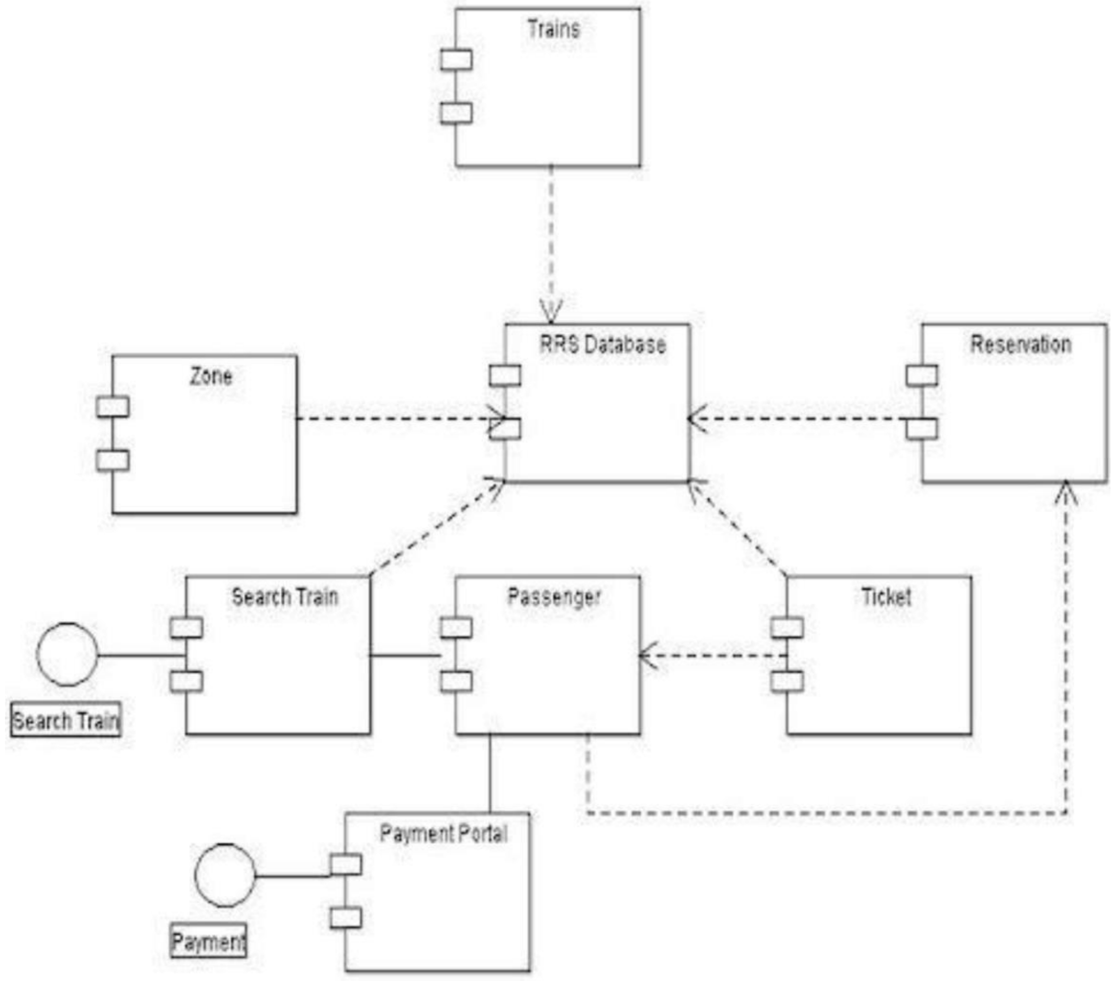
Since it is a special kind of a UML diagram, it holds distinct purposes. It describes all the individual components that are used to make the functionalities, but not the functionalities of the system. It visualizes the physical components inside the system. The components can be a library, packages, files, etc.

The component diagram also describes the static view of a system, which includes the organization of components at a particular instant. The collection of component diagrams represents a whole system.

The main purpose of the component diagram are enlisted below:

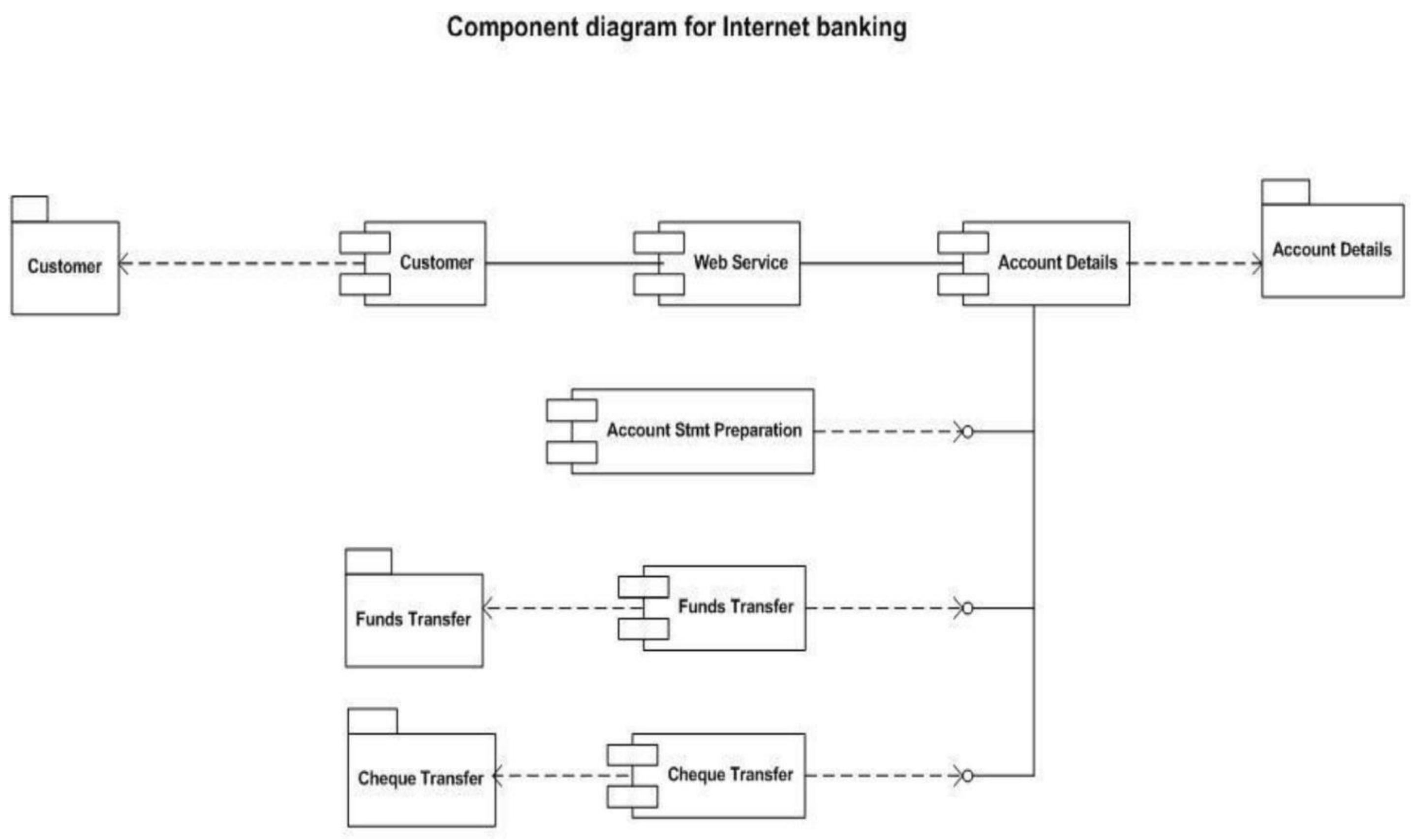
1. It envisions each component of a system.
2. It constructs the executable by incorporating forward and reverse engineering.

Component DIAGRAM for Ticket Reservation System



dig. Component Diagram for Railway Reservation System

Component DIAGRAM for Bank management System



RESULT:

Thus the Component diagrams for the Ticket Reservation system and Bank management system has been designed, executed and output is verified

Post-Lab: Inference and Analysis

Analysis Questions:

1. How does the UML Component Diagram improve understanding of the LMS architecture?

    - **Visualizes System Structure** – Clearly represents key components like authentication, course management, and user interfaces.
    - **Shows Component Interactions** – Illustrates how modules communicate, such as students accessing courses or instructors uploading materials.
    - **Enhances Maintainability** – Helps identify independent and reusable components, making updates easier.
    - **Improves Collaboration** – Aids developers, designers, and stakeholders in understanding system dependencies and functionality.

2. What challenges did you encounter while integrating reused and new components?

    - **Compatibility Issues** – Reused components may not align with new system frameworks, technologies, or data formats.
    - **Dependency Conflicts** – Existing and new components might have conflicting dependencies, requiring careful version management.
    - **Customization Limitations** – Reused components may not fully meet new requirements, leading to extra modifications.
    - **Performance Optimization** – Ensuring seamless communication between old and new components without affecting system efficiency.

3. How does component-based development enhance system scalability and maintainability?
   - **Modular Structure** – Components are independent, allowing easy system expansion without major changes.
   - **Code Reusability** – Existing components can be reused, reducing development effort for new features.
   - **Easier Maintenance** – Faulty components can be updated or replaced without affecting the entire system.
   - **Flexible Upgrades** – New technologies or features can be integrated by modifying specific components rather than the whole system.

Inference:

- Discuss the role of component reuse in reducing development effort and time.
- Reflect on how UML diagrams facilitate clear communication among stakeholders.

Result:

By completing this experiment, I have:

1. Drawn the UML Component Diagram for the LMS.
2. Documentation of reused and new components, including their functionalities and interfaces has been completed.
3. Answered to the post-lab questions.

Exercise No: 9

Sketch the deployment diagrams to model the runtime architecture of your application.

Date:

Pre-Lab Questions:

1. What is the significance of a multi-tier architecture in application development?
    - **Scalability** – Separates concerns (e.g., UI, business logic, database), allowing independent scaling of each tier.
    - **Maintainability** – Modular design makes updates and debugging easier without affecting the entire system.
    - **Security** – Sensitive operations (e.g., authentication) can be handled in secure backend layers, reducing exposure.
    - **Performance Optimization** – Workload is distributed across tiers, improving system efficiency and responsiveness.

2. What are the key elements of a UML Deployment Diagram?
    - **Nodes** – Represent physical devices or servers (e.g., application server, database server).
    - **Artifacts** – Software components deployed on nodes (e.g., executables, databases, web services).
    - **Communication Paths** – Show connections between nodes, representing data exchange or network communication.
    - **Dependencies –** Indicate relationships between artifacts, such as software relying on a specific database or API.

3. How do external services enhance application functionality?

- **Expanded Features** – Adds advanced functionalities (e.g., payment processing, AI, or cloud storage) without building them from scratch.
- **Improved Performance** – Offloads resource-intensive tasks (e.g., image processing, authentication) to specialized external services.
- **Scalability** – Adapts to increasing workloads by leveraging cloud-based services like AWS, Firebase, or Azure.
- **Cost Efficiency** – Reduces development and maintenance costs by using prebuilt, well-maintained solutions.

In-Lab: Case Studies and Problem Statements

Case Study:

You are tasked with developing an application for a retail business. The application must:

- Provide a user-friendly web front-end for customers to browse products and place orders.
- Use a backend server to handle business logic, such as order processing and inventory management.
- Integrate a database to store product, customer, and transaction data.
- Utilize external services for payment processing and email notifications to confirm orders.

Problem Statement:

Develop an application with the following components:

1. **Web Front-End**: Provides the user interface for customers.
2. **Backend Server**: Processes user requests, manages business logic, and interacts with the database.
3. **Database**: Stores and retrieves data related to products, orders, and customers.
4. **External Services**:
    - **Payment Gateway**: Processes customer payments.
    - **Email Notification Service**: Sends order confirmations to customers.
5. **Deployment Diagram**: Illustrates the runtime architecture of the application.

Steps to Perform:

1. **Requirement Analysis:**
    - Define the functional requirements of the application.
    - Identify the components and external services to be included.
2. **System Design:**

   - Design the web front-end and backend server.
   - Define the database schema for storing application data.
   - Specify how external services will be integrated.
3. **UML Deployment Diagram:**
   - Model the physical nodes, such as client devices, web server, application server, database server, and external service endpoints.
   - Illustrate the deployment of components and their interactions.

Implementation Example:

Deployment Diagram Elements:

- **Nodes:**
  - Client Device (Web Browser)
  - Web Server
  - Application Server
  - Database Server
  - Payment Gateway
  - Email Service
- **Artifacts:**
  - Web Application
  - Backend Application
  - Database
  - PaymentProcessingAPI
  - EmailNotificationAPI
- **Connections:**
  - Depict communication paths between nodes (e.g., HTTP, SQL, API calls).

EX NO.9 DATE:	**Draw deployment diagrams to model the runtime architecture of your application**

AIM:

To draw the Deployment Diagrams for Ticket Reservation System and Bank Management System

HARDWARE REQUIREMENTS:

Intel Pentium Processor 3

SOFTWARE REQUIREMENTS:

AGROUML

PROJECT DESCRIPTION:

The deployment diagram visualizes the physical hardware on which the software will be deployed. It portrays the static deployment view of a system. It involves the nodes and their relationships.

It ascertains how software is deployed on the hardware. It maps the software architecture created in design to the physical system architecture, where the software will be executed as a node. Since it involves many nodes, the relationship is shown by utilizing communication paths.

Purpose of Deployment Diagram

The main purpose of the deployment diagram is to represent how software is installed on the hardware component. It depicts in what manner a software interacts with hardware to perform its execution.

Both the deployment diagram and the component diagram are closely interrelated to each other as they focus on software and hardware components. The component diagram represents the components of a system, whereas the deployment diagram describes how they are actually deployed on the hardware.

The deployment diagram does not focus on the logical components of the system, but it put its attention on the hardware topology.

Following are the purposes of deployment diagram enlisted

below: To envision the hardware topology of the

system.

To represent the hardware components on which the software components are installed.

To describe the processing of nodes at the runtime.

Symbol and notation of Deployment diagram

The deployment diagram consist of the following

notations: A component

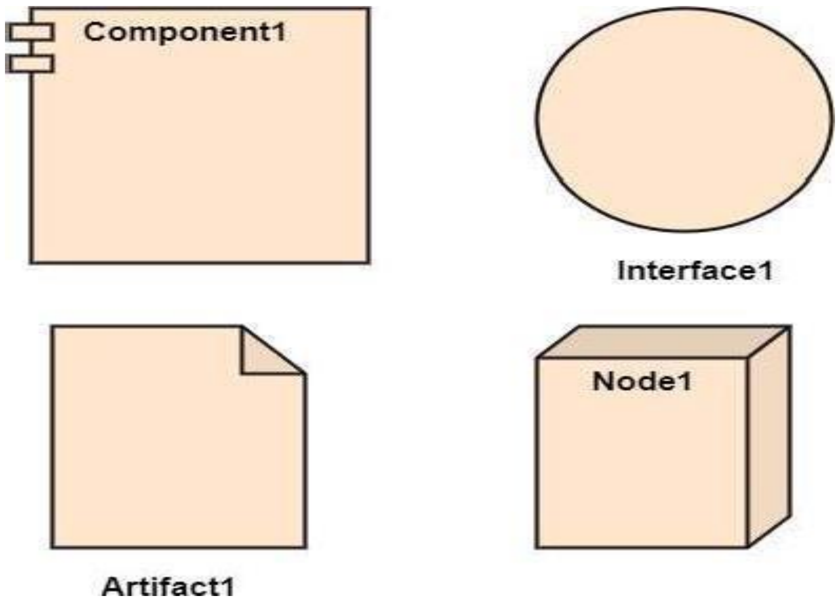
An

artifact

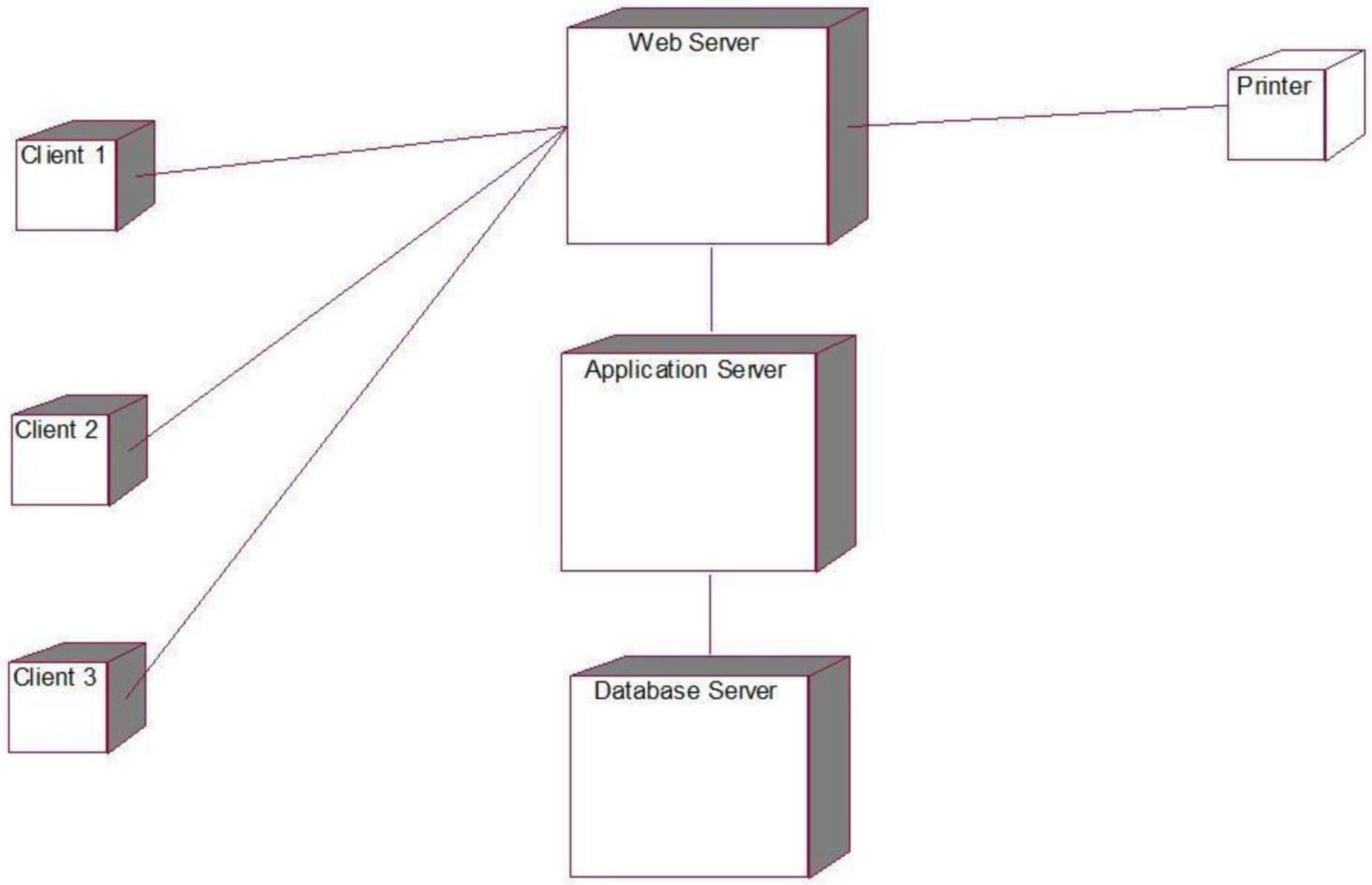
An

interface

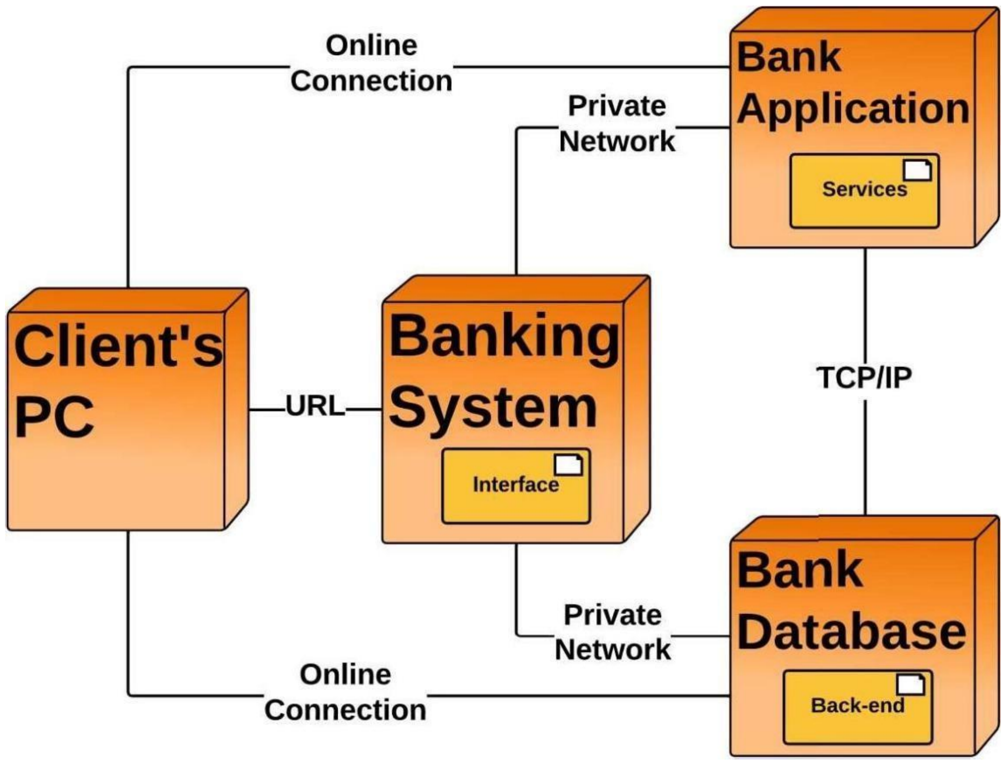
A node



Deployment Diagram for Railway Reservation System



Deployment Diagram for Bank Management System



RESULT:

Thus the Deployment diagrams for the Ticket Reservation system and Bank management system has been designed, executed and output is verified

Post-Lab: Inference and Analysis

Analysis Questions:

1. How does the UML Deployment Diagram help in visualizing the runtime architecture of the application?

    - **Represents Physical Structure** – Shows how hardware nodes (servers, databases, cloud services) interact at runtime.
    - **Illustrates Component Deployment** – Maps software artifacts (applications, APIs, databases) to their respective nodes.
    - **Defines Communication Paths** – Depicts network connections and data flow between different system components.
    - **Improves System Optimization** – Helps identify bottlenecks, scalability issues, and performance improvements in deployment.

2. What are the benefits and challenges of integrating external services into the application?

    - **Enhanced Functionality** – Adds features like payments, authentication, and cloud storage without extra development.
    - **Faster Development** – Saves time by using prebuilt APIs instead of coding from scratch.
    - **Dependency Risks** – Service outages or API changes can disrupt application performance.
    - **Security Concerns** – Handling sensitive data with third-party services requires strict security measures.

3. How can you ensure the scalability and reliability of the multi-tier architecture?

   - **Load Balancing** – Distribute traffic across multiple servers to prevent overload and ensure availability.
   - **Database Optimization** – Use indexing, caching, and replication to handle high query loads efficiently.
   - **Horizontal and Vertical Scaling** – Add more servers (horizontal scaling) or upgrade hardware (vertical scaling) as needed.
   - **Failover and Redundancy** – Implement backup servers and automatic failover mechanisms to ensure reliability.

Inference:

- Reflect on the importance of deployment diagrams in planning and maintaining system infrastructure.
- Discuss the impact of using external services on the development and operation of the application.

Result:

By completing this experiment, I have:

1. Drawn UML Deployment Diagram for the application.
2. Documentation of system components and their roles has been done.
3. Description of external service integration is completed.
4. Answers to post-lab questions has been updated.

Exercise No: 10

Apply appropriate design patterns to improve the reusability and maintainability of the software system.

Date:

Pre-Lab Questions:

1. What are design patterns, and why are they important in software development?
    - **Definition** – Design patterns are proven solutions to common software design problems, providing reusable templates for efficient development.
    - **Code Reusability** – Encourages the use of standardized solutions, reducing redundant code and improving maintainability.
    - **Scalability and Flexibility** – Helps design systems that can adapt and scale efficiently as requirements change.
    - **Improved Code Quality** – Promotes best practices, ensuring well-structured, readable, and maintainable code.

2. Identify the three main categories of design patterns and provide an example of each.
    - **Creational Patterns** – Focus on object creation, ensuring flexibility and reusability.
      Example: Singleton Pattern – Ensures only one instance of a class exists (e.g., database connection management).
    - **Structural Patterns** – Help in organizing classes and objects to form larger structures.
      Example: Adapter Pattern – Allows incompatible interfaces to work together (e.g., converting XML data to JSON format).
    - **Behavioral Patterns** – Define communication and interaction between objects.
      Example: Observer Pattern – Allows multiple objects to respond to changes in another object (e.g., event listeners in UI design).

3. How do design patterns contribute to software modularity and reusability?

- **Encapsulation of Functionality** – Patterns like Facade and Adapter separate concerns, making modules independent and easier to modify.
- **Code Reusability** – Creational patterns like Factory Method enable the reuse of object creation logic, reducing redundant code.
- **Improved Maintainability** – Behavioral patterns like Observer and Strategy allow flexible communication between modules, simplifying future changes.
- **Scalability and Extensibility** – Structural patterns like Decorator make it easy to add new features without modifying existing code, enhancing system scalability.

In-Lab: Case Studies and Problem Statements

Case Study:

You are tasked with improving an e-commerce application that includes the following components:

- **Catalog Module**: Manages product details and inventory.
- **Shopping Cart Module**: Handles user-selected items and calculates totals.
- **Order Processing Module**: Processes orders and payments.
- **Notification System**: Sends order confirmation emails and updates.

The current system suffers from low modularity and high coupling, making it difficult to add new features or reuse existing components.

Problem Statement:

Identify design patterns that can improve the modularity and reusability of the e-commerce system. Specifically:

1. Apply a **creational pattern** (e.g., Factory, Singleton) to streamline object creation.
2. Use a **structural pattern** (e.g., Adapter, Composite) to manage relationships between components.
3. Implement a **behavioral pattern** (e.g., Observer, Strategy) to enhance inter-component communication.

Steps to Perform:

1. **Requirement Analysis:**
    - Identify key issues in the existing system (e.g., tight coupling, redundant code).
    - Determine which design patterns can address these issues.
2. **Pattern Implementation:**
    - Apply appropriate design patterns to the Catalog, Shopping Cart, and Notification modules.
    - Example:

        - **Factory Pattern**: Simplify object creation for different product types.
        - **Observer Pattern**: Notify the Notification System when an order is placed.
        - **Adapter Pattern**: Integrate a new payment gateway with minimal changes.

3. **Model Improved Architecture:**
    - Use UML diagrams (e.g., Class Diagram, Sequence Diagram) to illustrate the application of design patterns.

EX NO.10 DATE:	**Apply appropriate design patterns to improve the reusability and maintainability of the software system**

SOFTWARE PERSONNEL MANAGEMENT SYSTEM AIM:
To implement a software for software personnel management system

PROBLEMSTATEMENT:

Human Resource management system project involves new and/or system upgrades of software of send to capture information relating to the hiring termination payment and management of employee. He uses system to plan and analyze all components and performance of metrics driven human resource functions, including recruitment, attendance, compensation, benefits and education. Human resources management systems should align for maximum operating efficiency with financial accounting operations customer relationship management, security and business lines as organization.

(I) SOFTWARE REQUIREMENT SPECIFICATION:

2.1 SOFTWAREINTERFACE

- **Front End Client-** The applicant and Administrator online interface is built using JSP and HTML. The HR's local interface is built using Java.

Server - Glassfish application server (SQLCorporation).

Back End SQL database.

2.2 HARDWARE INTERFACE

The server is directly connected to the client systems. The client systems have access to the database in the server.

(II) USECASE DIAGRAM:

The HR of an organization involves recruitment training, monitoring and motivation ofan employee. The HR also involves gives salary as observed in the payroll sheet. The employee undergoes training, receives the salary, gives the expected performance and manages time in order to complete a given task within the required period.

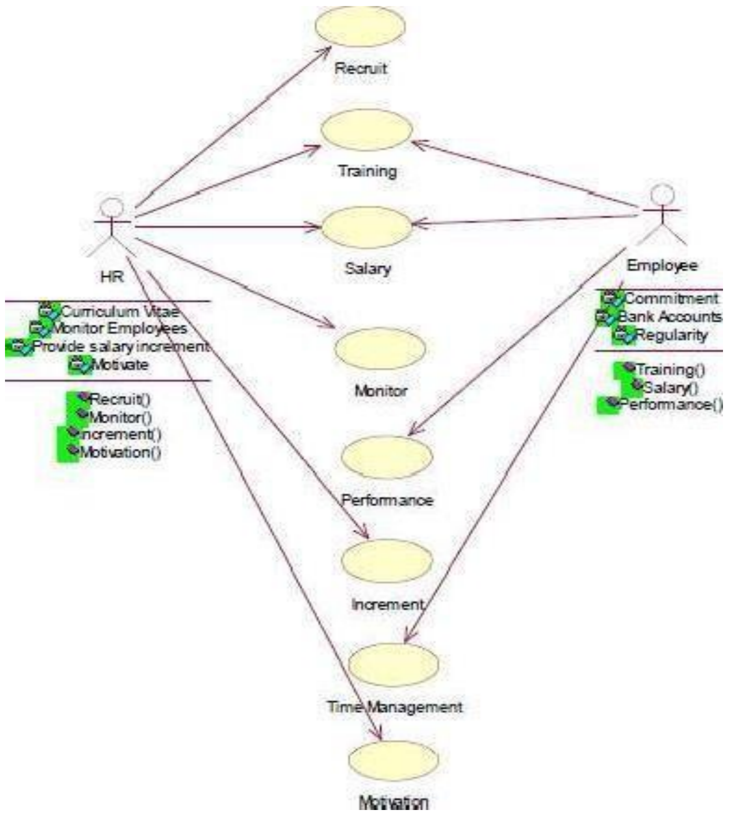


Fig.3. USE CASE DIAGRAM

(III) ACTIVITYDIAGRAM:

The activity diagram action, partition, for k join and object node. Most of the notation is self-explanatory, two subtle points. Once an action finished, there is an automatic outgoing transaction. The diagram can show both control flow and dataflow.

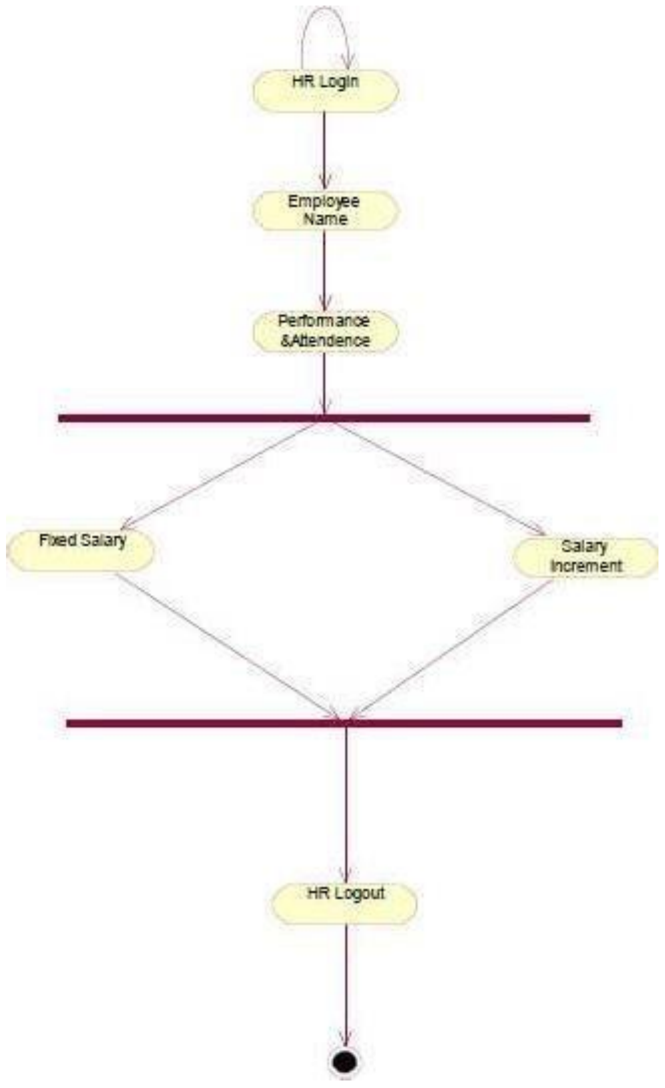


Fig.4. ACTIVITY DIAGRAM

(IV) CLASSDIAGRAM:

The class diagram, also referred to as object modeling is the main static analysis diagram.Themaintaskofobjectmodelingistographicallyshowwhateachobjectwilldo in the problem domain. The problem domain describes the structure and the relationships among objects.

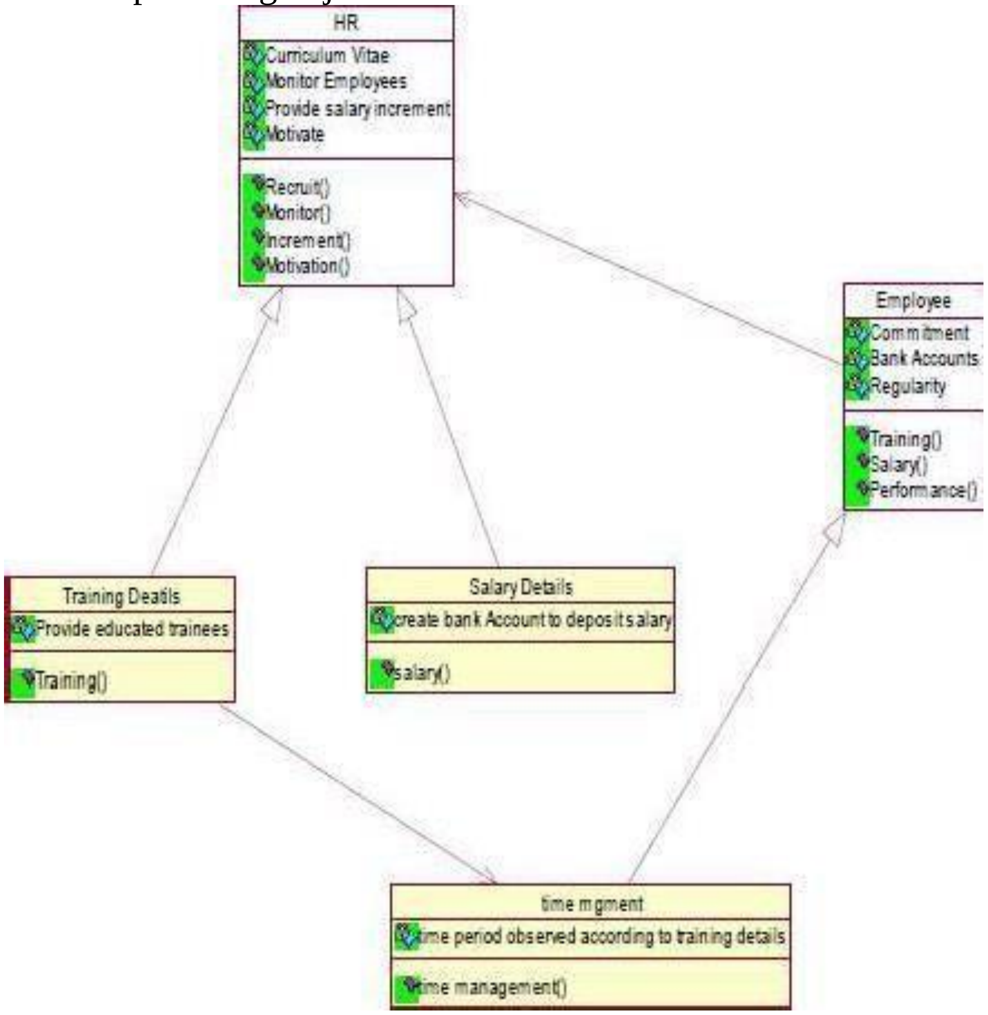


Fig.5. CLASS DIAGRAM

(V) INTERACTION DIAGRAM:

A sequence diagram represents the sequence and interactions of a given USE- CASE or scenario. Sequence diagrams can capture most of the information about the system.Mostobjecttoobjectinteractionsandoperationsareconsideredeventsandevents include signals, inputs, decisions, interrupts, transitions and actions to or from users or external devices.

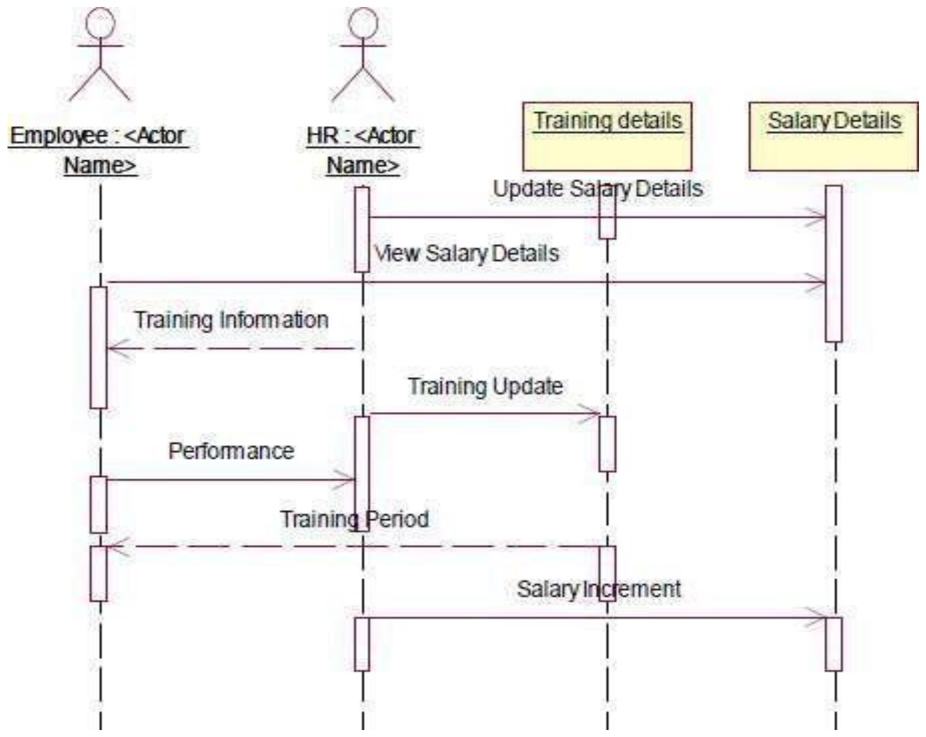


Fig.6.1.SEQUENCE DIAGRAM

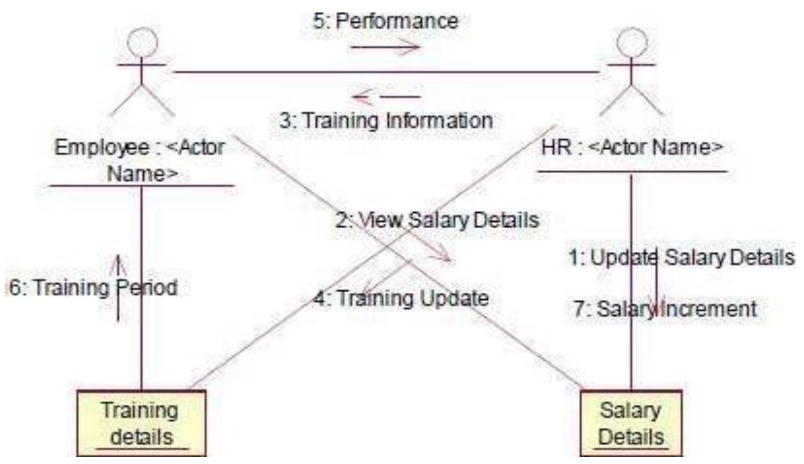


Fig.6.2.COLLABORATION DIAGRAM

(VI) State Transition Diagram

States of object are represented as rectangle with round corner, the transaction between the different states. A transition is a relationship between two state that indicates that when an event occur the object moves from the prior state to the subsequent.

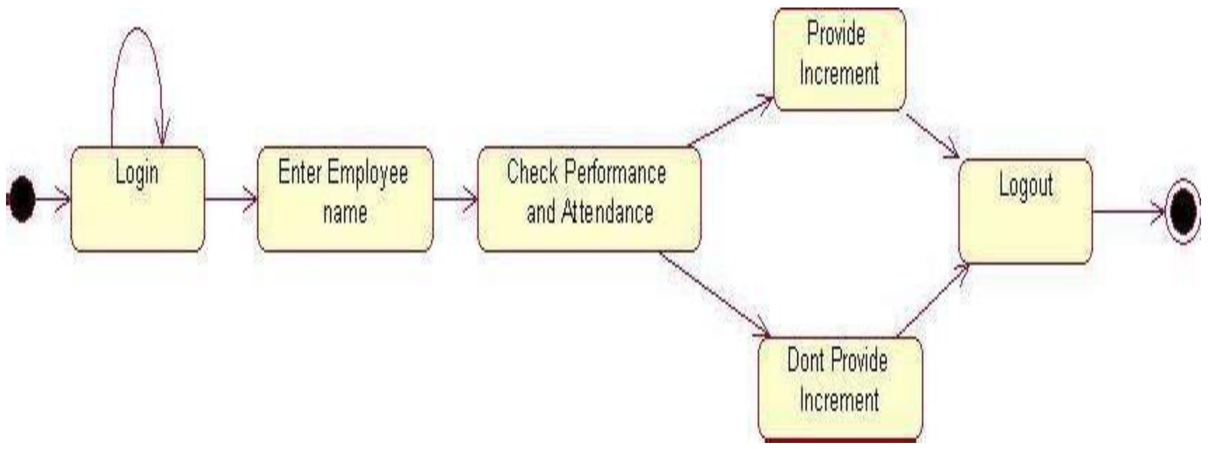


Fig.6.3 STATE TRANSITION DIAGRAM

(VII) DEPLOYMENT DIAGRAM AND COMPONENTDIAGRAM

HR recruits employee for a company employee recruited by HR goes undertraining before actually working. Training period is given to the employee with the training details. The salary details for the employee are provided.

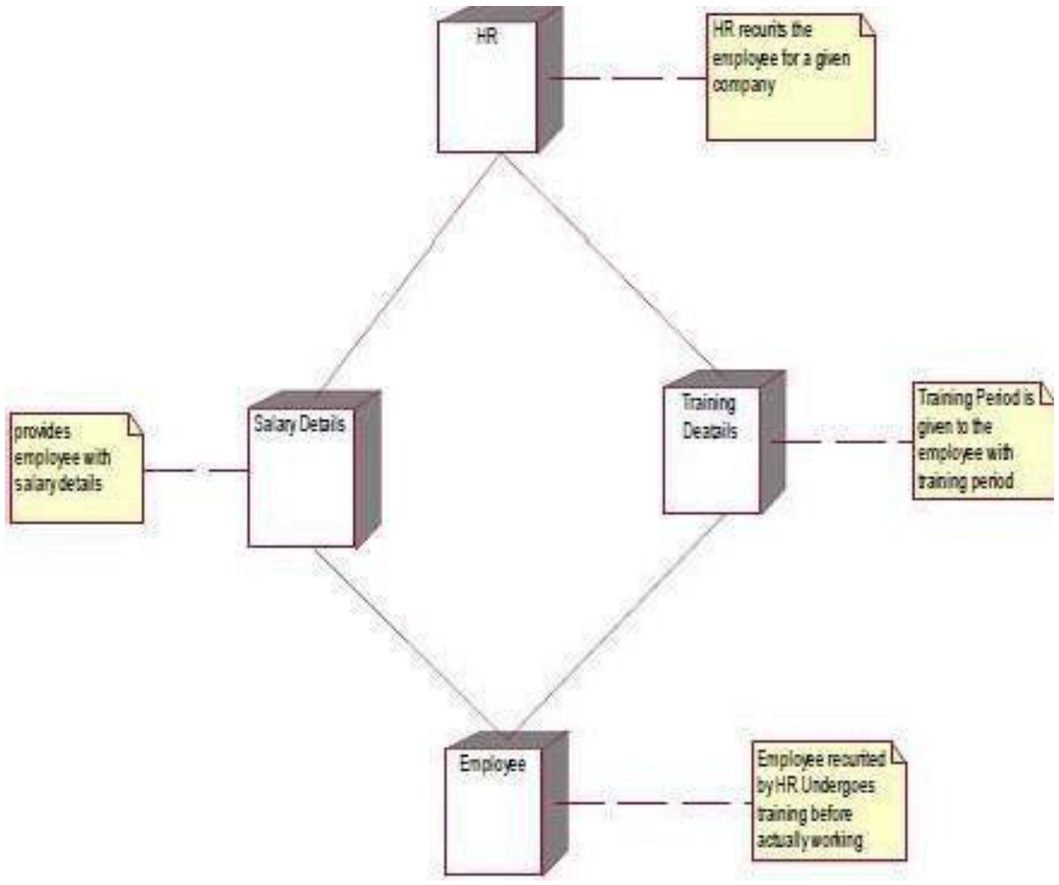

HR
HR recurits the employee for a given company
provides employee with salary details
Salary Details
Training Deatails
Training Period is given to the employee with training period
Employee
Employee recurited by HR Undergoes training before actually working

Fig.8.1.DEPLOYMENT DIAGRAM

COMPONENT DIAGRAM

The HR recruits, motivate and monitor the employee, HR also update the salary details and training details for reference. The employee are those who are recruited by HR and work for the company. The training details provide employees with training details which is updated by HR

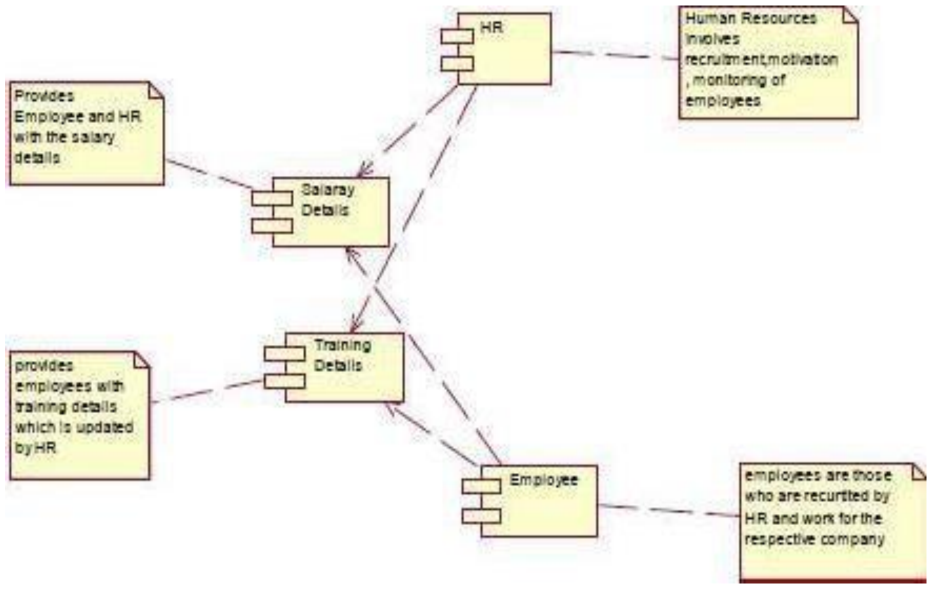

HR
Human Resources involves recruitment,motivation ,monitoring of employees
Provides Employee and HR with the salary details
Salaray Details
Training Details
provides employees with training details which is updated by HR
Employee
employees are those who are recurtited by HR and work for the respective company

Fig.8.2.COMPONENT DIAGRAM

RESULT:

Thus the project for software personnel management system has been successfully executed and codes are generated.

Post-Lab: Inference and Analysis

Analysis Questions:

1. Which design patterns were applied, and how did they improve the system's modularity and reusability?

    - **Singleton Pattern** – Ensured a single instance of essential services (e.g., database connection), preventing redundancy and improving efficiency.
    - **Factory Pattern** – Encapsulated object creation, allowing flexible instantiation of different components without modifying core logic.
    - **Observer Pattern** – Enabled event-driven communication (e.g., notifying users of order updates), improving modularity by reducing direct dependencies.
    - **Facade Pattern** – Provided a simplified interface for complex subsystems, making the system easier to extend and maintain.

2. What challenges did you face while implementing the design patterns?

    - **Selecting the Right Pattern** – Choosing the most suitable pattern for a problem was difficult, as some patterns had overlapping functionalities.
    - **Increased Complexity** – Some patterns, like Factory or Observer, introduced additional layers, making the code more abstract and harder to understand initially.
    - **Integration with Existing Code** – Adapting existing components to fit design patterns required refactoring, which was time-consuming and risked breaking functionality.
    - **Performance Overhead** – Certain patterns, such as Singleton and Decorator, added processing steps, requiring careful optimization to maintain efficiency.

3. How do design patterns contribute to the scalability and maintainability of the system?

    - **Encapsulation and Modularity** – Patterns like Facade and Adapter separate concerns, making it easier to modify or extend specific modules without affecting the entire system.
    - **Code Reusability** – Factory Method and Singleton reduce redundant code by centralizing object creation, ensuring consistency across the application.
    - **Flexibility and Extensibility** – Decorator and Observer allow adding new features dynamically without modifying existing code, supporting scalability.
    - **Simplified Maintenance** – Strategy and Command patterns enable replacing or updating functionality with minimal code changes, making debugging and updates more efficient.

Inference:

- Reflect on how applying design patterns simplifies system design and reduces complexity.
- Discuss the trade-offs of using design patterns, such as increased initial development effort.

Result:

By completing the exercise I have:

1. Learned the description of the identified problems and applied design patterns.
2. Drawn the UML diagrams illustrating the improved system architecture.
3. Answered to the post-lab analysis questions.