SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

FACULTY OF SCIENCE AND HUMANITIES

DEPARTMENT OF COMPUTER APPLICATIONS

PRACTICAL RECORD NOTE

STUDENT NAME :

REGISTER NUMBER :

CLASS : **BCA** **SECTION:**

YEAR & SEMESTER : **III YEAR & V SEM**

SUBJECT CODE : **UCA23503J**

SUBJECT TITLE : **OBJECT ORIENTED ANALYSIS AND DESIGN**

OCTOBER 2025

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

FACULTY OF SCIENCE AND HUMANITIES

DEPARTMENT OF COMPUTER APPLICATIONS

SRM Nagar, Kattankulathur – 603 203

CERTIFICATE

Certified to be the bonafide record of practical work done by

_____

*Register No.*_____*of* **BCA** *Degree programme for*

UCA23503J – OBJECT ORIENTED ANALYSIS AND DESIGN *in the Computer lab in SRM Institute of Science and Technology during the academic year 2025-2026.*

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

*Submitted for Semester Practical Examination held on*_____.

Internal Examiner ***External Examiner***

INDEX

S. No	Date	Name of the Experiment	Page No	Sign
1		ATM system		
2		Payroll Management System		
3		Quiz System		
4		Stock Maintenance System		
5		Passport Registration System		
6		Student Mark Analysis System		
7		Library Management System		
8		Railway Ticket Reservation System		
9		Placement Registration System		
10		Exam Registration System		

ATM SYSTEM

Ex No :

Date :

AIM

ALGORITHM

USE CASE DIAGRAM

Step 1: Identify Actors

Actors represent entities that interact with the system. For an ATM system, the actors are:

- **Client**: The person using the ATM.
- **ATM**: The machine the client interacts with.
- **System/Bank**: The bank's backend system that processes requests.

Step 2: Identify Use Cases

Use cases describe the interactions or services provided by the system. For the ATM system, the use cases are:

1. **Deposit Amount**: Client deposits money into their account.
2. **Withdraw Amount**: Client withdraws money from their account.
3. **Review Details**: Client checks account details (e.g., balance, recent transactions).
4. **Update Details**: Client updates account information (e.g., phone number, email).
5. **Terminate Transaction**: The transaction is completed and the system logs out the client.
6. **Cancel Transaction**: The client cancels an ongoing transaction.
7. **Check PIN/Card**: The system verifies the client's card and PIN for security.

Step 3: Define Actor-Use Case Relationships

- **Client**: Interacts with all use cases.
- **ATM**: Manages deposit, withdrawal, transaction termination, cancellation, and PIN check.
- **System/Bank**: Processes transactions and handles account details.

Step 4: Draw the Diagram

- **Draw actors**: Place actors (Client, ATM, System/Bank) outside the system boundary.
- **Draw use cases**: Inside the system boundary, use ovals for each use case.
- **Connect**: Draw lines between actors and relevant use cases.

Step 5: Define Optional Relationships (if needed)

- Use **<<include>>** for mandatory steps (e.g., Check PIN/Card before transactions).
- Use **<<extend>>** for optional actions (e.g., Cancel Transaction).

Step 6: Review

Ensure all actors are correctly linked to the relevant use cases

CLASS DIAGRAM

Step 1: Identify Key Classes

- ATM, Client, BankAccount, Transaction, Bank

Step 2: Define Attributes and Methods

1. **ATM**:
    - **Attributes**: ATMID, location, currentBalance
    - **Methods**: validateCard(), deposit(), withdraw()
2. **Client**:
    - **Attributes**: clientID, cardNumber, PIN
    - **Methods**: enterPIN(), requestTransaction()
3. **BankAccount**:
    - **Attributes**: accountNumber, balance
    - **Methods**: getBalance(), deposit(), withdraw()
4. **Transaction**:
    - **Attributes**: transactionID, type, amount
    - **Methods**: execute(), cancel()
5. **Bank**:
    - **Attributes**: bankName, bankID
    - **Methods**: processTransaction(), verifyPIN()

Step 3: Define Relationships

- **ATM** interacts with **Client** and **BankAccount** (association).
- **Client** aggregates with **BankAccount**.
- **Transaction** associates with **ATM** and **BankAccount**.
- **Bank** oversees all components.

Step 4: Draw the Diagram

- Draw rectangles for each class with attributes and methods.
- Connect the classes with lines to show relationships (association, aggregation).

Step 5: Finalize

Ensure the correct relationships and elements are represented.

ACTIVITY DIAGRAM

Step 1: Identify the Process

For an ATM system, the typical process flow involves a client performing transactions like withdrawing money or checking their balance. The steps can include:

1. **Insert Card**
2. **Enter PIN**
3. **Select Transaction**
4. **Process Transaction** (Deposit, Withdraw, Check Balance, etc.)
5. **End Transaction**

Step 2: Define Activities

Each process involves several activities. Common activities for an ATM system include:

- **Insert Card**
- **Validate Card**
- **Enter PIN**
- **Validate PIN**
- **Select Transaction Type**
- **Perform Transaction** (Withdraw, Deposit, etc.)
- **Check Balance**
- **Print Receipt**
- **End Session**

Step 3: Define Decision Points

Decision points represent choices the client can make:

- **Valid Card?**: If invalid, eject the card.
- **Correct PIN?**: If incorrect, ask to re-enter PIN or terminate.
- **Transaction Type?**: Deposit, Withdraw, Check Balance, etc.

Step 4: Draw the Diagram

1. **Start Node**: Draw a solid black circle to represent the start of the activity.

2. **Actions/Activities**: Draw rounded rectangles for each activity (e.g., Insert Card, Enter PIN, Withdraw Money).
3. **Decision Points**: Represent decisions with diamonds. Label the outcomes with arrows (e.g., Yes/No, Valid/Invalid).
4. **Flow Arrows**: Use arrows to show the flow between activities.
5. **End Node**: Draw a solid black circle with a ring around it to represent the end of the process.

SEQUENCE DIAGRAM

Step 1: Identify Participants

Identify the main objects involved in the ATM transaction process:

- **Client**
- **ATM**
- **Bank System**

Step 2: Define the Sequence of Messages

Outline the sequence of interactions between the participants. Typical interactions for an ATM transaction include:

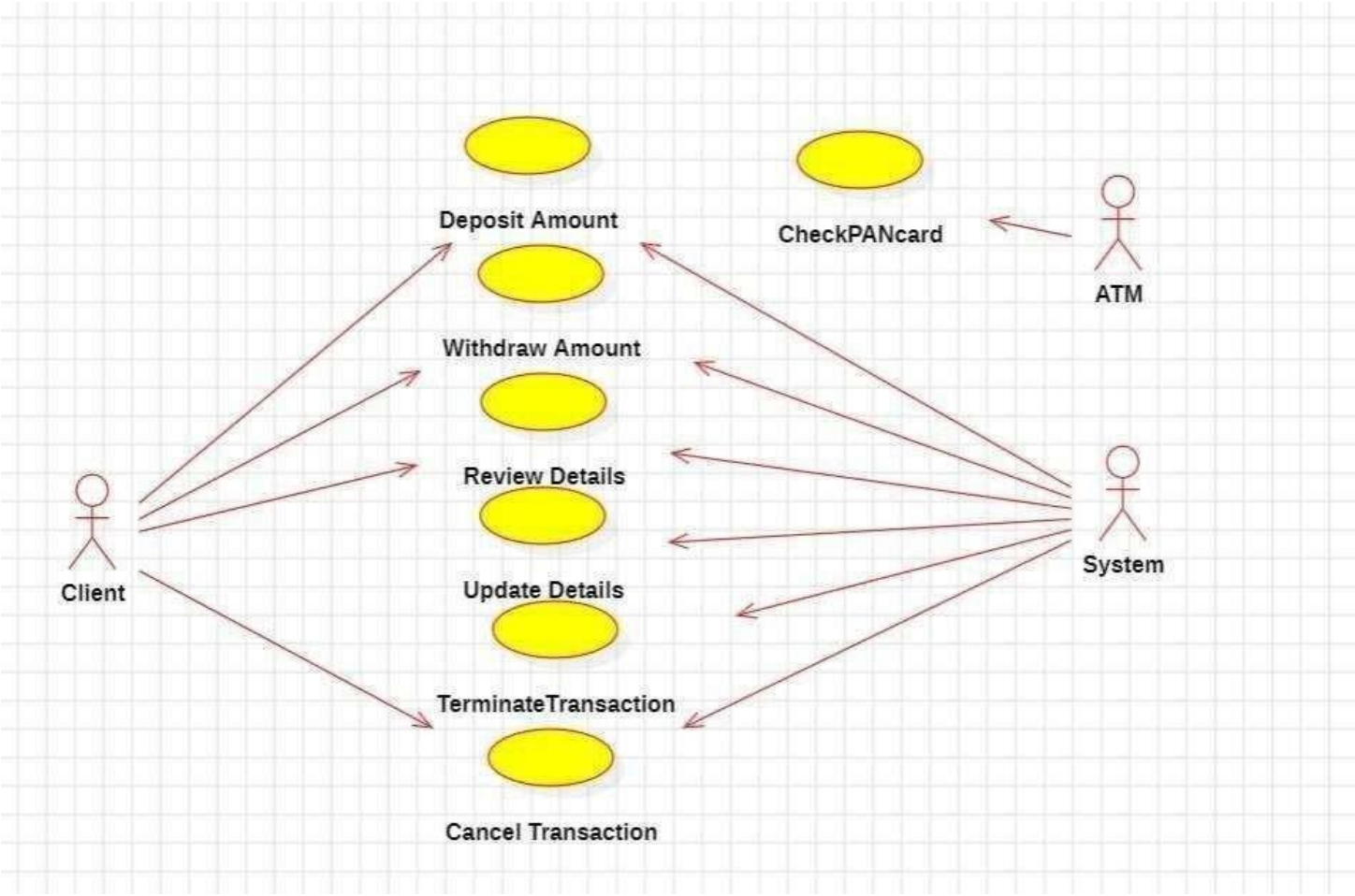
1. **Client** inserts card.
2. **ATM** validates card.
3. **Client** enters PIN.
4. **ATM** validates PIN.
5. **Client** selects transaction type (e.g., Withdraw, Deposit).
6. **ATM** processes transaction.
7. **Bank System** updates account balance.
8. **ATM** provides confirmation or receipt to the **Client**.
9. **Client** ends session.

Step 3: Draw the Diagram

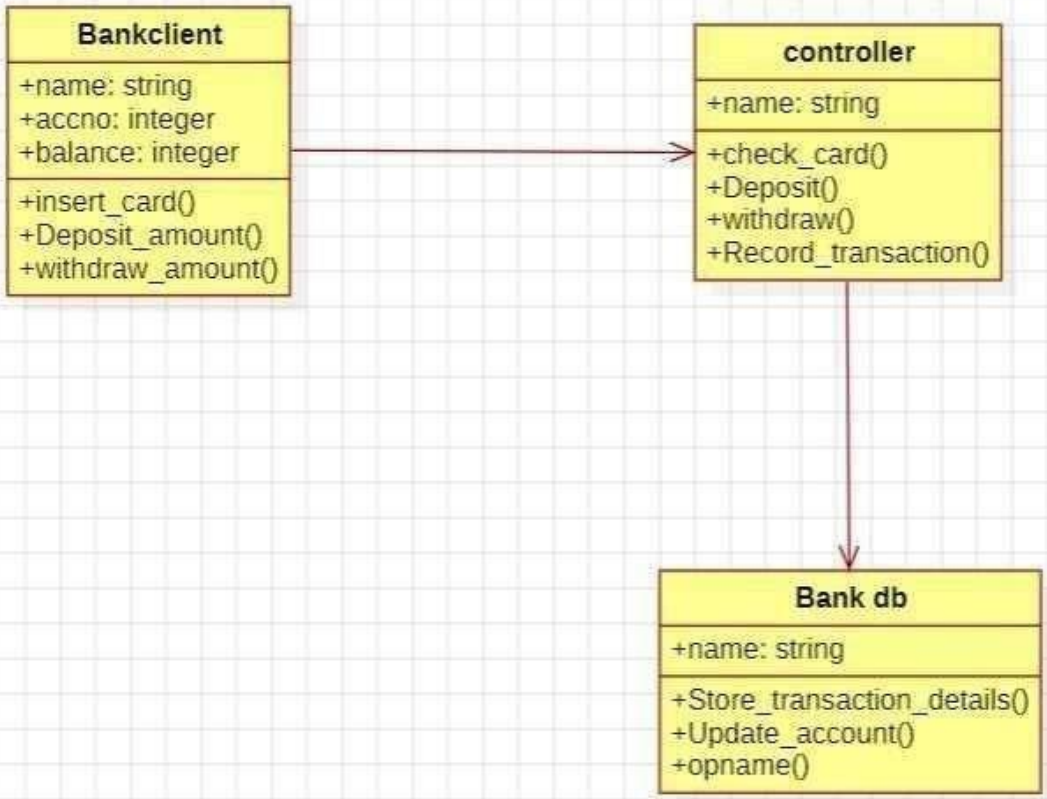
1. **Lifelines**: Draw vertical dashed lines for each participant (Client, ATM, Bank System).
2. **Activation Bars**: Use rectangles to represent the active period of each participant during the interaction.
3. **Messages**: Draw horizontal arrows to represent messages or actions:
    - From **Client** to **ATM**: Insert Card, Enter PIN, Select Transaction.
    - From **ATM** to **Bank System**: Validate PIN, Process Transaction, Update Account.
    - Responses from **ATM** back to **Client**: Display Receipt, Confirm Transaction.

Step 4: Review and Finalize

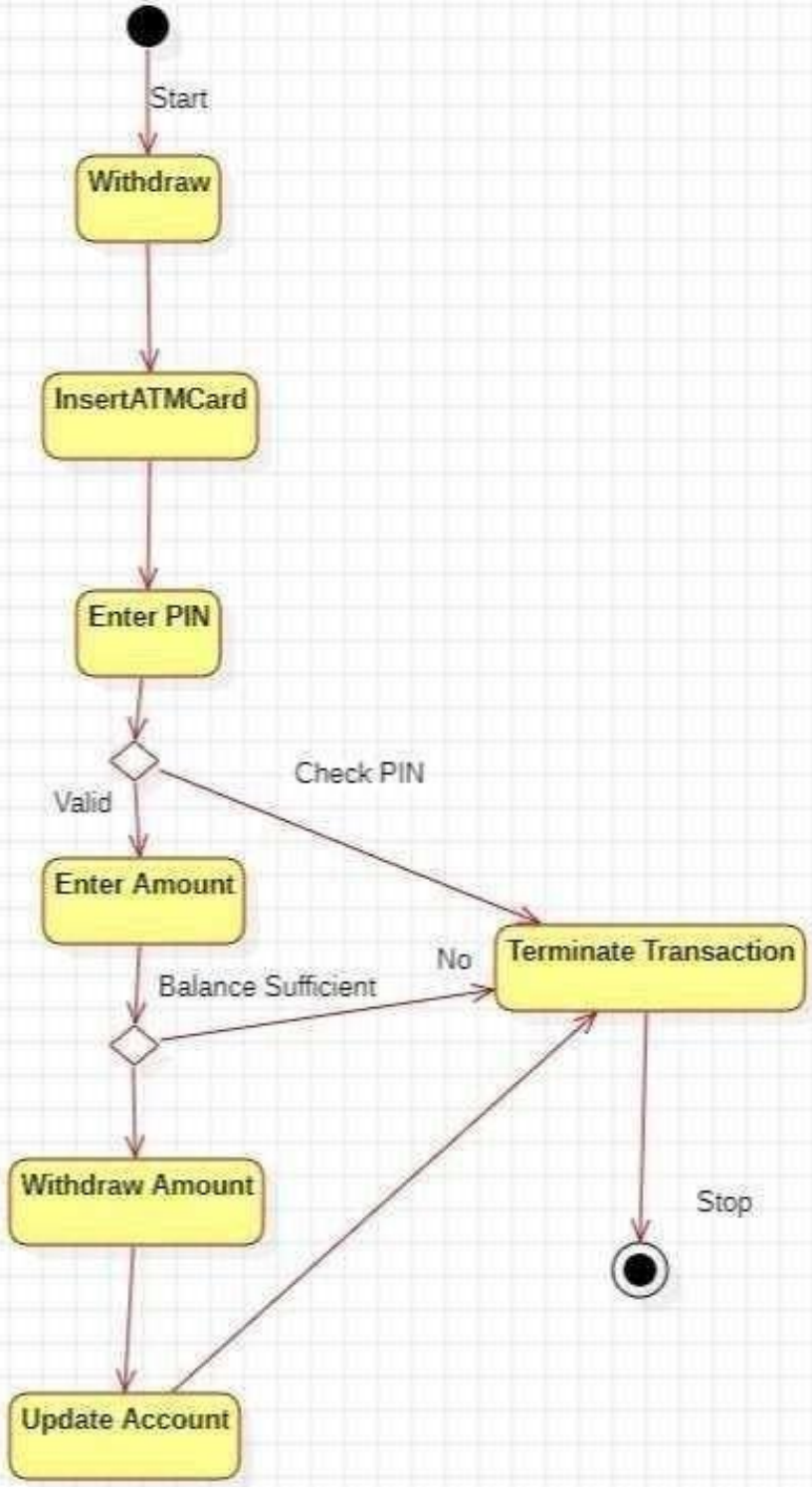
Ensure that the sequence accurately reflects the flow of interactions. Check for the correct order of messages and that all necessary interactions are included.

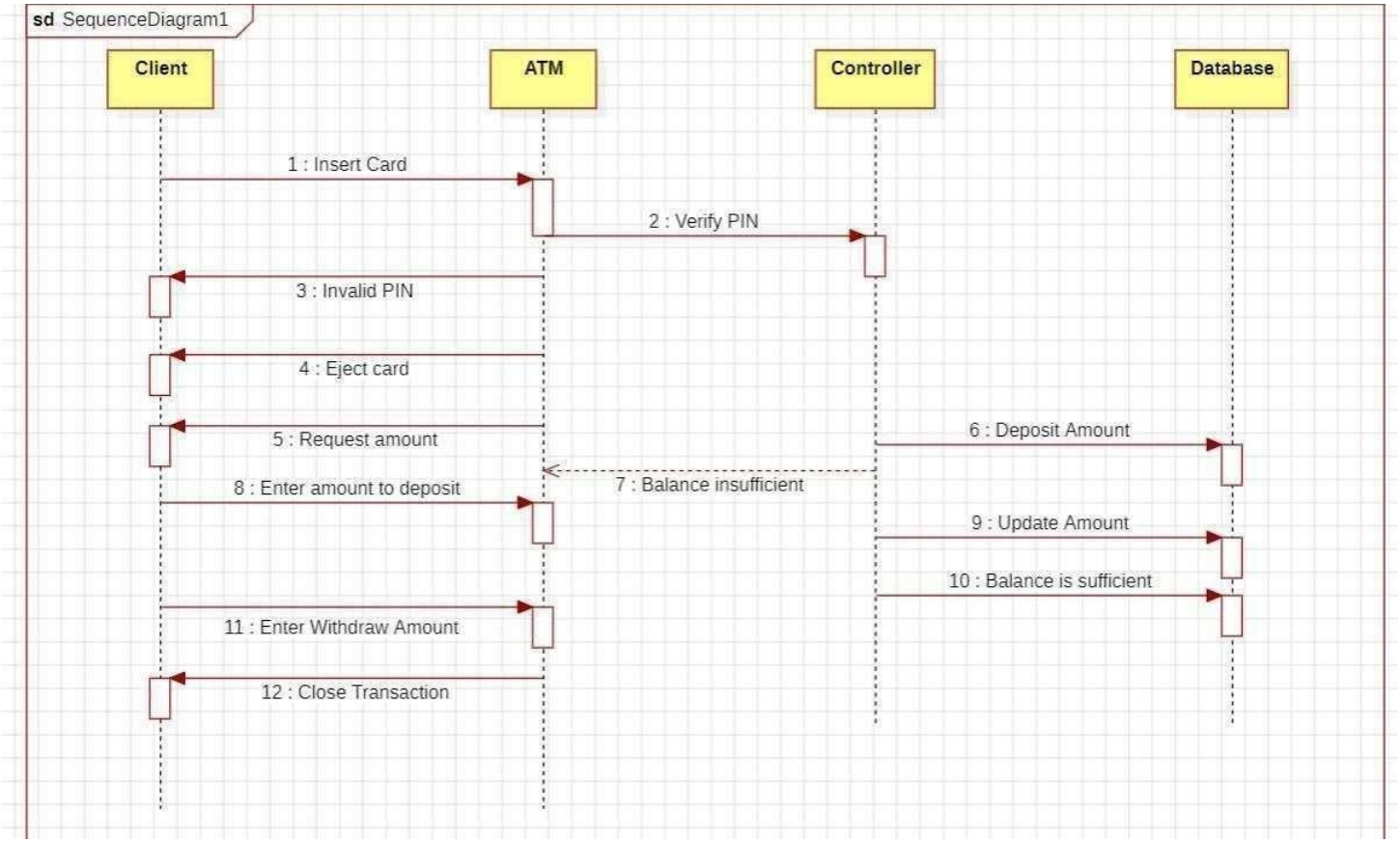

Deposit Amount
CheckPANcard
ATM
Withdraw Amount
Review Details
Client
System
Update Details
TerminateTransaction
Cancel Transaction

CLASS DIAGRAM:



Activity Diagram :




sd SequenceDiagram1
Client
ATM
Controller
Database
1 : Insert Card
2 : Verify PIN
3 : Invalid PIN
4 : Eject card
5 : Request amount
6 : Deposit Amount
7 : Balance insufficient
8 : Enter amount to deposit
9 : Update Amount
10 : Balance is sufficient
11 : Enter Withdraw Amount
12 : Close Transaction

RESULT:

PAYROLL MANAGEMENT SYSTEM

Ex No :

Date :

AIM

ALGORITHM

USE CASE DIAGRAM

Step 1: Identify Actors

- Identify the external entities interacting with the system:
  - **Employee**
  - **Administrator**

Step 2: Identify Use Cases

- List out the primary use cases in the system:
  - Employee Information
  - Time Card Information
  - Purchase of Resources
  - Payment Method Details
  - Reports
  - Management and Updation
  - Payroll Generation

Step 3: Define Relationships

- Define which actor interacts with which use case:
  - **Employee** interacts with:
    - Employee Information
    - Time Card Information
    - Purchase of Resources
    - Payment Method Details
    - Reports
  - **Administrator** interacts with:
    - Employee Information
    - Management and Updation
    - Reports
    - Payroll Generation

Step 4: Draw the Diagram

- Place actors on the diagram and position use cases in the center.
- Draw lines connecting actors to their use cases.

Step 5: Refine and Add Details

- Include relationships like «extend» and «include» where appropriate for refining use cases.

CLASS DIAGRAM

Step 1: Identify Key Classes

- List out main classes based on your use cases:
  - **Employee**
  - **TimeCard**
  - **Purchase**
  - **PaymentMethod**
  - **Report**
  - **Payroll**

Step 2: Define Class Attributes and Methods

- Define attributes and methods for each class:
  - **Employee**:
    - Attributes: employeeID, name, salary
    - Methods: getDetails(), updateDetails()

Step 3: Determine Relationships

- Identify relationships such as inheritance or association:
  - **Employee** associates with **TimeCard**, **Purchase**, and **PaymentMethod**.
  - **Administrator** manages **Payroll** and **Reports**.

Step 4: Draw the Diagram

- Draw classes as boxes with attributes and methods.
- Add lines to show associations, inheritance, or multiplicity between classes.

Step 5: Refine with Multiplicity and Associations

- Show multiplicity (e.g., "1 Employee to many TimeCards") and label the relationships.

ACTIVITY DIAGRAM

Step 1: Choose a Process

- Pick a specific process like **Employee Submits Time Card**.

Step 2: Define Key Activities

- Break down the process into actions:
  - Log in
  - Enter Time Card
  - Verify Time Card
  - Save Time Card
  - Log out

Step 3: Sequence the Activities

- Organize the activities in a logical order from start to finish.

Step 4: Draw the Diagram

- Draw action nodes for each activity.
- Connect them with arrows to show the flow of actions.

Step 5: Add Decision Points and End

- Add decision nodes (e.g., verification of time card) and finish with an end node.

SEQUENCE DIAGRAM

Step 1: Pick a Scenario

- Select a scenario such as **Employee Requests Payroll Details**.

Step 2: Identify Objects

- Identify objects involved:
  - **Employee**, **PayrollSystem**, **Database**

Step 3: Define Message Flow

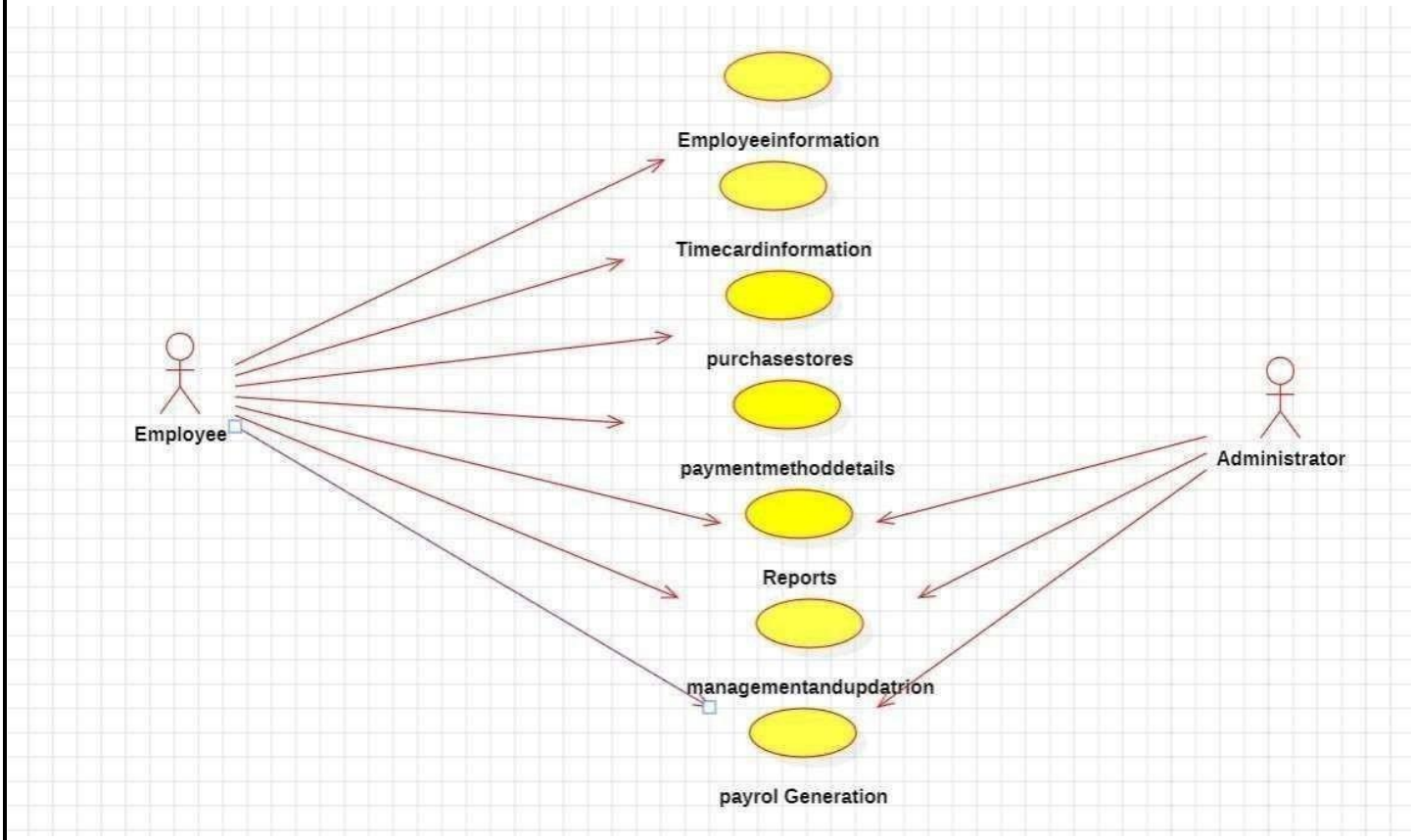
- Sequence the interactions between objects:
  - Employee -> PayrollSystem: requestPayrollDetails()
  - PayrollSystem -> Database: getPayrollDetails(employeeID)
  - Database -> PayrollSystem: returnPayrollDetails()

Step 4: Draw Lifelines and Messages

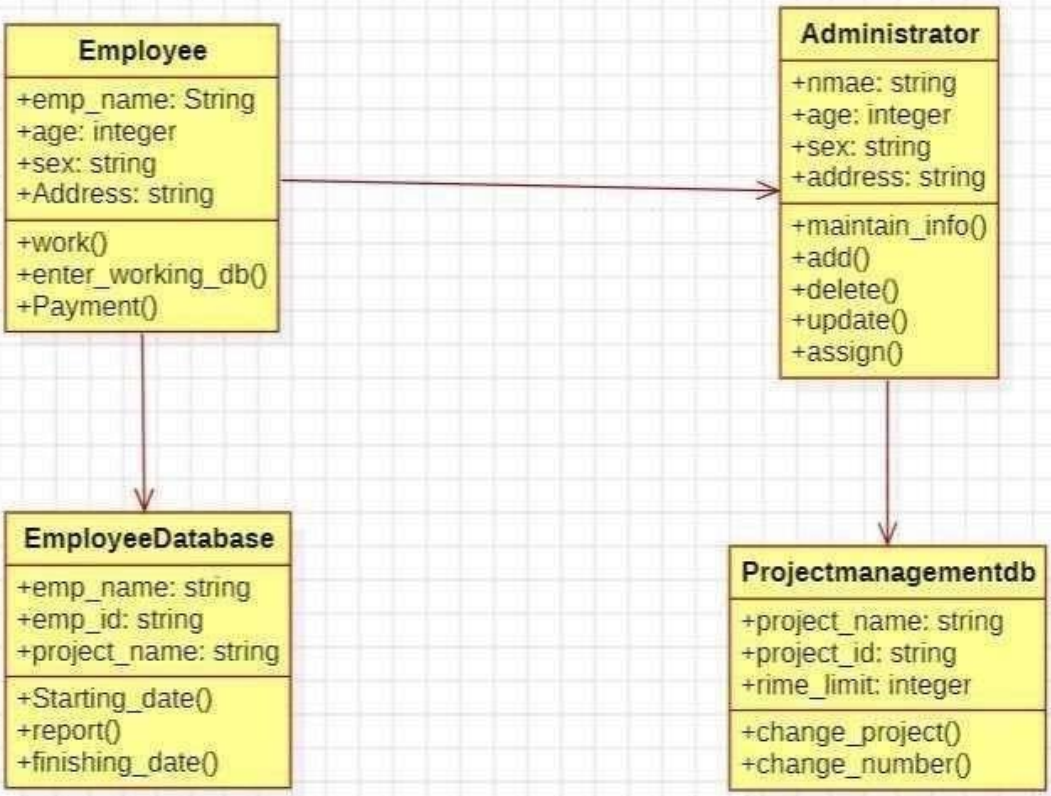
- Draw the objects and lifelines vertically, with arrows representing the messages exchanged.

Step 5: Add Activation Bars

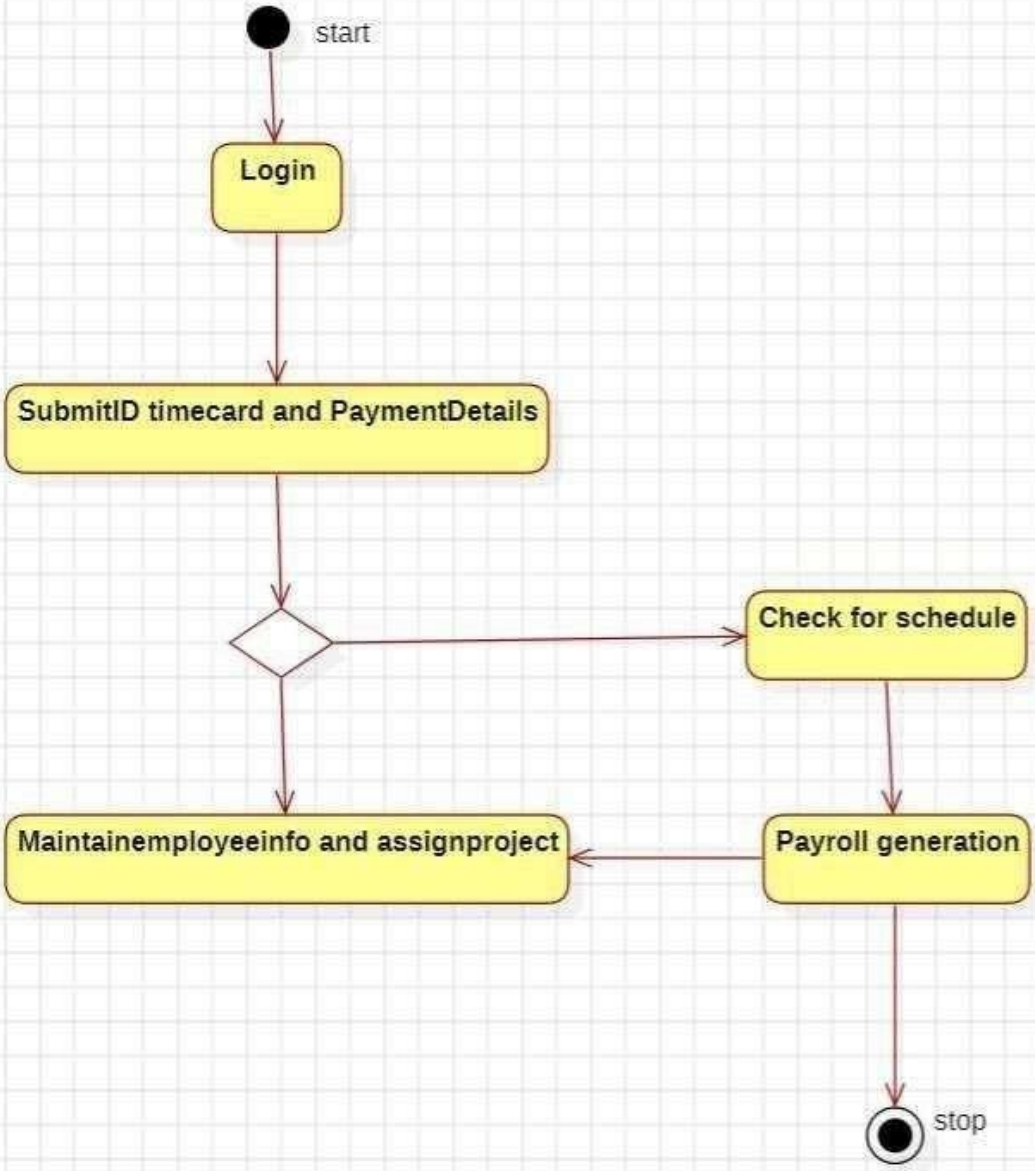
- Add activation bars to show when an object is active and processing a request

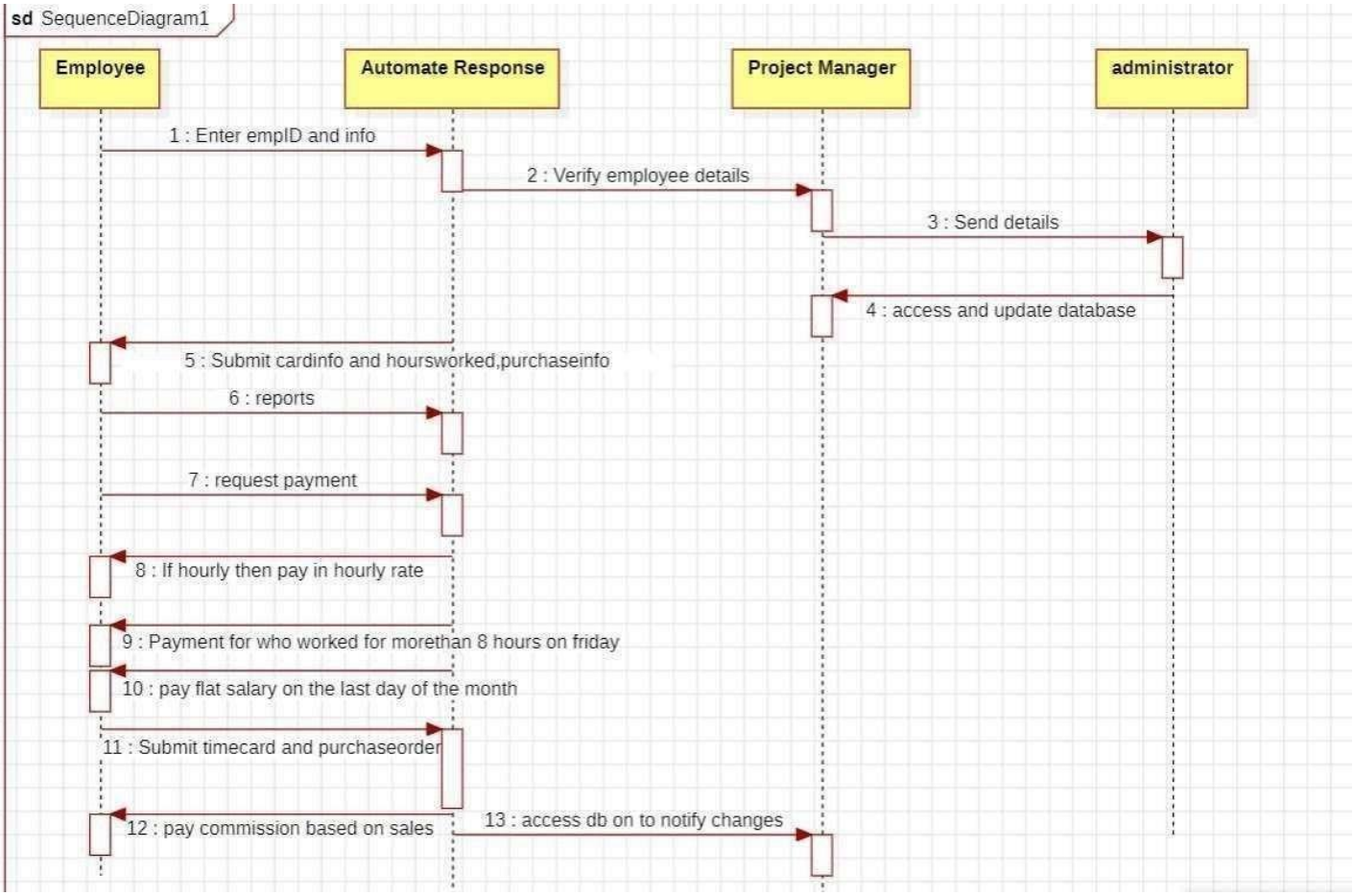

Employeeinformation
Timecardinformation
purchasestores
paymentmethoddetails
Reports
managementandupdatrion
payrol Generation
Employee
Administrator

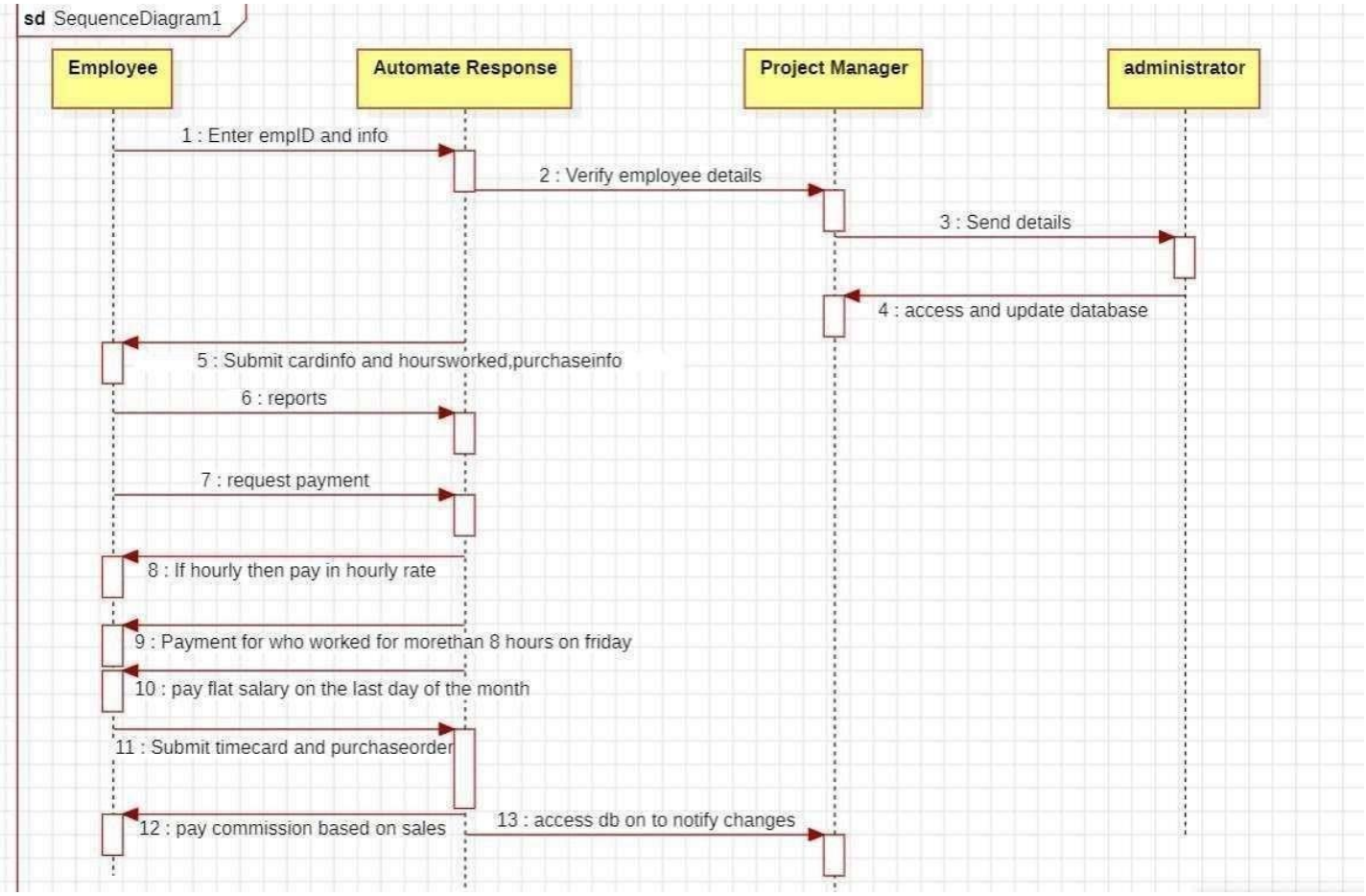
Class Diagram:

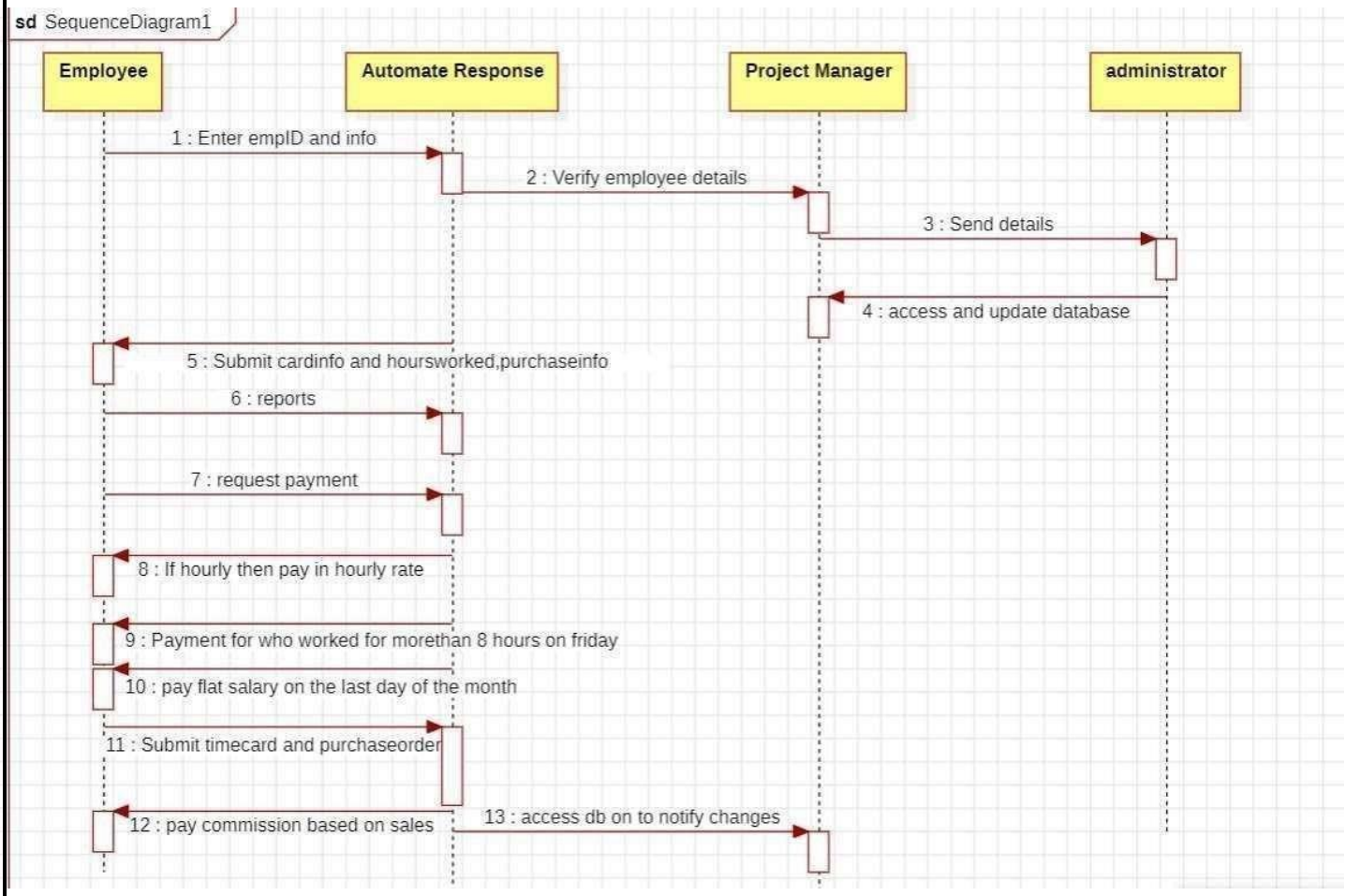


Activity Diagram:




sd SequenceDiagram1
Employee
Automate Response
Project Manager
administrator
1 : Enter empID and info
2 : Verify employee details
3 : Send details
4 : access and update database
5 : Submit cardinfo and hoursworked,purchaseinfo
6 : reports
7 : request payment
8 : If hourly then pay in hourly rate
9 : Payment for who worked for morethan 8 hours on friday
10 : pay flat salary on the last day of the month
11 : Submit timecard and purchaseorder
12 : pay commission based on sales
13 : access db on to notify changes


sd SequenceDiagram1
Employee
Automate Response
Project Manager
administrator
1 : Enter empID and info
2 : Verify employee details
3 : Send details
4 : access and update database
5 : Submit cardinfo and hoursworked,purchaseinfo
6 : reports
7 : request payment
8 : If hourly then pay in hourly rate
9 : Payment for who worked for morethan 8 hours on friday
10 : pay flat salary on the last day of the month
11 : Submit timecard and purchaseorder
12 : pay commission based on sales
13 : access db on to notify changes


sd SequenceDiagram1
Employee
Automate Response
Project Manager
administrator
1 : Enter empID and info
2 : Verify employee details
3 : Send details
4 : access and update database
5 : Submit cardinfo and hoursworked,purchaseinfo
6 : reports
7 : request payment
8 : If hourly then pay in hourly rate
9 : Payment for who worked for morethan 8 hours on friday
10 : pay flat salary on the last day of the month
11 : Submit timecard and purchaseorder
12 : pay commission based on sales
13 : access db on to notify changes

RESULT:

QUIZ SYSTEM

Ex No :

Date :

AIM

ALGORITHM

USE CASE DIAGRAM

Step 1: Identify Actors

- **Actors:**
  - Participant
  - Quiz Master

Step 2: Identify Use Cases

- **Use Cases:**
  - Registration
  - Prepare Questions
  - Participation
  - Scoring
  - Report

Step 3: Define Relationships

- **Relationships:**
  - Participant interacts with:
    - Registration (create account)
    - Participation (take quiz)
  - Quiz Master interacts with:
    - Prepare Questions (create/edit questions)
    - Scoring (evaluate quiz)
    - Report (generate results)

Step 4: Draw Diagram

- Create a visual representation:
  - Place actors outside the system boundary.
  - List use cases inside the boundary.
  - Connect actors to relevant use cases with lines.

Step 5: Review and Refine

- Ensure all use cases and relationships are accurately represented. Check for any missing interactions or actors.

CLASS DIAGRAM

Step 1: Identify Classes

- **Classes:**
  - Participant
  - QuizMaster
  - Quiz
  - Question
  - Score
  - Report

Step 2: Define Attributes and Methods

- **Participant:**
  - Attributes: name, email
  - Methods: register(), participate()
- **QuizMaster:**
  - Attributes: name, email
  - Methods: prepareQuestions(), scoreQuiz(), generateReport()
- **Quiz:**
  - Attributes: title, questions[]
  - Methods: start(), end()
- **Question:**
  - Attributes: text, options[], correctAnswer
  - Methods: validateAnswer()
- **Score:**
  - Attributes: participantId, quizId, points
  - Methods: calculateScore()
- **Report:**
  - Attributes: quizId, results[]
  - Methods: generateReport()

Step 3: Define Relationships

- **Relationships:**
  - Participant (1) ↔ (0..*) Score
  - QuizMaster (1) ↔ (0..*) Quiz
  - Quiz (1) ↔ (0..*) Question
  - Score (1) ↔ (1) Participant and Quiz

Step 4: Draw Diagram

- Represent each class with a box, showing attributes and methods. Use lines to depict relationships, including multiplicity.

Step 5: Review and Refine

- Verify that all necessary classes, attributes, methods, and relationships are included and accurate.

ACTIVITY DIAGRAM

Step 1: Identify Activities

- **Activities:**
  - Registration
  - Prepare Questions
  - Participation
  - Scoring
  - Reporting

Step 2: Define Flow

- Example flow for Registration:
  - Start → Input Info → Validate → Complete Registration

Step 3: Identify Decision Points

- Example: Validation checks during registration and question preparation.

Step 4: Draw Diagram

- Use UML symbols to represent activities (rounded rectangles), decisions (diamonds), and flows (arrows) to visualize the process.

Step 5: Review and Refine

- Ensure the flow accurately represents all activities and decisions, checking for logical progression.

SEQUENCE DIAGRAM

Step 1: Identify Objects

- **Objects:**
  - Participant
  - QuizMaster
  - Quiz
  - Question
  - Score
  - Report

Step 2: Define Interactions

- Example interactions:
  - Participant registers → QuizMaster confirms.
  - QuizMaster prepares questions → System saves questions.
  - Participant answers questions → System submits answers.
  - QuizMaster calculates scores → System saves scores.
  - QuizMaster generates report → System displays report.

Step 3: Arrange Lifelines

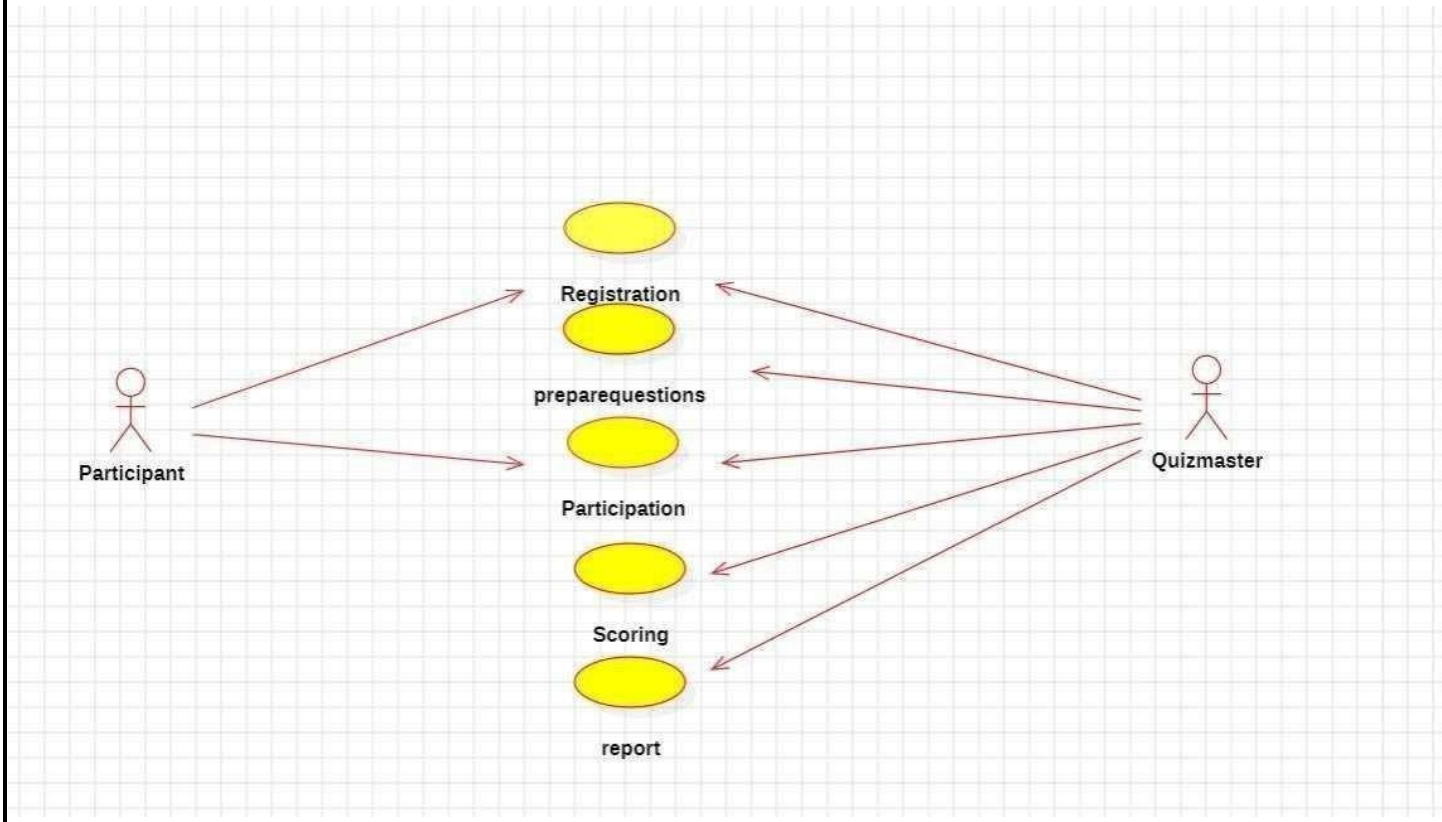
- Place objects horizontally and draw vertical dashed lines (lifelines) for each.

Step 4: Draw Messages

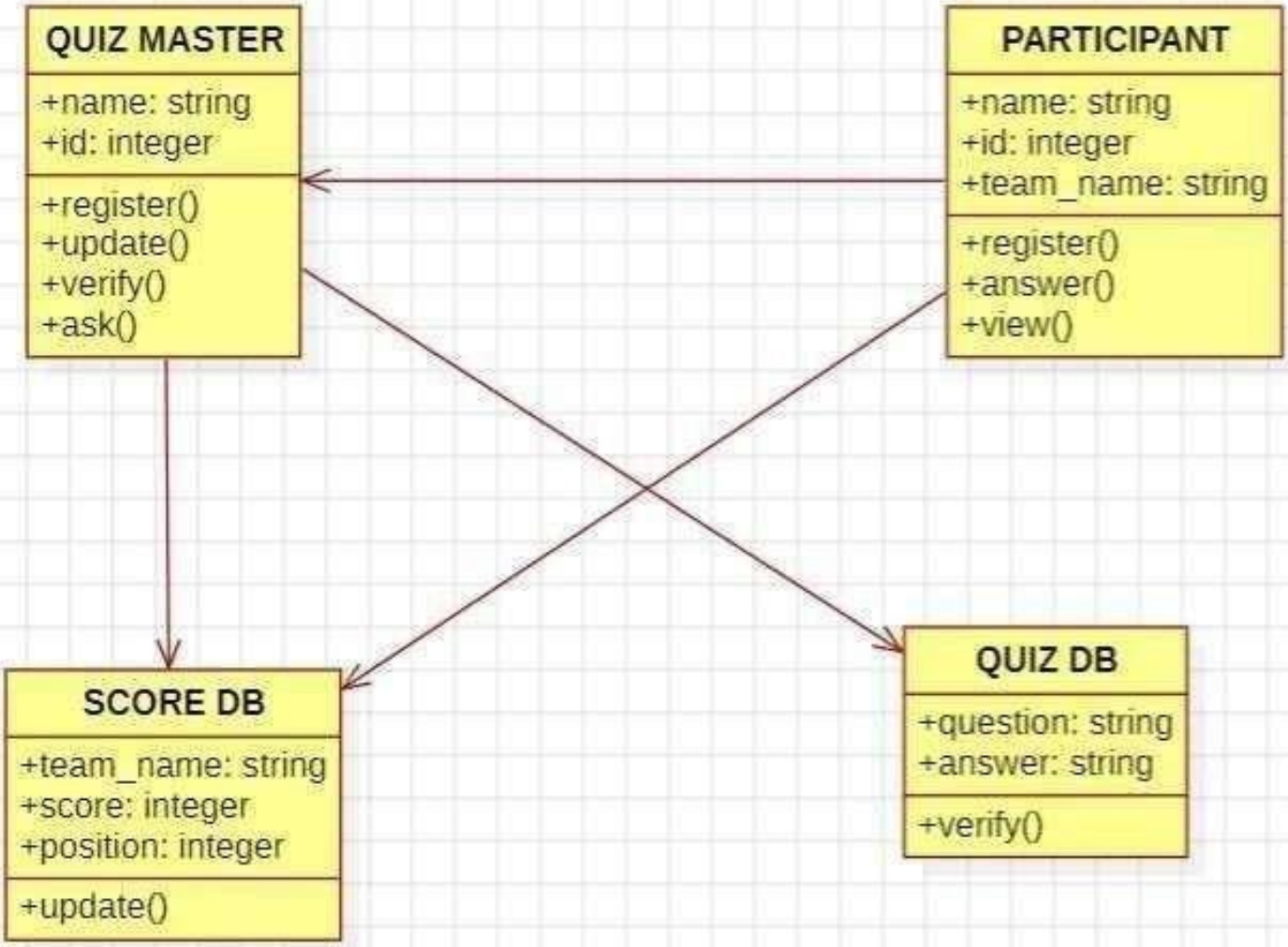
- Use arrows to represent messages exchanged between objects, indicating the order of interactions.

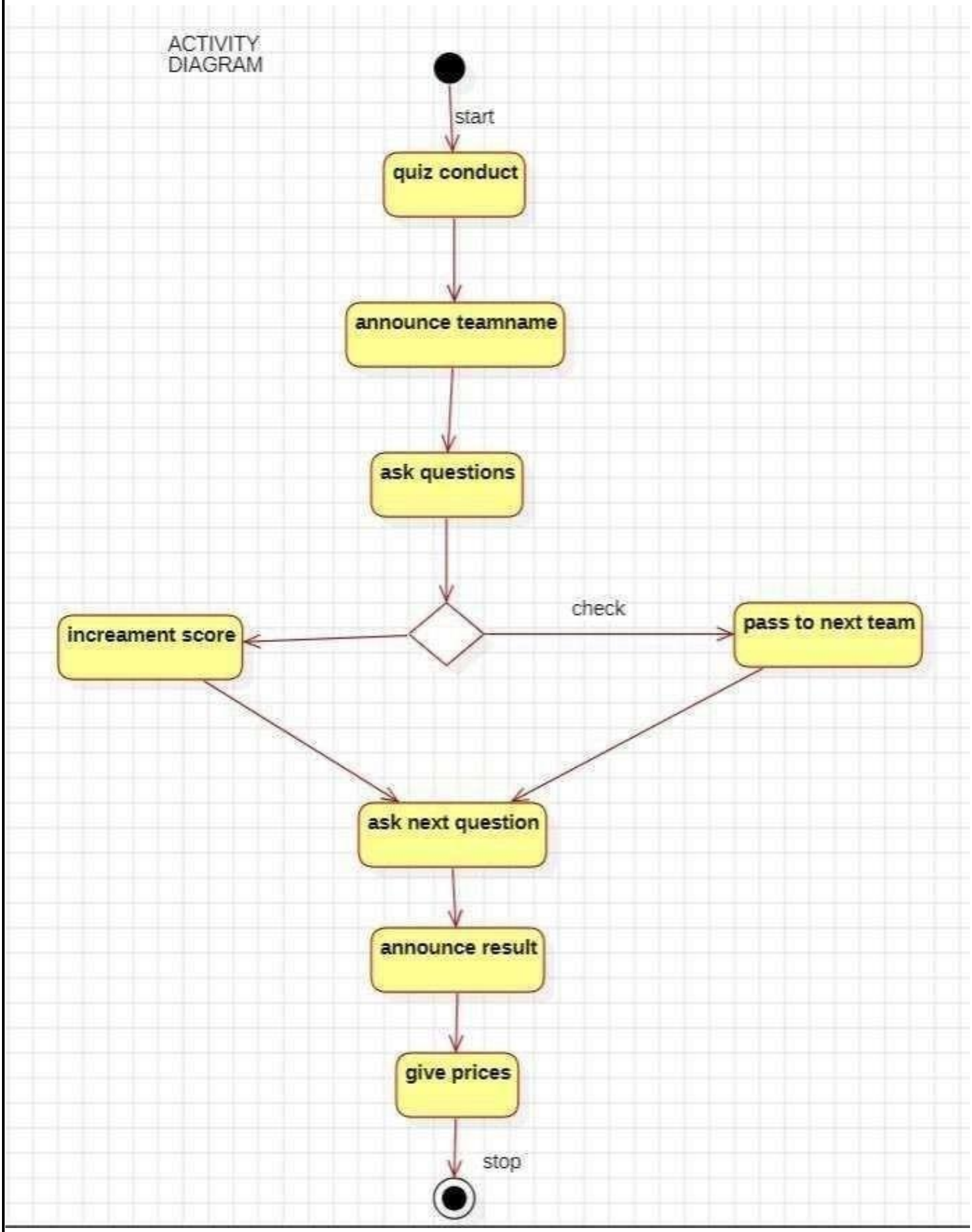
Step 5: Review and Refine

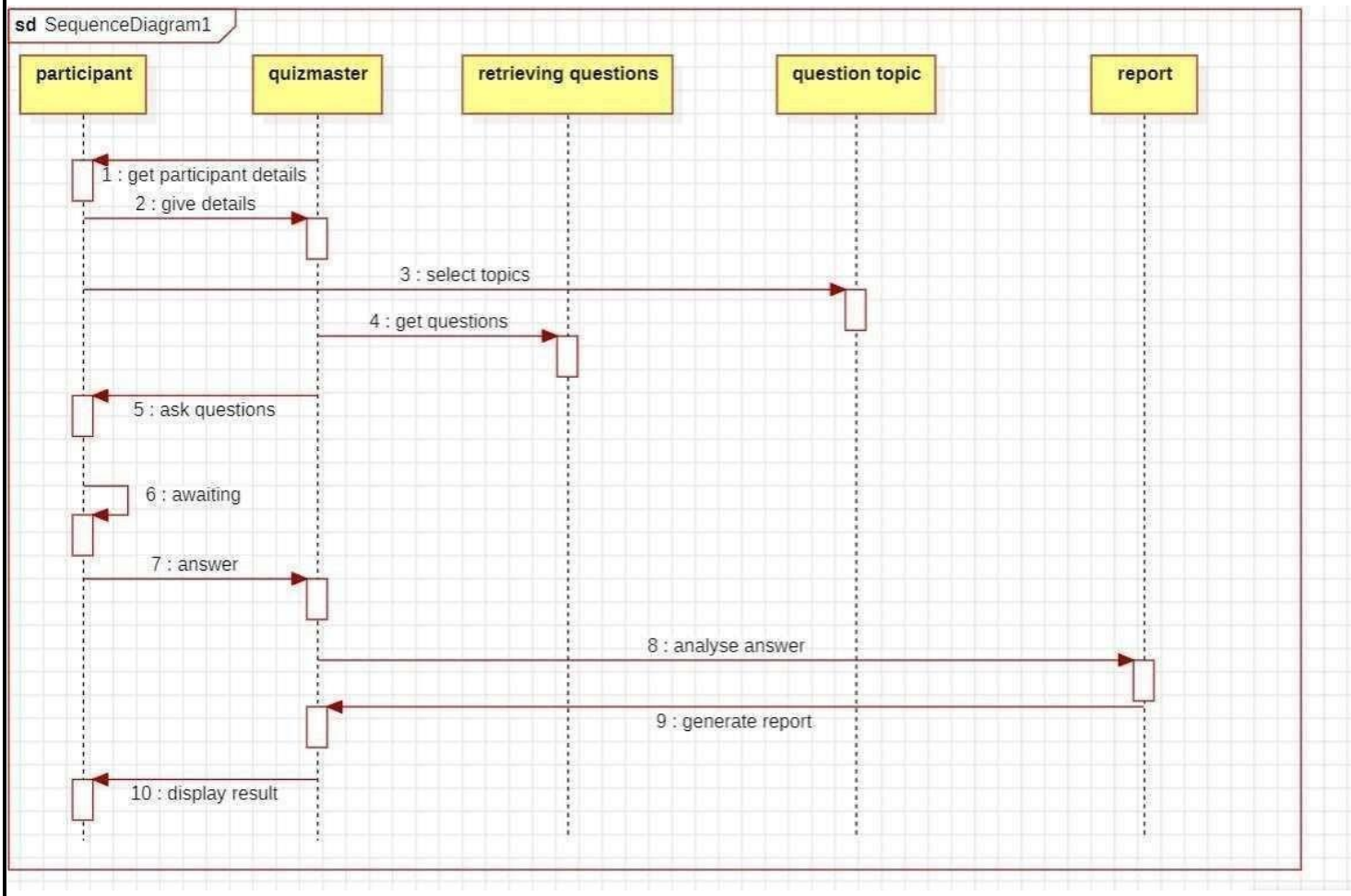
- Ensure that the sequence and timing of interactions make sense and are complete.


Registration
preparequestions
Participant
Quizmaster
Participation
Scoring
report

CLASS DIAGRAM:




ACTIVITY
DIAGRAM
start
quiz conduct
announce teamname
ask questions
check
increament score
pass to next team
ask next question
announce result
give prices
stop


sd SequenceDiagram1
participant
quizmaster
retrieving questions
question topic
report
1 : get participant details
2 : give details
3 : select topics
4 : get questions
5 : ask questions
6 : awaiting
7 : answer
8 : analyse answer
9 : generate report
10 : display result

RESULT:

STOCK MAINTENANCE SYSTEM

Ex No :

Date :

AIM

ALGORITHM

USE CASE DIAGRAM

Step 1: Identify Actors

- **Actors:**
  - Customer
  - Agent
  - Stock Person

Step 2: Identify Use Cases

- **Use Cases:**
  - Make Order
  - Collect Company's Customer Information
  - Check for Customer Records
  - Verify Product Order
  - Update Billing
  - Receive Packing Order
  - Retrieve Stock
  - Delivery

Step 3: Define Relationships

- **Relationships:**
  - Customer interacts with:
    - Make Order
    - Collect Company's Customer Information
    - Check for Customer Records
  - Agent interacts with:
    - Verify Product Order
    - Update Billing
    - Receive Packing Order
    - Delivery
  - Stock Person interacts with:
    - Retrieve Stock

Step 4: Draw Diagram

- Create a diagram:
  - Place actors outside the system boundary.
  - List use cases inside the boundary.
  - Connect actors to relevant use cases with lines.

Step 5: Review and Refine

- Ensure all use cases and relationships are accurately represented and check for any missing interactions.

CLASS DIAGRAM

Step 1: Identify Classes

- **Classes:**
  - Customer
  - Agent
  - StockPerson
  - Order
  - CustomerRecord
  - Billing
  - PackingOrder
  - Stock

Step 2: Define Attributes and Methods

- **Customer:**
  - Attributes: name, contactInfo
  - Methods: makeOrder(), collectCustomerInfo()
- **Agent:**
  - Attributes: name, agentId
  - Methods: verifyProductOrder(), updateBilling(), receivePackingOrder(), deliver()
- **StockPerson:**
  - Attributes: name, employeeId
  - Methods: retrieveStock()
- **Order:**
  - Attributes: orderId, productDetails, status
  - Methods: createOrder(), checkOrderStatus()
- **CustomerRecord:**
  - Attributes: customerId, recordDetails
  - Methods: checkRecords()
- **Billing:**
  - Attributes: billingId, amount
  - Methods: generateInvoice()
- **PackingOrder:**
  - Attributes: packingId, orderId
  - Methods: createPackingOrder()
- **Stock:**
  - Attributes: productId, quantity
  - Methods: updateStock(), checkStock(

Step 3: Define Relationships

- **Relationships:**
  - Customer (1) ↔ (0..*) Order
  - Agent (1) ↔ (0..*) PackingOrder
  - StockPerson (1) ↔ (0..*) Stock
  - Order (1) ↔ (1) CustomerRecord
  - Billing (1) ↔ (1) Order

Step 4: Draw Diagram

- Create class boxes with attributes and methods. Use lines to depict relationships, including multiplicity.

Step 5: Review and Refine

- Verify that all classes, attributes, methods, and relationships are included and accurate.

ACTIVITY DIAGRAM

Step 1: Identify Activities

- **Activities:**
  - Make Order
  - Collect Customer Information
  - Check Customer Records
  - Verify Product Order
  - Update Billing
  - Receive Packing Order
  - Retrieve Stock
  - Delivery

Step 2: Define Flow

- Example flow for Make Order:
  - Start → Input Order Details → Validate Order → Confirm Order → End

Step 3: Identify Decision Points

- Example: Validate order status, check stock availability.

Step 4: Draw Diagram

- Use UML symbols to represent activities (rounded rectangles), decisions (diamonds), and flows (arrows).

Step 5: Review and Refine

- Ensure the flow accurately captures all activities and decision points, checking for logical progression

SEQUENCE DIAGRAM

Step 1: Identify Objects

- **Objects:**
  - Customer
  - Agent
  - StockPerson
  - Agent Database
  - CustomerDatabase
  - Stock Database
  - Shipping Agents
  - Accountant Database
  - Purchase Database

Step 2: Define Interactions

- Example interactions:
  - Customer makes an order → Agent verifies product → Agent updates billing → StockPerson retrieves stock → Delivery is made.

Step 3: Arrange Lifelines

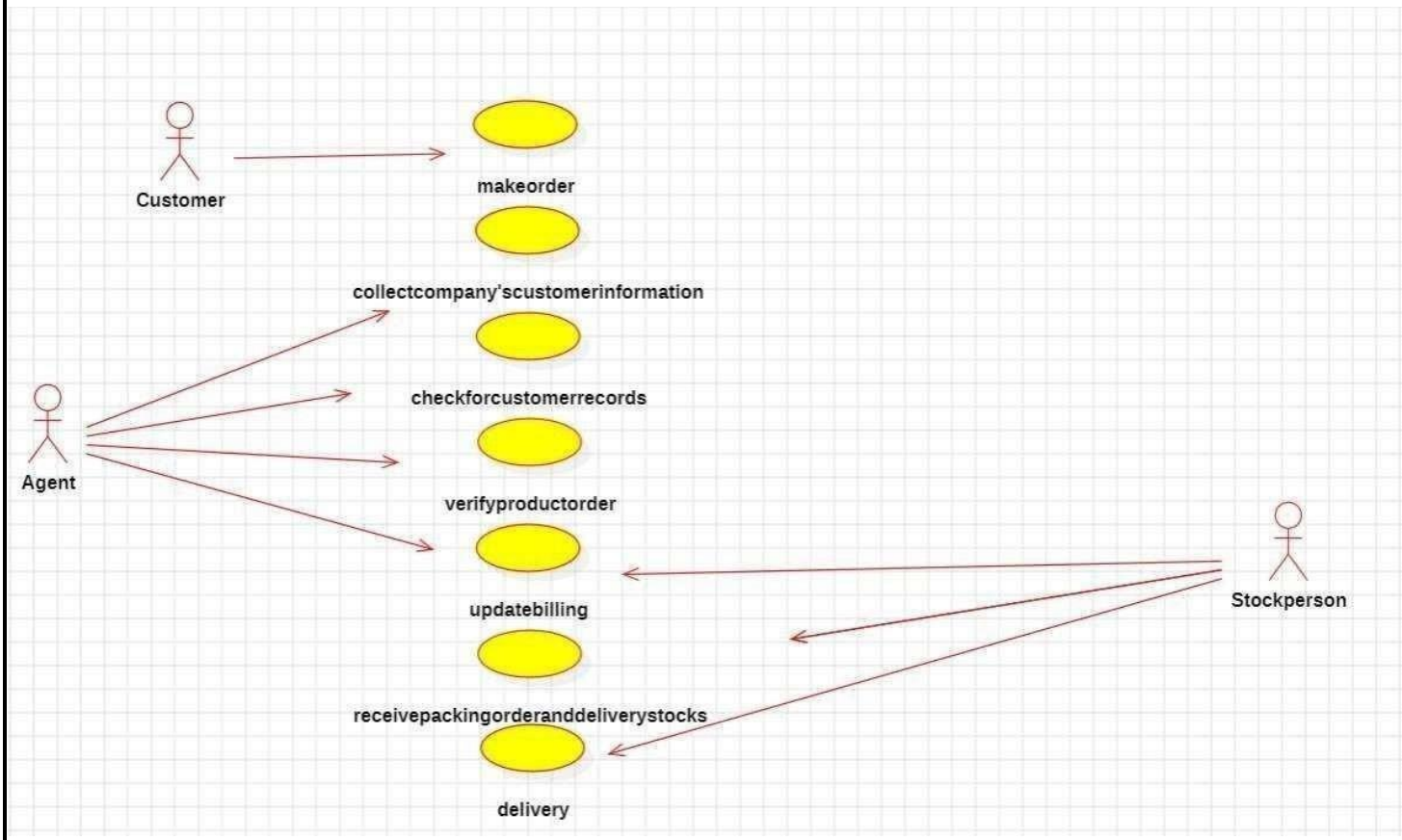
- Place objects horizontally, each with a vertical dashed line (lifeline).

Step 4: Draw Messages

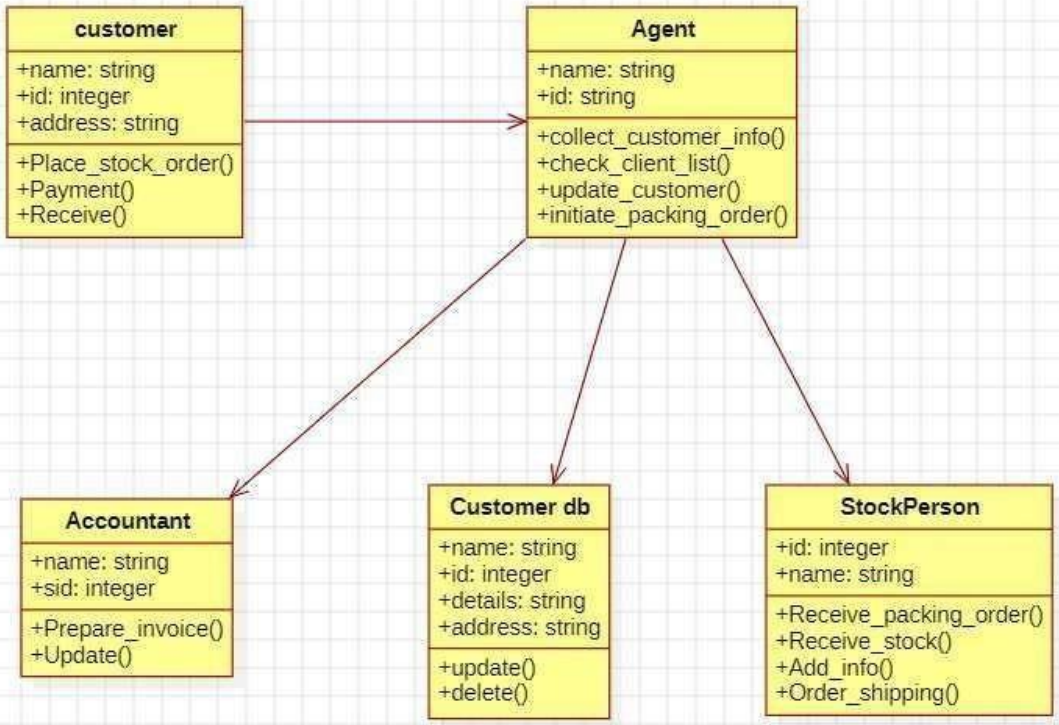
- Use arrows to represent messages exchanged between objects, indicating the order of interactions.

Step 5: Review and Refine

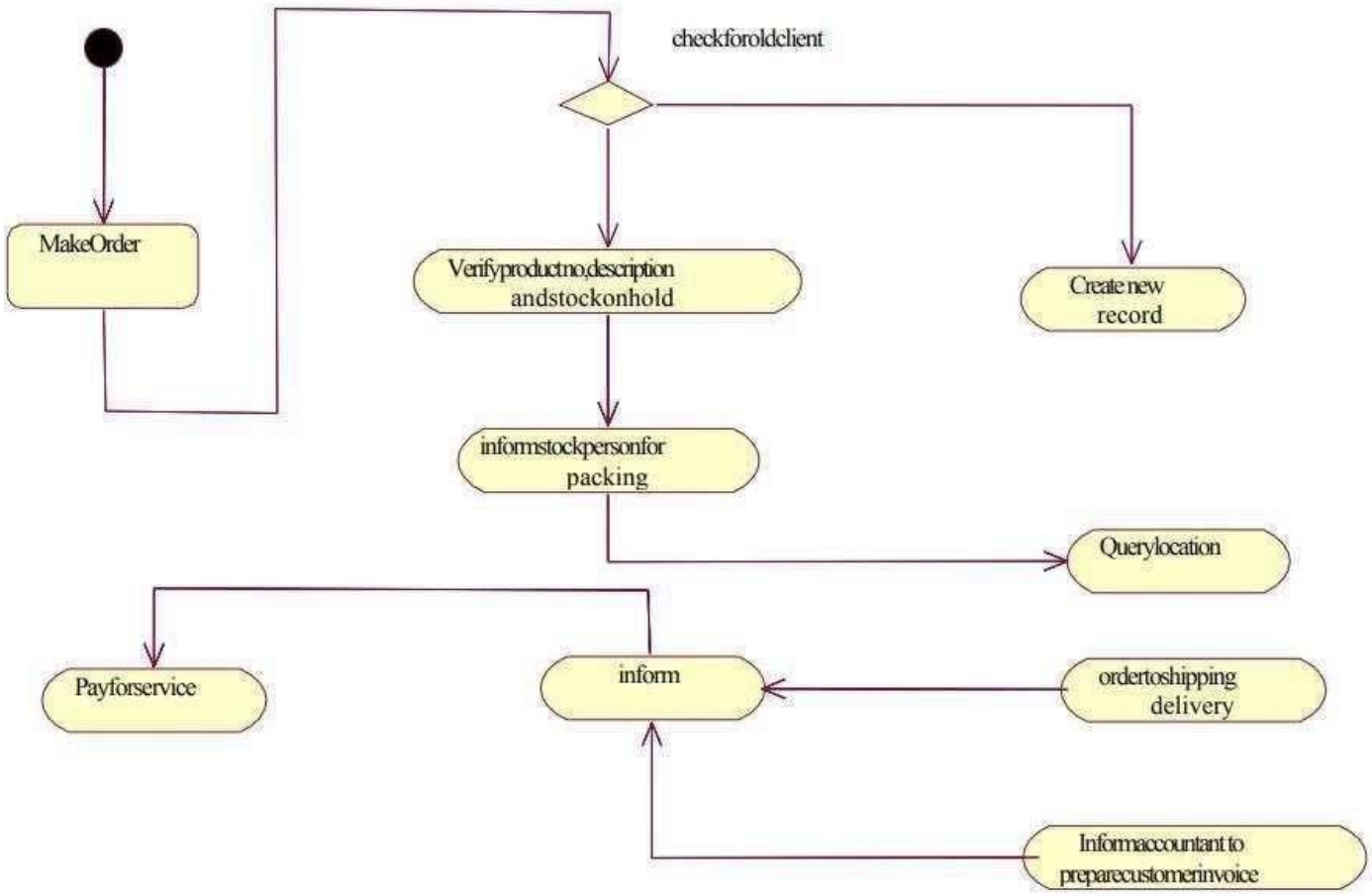
- Ensure that the sequence and timing of interactions are logical and complete.


Customer
makeorder
collectcompany'scustomerinformation
Agent
checkforcustomerrecords
verifyproductorder
Stockperson
updatebilling
receivepackingorderanddeliverystocks
delivery

CLASS DIAGRAM:



ACTIVITY DIAGRAM



RESULT:

PASSPORT REGISTRATION SYSTEM

Ex.No :

Date :

AIM

ALGORITHM

USE CASE DIAGRAM

Step 1: Identify Actors

- **Actors:**
  - Applicant
  - Administrator

Step 2: Identify Use Cases

- **Use Cases:**
  - Registration
  - Check Status
  - Process Application
  - Dispatch Passport

Step 3: Define Relationships

- **Relationships:**
  - Applicant interacts with:
    - Registration
    - Check Status
  - Administrator interacts with:
    - Process Application
    - Dispatch Passport

Step 4: Draw Diagram

- Create a diagram:
  - Place actors (Applicant, Administrator) outside the system boundary.
  - List use cases (Registration, Check Status, Process Application, Dispatch Passport) inside the boundary.
  - Connect actors to relevant use cases with lines.

Step 5: Review and Refine

- Ensure all use cases and relationships are accurately represented and check for any missing interactions.

CLASS DIAGRAM

Step 1: Identify Classes

- **Classes:**
  - Applicant
  - Administrator
  - Application
  - Passport
  - Status

Step 2: Define Attributes and Methods

- **Applicant:**
  - Attributes: `name`, `contactInfo`, `applicationId`
  - Methods: `register()`, `checkStatus()`
- **Administrator:**
  - Attributes: `name`, `adminId`
  - Methods: `processApplication()`, `dispatchPassport()`
- **Application:**
  - Attributes: `applicationId`, `applicantId`, `status`
  - Methods: `submitApplication()`, `updateStatus()`
- **Passport:**
  - Attributes: `passportId`, `applicantId`, `issueDate`, `expiryDate`
  - Methods: `generatePassport()`
- **Status:**
  - Attributes: `statusId`, `applicationId`, `currentStatus`
  - Methods: `getStatus()`

Step 3: Define Relationships

- **Relationships:**
  - Applicant (1) ↔ (0..*) Application
  - Administrator (1) ↔ (0..*) Application
  - Application (1) ↔ (1) Passport
  - Application (1) ↔ (1) Status

Step 4: Draw Diagram

- Create class boxes with attributes and methods. Use lines to depict relationships, including multiplicity.

Step 5: Review and Refine

- Verify that all classes, attributes, methods, and relationships are included and accurate.

ACTIVITY DIAGRAM

Step 1: Identify Activities

- **Activities:**
  - Login
  - Registration
  - Application Form
  - Verification
  - Enquiry
  - Payment
  - Getting Passport

Step 2: Define Flow

- Example flow for Registration:
  - Start → Fill Application Form → Submit Application → Confirmation

Step 3: Identify Decision Points

- Example: Check if application is complete before processing.

Step 4: Draw Diagram

- Use UML symbols to represent activities (rounded rectangles), decisions (diamonds), and flows (arrows).

Step 5: Review and Refine

- Ensure the flow accurately captures all activities and decision points, checking for logical progression.

SEQUENCE DIAGRAM

Step 1: Identify Objects

- **Objects:**
  - Administrator
  - System
  - Admin Panel
  - Database
  - Applicant

Step 2: Define Interactions

- Example interactions:
  - Applicant registers → Administrator processes application → Status is checked → Passport is dispatched.

Step 3: Arrange Lifelines

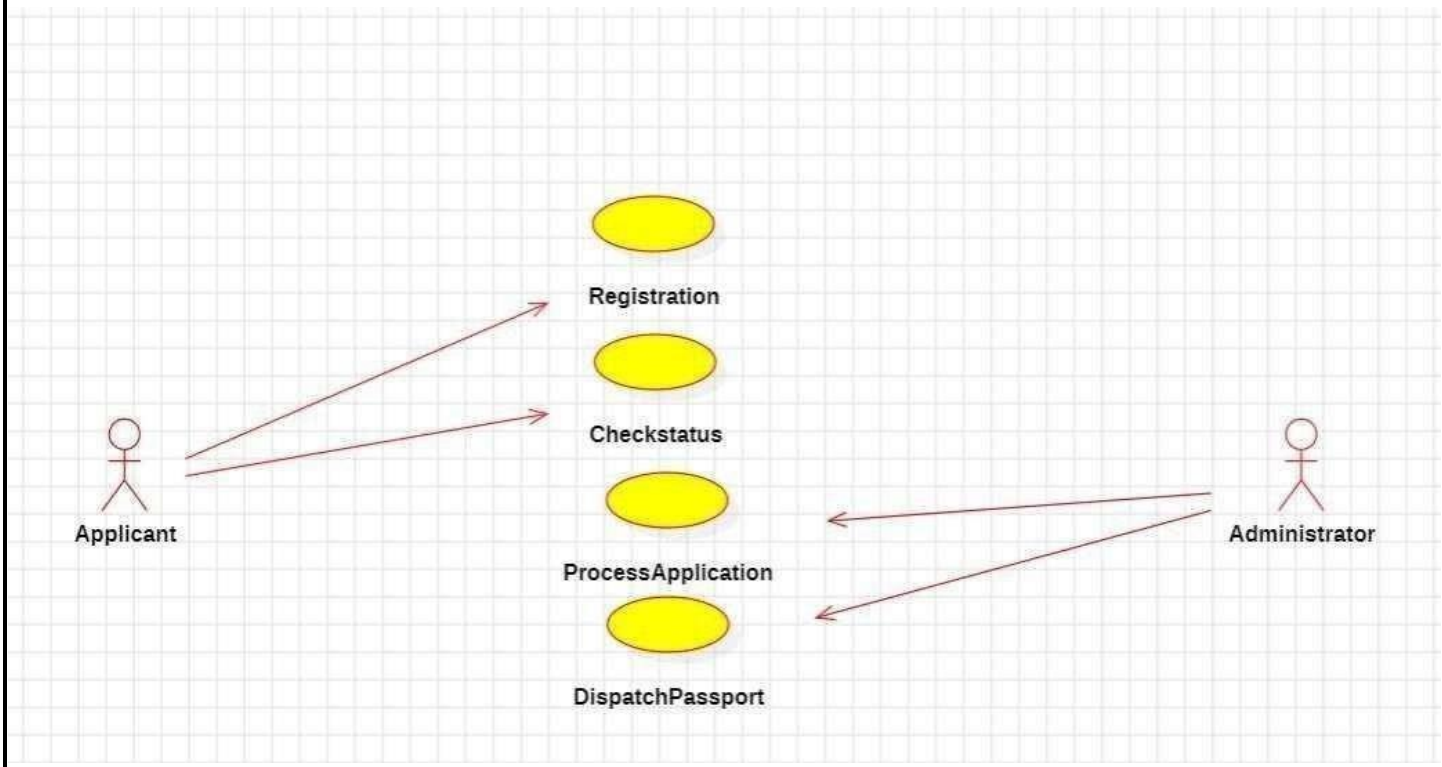
- Place objects horizontally, each with a vertical dashed line (lifeline).

Step 4: Draw Messages

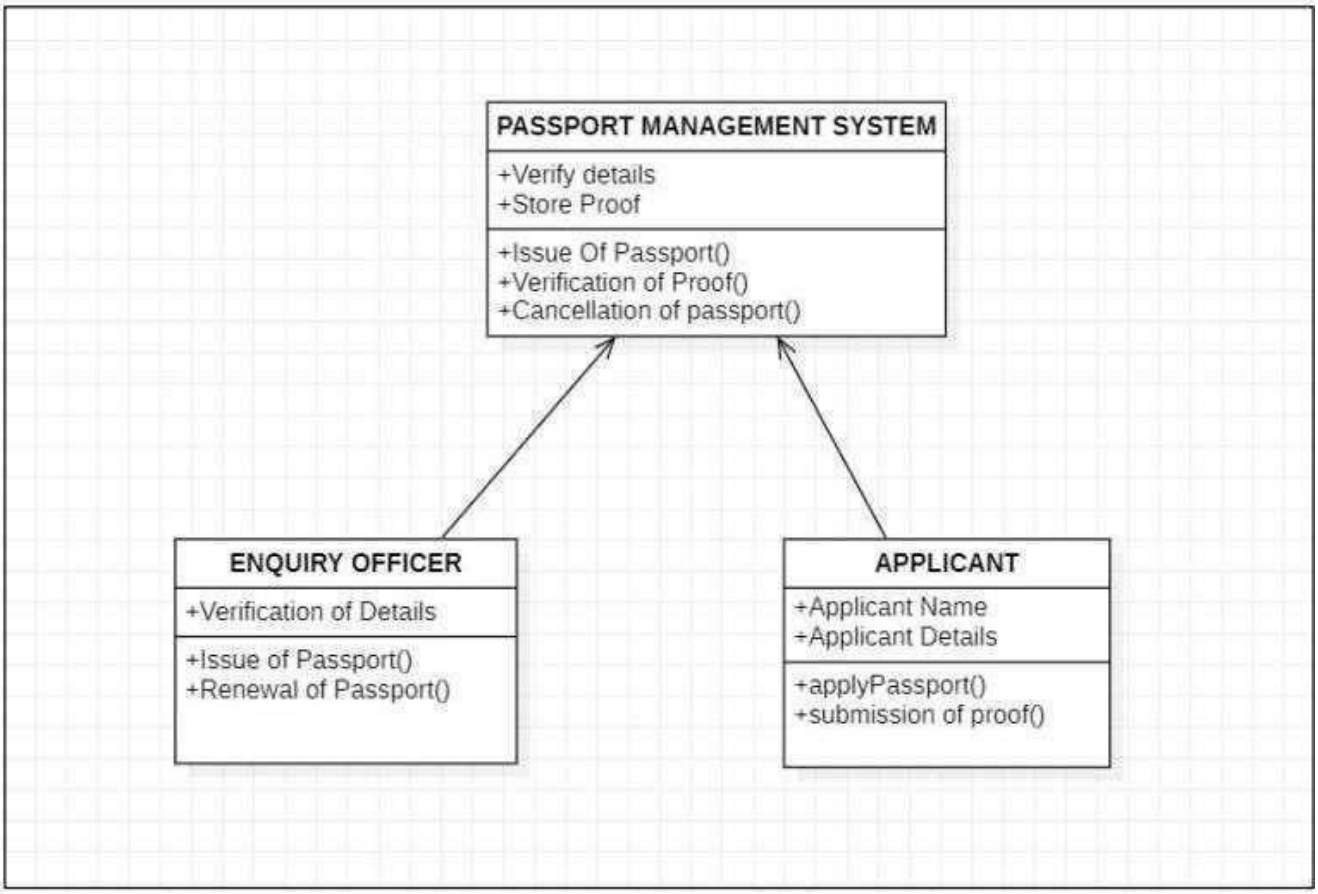
- Use arrows to represent messages exchanged between objects, indicating the order of interactions.

Step 5: Review and Refine

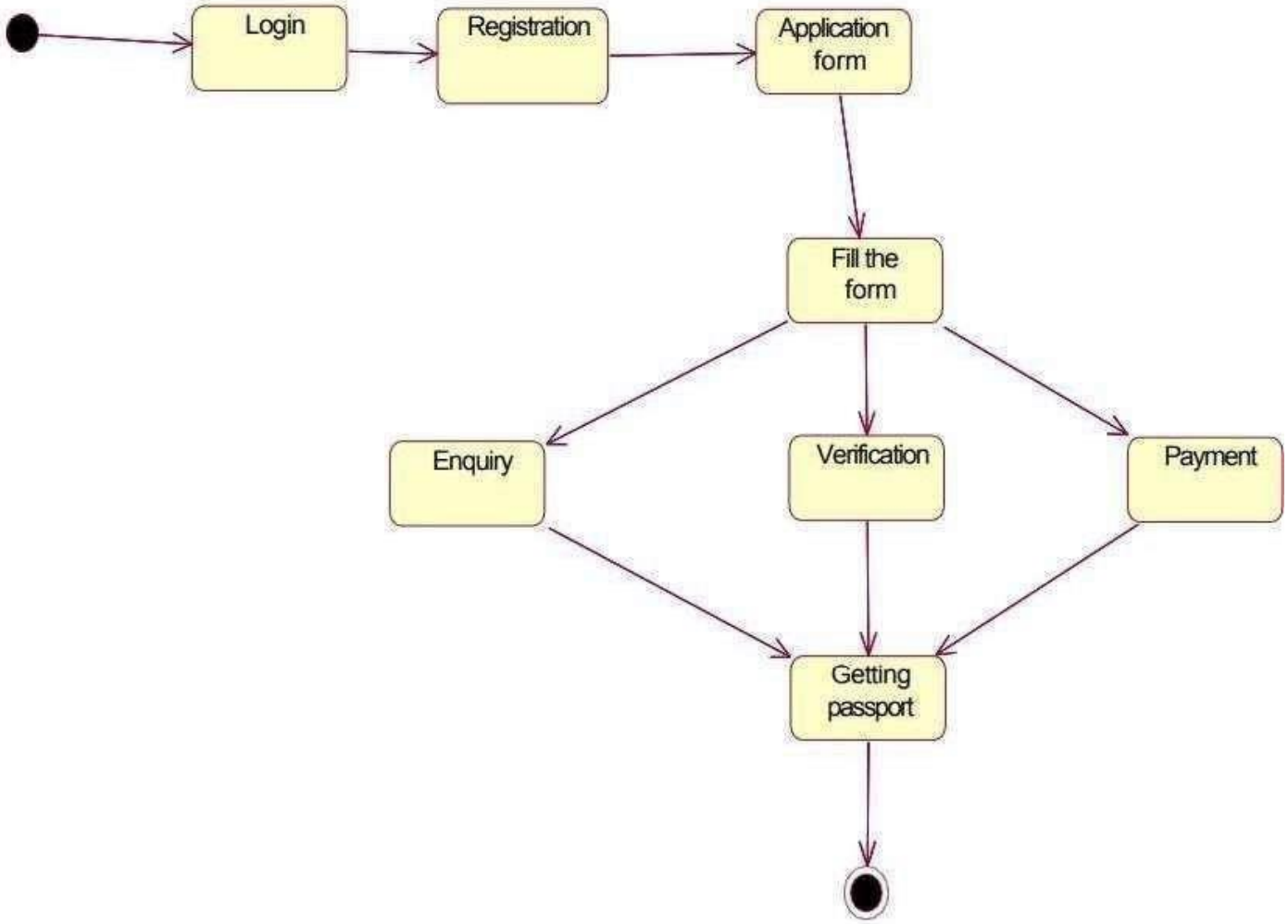
- Ensure that the sequence and timing of interactions are logical and complete.

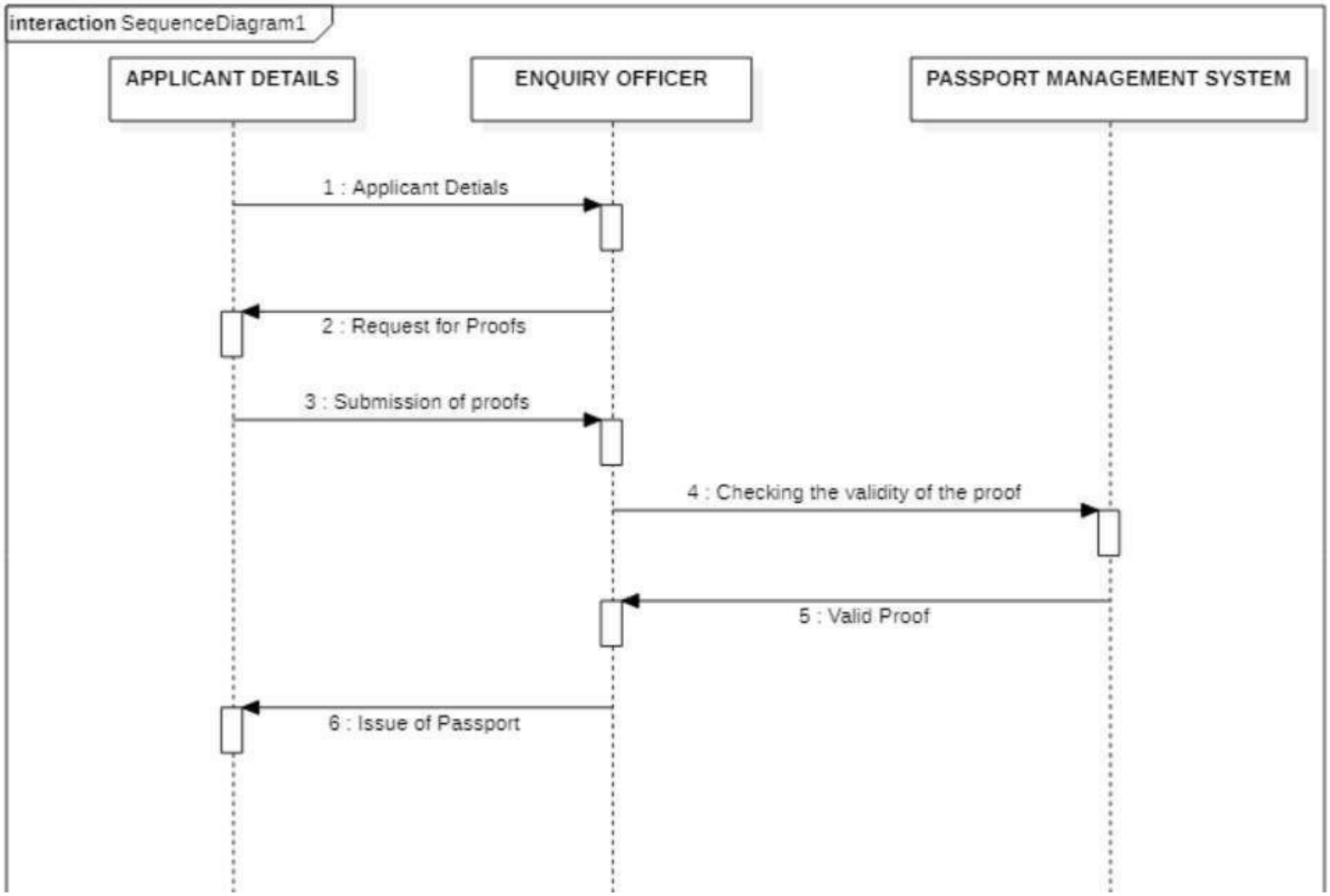

Registration
Checkstatus
Applicant
Administrator
ProcessApplication
DispatchPassport

Class Diagram :



ACTIVITY DIAGRAM




interaction SequenceDiagram1
APPLICANT DETAILS
ENQUIRY OFFICER
PASSPORT MANAGEMENT SYSTEM
1 : Applicant Detials
2 : Request for Proofs
3 : Submission of proofs
4 : Checking the validity of the proof
5 : Valid Proof
6 : Issue of Passport

RESULT:

STUDENT MARK ANALYSIS SYSTEM

Ex No:

Date:

AIM

ALGORITHM

USE CASE DIAGRAM

Step 1: Identify the Actors

- **Student:** Views grades and subjects.
- **Staff:** Manages students, subjects, marks, and assigns grades**.**

Step 2: Identify Use Cases

- **Student Use Cases:**
  - View subjects
  - View marks
  - View total marks
  - View grade
- **Staff Use Cases:**
  - Add/update subjects
  - Enter marks
  - Calculate total marks
  - Assign grade
  - Manage student records (add/remove students)

Step 3: Define Relationships

- **Student interacts with:**
  - View subjects, marks, total marks, and grade.
- **Staff interacts with:**
  - Add subjects, enter marks, calculate total, assign grades, and manage students.

Step 4: Draw the Diagram

- Draw Student and Staff as actors.
- Add use cases in ellipses inside the system boundary.
- Connect the actors to their related use cases.

Step 5: Add Relationships

- Use include between related use cases, e.g., “Calculate total” includes “Enter marks.”
- Use extend for optional use cases, e.g., “Assign grade” extends “Calculate total.”

CLASS DIAGRAM

Step 1: Identify the Classes

- **Classes:** Student, Staff, Subject, Marks, Total, Grade**.**

Step 2: Define Attributes and Methods

- **Student:**
  - **Attributes:** studentID, name, totalMarks, grade.
  - **Methods:** viewMarks(), viewTotal(), viewGrade().
- **Staff:**
  - **Attributes:** staffID, name.
  - **Methods:** addStudent(), enterMarks(), assignGrade().
- **Subject:**
  - **Attributes:** subjectID, subjectName.
  - **Methods:** addSubject(), viewSubject().
- **Marks:**
  - **Attributes:** studentID, subjectID, mark.
  - **Methods:** enterMark().
- **Total:**
  - **Attributes:** studentID, totalMarks.
  - **Methods:** calculateTotal().
- **Grade:**
  - **Attributes:** gradeID, grade.
  - **Methods:** assignGrade().

Step 3: Identify Relationships

- A Student has many Marks.
- Marks are associated with a Subject.
- Staff manages Student records and enters Marks.
- Total is calculated from Marks.

Step 4: Draw the Diagram

- Each class is represented as a box containing its attributes and methods.
- Use association lines to show relationships between Student, Marks, Subject, Grade, and Total.

Step 5: Add Multiplicity and Associations

- Example: A Student is associated with many Marks, and each Mark is related to a specific Subject.

ACTIVITY DIAGRAM

Step 1: Choose a Process

- Example: "Calculate and Assign Grades"

Step 2: Define Key Activities

- **Student:** Views total marks and grades.
- **Staff:** Enters marks, calculates total, assigns grades.

Step 3: Sequence the Activities

- Staff enters marks → System calculates total → System assigns grades → Student views marks and grades.

Step 4: Draw the Diagram

- Use action nodes for activities: “Enter Marks,” “Calculate Total,” “Assign Grades,” “View Marks/Grades.”
- Connect the nodes with arrows to indicate flow.

Step 5: Add Decision Points

- Add a decision point for assigning grades based on the total marks (e.g., "If total > 90, assign grade A").

SEQUENCE DIAGRAM

Step 1: Pick a Scenario

- Example: Staff enters marks, system calculates total, and Student views grade.

Step 2: Identify Objects

- Staff, Student, System, Marks, Total, Grade.

Step 3: Define Message Flow

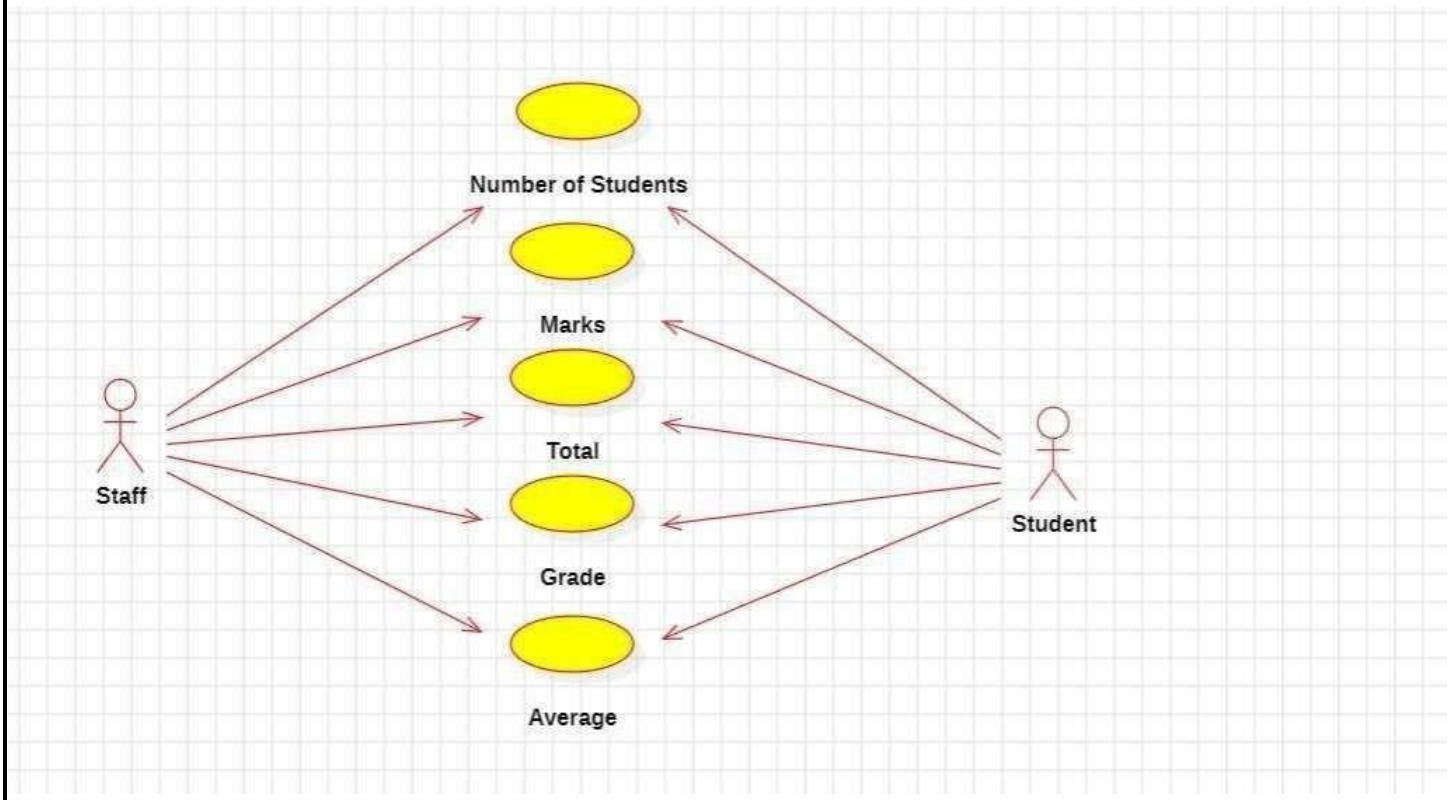
- Staff → System: enterMarks()
- System → Marks: storeMarks()
- System → Total: calculateTotal()
- System → Grade: assignGrade()
- Student → System: viewGrade()

Step 4: Draw Lifelines

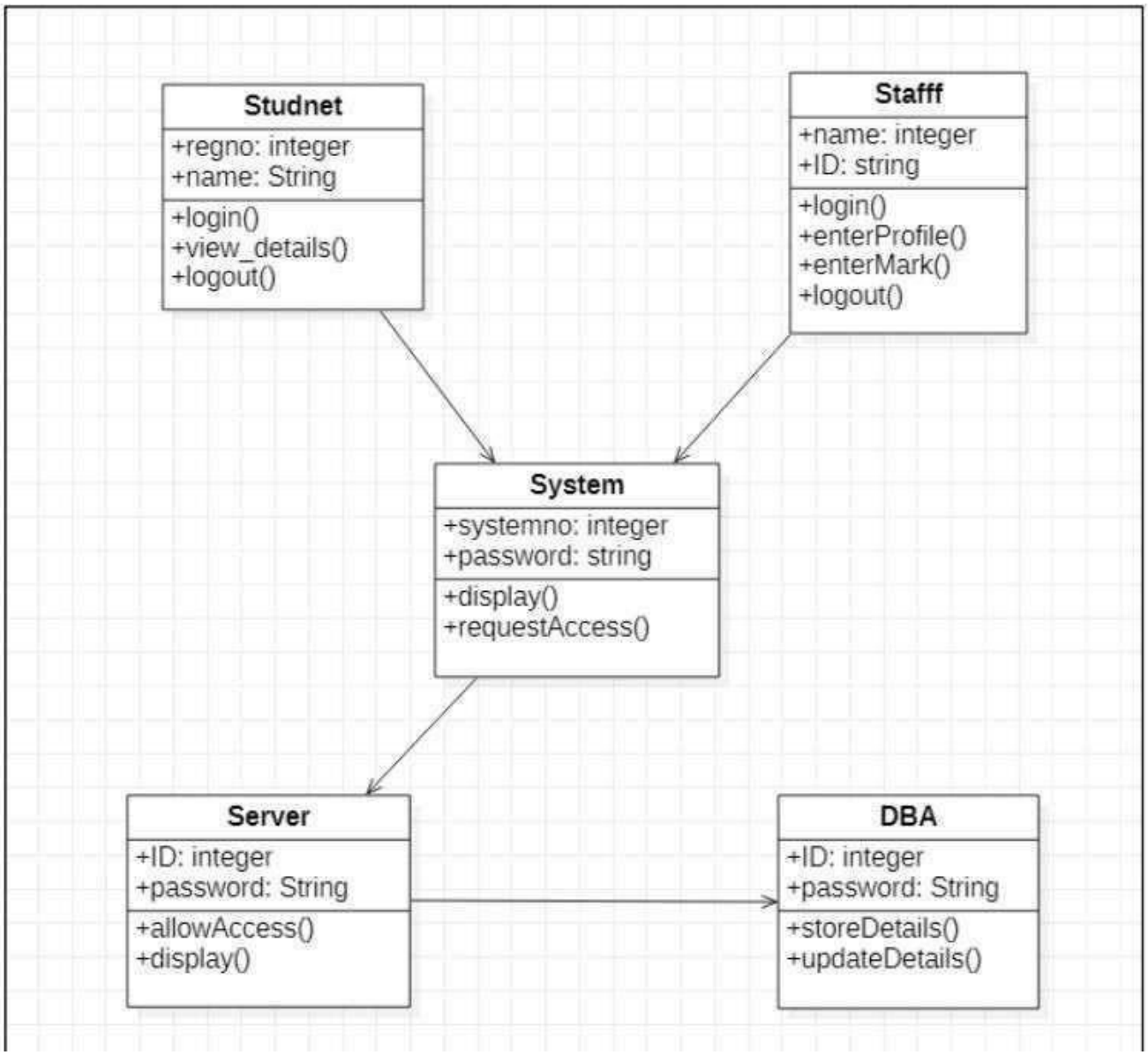
- Add vertical lifelines for Staff, Student, System, etc.
- Draw arrows between lifelines for each interaction.

Step 5: Add Activation Bars

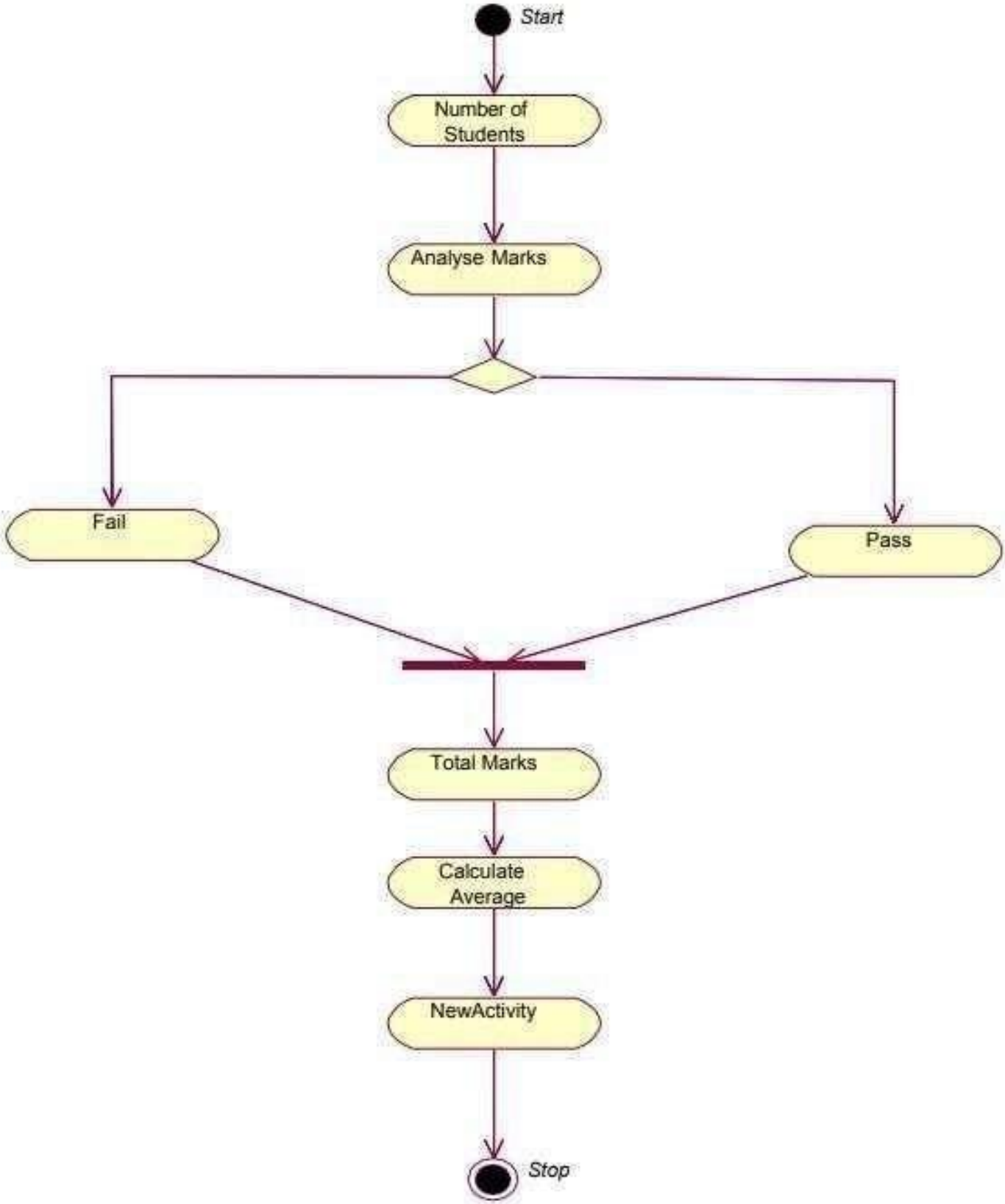
- Activation bars show when an object is processing (e.g., calculating total marks, assigning grade).

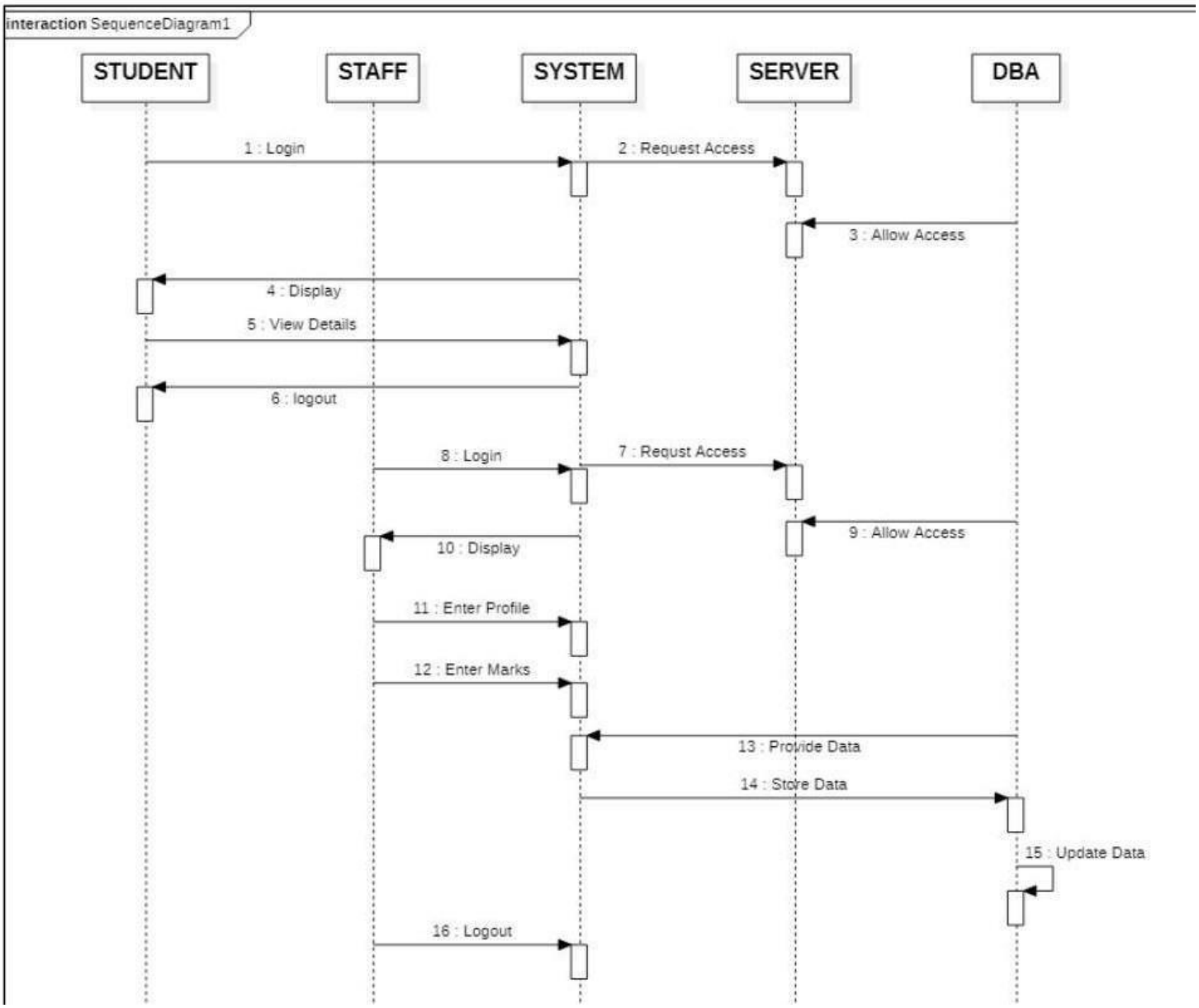

Number of Students
Marks
Total
Grade
Average
Staff
Student

Class Diagram :



ACTIVITY DIAGRAM




interaction SequenceDiagram1
STUDENT
STAFF
SYSTEM
SERVER
DBA
1 : Login
2 : Request Access
3 : Allow Access
4 : Display
5 : View Details
6 : logout
8 : Login
7 : Requst Access
9 : Allow Access
10 : Display
11 : Enter Profile
12 : Enter Marks
13 : Provide Data
14 : Store Data
15 : Update Data
16 : Logout

RESULT:

LIBRARY MANAGEMENT SYSTEM

Ex No:

Date:

AIM

ALGORITHM

USE CASE DIAGRAM

Step 1: Identify Actors

- **Student**: The individual who submits ID cards and requests to issue books.
- **Librarian**: The person who verifies ID cards, issues books, and manages the library system.
- **Supplier**: The external entity that provides books to the library.

Step 2: Identify Use Cases

- **Submit ID Card**: The student submits their ID card.
- **Verify ID Card**: The librarian verifies the submitted ID card.
- **Grant Permission Only if ID Card is Valid**: Librarian grants permission based on ID card validity.
- **Request to Issue Book**: Student requests to issue a book.
- **Issue Book**: Librarian issues the requested book.
- **Return the Book**: Student returns a previously issued book.
- **Search the Database for Book**: Librarian searches the library database for book availability.
- **Check Whether the Book is Available or Not**: Librarian checks if the book is available.
- **Issue the Book if Available**: Librarian issues the book if it's available.

Step 3: Create System Boundary

- Draw a rectangle labeled "Library Management System" to represent the system boundary.

Step 4: Place Actors Outside the System

- Position the **Student** on the left, **Librarian** in the middle, and **Supplier** on the right side of the rectangle.

Step 5: Connect Actors to Use Cases

- Draw ovals for each use case inside the rectangle and connect actors to the relevant use cases:
  - **Student** connects to:
    - Submit ID Card
    - Request to Issue Book
    - Return the Book
  - **Librarian** connects to:
    - Verify ID Card
    - Grant Permission Only if ID Card is Valid
    - Issue Book
    - Search the Database for Book
    - Check Whether the Book is Available or Not
    - Issue the Book if Available

CLASS DIAGRAM

Step 1: Identify Classes

- Student
- Librarian
- Book
- IDCard
- Supplier
- LibraryDatabase

Step 2: Define Attributes and Methods

- **Student**:
  - Attributes: name, studentID, submittedIDCard
  - Methods: submitIDCard(), requestBook(), returnBook()
- **Librarian**:
  - Attributes: librarianID, password
  - Methods: verifyIDCard(), grantPermission(), issueBook(), searchDatabase(), checkAvailability()
- **Book**:
  - Attributes: title, author, ISBN, availabilityStatus
  - Methods: markAsIssued(), markAsReturned()
- **IDCard**:
  - Attributes: idCardNumber, validityStatus
  - Methods: verify()
- **Supplier**:
  - Attributes: supplierID, name
  - Methods: supplyBook()

- **LibraryDatabase**:
  - Attributes: books (list of Book objects)
  - Methods: searchBook(), updateBookStatus()

Step 3: Draw Relationships

- Connect **Student** to **IDCard** (1-to-1).
- Connect **Librarian** to **Book** (1-to-many).
- Connect **Book** to **LibraryDatabase** (1-to-1).
- Connect **Supplier** to **Book** (1-to-many).

Step 4: Layout Classes

- Arrange the classes logically to show their relationships.

Step 5: Add Visibility Modifiers

- Indicate access modifiers for attributes and methods (public +, private -, protected #).

ACTIVITY DIAGRAM

Step 1: Start Node

- Begin with a filled circle representing the start of the process.

Step 2: Define Activities

- For **Submit ID Card**:
  - Fill out ID card submission form
  - Submit ID card
- For **Verify ID Card**:
  - Librarian retrieves submitted ID card
  - Librarian verifies ID card
- For **Grant Permission**:
  - If ID card is valid: grant permission
  - If ID card is invalid: notify student
- For **Request to Issue Book**:
  - Student requests a book
- For **Issue Book**:
  - Librarian checks availability
  - If available, issue book
  - If not available, notify student
- For **Return the Book**:
  - Student returns a book

Step 3: Control Flows

- Draw arrows to indicate the flow from one activity to the next.

Step 4: Decision Nodes

- Use diamond shapes to represent decisions (valid or invalid ID, availability of the book).

Step 5: End Node

- Connect activities to an end node (represented by a circle with a ring).

SEQUENCE DIAGRAM

Step 1: Identify Objects

- **Student**
- **Librarian**
- **Supplier**
- **Database**

Step 2: Determine Sequence of Messages

- For **Submit ID Card**:
  - Student → IDCard: submitIDCard()
  - IDCard → Librarian: notifySubmission()
- For **Verify ID Card**:
  - Librarian → IDCard: verify()
  - IDCard → Librarian: returnValidity()
- For **Grant Permission**:
  - Librarian → Student: grantPermission() (if valid)
  - Librarian → Student: notifyInvalid() (if invalid)
- For **Request to Issue Book**:
  - Student → Librarian: requestBook()
  - Librarian → Book: checkAvailability()
  - Book → Librarian: returnAvailabilityStatus()
- For **Issue Book**:
  - If available:
    - Librarian → Book: issueBook()
    - Book → Librarian: markAsIssued()
    - Librarian → Student: confirmIssuance()
  - If not available:
    - Librarian → Student: notifyNotAvailable()

Step 3: Draw Lifelines

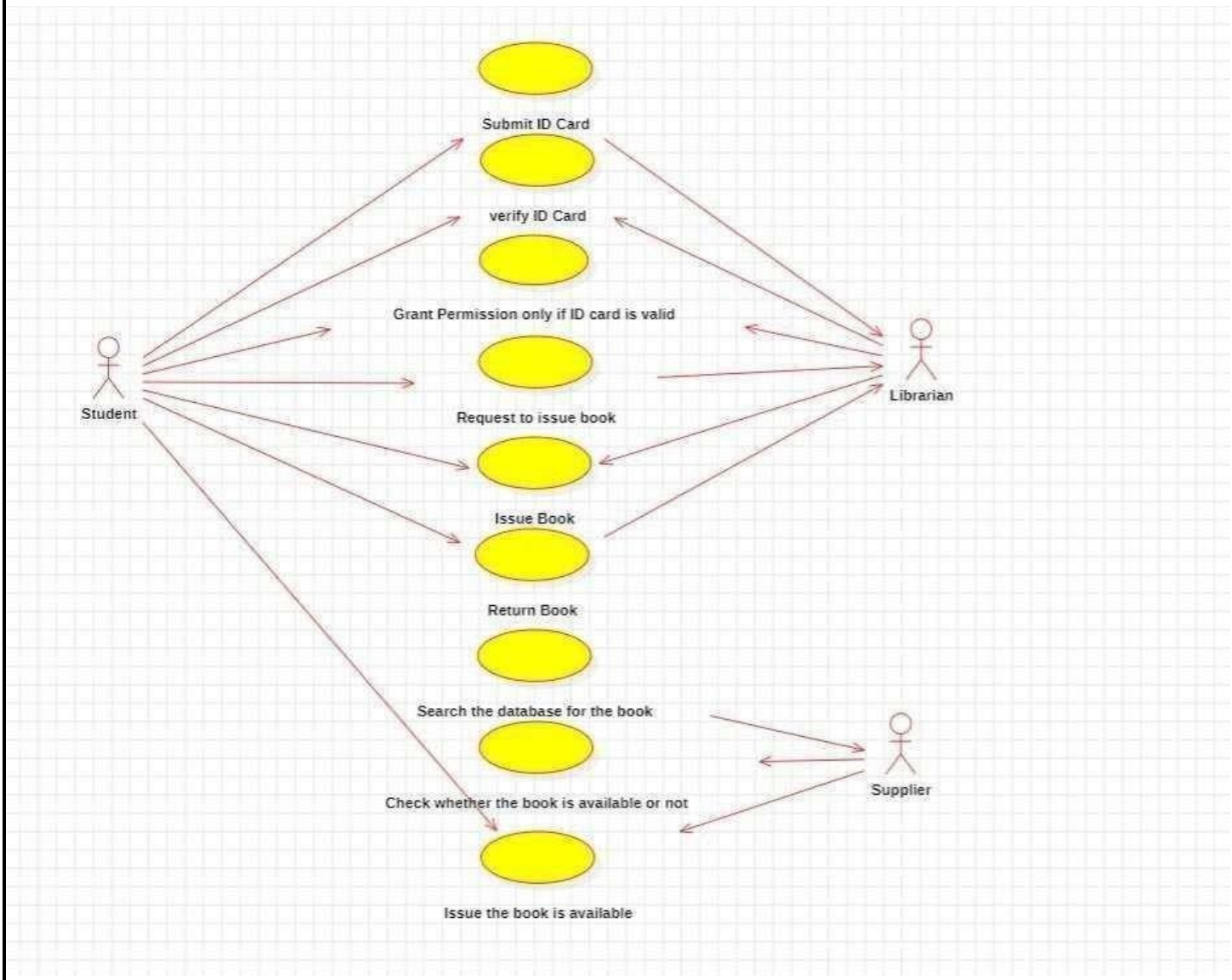
- Vertical dashed lines for each object representing their existence over time.

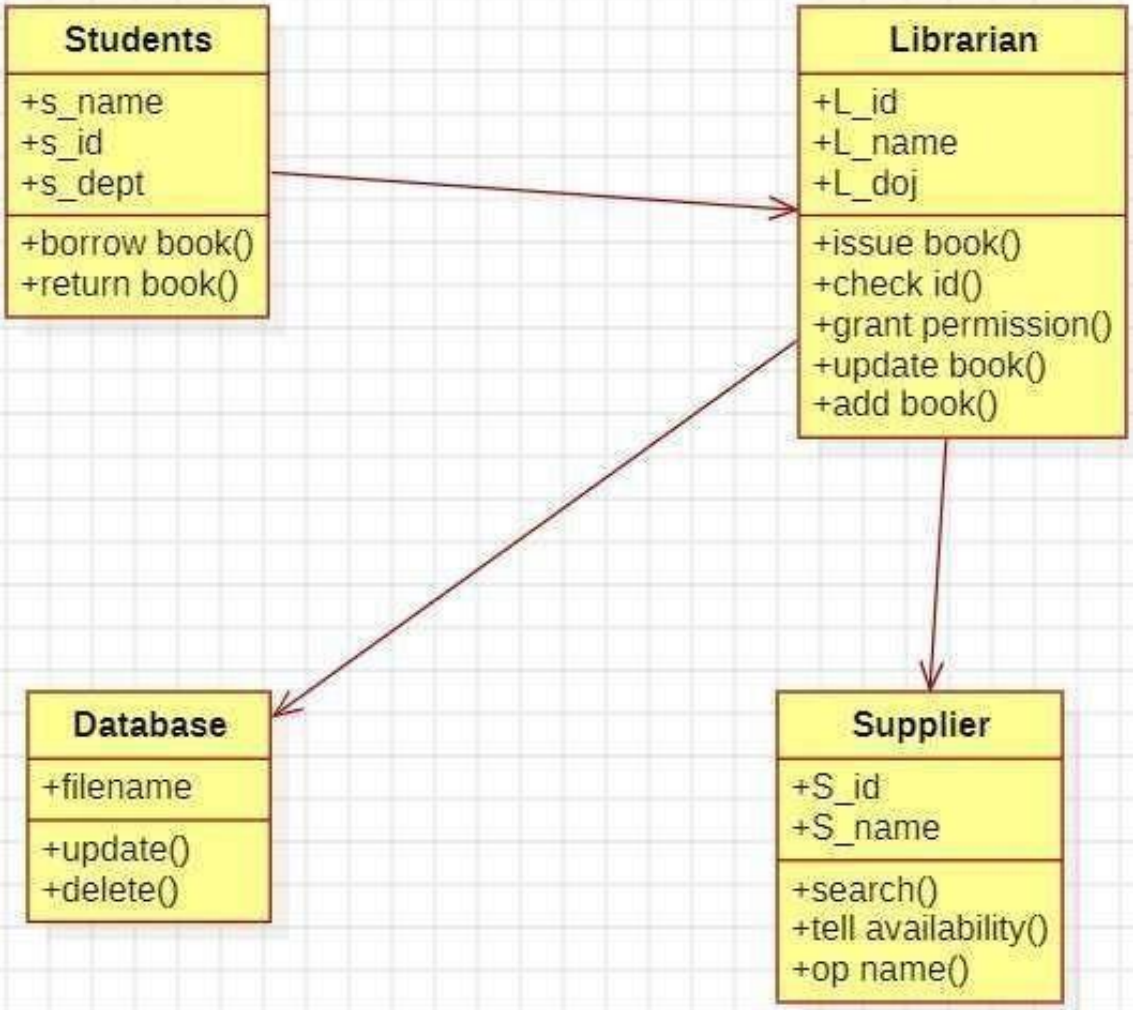
Step 4: Represent Messages

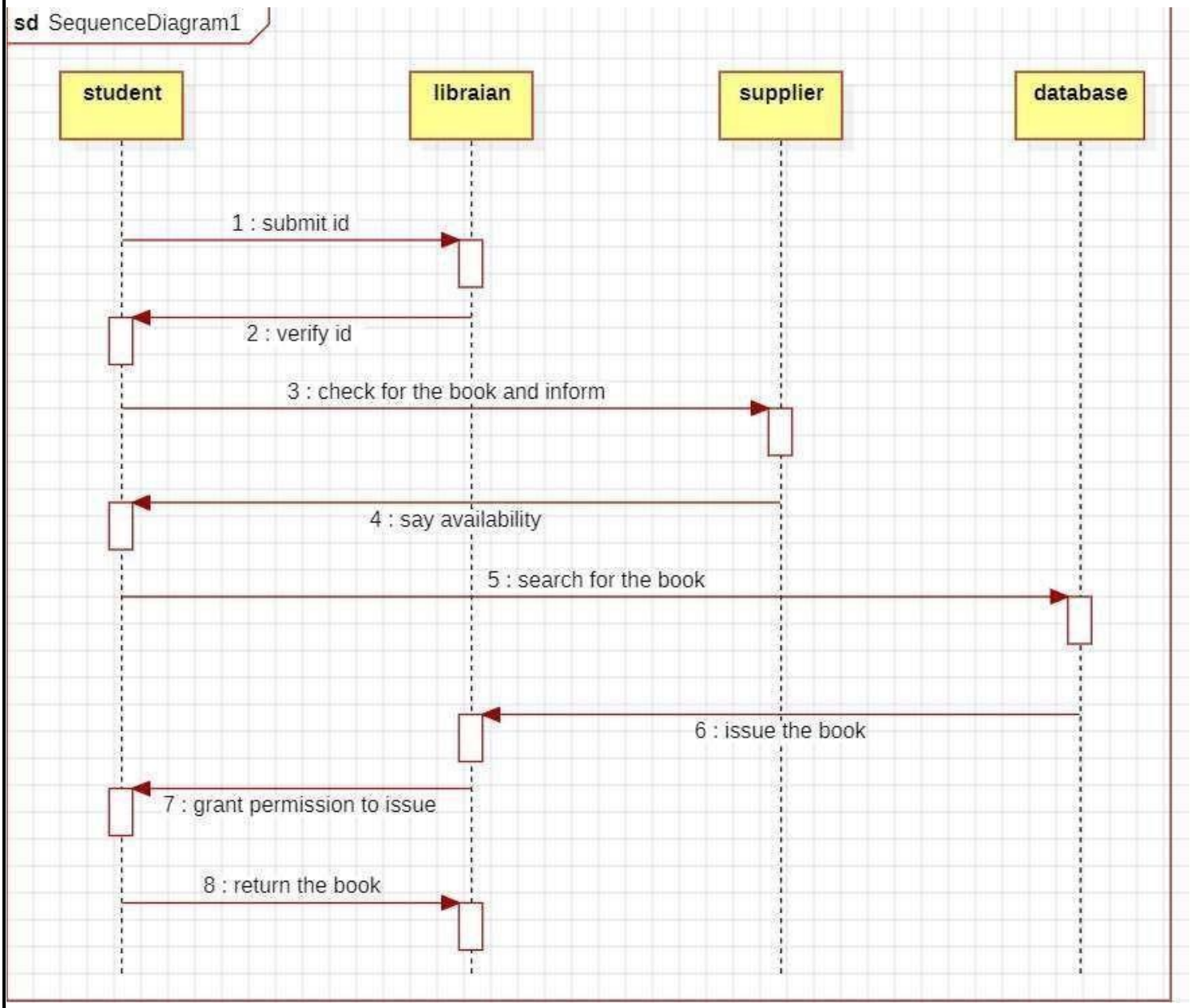
- Horizontal arrows showing interactions over time, labeled with message names.

Step 5: Return Messages

- Use dashed arrows for responses to indicate the flow of control back to the calling object.


Submit ID Card
verify ID Card
Grant Permission only if ID card is valid
Student
Librarian
Request to issue book
Issue Book
Return Book
Search the database for the book
Supplier
Check whether the book is available or not
Issue the book is available


Students
+s_name
+s_id
+s_dept
+borrow book()
+return book()
Librarian
+L_id
+L_name
+L_doj
+issue book()
+check id()
+grant permission()
+update book()
+add book()
Database
+filename
+update()
+delete()
Supplier
+S_id
+S_name
+search()
+tell availability()
+op name()


sd SequenceDiagram1
student
libraian
supplier
database
1 : submit id
2 : verify id
3 : check for the book and inform
4 : say availability
5 : search for the book
6 : issue the book
7 : grant permission to issue
8 : return the book

RESULT:

RAILWAY TICKET RESERVATION SYSTEM

Ex No :
Date :

AIM

ALGORITHM

USE CASE DIAGRAM

Step 1: Identify Actors

- **Passenger**: Books and cancels tickets, views train details and fares.
- **Administrator**: Updates train details and manages other system functionalities.

Step 2: Identify Use Cases

- **Passenger Use Cases**:
  - Reserve Ticket
  - Cancel Ticket
  - View Ticket Fare
  - View Train Details
- **Administrator Use Cases**:
  - Update Train Details

Step 3: Define Relationships

- **Passenger** interacts with:
  - Reserve Ticket
  - Cancel Ticket
  - View Ticket Fare
  - View Train Details
- **Administrator** interacts with:
  - Update Train Details

Step 4: Draw the Diagram

- Place **Passenger** and **Administrator** on the diagram as actors.
- Use ellipses for each use case and connect actors to the appropriate use cases.

Step 5: Refine and Add Details

- Use «include» if needed for overlapping functionalities, such as ticket reservation depending on fare viewing.
- You can also use «extend» to show optional interactions like adding extra services to a ticket reservation.

CLASS DIAGRAM

Step 1: Identify Key Classes

- **Passenger**
- **Administrator**
- **Train**
- **Ticket**
- **Fare**

Step 2: Define Class Attributes and Methods

- **Passenger**:
  - Attributes: passengerID, name, email, contactNumber
  - Methods: reserveTicket(), cancelTicket(), viewFare(), viewTrainDetails()
- **Administrator**:
  - Attributes: adminID, name
  - Methods: updateTrainDetails()
- **Train**:
  - Attributes: trainID, trainName, source, destination, availableSeats
  - Methods: updateTrainDetails(), getTrainDetails()
- **Ticket**:
  - Attributes: ticketID, passengerID, trainID, fare, reservationStatus
  - Methods: reserve(), cancel()
- **Fare**:
  - Attributes: fareID, trainID, amount
  - Methods: calculateFare()

Step 3: Determine Relationships

- **Passenger** is associated with **Ticket**.
- **Administrator** is associated with **Train**.
- **Ticket** and **Fare** are associated with **Train**.

Step 4: Draw the Diagram

- Represent each class as a box containing attributes and methods.
- Show associations between **Passenger**, **Administrator**, **Train**, **Ticket**, and **Fare**.

Step 5: Refine with Multiplicity and Associations

- Example: One **Passenger** can reserve multiple **Tickets**, and one **Ticket** corresponds to one **Train**.

ACTIVITY DIAGRAM

Step 1: Choose a Process

- **Passenger Reserves a Ticket**

Step 2: Define Key Activities

- **Passenger** enters journey details, selects train, views fare, confirms booking, and makes payment.
- **System** reserves the seat and generates the ticket.

Step 3: Sequence the Activities

- **Passenger**:
    1. Log in
    2. Select Journey Details (Source & Destination)
    3. View Available Trains
    4. View Fare
    5. Reserve Ticket
    6. Make Payment
    7. Receive Confirmation

Step 4: Draw the Diagram

- Use action nodes for each activity, connecting them with arrows showing flow from start to end.

Step 5: Add Decision Points and End

- Add a decision point between selecting journey details and confirming reservation based on seat availability.

SEQUENCE DIAGRAM

Step 1: Pick a Scenario

- Passenger Cancels a Reserved Ticket

Step 2: Identify Objects

- Passenger, Reservation System, Payment System, Train, Ticket

Step 3: Define Message Flow

- **Passenger** -> **Reservation System**: cancelTicket()
- **Reservation System** -> **Ticket**: verifyTicket()

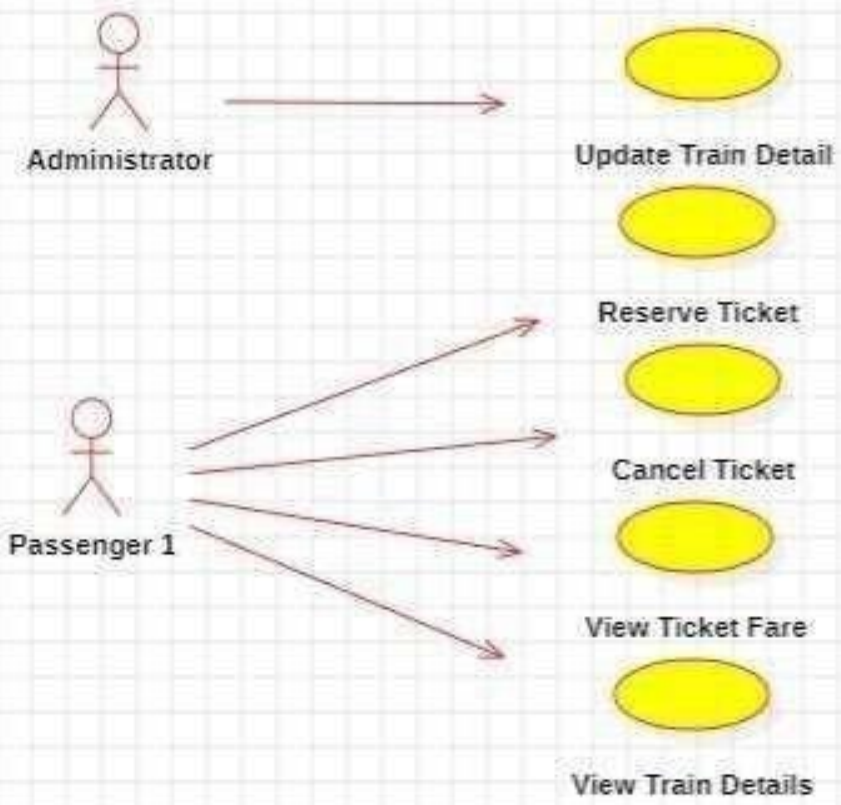
- **Reservation System -> Payment System:** processRefund()
- **Payment System -> Reservation System:** refundConfirmation()
- **Reservation System -> Passenger**: cancelConfirmation()

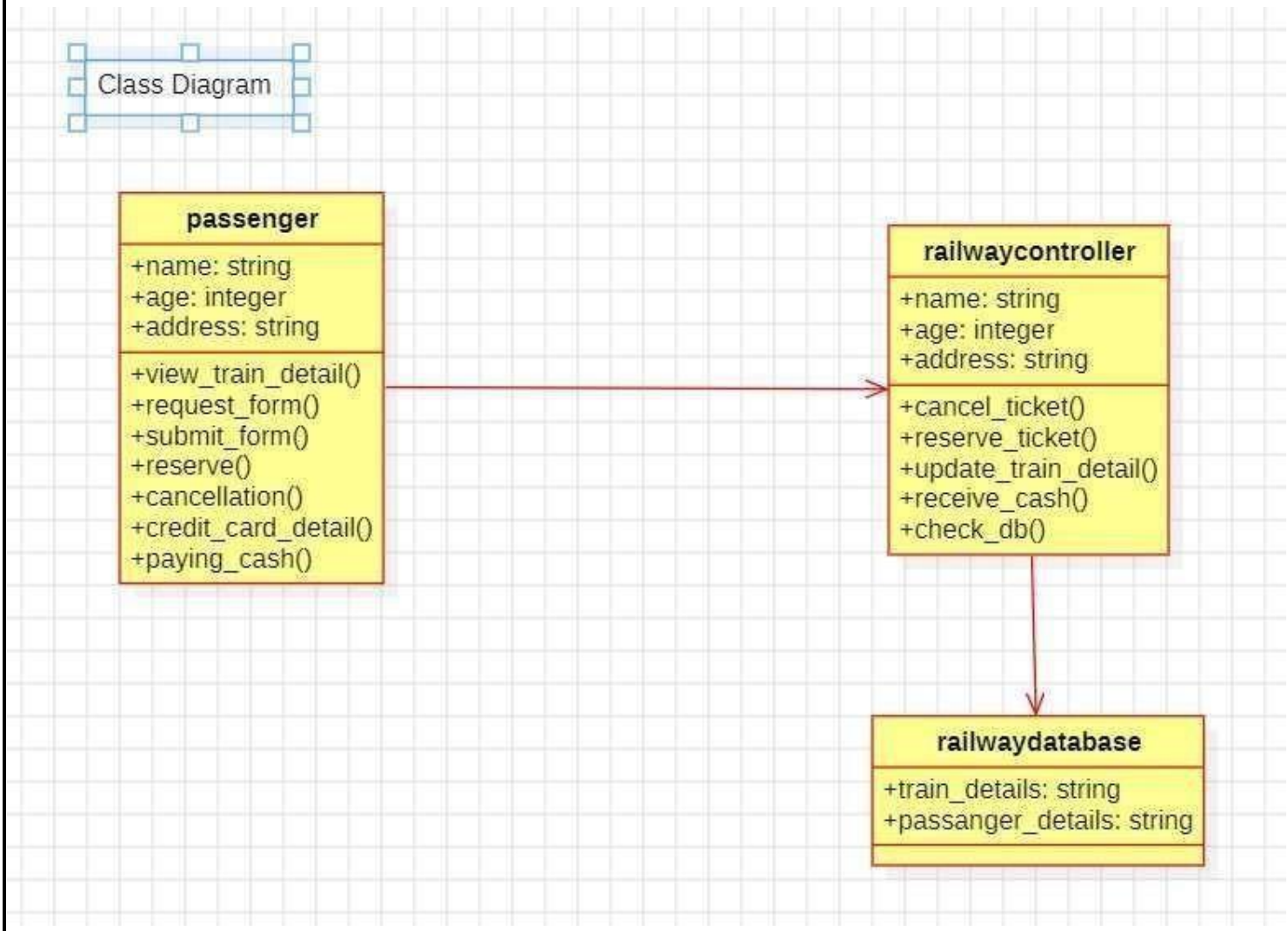
Step 4: Draw Lifelines and Messages

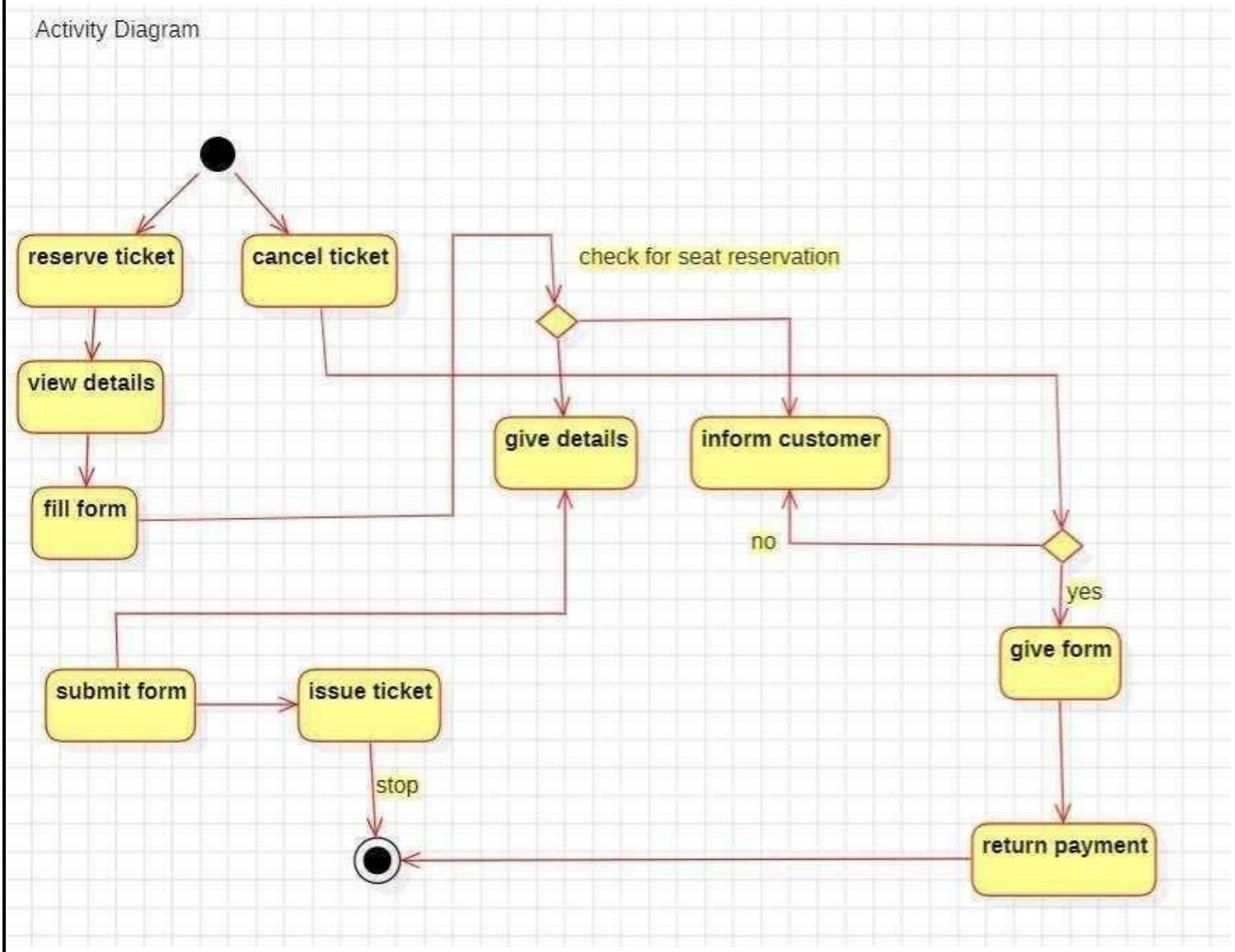
- Draw vertical lifelines for each object, and use arrows to represent the message flow between them.

Step 5: Add Activation Bars

- Activation bars show when objects are processing the message, such as the reservation system verifying the ticket or the payment system processing the refund.


Administrator
Update Train Detail
Reserve Ticket
Passenger 1
Cancel Ticket
View Ticket Fare
View Train Details


Class Diagram
passenger
+name: string
+age: integer
+address: string
+view_train_detail()
+request_form()
+submit_form()
+reserve()
+cancellation()
+credit_card_detail()
+paying_cash()
railwaycontroller
+name: string
+age: integer
+address: string
+cancel_ticket()
+reserve_ticket()
+update_train_detail()
+receive_cash()
+check_db()
railwaydatabase
+train_details: string
+passanger_details: string


Activity Diagram
reserve ticket
cancel ticket
check for seat reservation
view details
give details
inform customer
fill form
no
yes
give form
submit form
issue ticket
stop
return payment

RESULT:

PLACEMENT REGISTRATION SYSTEM

Ex No:
Date:

AIM

ALGORITHM

USE CASE DIAGRAM

Step 1: Identify Actors

- **Applicant**
- **HR**

Step 2: Identify Use Cases

- **Register**
- **Know Status**
- **Admin Panel**
- **Send Response**
- **Delete Application**

Step 3: Create System Boundary

- Draw a rectangle to represent the system, labeling it "Placement Registration System".

Step 4: Place Actors Outside the System

- Position the **Applicant** on the left side and **HR** on the right side of the rectangle.

Step 5: Connect Actors to Use Cases

- Draw ovals inside the rectangle for each use case.
- Connect **Applicant** to:
  - Register
  - Know Status
- Connect **HR** to:
  - Admin Panel
  - Send Response
  - Delete Application

CLASS DIAGRAM

Step 1: Identify Classes

- **Applicant**
- **Application**
- **HR**
- **Status**

Step 2: Define Attributes and Methods

- **Applicant**:
  - Attributes: name, rollNumber, email, phone, course, year, cgpa
  - Methods: register(), checkStatus()
- **Application**:
  - Attributes: applicationID, applicant, status
  - Methods: updateStatus(), delete()
- **HR**:
  - Attributes: hrID, password
  - Methods: login(), sendResponse(), deleteApplication()
- **Status**:
  - Attributes: applicationID, currentStatus
  - Methods: notifyApplicant()

Step 3: Draw Relationships

- Connect **Applicant** to **Application** (1-to-many).
- Connect **Application** to **Status** (1-to-1).
- Connect **HR** to **Application** (1-to-many).

Step 4: Layout Classes

- Arrange the classes logically, indicating their relationships clearly.

Step 5: Add Visibility Modifiers

- Indicate access modifiers for attributes and methods (public +, private -, protected #).

ACTIVITY DIAGRAM

Step 1: Start Node

- Begin with a filled circle representing the start of the process.

Step 2: Define Activities

- For **Register**:
  - Fill out registration form
  - Submit application
  - Confirmation received

- For **Know Status**:
  - Enter email
  - Retrieve application status
- For **Admin Panel**:
  - HR logs in
  - View applications
- For **Send Response**:
  - HR selects application
  - Update status
  - Notify applicant
- For **Delete Application**:
  - HR selects application
  - Confirm deletion

Step 3: Control Flows

- Draw arrows to indicate the flow from one activity to the next.

Step 4: End Node

- Connect activities to an end node (represented by a circle with a ring).

Step 5: Parallel Activities (if applicable)

- Use a fork node to show activities that can occur simultaneously.

SEQUENCE DIAGRAM

Step 1: Identify Objects

- **Applicant**
- **Application System**
- **HR**

Step 2: Determine Sequence of Messages

- For **Register**:
  - Applicant → Application System: register()
  - Application System → Applicant: confirmation()
- For **Know Status**:
  - Applicant → Application System: checkStatus()
  - Application System → Applicant: statusResult()
- For **Send Response**:
  - HR → Application System: sendResponse()
  - Application System → Applicant: notifyStatus()
- For **Delete Application**:
  - HR → Application System: deleteApplication()
  - Application System → HR: confirmation()

Step 3: Draw Lifelines

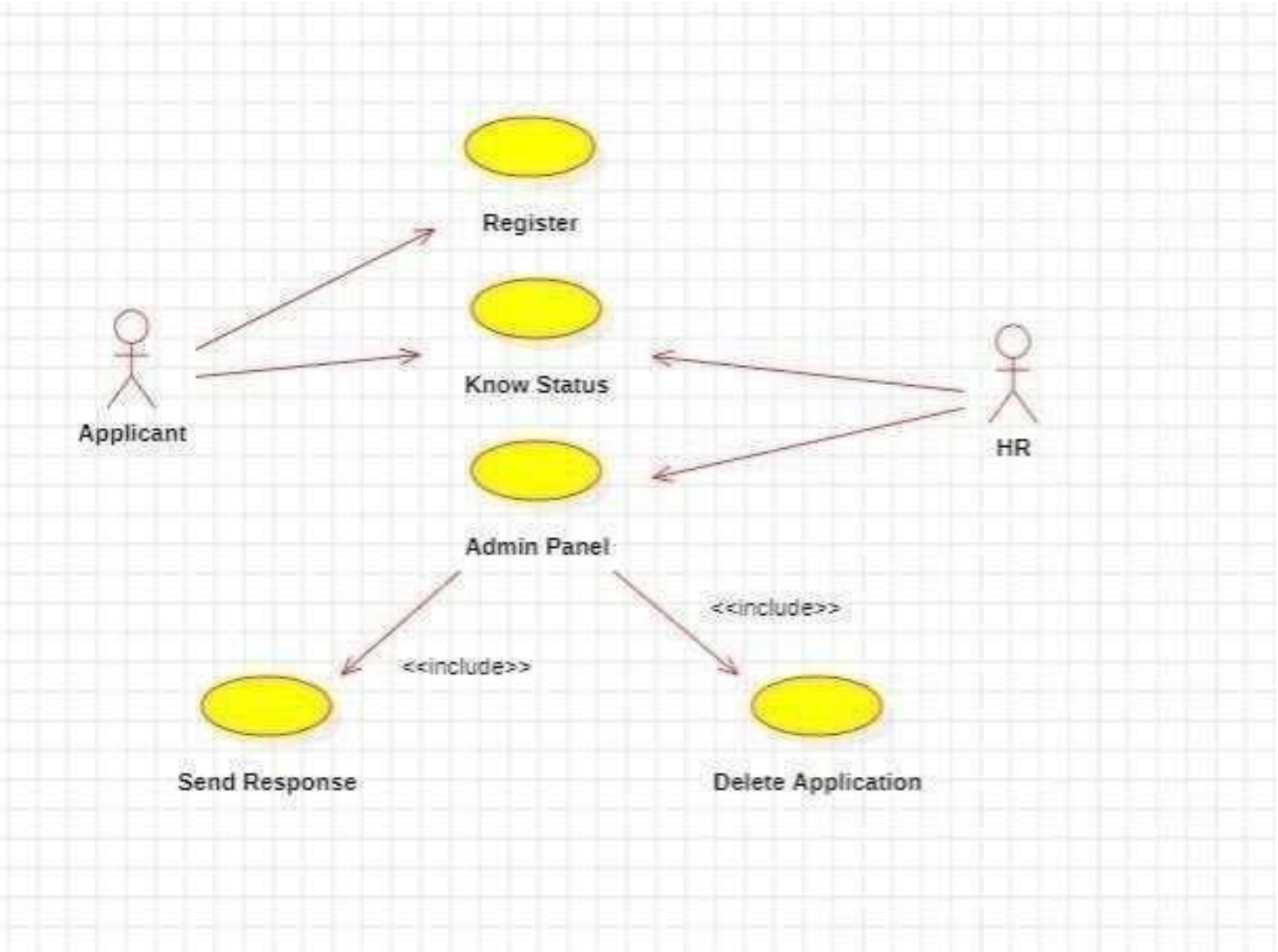
- Vertical dashed lines for each object representing their existence over time.

Step 4: Represent Messages

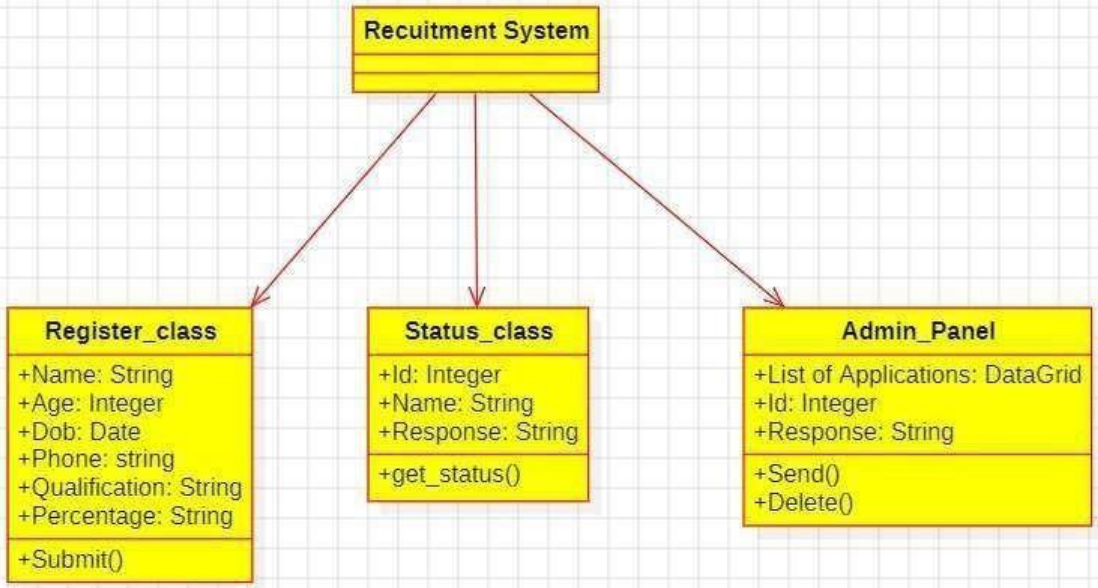
- Horizontal arrows showing interactions over time, labeled with message names.

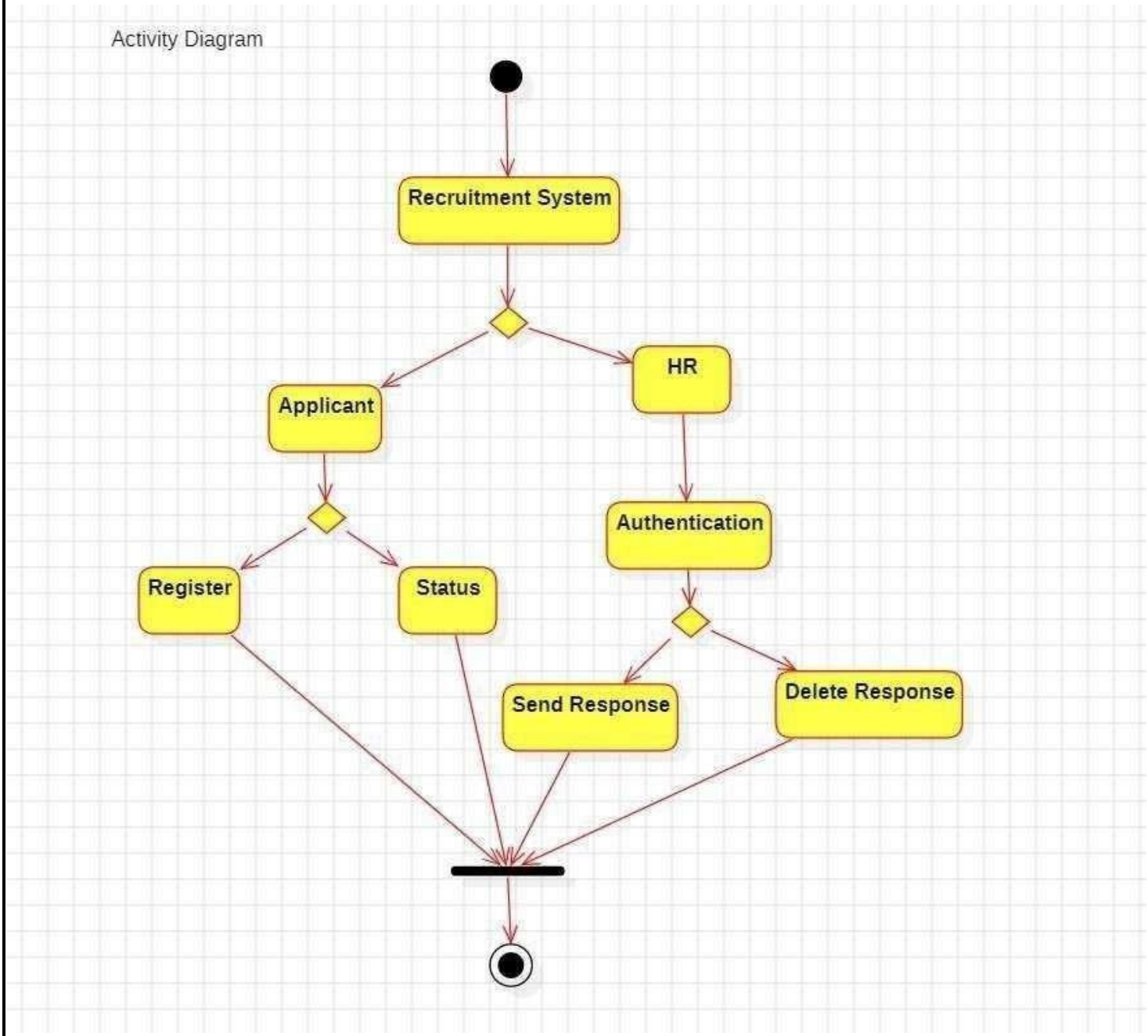
Step 5: Return Messages

- Use dashed arrows for responses to indicate the flow of control back to the calling object.

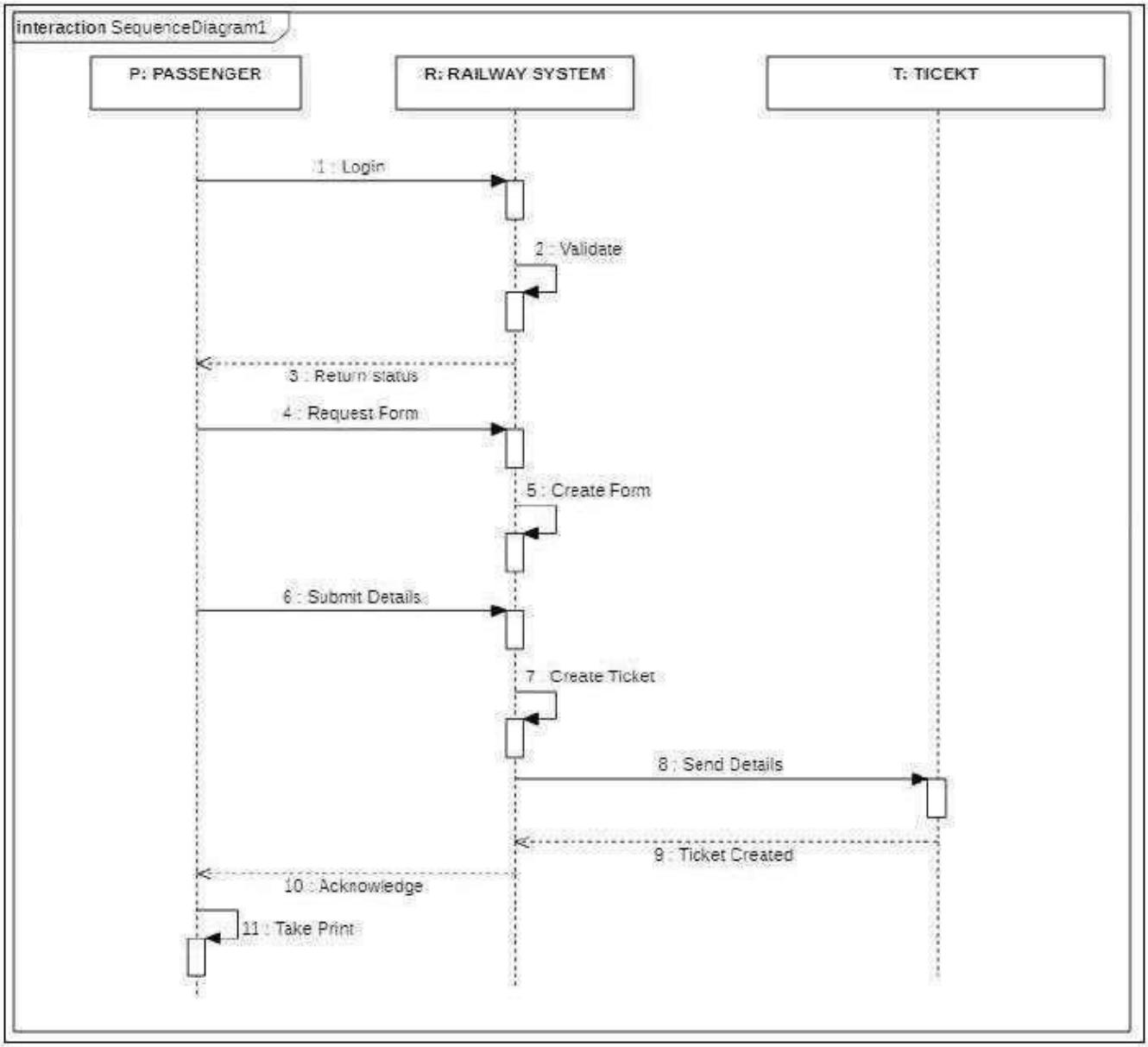

Register
Know Status
Admin Panel
Applicant
HR
<<include>>
<<include>>
Send Response
Delete Application

Class Diagram




Activity Diagram
Recruitment System
Applicant
HR
Authentication
Register
Status
Send Response
Delete Response

Sequence Diagram :



RESULT:

EXAM REGISTRATION SYSTEM

Ex.No :
Date :

AIM

ALGORITHM

USE CASE DIAGRAM

Step 1: Identify Actors

- **Student**: Submits details, proof, and fees for exam registration.
- **Education Officer**: Verifies proof and issues hall tickets.

Step 2: Identify Use Cases

- **Student Use Cases**:
  - Enter Student Details
  - Upload Student Photo
  - Verify Student Proof
  - Pay Exam Fees
  - Download Hall Ticket
- **Education Officer Use Cases**:
  - Issue Hall Ticket
  - Submit Student Proof

Step 3: Define Relationships

- **Student** interacts with:
  - Enter Student Details
  - Upload Student Photo
  - Verify Student Proof
  - Pay Exam Fees
  - Download Hall Ticket
- **Education Officer** interacts with:
  - Submit Student Proof
  - Issue Hall Ticket

Step 4: Draw the Diagram

- Place **Student** and **Education Officer** on the diagram as actors.
- Use ellipses for each use case and connect actors to the appropriate use cases.

Step 5: Refine and Add Details

- Use «extend» for the dependency of **"Download Hall Ticket"** on **"Pay Exam Fees"**.

Use «include» for related tasks like **"Submit Student Proof"** and **"Verify Student Proof"**

CLASS DIAGRAM

Step 1: Identify Key Classes

- **Student**
- **Education Officer**
- **Proof**
- **Payment**
- **Hall Ticket**

Step 2: Define Class Attributes and Methods

- **Student**:
  - Attributes: studentID, name, photo, proof, feesPaid
  - Methods: enterDetails(), uploadPhoto(), submitProof(), payFees(), downloadHallTicket()
- **Education Officer**:
  - Attributes: officerID, name
  - Methods: verifyProof(), issueHallTicket()
- **Proof**:
  - Attributes: proofID, proofType, proofStatus
  - Methods: submitProof(), verifyProof()
- **Payment**:
  - Attributes: paymentID, amount, paymentStatus
  - Methods: makePayment()
- **HallTicket**:
  - Attributes: ticketID, studentID, issueDate
  - Methods: generateTicket()

Step 3: Determine Relationships

- **Student** is associated with **Proof** and **Payment**.
- **Education Officer** verifies **Proof** and issues **HallTicket**.
- **Student** downloads **HallTicket** after payment is completed.

Step 4: Draw the Diagram

- Represent each class as a box containing attributes and methods.
- Show associations between **Student**, **Proof**, **Payment**, **Education Officer**, and **HallTicket**.

Step 5: Refine with Multiplicity and Associations

- Example: One **Student** can submit multiple **Proofs**, but each **Proof** is verified by one **Education Officer**.

ACTIVITY DIAGRAM

Step 1: Choose a Process

- **Student Submits Exam Registration**

Step 2: Define Key Activities

- **Student** enters details, uploads photo, submits proof, pays fees, and downloads hall ticket.
- **Education Officer** verifies the proof and issues the hall ticket.

Step 3: Sequence the Activities

- **Student**:
  1. Log in
  2. Enter Student Details
  3. Upload Photo
  4. Submit Proof
  5. Pay Fees
  6. Download Hall Ticket
- **Education Officer**:
  1. Verify Proof
  2. Issue Hall Ticket

Step 4: Draw the Diagram

- Use action nodes for each activity, connecting them with arrows showing flow from start to end.

Step 5: Add Decision Points and End

- Add a decision point between proof submission and verification. If the proof is valid, proceed to fees payment; otherwise, request re-submission of proof.

SEQUENCE DIAGRAM

Step 1: Pick a Scenario

- Student Requests Hall Ticket after Payment

Step 2: Identify Objects

- Student, Education System, Payment System, Education Officer, Hall Ticket System

Step 3: Define Message Flow

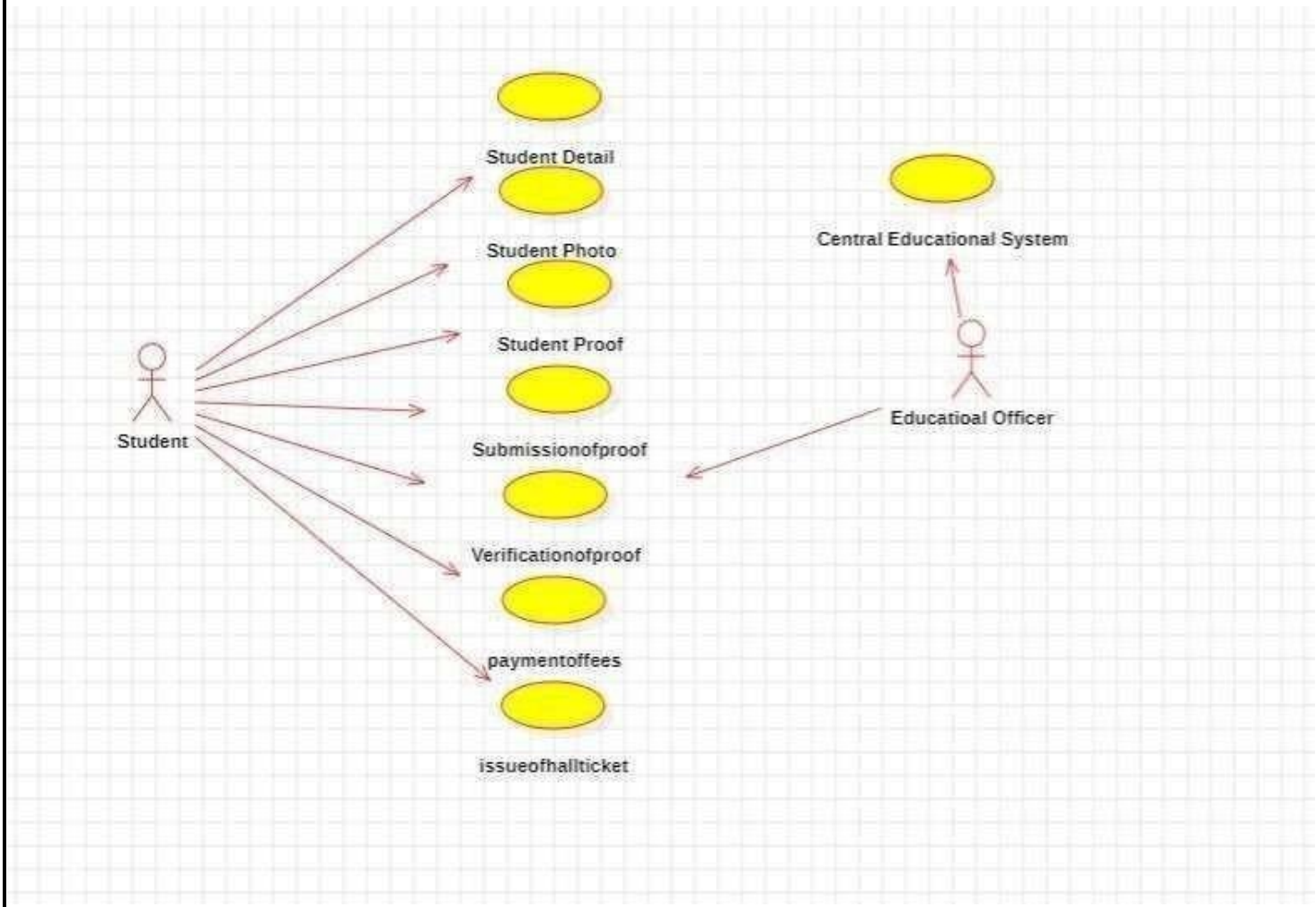
- **Student** -> **Education System**: requestHallTicket()
- **Education System** -> **Payment System**: verifyPayment()
- **Payment System** -> **Education System**: paymentVerified()
- **Education System** -> **Education Officer**: verifyProof()
- **Education Officer** -> **Education System**: issueHallTicket()
- **Education System** -> **Student**: provideHallTicket()

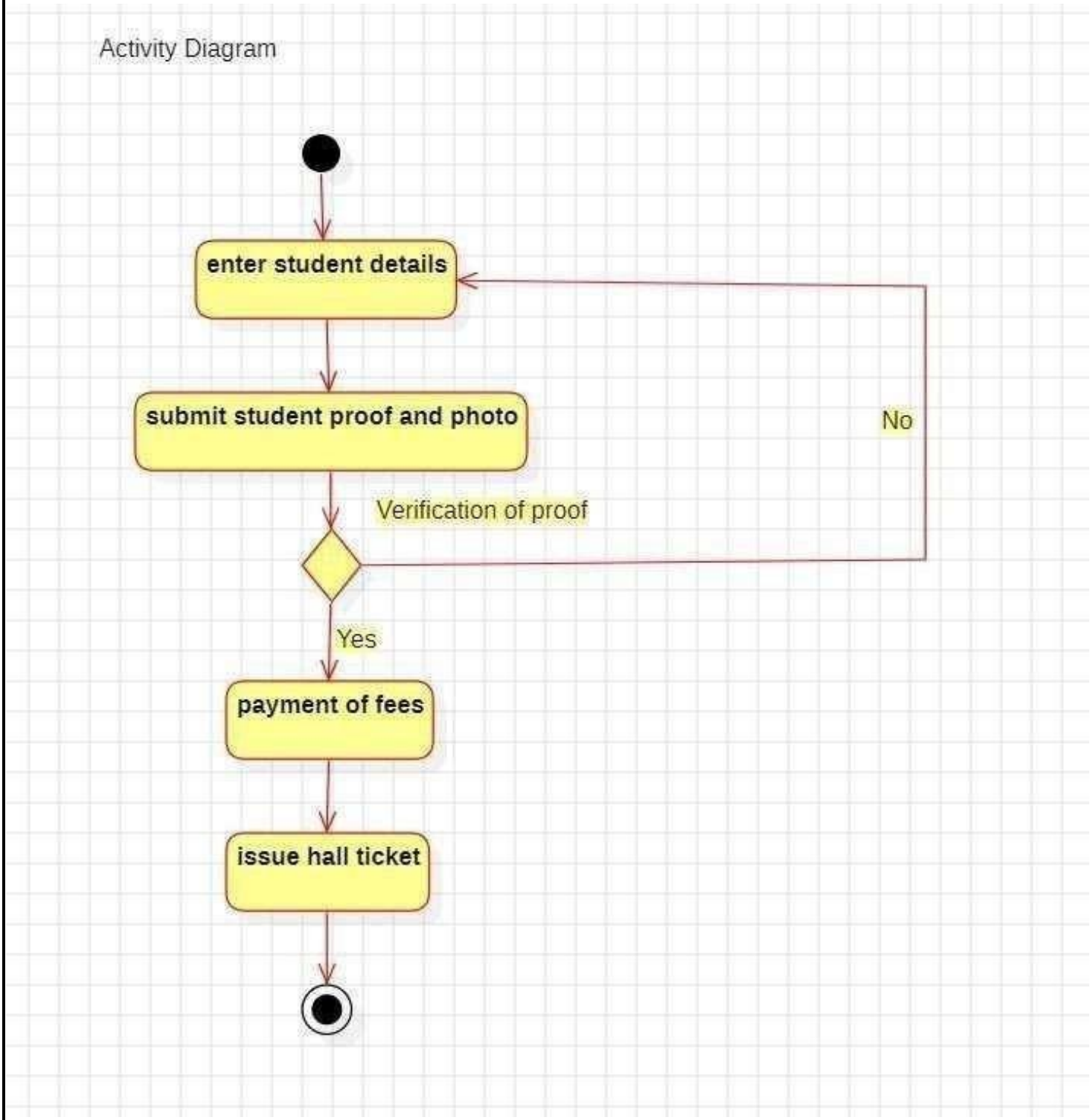
Step 4: Draw Lifelines and Messages

- Draw vertical lifelines for each object, and use arrows to represent the message flow between them.

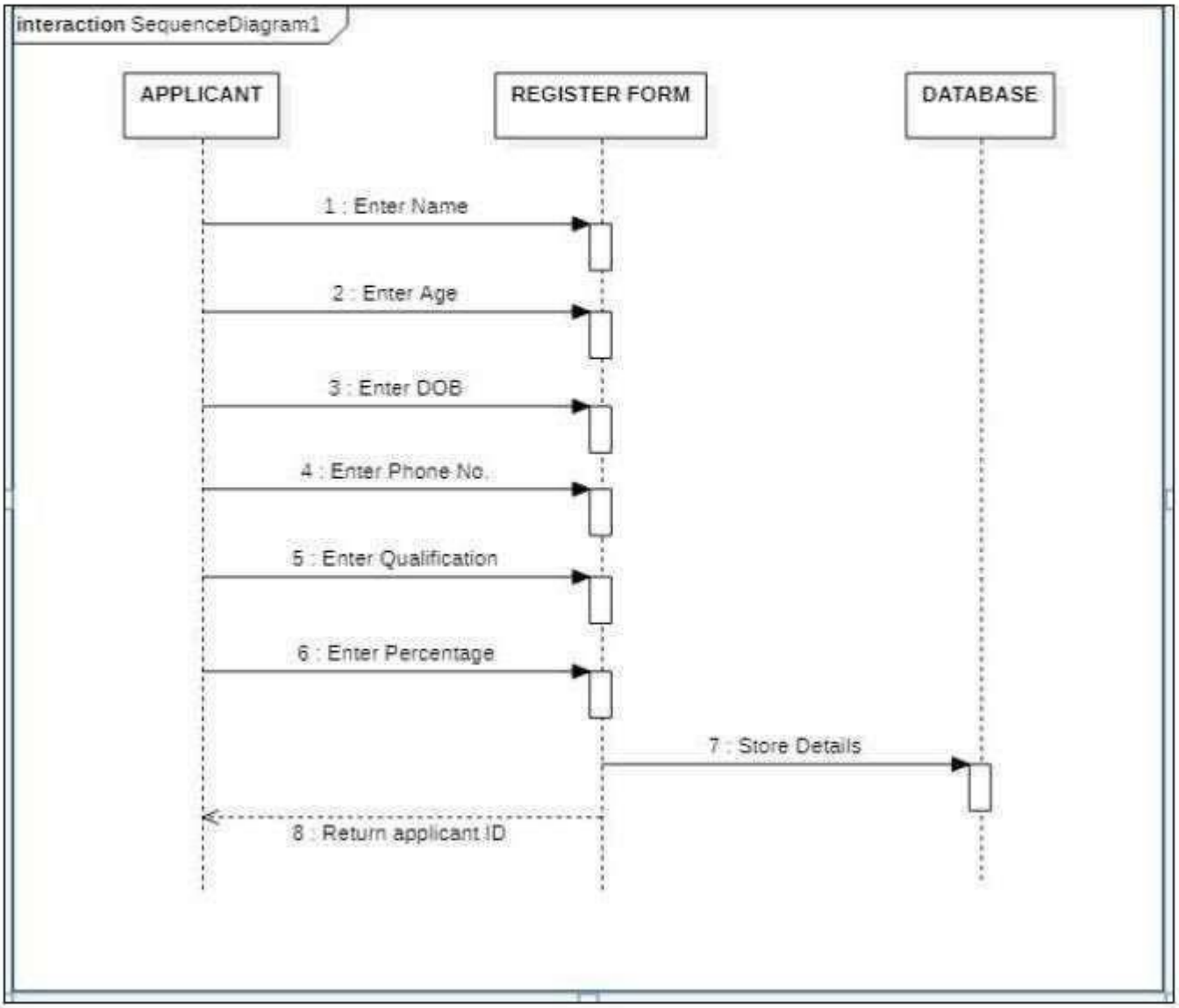
Step 5: Add Activation Bars

- Activation bars show when objects are processing the message, e.g., when the system verifies payment and when the officer verifies proof.

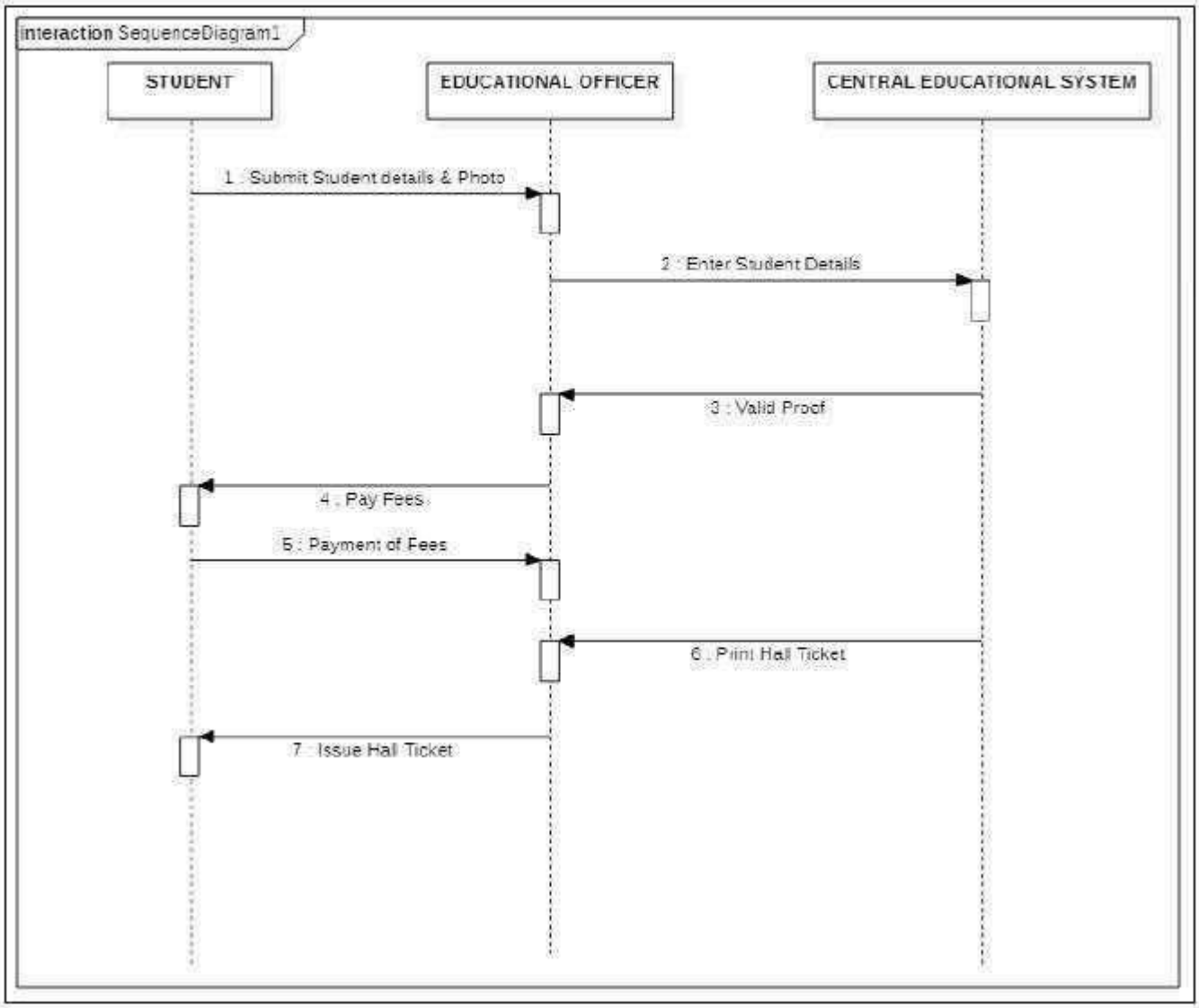

Student Detail
Student Photo
Student Proof
Submissionofproof
Verificationofproof
paymentoffees
issueofhallticket
Central Educational System
Student
Educatioal Officer


Activity Diagram
enter student details
submit student proof and photo
No
Verification of proof
Yes
payment of fees
issue hall ticket

Sequence Diagram :



Sequence Diagram :



RESULT: