B.M.S. COLLEGE OF ENGINEERING BENGALURU
Autonomous Institute, Affiliated to VTU

Lab Record

Software Engineering and Object-Oriented Modeling

Submitted in partial fulfillment for the 5th Semester Laboratory

Bachelor of Engineering
in
Computer Science and Engineering

Submitted by:

Parnika Deepak Bhat
1BM23CS226

Department of Computer Science and Engineering
B.M.S. College of Engineering
Bull Temple Road, Basavanagudi, Bangalore 560 019
August 2025-December 2025

B.M.S. COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the Object-Oriented Analysis and Design(22CS6PCSEO) laboratory has been carried out by Parnika Deepak Bhat(1BM23CS226) during the 5th Semester August 2025-December 2025

.

Signature of the Faculty Incharge:

Name of Your Batch Incharge and designation: Dr. Adarsha Sagar H V

(Assistant Professor)
Department of Computer Science and Engineering
B.M.S. College of Engineering, Bangalore

Table of Contents

1. Hotel Management System

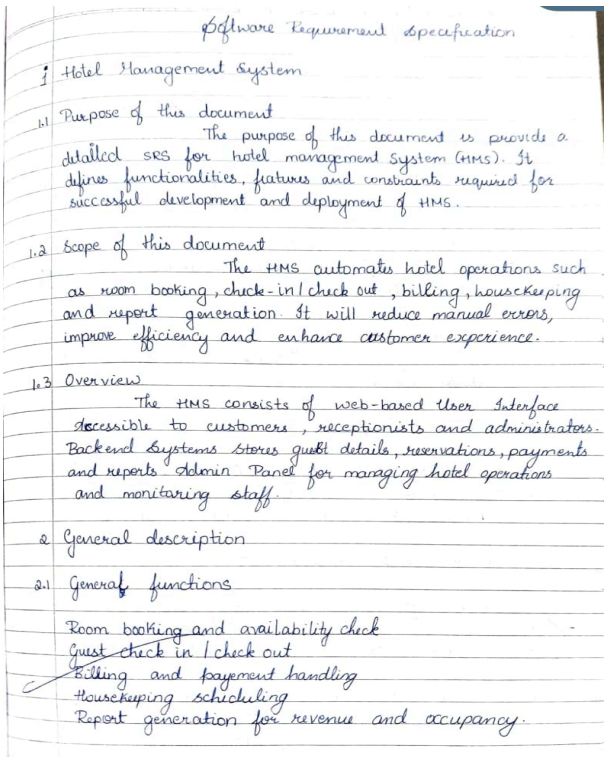
Problem Statement
SRS-Software Requirements Specification
Class Diagram Requirements: Minimum 7 classes, 4 attributes and 4 operations in each class, associations, association name, association end names, multiplicity, association class, enumeration, qualified association, aggregation, composition, generalization, ordered, sequencing, multiplicity of attribute, brief description of the Class diagram
State Diagram: one simple state diagram, one advance state diagram (either Sub Machine or Nested State, include any one of the concurrency in your design) Requirements: minimum of 6 states for both the state diagram with state name, do activities, events, guard condition, brief description of State diagram
Use-Case Diagram: Minimum 5 use cases to be identified, and design one simple use case diagram and one advanced use case diagram(ie using include,extend and generalization relationship) and explanation as there in the solution manual.
Sequence Diagram: Write the scenario for any two use case transaction completely. and design the simple sequence diagram with minimum of 5 objects and communication between them. Design advanced sequence diagram using passive and transient objects. Brief explanation for each
Activity Diagram: Design the simple activity diagram showing the algorithm or workflow. Design the advanced activity diagram with swimlanes showing the responsible person for swimlane and also write the brief explanation for both.

1.1Problem Statement

The Hotel Management System (HMS) is designed to computerize and streamline core hotel operations such as room booking, check-in/check-out, housekeeping, billing, payment processing, and reservation management.
The system should eliminate manual errors, improve efficiency, maintain accurate real-time

room availability, and enhance the overall guest experience.
This system helps hotel staff manage guest details, room status, payments, and housekeeping schedules in a centralized, user-friendly manner.

1.2 SRS-Software Requirements Specification



Software Requirement Specification

1 Hotel Management System

1.1 Purpose of this document
The purpose of this document is provide a detailed SRS for hotel management system (HMS). It defines functionalities, features and constraints required for successful development and deployment of HMS.

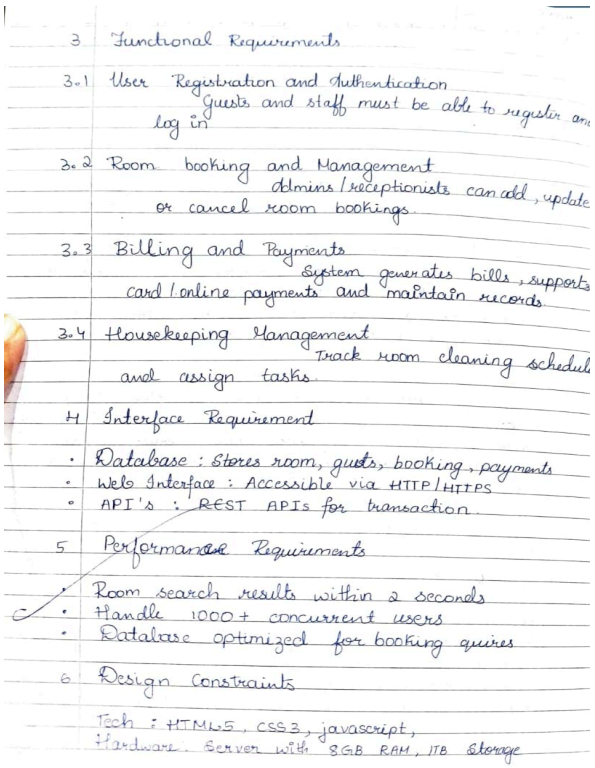
1.2 Scope of this document
The HMS automates hotel operations such as room booking, check-in/check out, billing, housekeeping and report generation. It will reduce manual errors, improve efficiency and enhance customer experience.

1.3 Overview
The HMS consists of web-based User Interface accessible to customers, receptionists and administrators. Backend Systems stores guest details, reservations, payments and reports. Admin Panel for managing hotel operations and monitoring staff.

2 General description

2.1 General functions

Room booking and availability check
Guest check in / check out
Billing and payment handling
Housekeeping scheduling
Report generation for revenue and occupancy.



3 Functional Requirements

3.1 User Registration and Authentication
Guests and staff must be able to register and log in

3.2 Room booking and Management
Admins / receptionists can add, update or cancel room bookings.

3.3 Billing and Payments
System generates bills, supports card / online payments and maintain records.

3.4 Housekeeping Management
Track room cleaning schedule and assign tasks.

4 Interface Requirement

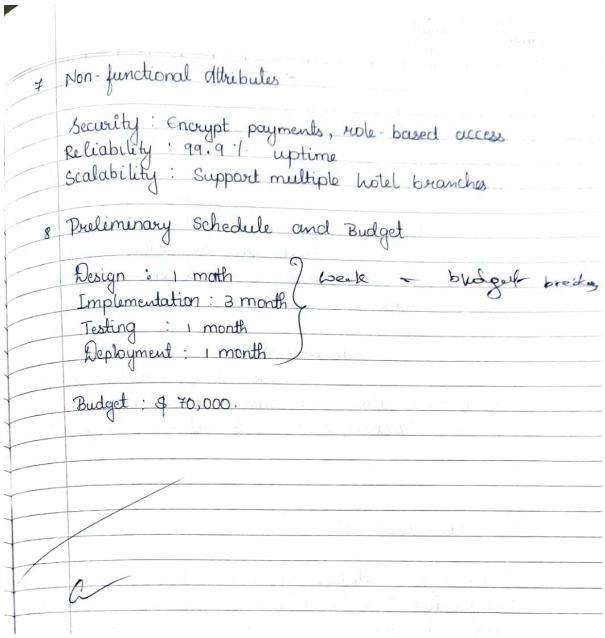
- Database : Stores room, guests, booking, payments
- Web Interface : Accessible via HTTP / HTTPS
- API's : REST APIs for transaction

5 Performance Requirements

- Room search results within 2 seconds
- Handle 1000+ concurrent users
- Database optimized for booking queries

6 Design Constraints

Tech : HTML5, CSS3, javascript,
Hardware : Server with 8GB RAM, 1TB Storage



7 Non-functional Attributes

Security : Encrypt payments, role-based access
Reliability : 99.9 % uptime
Scalability : Support multiple hotel branches

8 Preliminary Schedule and Budget

Design : 1 month
Implementation : 3 month
Testing : 1 month
Deployment : 1 month

Budget : $ 70,000.

1.3 Class Diagram

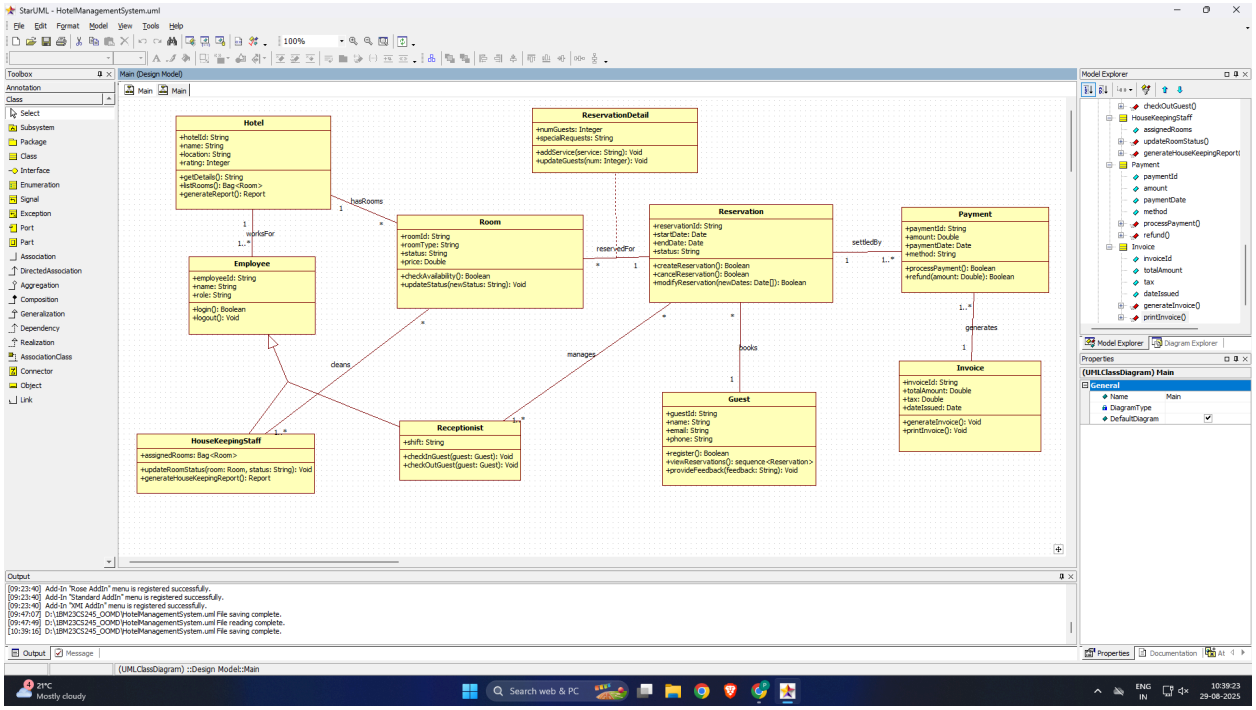


Fig 1.1

The class diagram for the Hotel Management System models the primary entities involved in hotel operations and their relationships.
It includes classes such as **Hotel, Room, Employee, Receptionist, HouseKeepingStaff, Guest, Reservation, ReservationDetail, Payment, and Invoice**, along with the enumeration **RoomStatus** and **Status**.

2.1 Major Classes & Their Responsibilities

1. Hotel

- **Attributes:** hotelId, name, location, rating
- **Operations:** getDetails(), listRooms(), generateReport()
- Represents the hotel entity and provides high-level operations for retrieving details and generating reports.

2. Room

- **Attributes:** roomId, roomType, status (RoomStatus), price
- **Operations:** checkAvailability(), updateStatus()
- A central entity that stores room information and tracks availability.

3. Guest

- **Attributes:** guestId, name, email, phone

- **Operations:** register(), viewReservations(), provideFeedback()
- Represents the customer who makes reservations and stays at the hotel.

4. Employee (Parent Class)

- **Attributes:** employeeId, name, role
- **Operations:** login(), logout()
- Superclass for hotel staff.

5. Receptionist (Subclass of Employee)

- **Attributes:** shift
- **Operations:** checkInGuest(), checkOutGuest()
- Handles front-desk operations.

6. HouseKeepingStaff (Subclass of Employee)

- **Attributes:** assignedRooms
- **Operations:** updateRoomStatus(), generateHouseKeepingReport()
- Maintains room cleanliness and updates room condition.

7. Reservation

- **Attributes:** reservationId, startDate, endDate, status
- **Operations:** createReservation(), cancelReservation(), modifyReservation()
- Represents the booking made by a guest.

8. ReservationDetail

- **Attributes:** numGuests, specialRequests
- **Operations:** addService(), updateGuests()
- Stores additional guest requirements linked to a reservation.

9. Payment

- **Attributes:** paymentId, amount, paymentDate, method, paymentStatus
- **Operations:** processPayment(), refund()
- Tracks billing and payment settlement.

10. Invoice

- **Attributes:** invoiceId, totalAmount, tax, dateIssued
- **Operations:** generateInvoice(), printInvoice()

- Generates the final invoice for completed stays.

11. RoomStatus (Enumeration)

- Available
- Occupied
- Reserved
- OutOfService

12. Status (Enumeration for Payment)

- Pending
- Completed
- Refunded

2.2 Associations & Multiplicity Explanation

Hotel — Room

- **1 to * (one hotel has many rooms)**
- Represents that a hotel manages multiple rooms.

Employee — Hotel

- **1 to * (hotel employs multiple employees)**

Receptionist — Reservation

- Receptionist **manages** reservations (1 to *)

Guest — Reservation

- **1 guest can make many reservations**
- Multiplicity: *1.. on reservation side, 1 on guest side**

Room — Reservation

- A reservation is **for one room**, but a room can be reserved **multiple times** over time.

Reservation — ReservationDetail

- **1 to 1 association (composition)** → reservation detail cannot exist without the reservation.

Reservation — Payment

- One reservation can have **one or many payments** (installments/partial payments in advanced cases).

Payment — Invoice

- Payment **generates** an invoice (1..* → 1).

HouseKeepingStaff — Room

- One staff member can **clean multiple rooms**.
- Relationship: **1 to * (cleaning)**

2.3 Aggregation, Composition & Special Cases

Composition:

- **Reservation → ReservationDetail**
  ReservationDetail cannot exist without Reservation.

Aggregation:

- **HouseKeepingStaff → assignedRooms**
  Rooms are "assigned" but not owned by staff.

Inheritance:

- **Employee** superclass with subclasses **Receptionist** and **HouseKeepingStaff**

Qualified Association:

- Room search is often **qualified by roomId or roomType**.

Ordered Association:

- Guest's reservation list is normally **ordered by date**.

1.4 Advanced Class Diagram

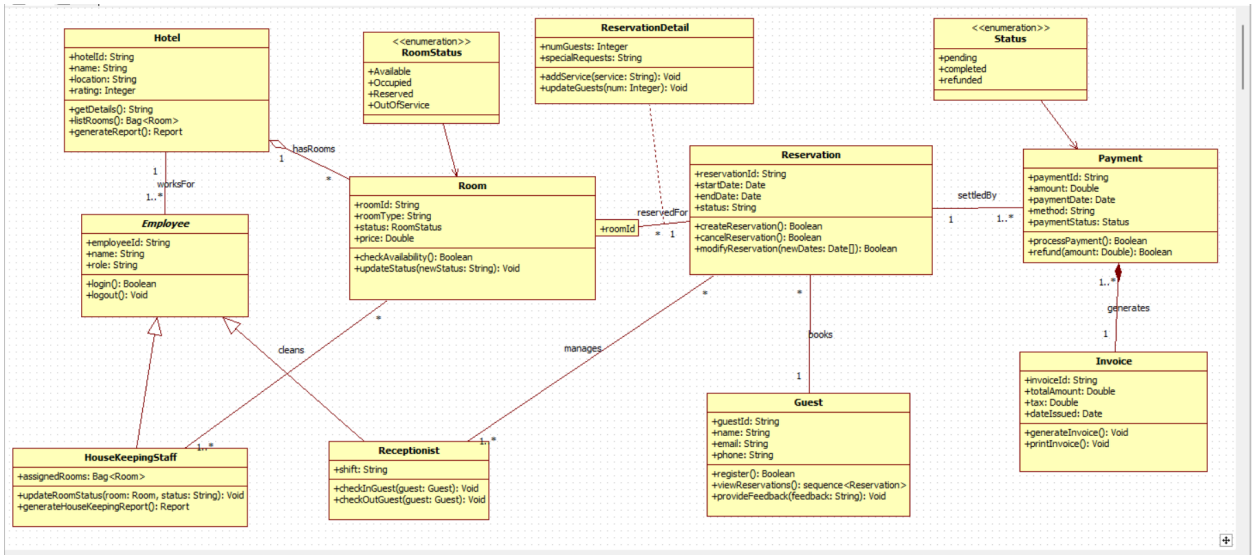


Fig 1.2

1.5 State Diagram

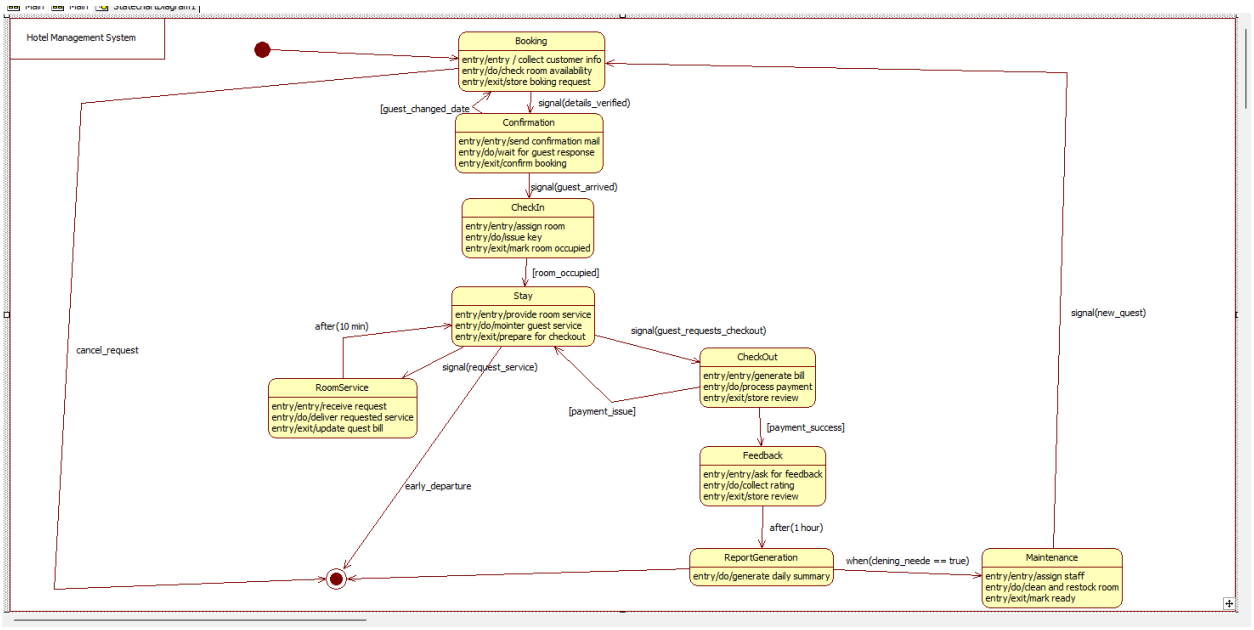


Fig 1.3

States:

1. Available
2. Reserved
3. Occupied
4. Under Cleaning
5. Out Of Service

Example transitions:

- *Available* → *Reserved* (on reservation creation)
- *Reserved* → *Occupied* (on guest check-in)
- *Occupied* → *Under Cleaning* (after check-out)
- *Under Cleaning* → *Available* (after housekeeping update)

1.6 Advanced State Diagram

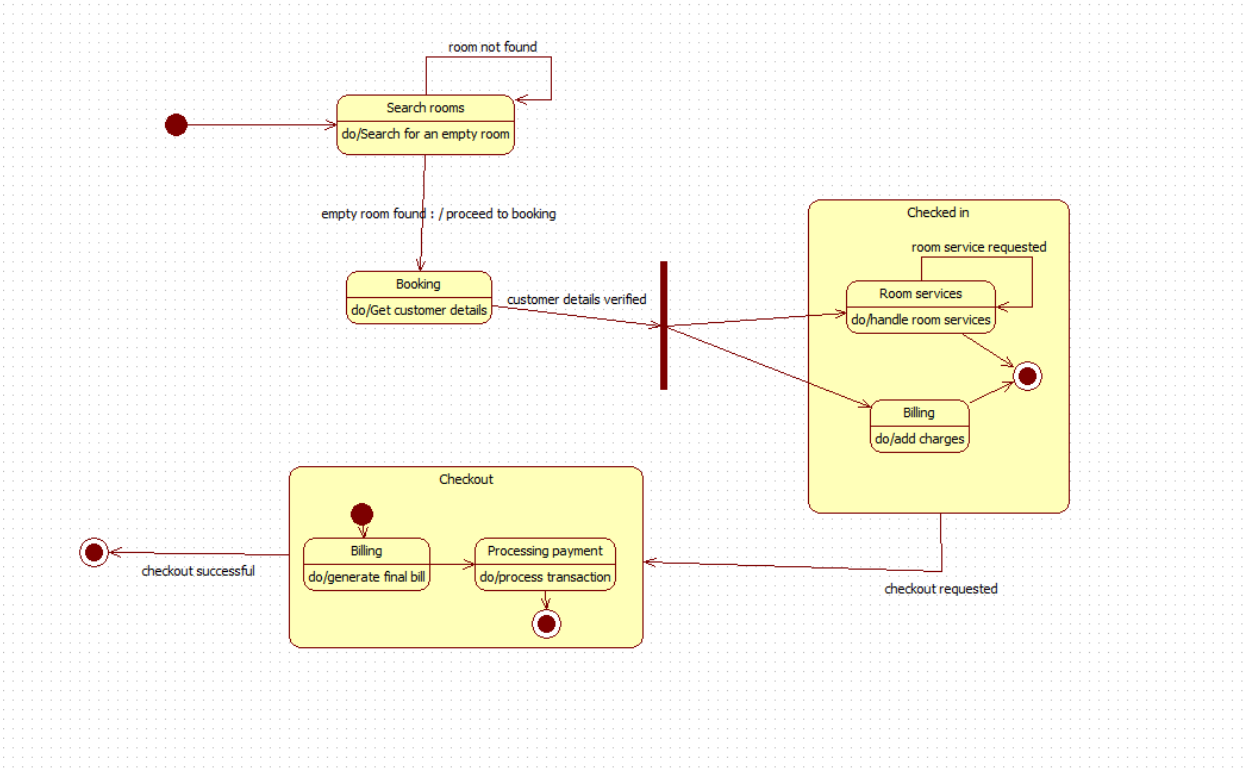


Fig 1.4

1.7 Use Case Diagram

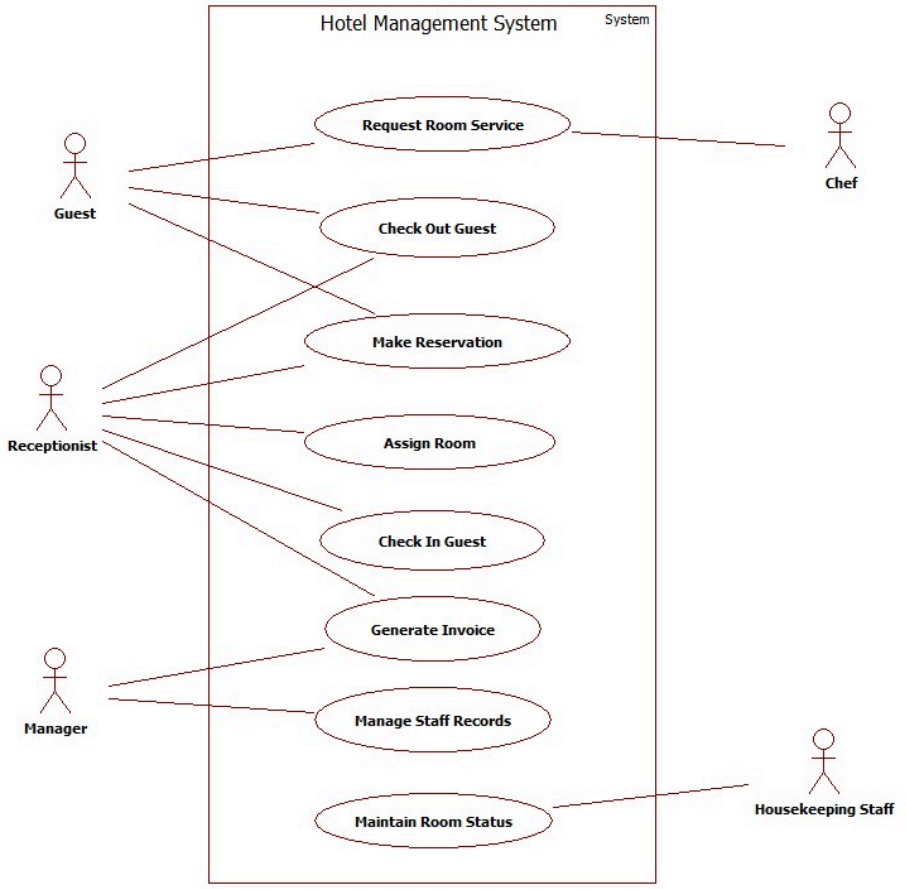


Fig 1.5

The simple use case diagram represents the basic hotel operations performed by the main users. The **Guest** can make reservations, request room service, and check out. The **Receptionist** handles back-end tasks such as check-in, check-out, assigning rooms, and generating invoices. The **Manager** manages staff-related operations, while **Housekeeping Staff** maintain room status. A **Chef** supports room service requests.

This diagram captures the core interactions between users and the system without showing detailed relationships like include/extend.

1.8 Advanced Use Case Diagram

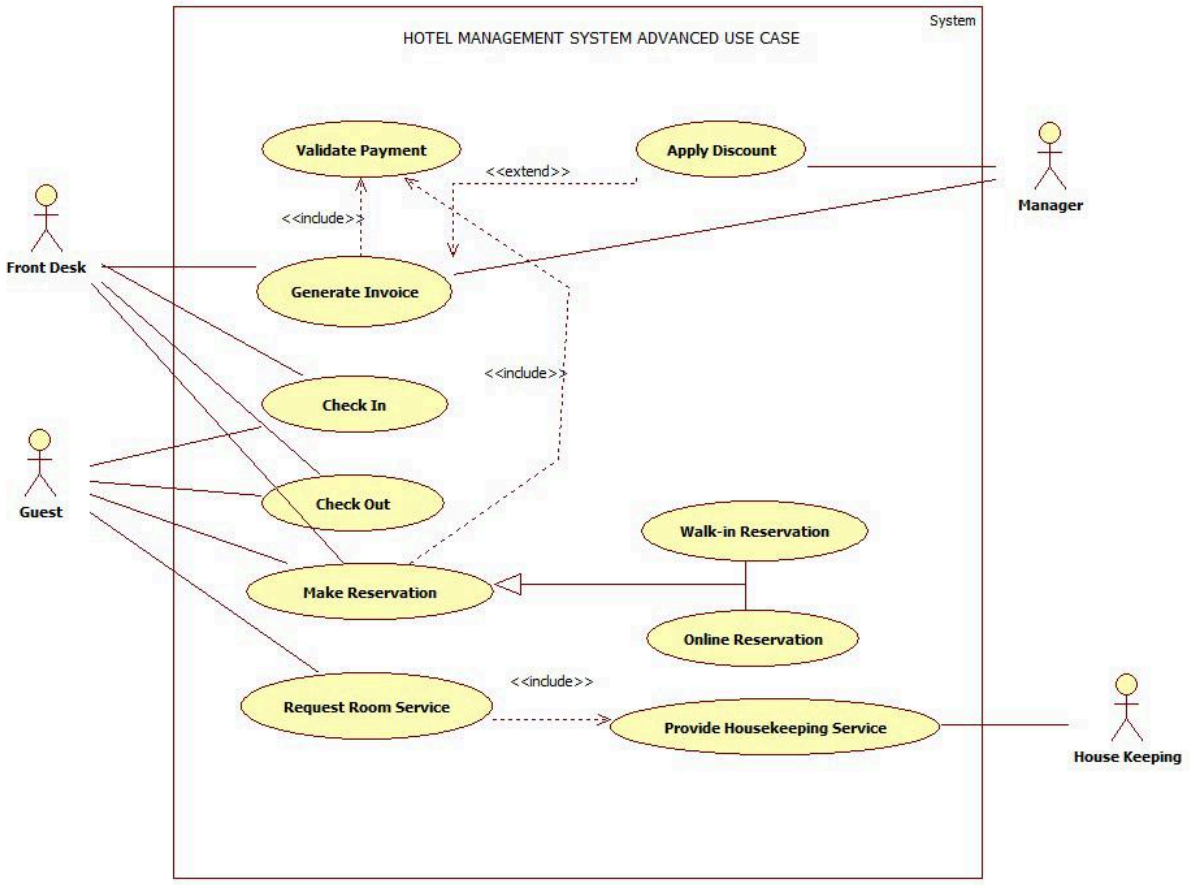


Fig 1.6

Include Relationships

These represent mandatory sub-functions.

- **Make Reservation** → *includes* **Validate Payment**
- **Check Out** → *includes* **Generate Invoice**
- **Request Room Service** → *includes* **Provide Housekeeping Service**

Extend Relationships

These represent optional or conditional tasks.

- **Validate Payment** → *extended by* **Apply Discount**
  (Manager applies discounts only under special conditions)

Generalization

Shows actor specialization:

- **Reservation** is generalized into:

- **Walk-in Reservation**
- **Online Reservation**

- **Front Desk** is a specialized form of **Receptionist**
- **Housekeeping** is a specialized form of service provider for service requests

1.9 Sequence Diagram

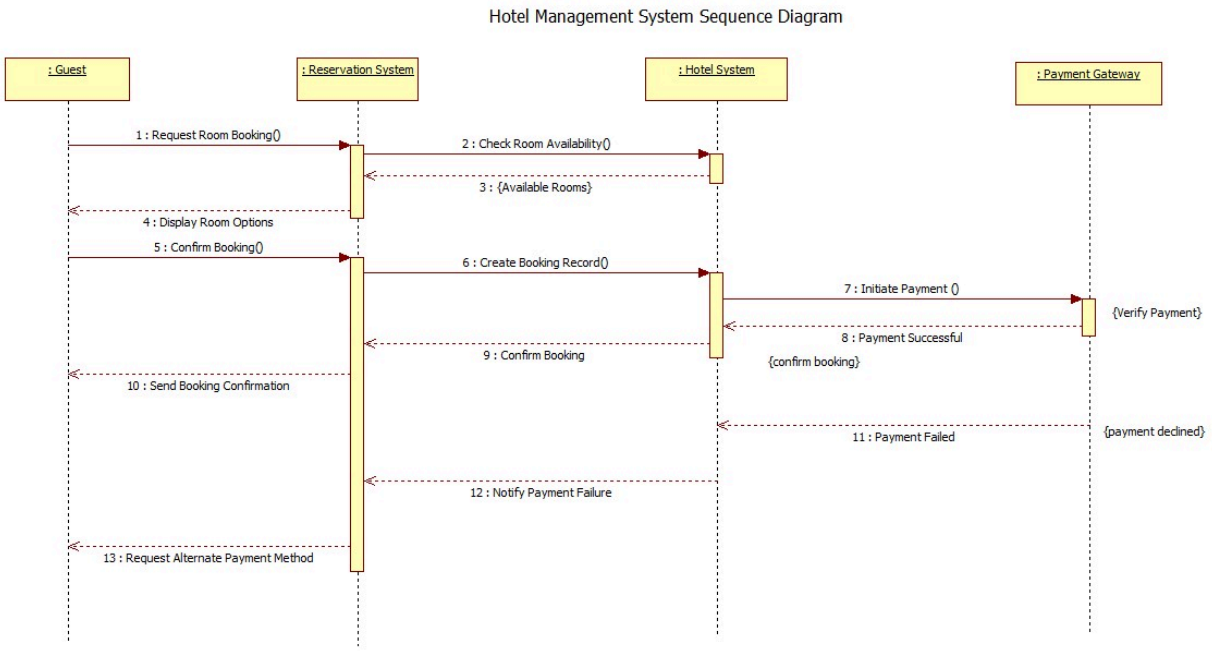


Fig 1.7

Scenario 1: Room Booking (Use Case 1)

Complete Scenario Description

1. The **Guest** initiates the booking process by sending a *Request Room Booking* message to the **Reservation System**.
2. The **Reservation System** contacts the **Hotel System** to *Check Room Availability*.
3. The **Hotel System** returns a list of *Available Rooms*.
4. The **Reservation System** displays available options to the **Guest**.
5. The **Guest** selects a room and sends *Confirm Booking*.
6. The **Reservation System** creates a *Booking Record* in the **Hotel System**.

7. The **Hotel System** initiates *Payment* by interacting with the **Payment Gateway**.
8. If the payment is successful, the **Payment Gateway** returns *Payment Successful*.
9. The **Hotel System** confirms the booking with the **Reservation System**.
10. The **Reservation System** sends *Booking Confirmation* to the **Guest**.
11. If payment fails, the **Payment Gateway** returns *Payment Failed*.
12. The **Reservation System** notifies the **Guest** of payment failure.
13. The **Guest** is prompted to use an alternate payment method.

Scenario 2: Guest Check-In and Service Request (Use Case 2)

Complete Scenario Description

1. The **Guest** sends a *requestRoom()* message to the **Reservation System**.
2. The **Reservation System** checks room availability with the **Hotel System**.
3. The **Hotel System** returns a list of available rooms.
4. The **Reservation System** sends these options back to the **Guest**.
5. Once the **Guest** confirms a room, the Reservation System creates a booking in the **Hotel System**.
6. The **Hotel System** responds with a *bookingId*.
7. The Reservation System initiates a **StaySession** object using the message <<create>>startStay().
8. The **Hotel System** assigns a room and returns the room number.
9. The **Services** subsystem issues the room key (issueKey()) and provides it to the **Guest**.
10. During the stay, the guest can request additional services (e.g., laundry, cleaning).
11. The **Guest** sends a service request via the **StaySession** to the **Services** subsystem.
12. The Services subsystem fulfills the request and updates the stay session.

13. At checkout time, the **Guest** initiates the *checkout()* request.
14. The **Hotel System** completes the stay, settles all services, and returns confirmation.
15. The **StaySession** object is destroyed (<<destroy>>).
16. Finally, the Reservation System sends a receipt containing invoice ID and transaction ID to the Guest.

1.10 Advanced Sequence Diagram

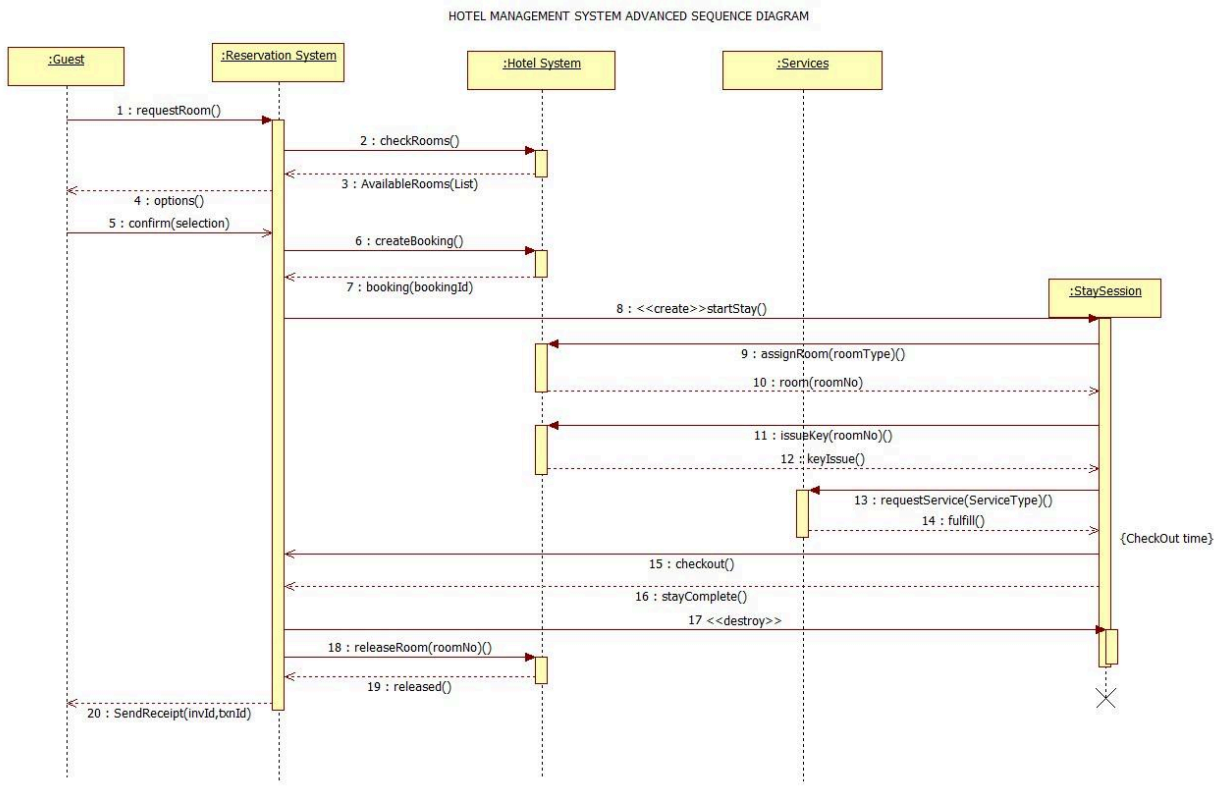


Fig 1.8

Transient Objects

These objects are **created temporarily** during the interaction and destroyed afterwards.

- **StaySession**
  - Created when the guest begins their stay.
  - Maintains temporary information such as room assignment and in-stay services.

- Destroyed once checkout is complete.
- Represented using <<create>> and <<destroy>>.

- **RoomKey** (implicit from keyIssue())
  - Exists only during room allocation.
  - Not stored permanently.

Passive Objects

Objects that **do not initiate actions** but participate by storing or providing data.

- **Booking Record**
  - Stores bookingId, room number, guest details.
  - Does not control the flow, only holds data.
- **Services**
  - Responds to requests like assigning rooms or issuing keys.
  - Does not initiate any sequence on its own.

1.11 Activity Diagram

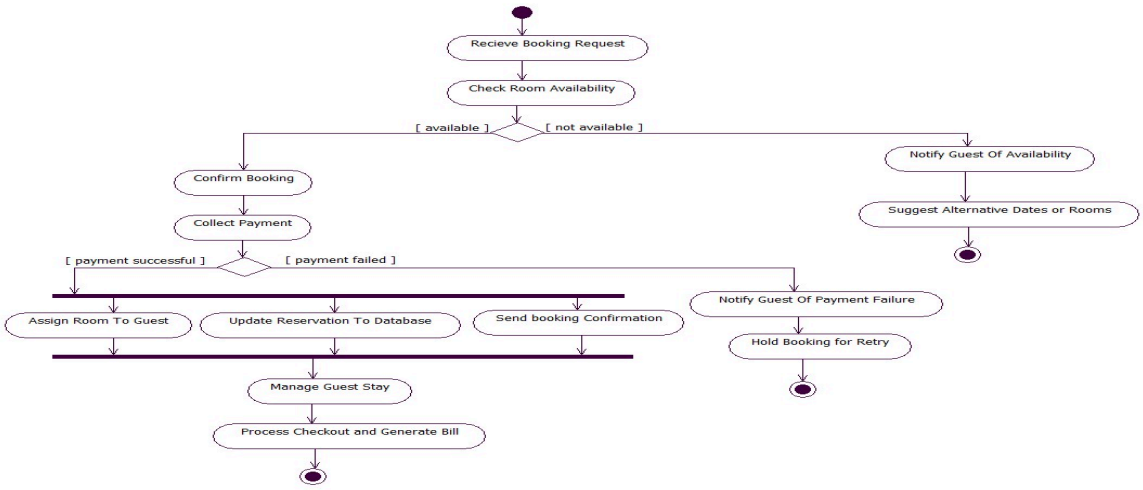


Fig 1.9

1. The process starts when the system receives a booking request from the guest.
2. The system checks room availability.
3. A decision is made:
4. If rooms are not available, the guest is notified and alternative dates/rooms are suggested.
5. If rooms are available, the system moves to confirm the booking.
6. The system then attempts to collect payment from the guest.
7. Payment status leads to two branches:
8. Payment successful:
    a. Room is assigned to the guest.
    b. Reservation details are updated in the database.
    c. Booking confirmation is sent to the guest.
9. Payment failed:
    a. The guest is notified about payment failure.
    b. Booking is placed on hold for retry.
10. After assignment, the system proceeds to manage the guest stay.
11. Finally, the system processes checkout, generates the bill, and the activity ends.

1.12 Advanced Activity Diagram

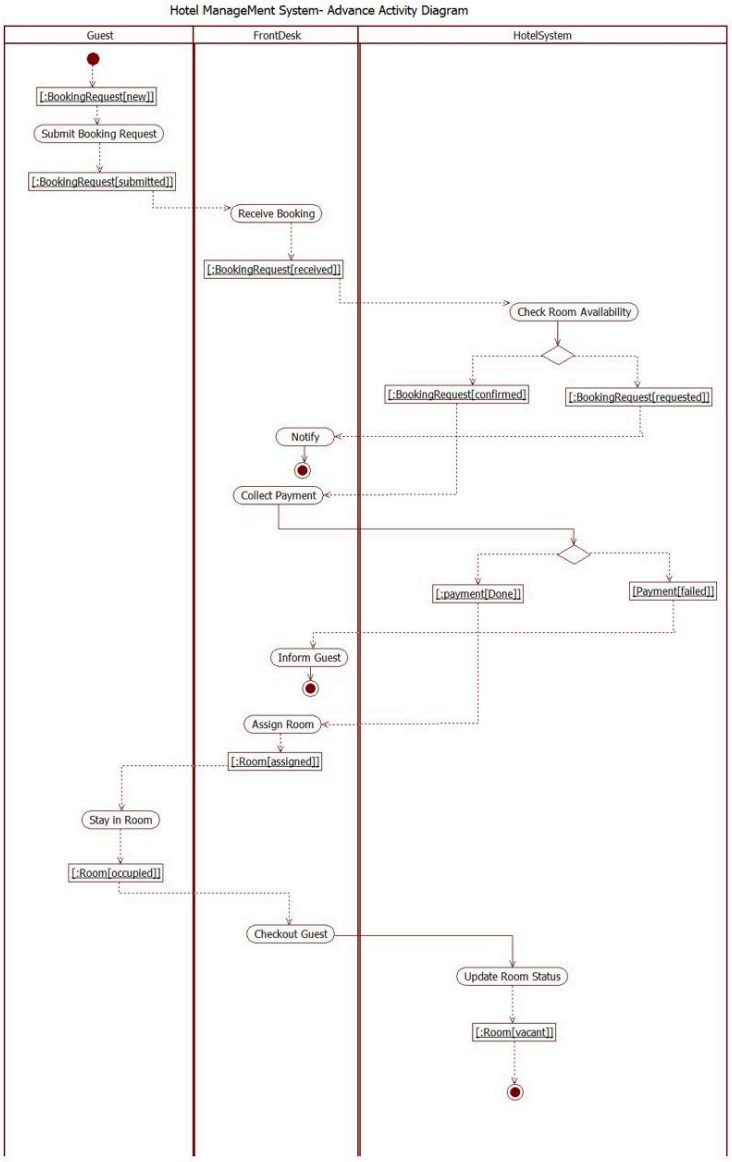


Fig 1.10

1. The **Guest** initiates the process by submitting a booking request.
2. The **Front Desk** receives the booking and acknowledges the request.
3. The **Hotel System** checks room availability and returns the result.
4. The Front Desk **notifies the guest** about room availability.
5. The Front Desk proceeds to **collect payment**.
6. Based on the payment status:

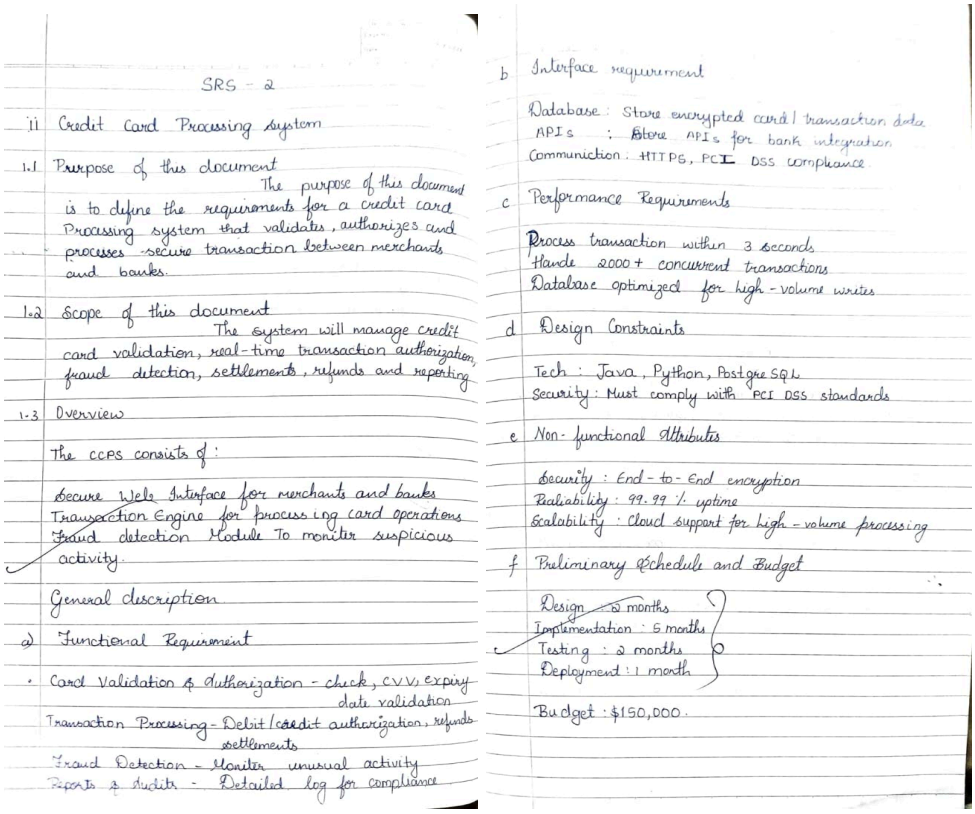
7. If **payment succeeds**, the Hotel System updates booking status.
8. If **payment fails**, the process branches to failure handling.
9. After confirmation, the Front Desk **assigns a room** to the guest.
10. The Guest then proceeds to **stay in the room**, marking the occupancy.
11. At checkout time, the Front Desk initiates **checkout processing**.
12. The Hotel System **updates room status** to vacant.
13. The activity ends after all states and updates are completed.

2.Credit Card Processing

2.1Problem Statement

The Credit Card Processing System enables secure online and offline payment transactions between customers and merchants. The system validates card details, checks available credit, authenticates customers, processes authorization requests through the payment gateway, and obtains approval from the bank. It also handles declined transactions, fraud checks, and generates transaction confirmations for both customers and merchants. The goal is to ensure fast, reliable, and secure credit card payments through proper verification and settlement mechanisms.

2.2 SRS-Software Requirements Specification



SRS - 2

ii Credit Card Processing System

1.1 Purpose of this document

The purpose of this document is to define the requirements for a credit card Processing system that validates, authorizes and processes secure transaction between merchants and banks.

1.2 Scope of this document

The system will manage credit card validation, real-time transaction authorization, fraud detection, settlements, refunds and reporting

1.3 Overview

The CCPS consists of:

Secure Web Interface for merchants and banks
Transaction Engine for processing card operations
Fraud detection Module To monitor suspicious activity.

General description

a) Functional Requirement

- Card Validation & Authorization - check, CVV, expiry date validation
Transaction Processing - Debit/credit authorization, refunds settlements
Fraud Detection - Monitor unusual activity
Reports & Audits - Detailed log for compliance

b Interface requirement

Database : Store encrypted card/transaction data
APIs : Store APIs for bank integration
Communication : HTTPS, PCI DSS compliance

c Performance Requirements

Process transaction within 3 seconds
Handle 2000+ concurrent transactions
Database optimized for high-volume writes

d Design Constraints

Tech : Java, Python, PostgreSQL
Security : Must comply with PCI DSS standards

e Non-functional Attributes

Security : End-to-End encryption
Reliability : 99.99 % uptime
Scalability : Cloud support for high-volume processing

f Preliminary Schedule and Budget

Design - 2 months
Implementation : 5 months
Testing : 2 months
Deployment : 1 month

Budget : $150,000.

2.3 Class Diagram

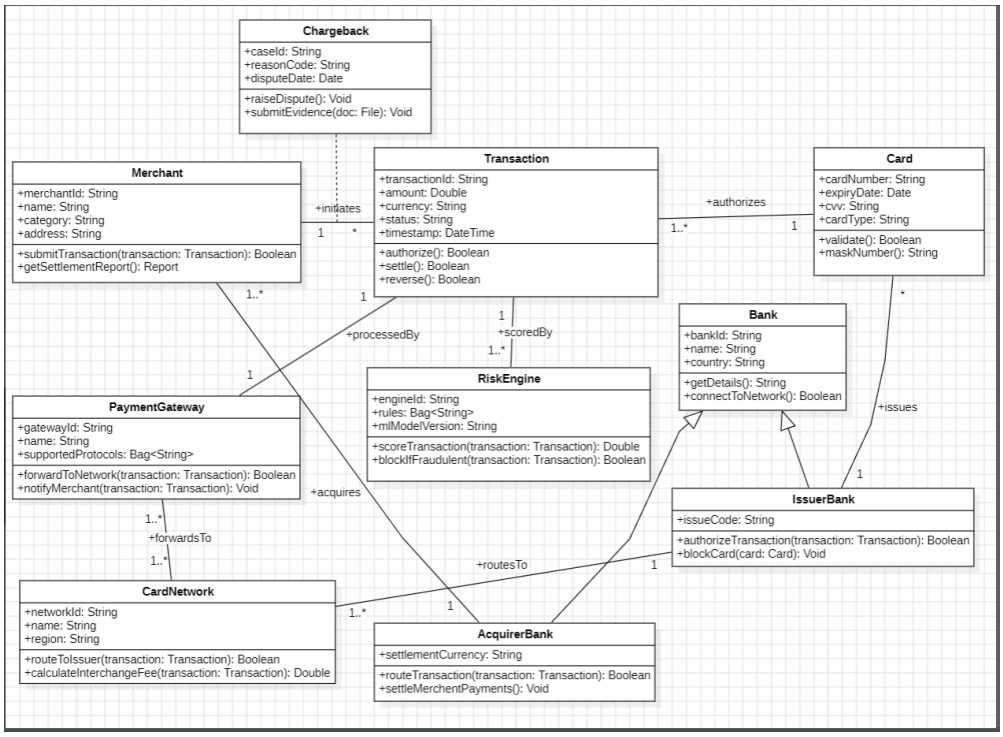


Fig 2.1

The **Merchant** class initiates transactions and sends information to the system. The **Transaction** class stores transaction details such as amount, currency, status, and timestamp. Each transaction is validated and processed with the help of the **Card** class, which contains card details like card number, expiry date, CVV, and card type.

The **PaymentGateway** class forwards transactions between the merchant and the banking network, and communicates with the **CardNetwork** for routing. The **IssuerBank** authorizes or declines the transaction based on card validity and available credit, while the **AcquirerBank** handles settlement for the merchant.

The **RiskEngine** evaluates transactions for fraud detection using rules and ML models. When disputes arise, the **Chargeback** class manages dispute raising and evidence submission. The **Bank** class maintains bank details and connects to the card network. Together, these classes represent a complete ecosystem for validating transactions, detecting fraud, authorizing payments, and settling funds securely.

2.4 Advanced Class Diagram

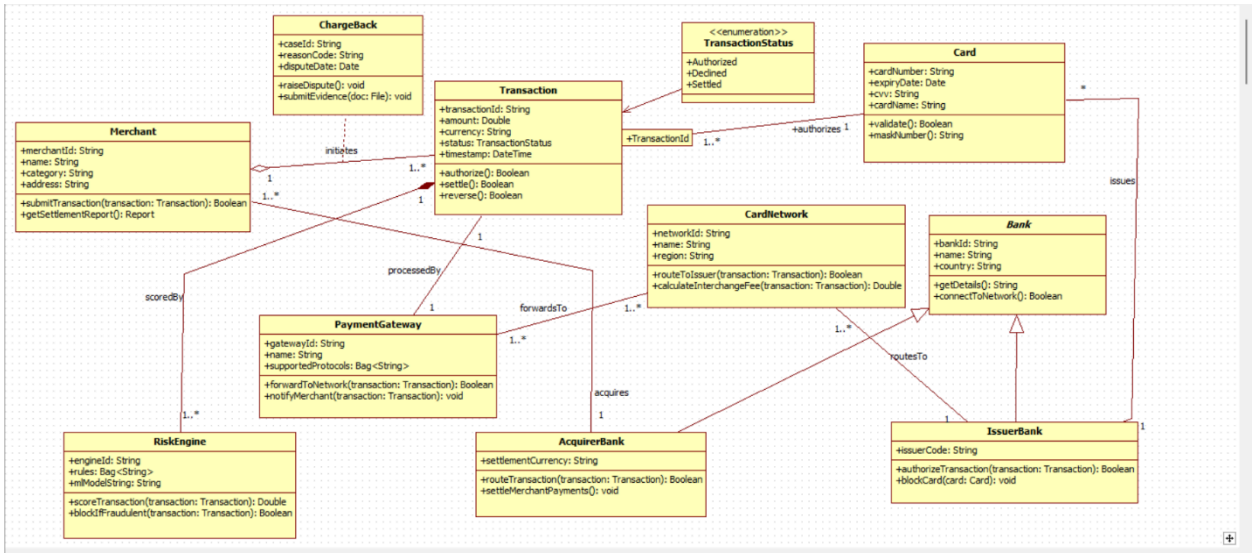


Fig 2.2

The advanced class diagram represents a complete credit card processing ecosystem involving merchants, banks, networks, and fraud detection components. The **Transaction** class is central, storing transaction details and supporting operations such as authorization, settlement, and reversal. A **Merchant** initiates multiple transactions, while the **Card** class holds sensitive card information and is linked to the **IssuerBank**, which authorizes or blocks the card.

The **PaymentGateway** forwards transactions to the **CardNetwork**, which routes them to the correct issuer and calculates interchange fees. The **AcquirerBank** settles payments for the merchant, whereas the **RiskEngine** scores transactions for fraud and can block risky ones. The **Chargeback** class manages disputes raised for incorrect or fraudulent transactions. Together, these components model a realistic financial workflow, ensuring secure routing, authorization, fraud analysis, and settlement within the credit card processing system.

2.5 State Diagram

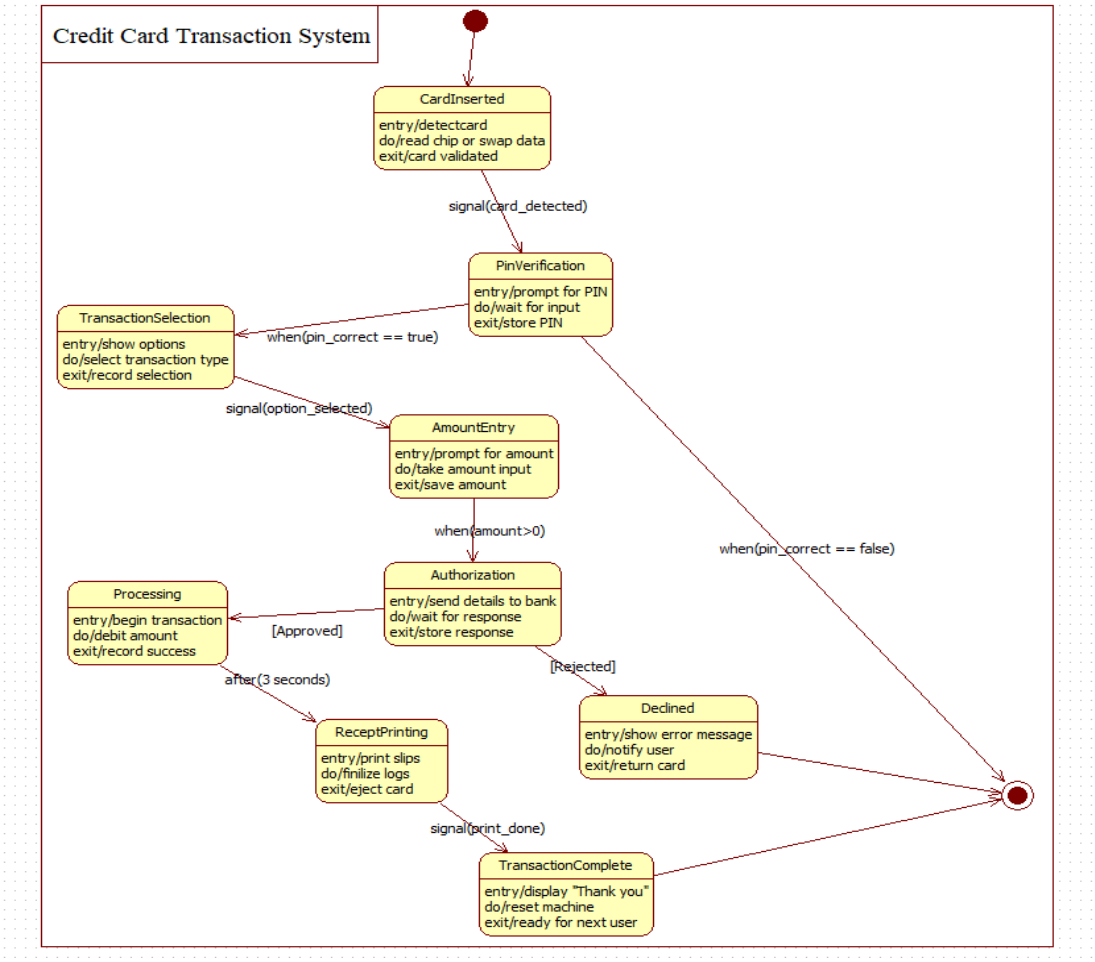


Fig 2.3

This state diagram describes the main states of a credit card transaction.
The system begins by receiving a transaction request and verifying card details. If invalid, the transaction is immediately rejected. If valid, the system checks the available credit limit.

With sufficient limit, the transaction enters the *Authorized* state and the amount is debited. After this, the system updates all records and sends notifications before generating the monthly statement. With insufficient credit, the transaction goes to *Declined* state and the user is notified.

2.6 Advanced State Diagram

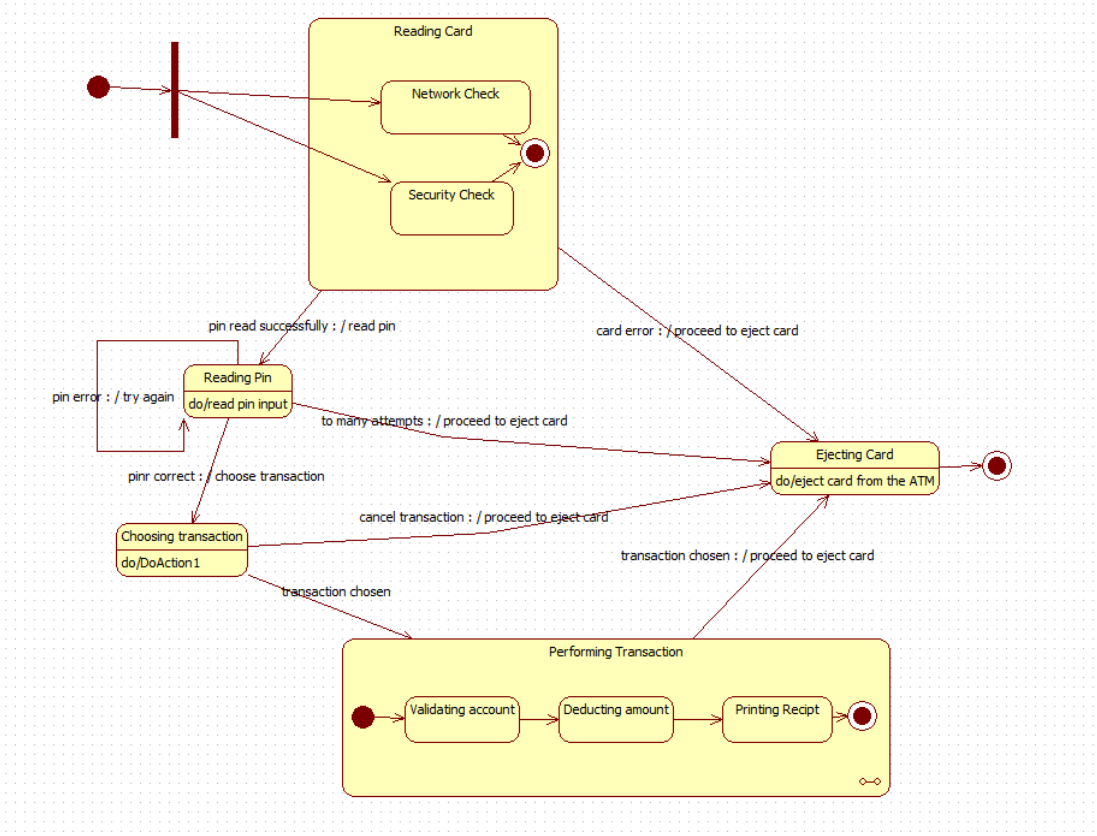


Fig 2.4

The advanced state diagram covers ATM-style credit card operations.
The flow begins with card insertion followed by network and security checks. The PIN verification state allows retries and failure transitions. Once the PIN is correct, users select a transaction type and enter the amount.

The authorization state communicates with the bank for approval. If approved, processing and receipt printing occur. If rejected, the user is notified and the card is returned. The flow ends with transaction completion and machine reset.

2.7 Use Case Diagram

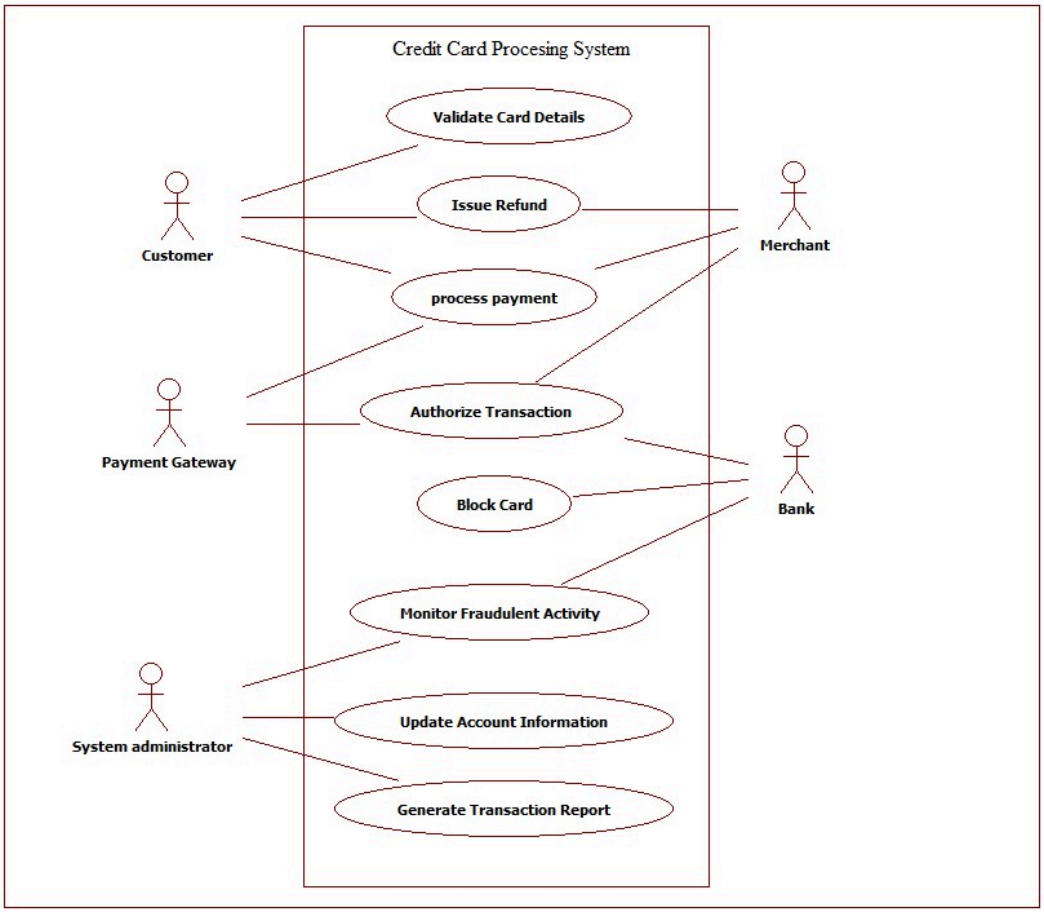


Fig 2.5

The simple use case diagram shows the high-level interactions between system actors. The **Customer** initiates payments, requests refunds, and validates card details. The **Merchant** processes payments and interacts with the bank for authorization. The **Payment Gateway** verifies card information and forwards authorization requests. The **Bank** authorizes transactions and blocks cards in case of suspicious activity. The **System Administrator** monitors fraudulent activities, updates account details, and generates reports.

The system supports essential use cases like card validation, payment processing, transaction authorization, card blocking, fraudulent activity monitoring, and report generation.

2.7 Advanced Use Case Diagram

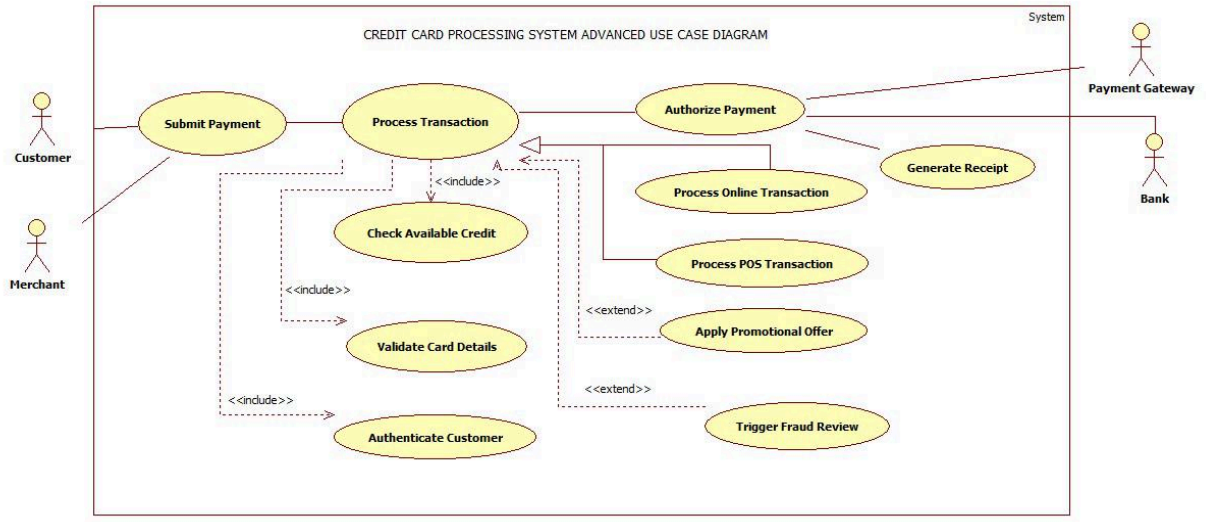


Fig 2.6

The advanced use case diagram provides a detailed set of operations.
The **Customer** submits a payment request, while the **Merchant** initiates transaction processing. The **Process Transaction** use case includes verifying available credit, validating card details, and authenticating the customer. The **Authorize Payment** use case further includes online transaction processing and POS transaction processing.

Promotional offers may extend from transaction processing, and fraud review may extend from authorization. The **Payment Gateway** generates receipts, while the **Bank** assists in authorization. This diagram captures a comprehensive flow of validation, authorization, fraud detection, and settlement, reflecting real-world processing steps.

2.8 Sequence Diagram

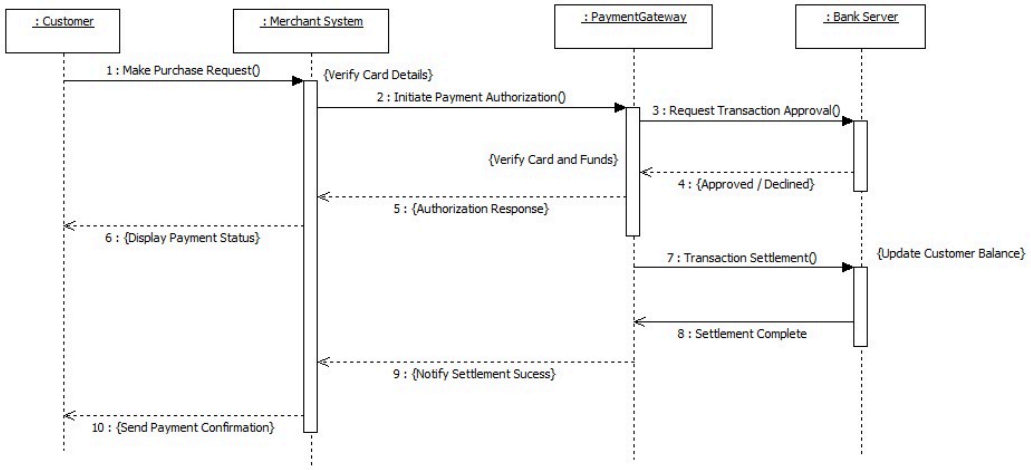


Fig 2.7

The simple sequence diagram shows the communication between the four main entities.
The **Customer** makes a purchase request, and the **Merchant System** verifies card details and sends an authorization request to the **Payment Gateway**. The gateway forwards the request to the **Bank Server**, which checks card validity and funds availability.

The bank sends back an authorization response (approved or declined). If approved, the gateway proceeds with the settlement process, and the merchant updates the customer with the payment status. This ensures proper authorization, fund verification, and completion of the transaction.

2.9 Advanced Sequence Diagram

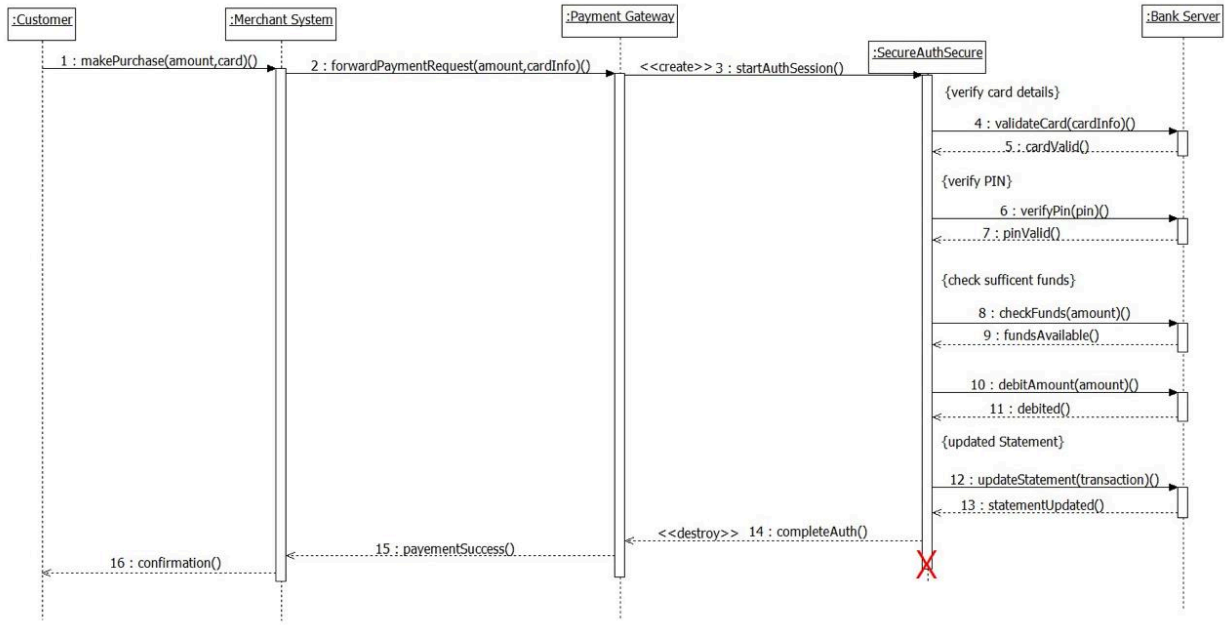


Fig 2.8

The advanced diagram includes more detailed authentication and verification steps.
The **Customer** initiates a payment request, and the **Merchant System** forwards the request to the **Payment Gateway**, which creates an authentication session. The **SecureAuthSecure** component validates the card, verifies the PIN, checks funds, debits the amount, and updates the account statement through the **Bank Server**.

After all steps succeed, the authentication session is destroyed, the payment gateway confirms success to the merchant, and the customer receives final confirmation. This detailed diagram reflects deeper security and fraud-prevention mechanisms.

2.10 Activity Diagram

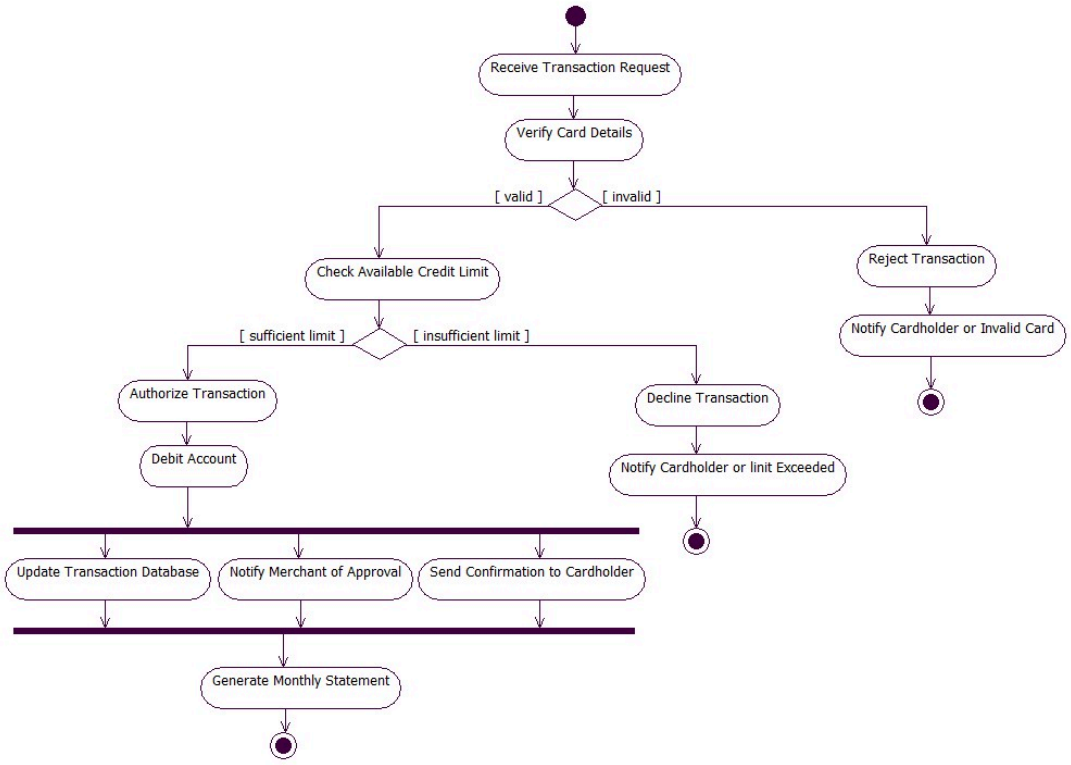


Fig 2.9

The simple activity diagram outlines basic transaction flow.
The process begins by receiving a transaction request and verifying card details. If the card is

invalid, the transaction is rejected. If valid, the system checks the available credit limit. If the limit is sufficient, the transaction is authorized and the account is debited.

Next, the system updates the transaction database, notifies the merchant, and sends confirmation to the cardholder. Finally, a monthly statement is generated. This diagram represents simple decision-based payment processing.

2.11 Advanced Activity Diagram

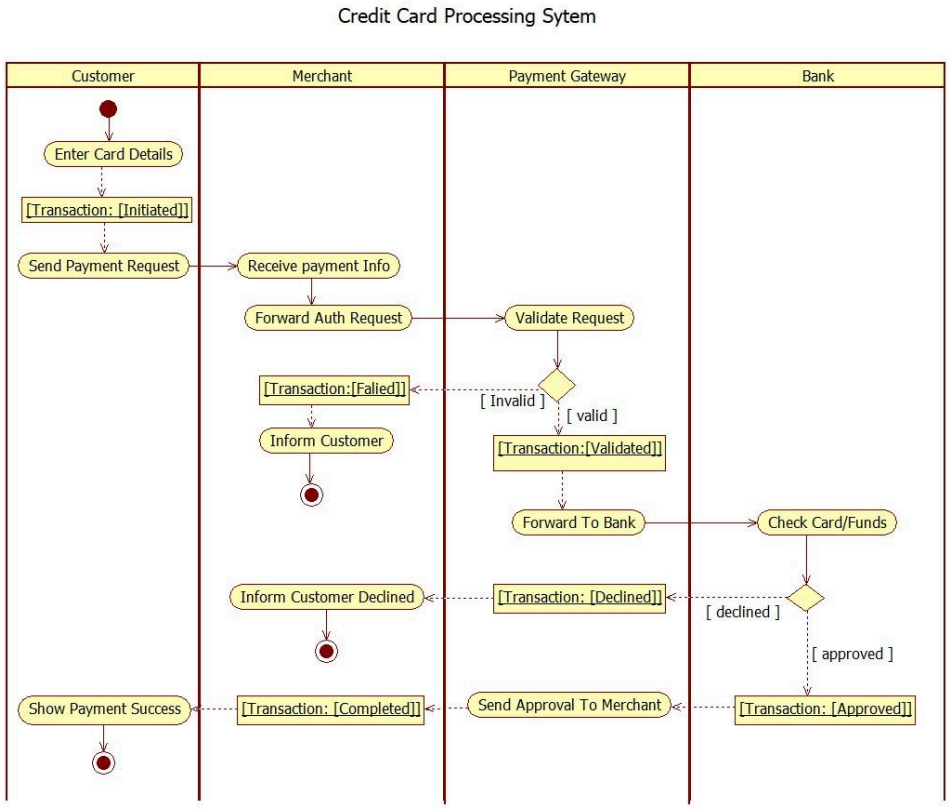


Fig 2.10

The advanced activity diagram includes more detailed swimlane-based flow.
The transaction starts with card reading and network/security checks. If errors occur, the card is ejected. On successful reading, the PIN is verified with retry options. A valid PIN leads to transaction selection.

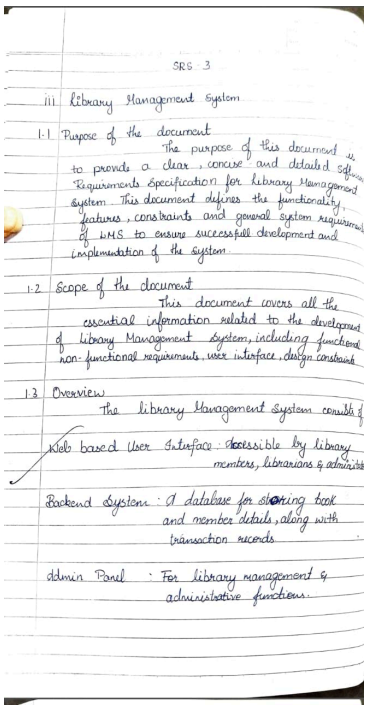
The **Performing Transaction** activity includes validating the account, deducting the amount, and printing the receipt. Any errors trigger card ejection. This diagram highlights detailed ATM-style credit card transaction behavior with error handling and multi-step verification.

3.Library Management System

3.1Problem Statement

The Library Management System (LMS) is designed to automate core library operations such as searching books, borrowing, returning, reserving, renewing, and paying fines. The system maintains information about books, members, staff, and loan records. It ensures efficient catalog management, availability checking, overdue fine calculation, and updating of inventory. LMS also supports staff functions like issuing and accepting returned books, managing memberships, and updating catalog records. The goal is to provide a smooth, accurate, and organized library workflow for both users and staff.

3.2 SRS-Software Requirements Specification



SRS - 3

1) Library Management System

1.1 Purpose of the document

The purpose of this document is to provide a clear, concise and detailed software Requirements Specification for Library Management System. This document defines the functionality, features, constraints and general system requirements of LMS to ensure successfull development and implementation of the system.

1.2 Scope of the document

This document covers all the essential information related to the development of Library Management System, including functional non-functional requirements, user interface, design constraints

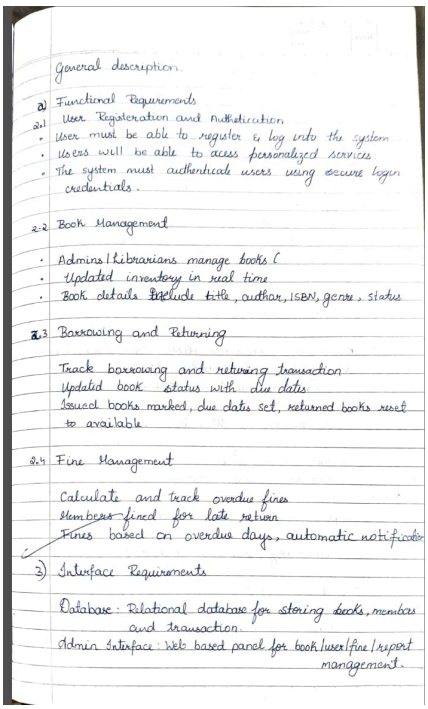
1.3 Overview

The library Management system consists of

Web based User Interface: Accessible by library members, librarians & administrators

Backend System: A database for storing book and member details, along with transaction records

Admin Panel: For library management & administrative functions.



General description

2) Functional Requirements

2.1 User Registeration and Authentication

- User must be able to register & log into the system
- Users will be able to access personalized services
- The system must authenticate users using secure login credentials.

2.2 Book Management

- Admins/Librarians manage books (
- Updated inventory in real time
- Book details include title, author, ISBN, genre, status

2.3 Borrowing and Returning

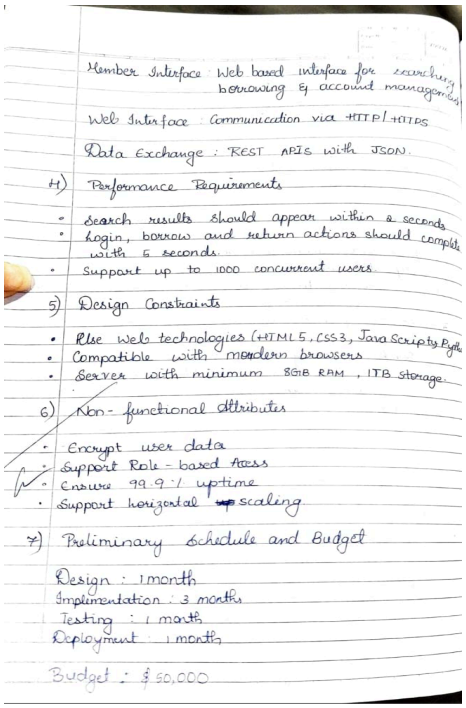
Track borrowing and returning transaction
Updated book status with due dates
Issued books marked, due dates set, returned books reset to available

2.4 Fine Management

Calculate and track overdue fines
Members fined for late return
Fines based on overdue days, automatic notification

3) Interface Requirements

Database: Relational database for storing books, members and transaction.
Admin Interface: Web based panel for book/user/fine/report management.



Member Interface: Web based interface for searching, borrowing & account management

Web Interface: Communication via HTTP/HTTPS

Data Exchange: REST APIs with JSON.

4) Performance Requirements

- Search results should appear within 2 seconds
- Login, borrow and return actions should complete with 5 seconds
- Support up to 1000 concurrent users

5) Design Constraints

- Use Web technologies (HTML5, CSS3, JavaScript, Python)
- Compatible with modern browsers
- Server with minimum 8GB RAM, 1TB storage

6) Non-functional attributes

- Encrypt user data
- Support Role-based Access
- Ensure 99.9% uptime
- Support horizontal scaling

7) Preliminary Schedule and Budget

Design: 1 month
Implementation: 3 months
Testing: 1 month
Deployment: 1 month

Budget: $50,000

3.3 Class Diagram

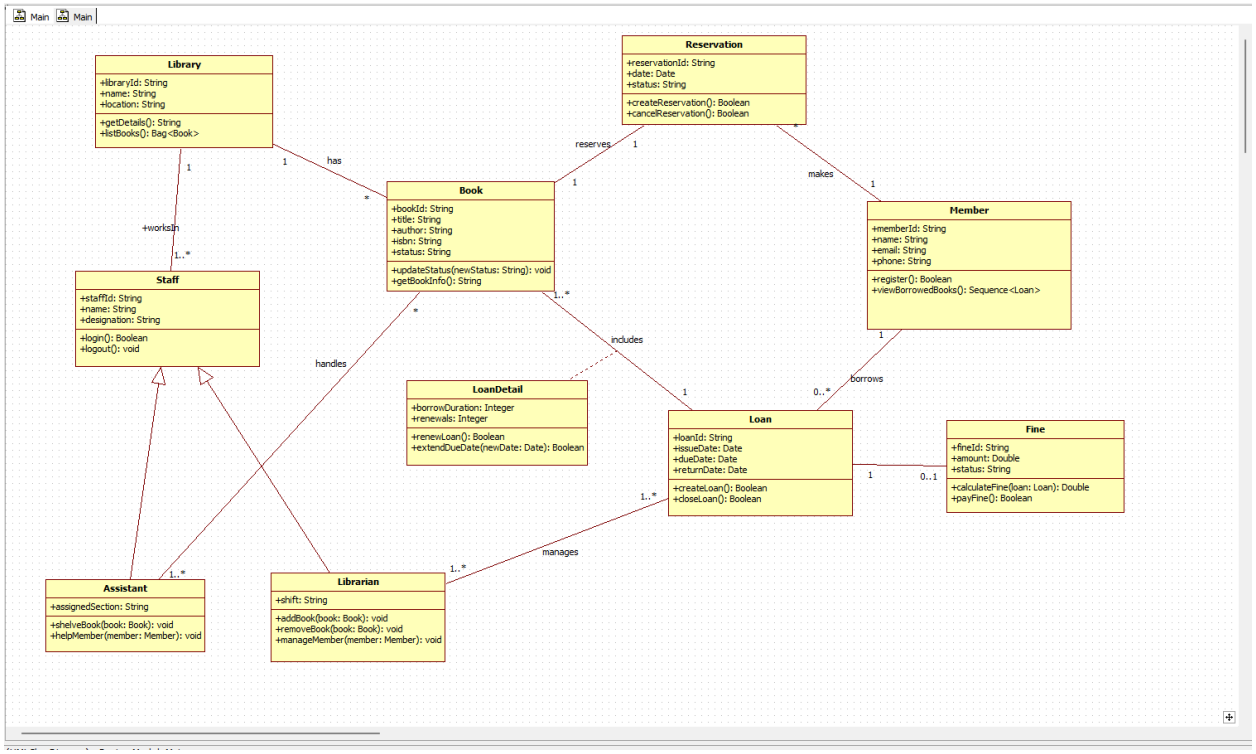


Fig 3.1

The class diagram models all major entities of the Library Management System.
The **Library** contains multiple books and staff members. The **Book** class stores details such as title, author, ISBN, and status (Available, Borrowed, Reserved). **Members** borrow books through **Loan** objects, which record issue date, due date, and return date. **LoanDetail** captures borrowing duration and renewals.

Staff are modeled through the **Staff** class, which has two specializations: **Librarian** and **Assistant**, each handling tasks such as adding/removing books or helping members. The **Reservation** class stores reservation requests. The **Fine** class manages fine amount, status, and calculations when books are returned late.

Together, these classes represent the complete structural framework required for managing library transactions, membership, staff actions, and inventory.

3.4 Advanced Class Diagram

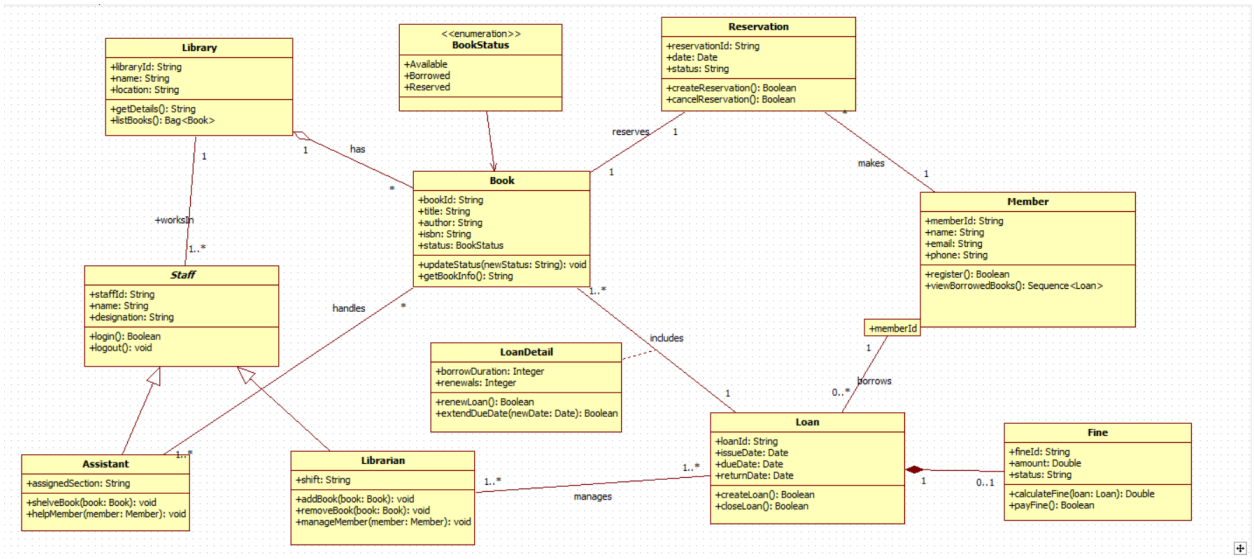


Fig 3.2

The advanced class diagram represents all major components and relationships within the Library Management System. The **Library** class maintains information about the library and has multiple **Book** objects. Each **Book** includes details such as title, author, ISBN, and a status from the **BookStatus** enumeration (Available, Borrowed, Reserved).

Members are represented through the **Member** class, which stores user information and allows viewing of borrowed books. A member can borrow multiple books through the **Loan** class, which records issue date, due date, return date, and is linked with **LoanDetail** for handling renewals and borrow duration. Overdue returns generate a **Fine**, which stores fine amount, status, and provides operations to calculate and pay fines.

Reservations are captured using the **Reservation** class, allowing members to reserve a book when unavailable. The **Staff** class models employees working in the library and is specialized into **Librarian** and **Assistant**. Librarians manage adding/removing books and member operations, while Assistants help with shelving and member support.

Overall, this advanced diagram shows a complete structural model capturing inventory management, borrowing, returning, reservations, renewals, fines, and staff operations necessary for an efficient and automated library system

3.5 State Diagram

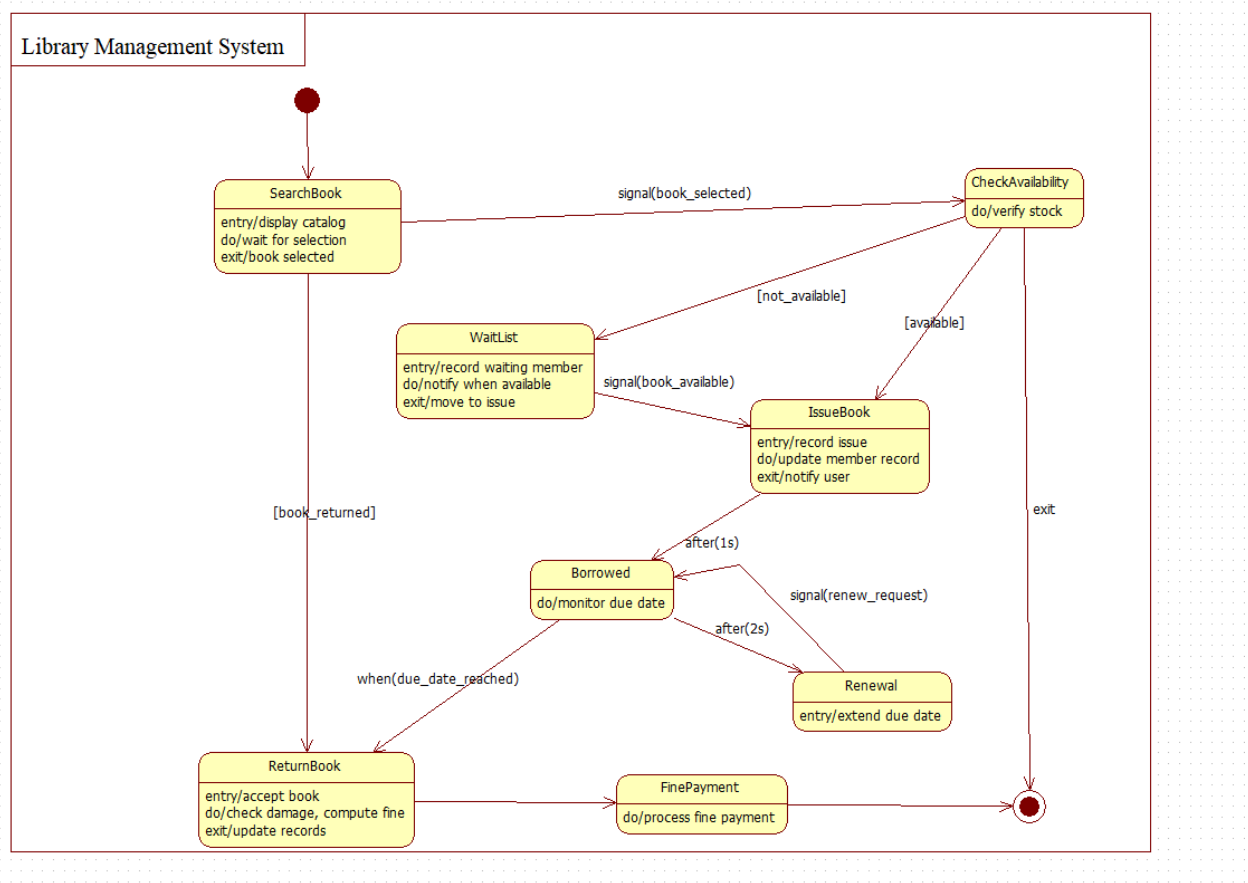


Fig 3.3

The simple state diagram shows the main states of a book request.
The process begins with a book request, followed by member ID verification. If the ID is valid, the system checks book availability. If available, the book is issued and the borrow record updated. If unavailable, the user is notified or added to a waitlist. After reading, the return state updates book inventory and handles fine calculation if overdue. This diagram expresses the lifecycle of a single borrow cycle.

3.6 Advanced State Diagram

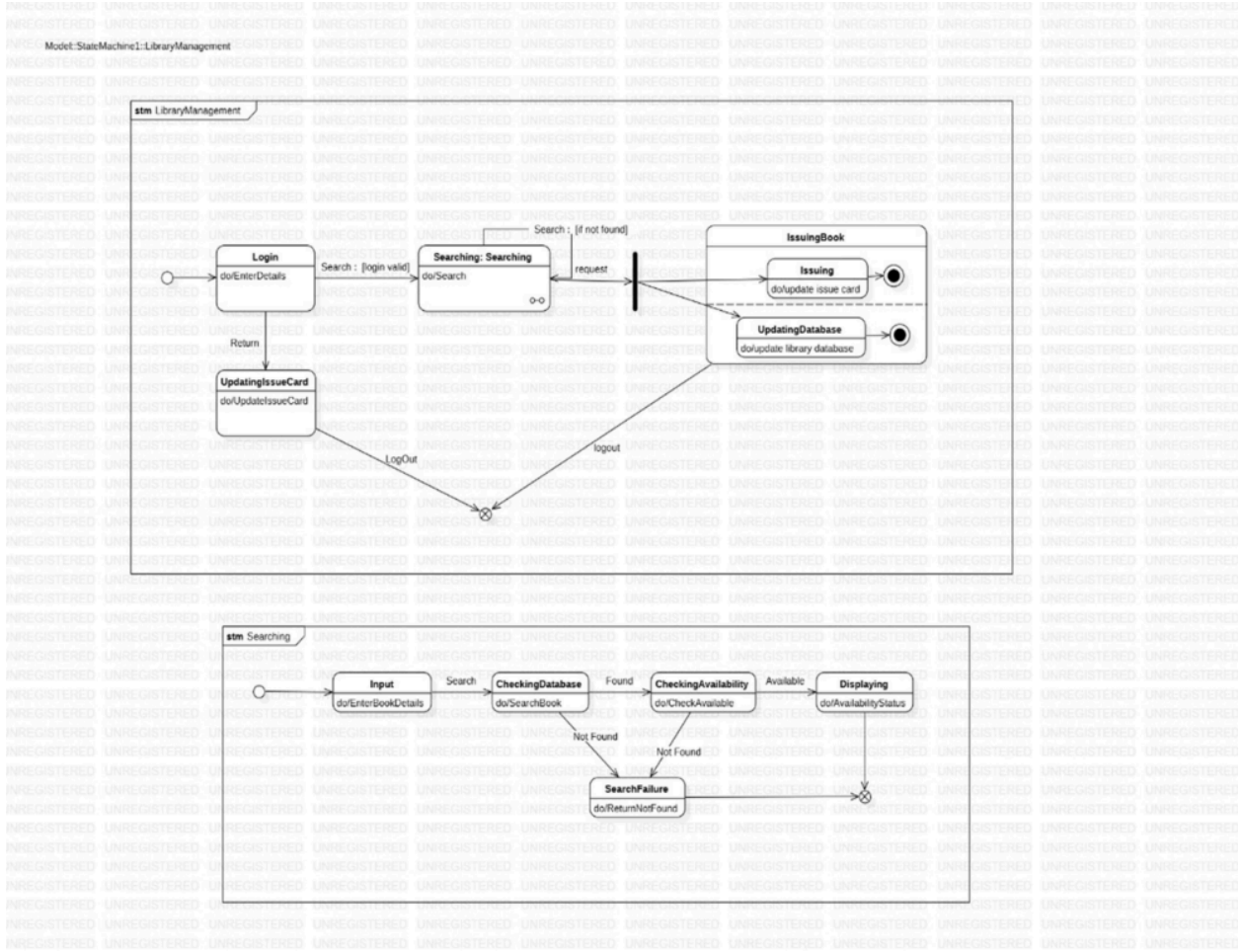


Fig 3.4

The advanced state diagram models detailed transitions for searching, issuing, borrowing, renewing, and returning books.
The process begins with **SearchBook**, followed by availability checking. If not available, the system moves to the **WaitList** state. If available, the system proceeds to **IssueBook**, then transitions to the **Borrowed** state where the due date is monitored. Members may request renewal, leading to the **Renewal** state. After reading, the user returns the book, triggering the **ReturnBook** state, which handles fine assessment and updates records. This diagram provides a more realistic model of the library's operational states.

3.7 Use Case Diagram

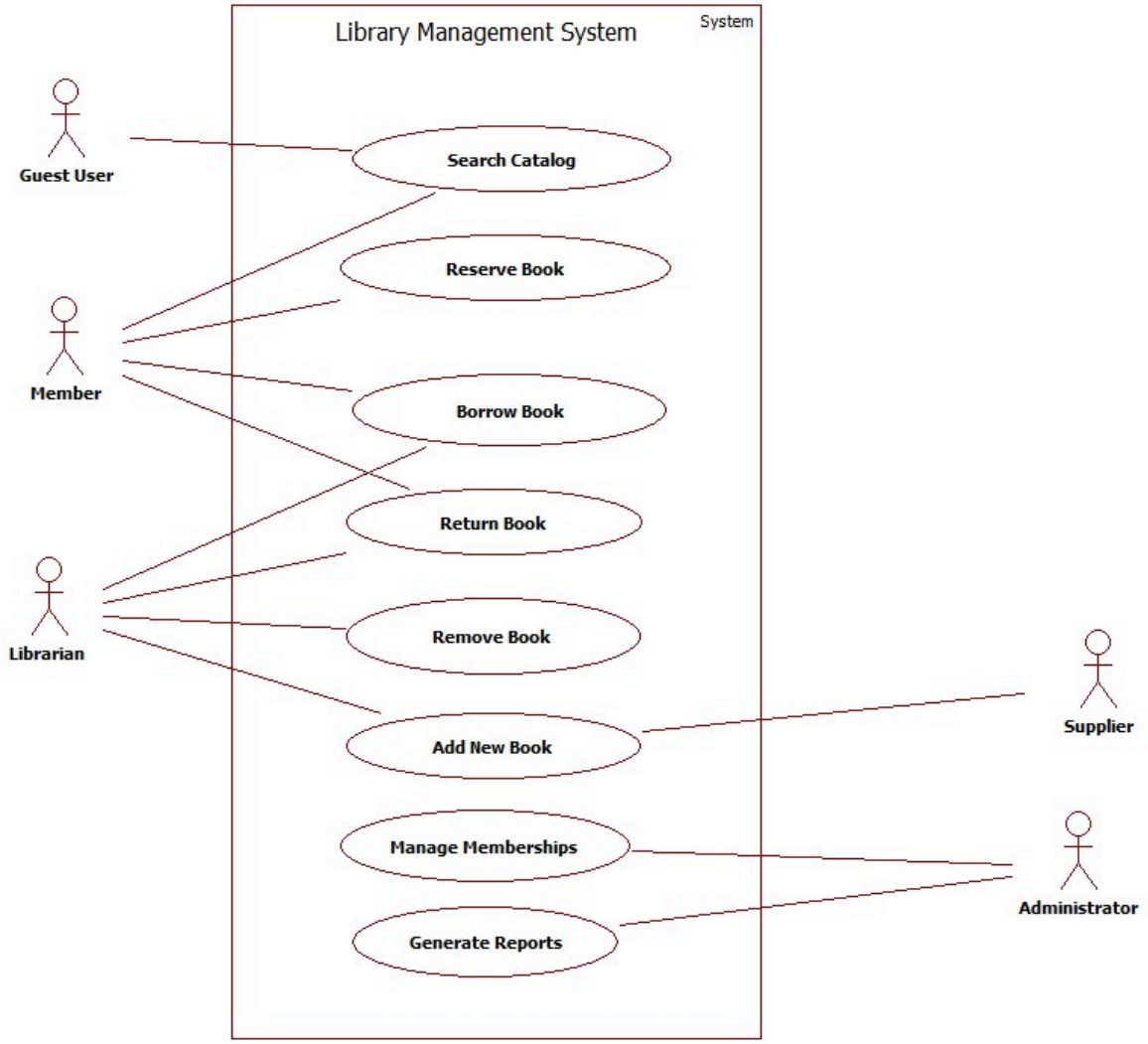


Fig 3.5

The use case diagram identifies different actors and their interactions with the system.
Members can search, reserve, borrow, renew, return books, and pay fines. **Librarians** manage circulation tasks, issue books, accept returns, remove or add books, and manage member records. **Suppliers** provide new books, and **Administrators** handle catalog management, inventory updates, and send due-date reminders. **Guest Users** can search the catalog without borrowing privileges. The diagram clearly shows the major functionalities required for efficient library operations.

3.8 Advanced Use Case Diagram

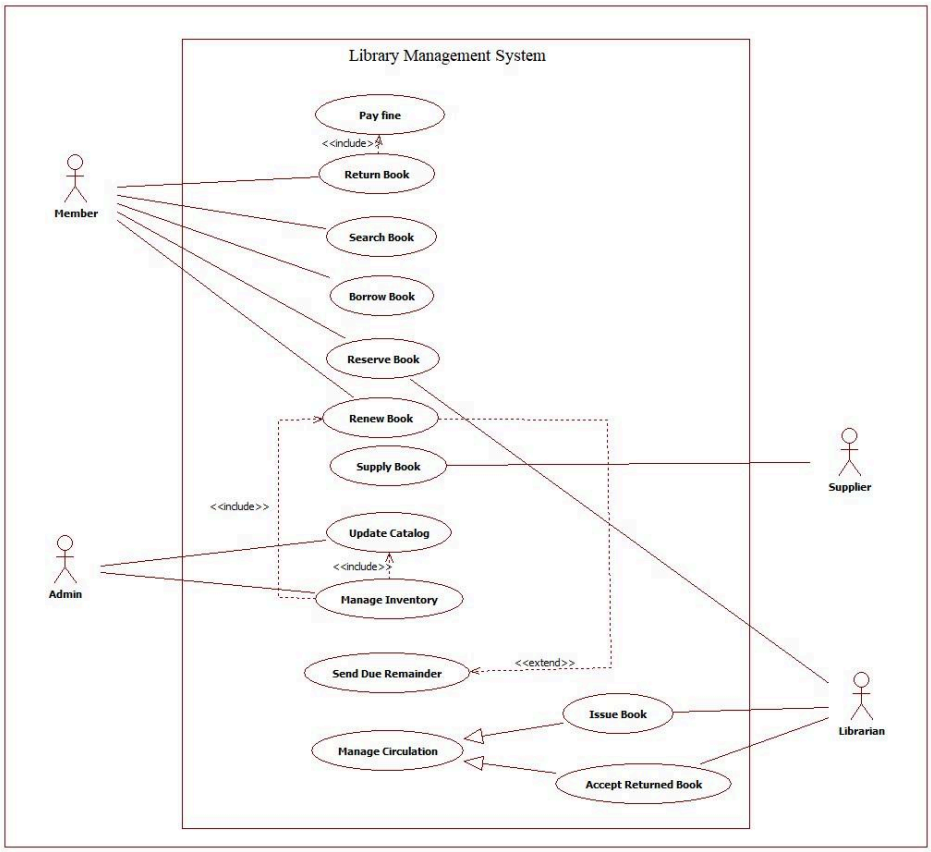


Fig 3.6

The advanced use case diagram captures all major interactions between different actors and the Library Management System. **Members** can search, reserve, borrow, renew, and return books, as well as pay fines. **Librarians** handle issuing and accepting returned books, managing memberships, adding or removing books, and maintaining the catalog. **Administrators** manage inventory, update catalog records, and generate system reports. **Suppliers** provide new books to update the catalog.

Some use cases like *Pay Fine* and *Update Catalog* are included within larger operations such as *Return Book* and *Manage Inventory*. The *Send Due Reminder* use case may extend the borrowing process to notify members before due dates. This advanced diagram clearly shows a complete set of functional requirements, covering user services, staff responsibilities, and system-level management within the library.

3.9 Sequence Diagram

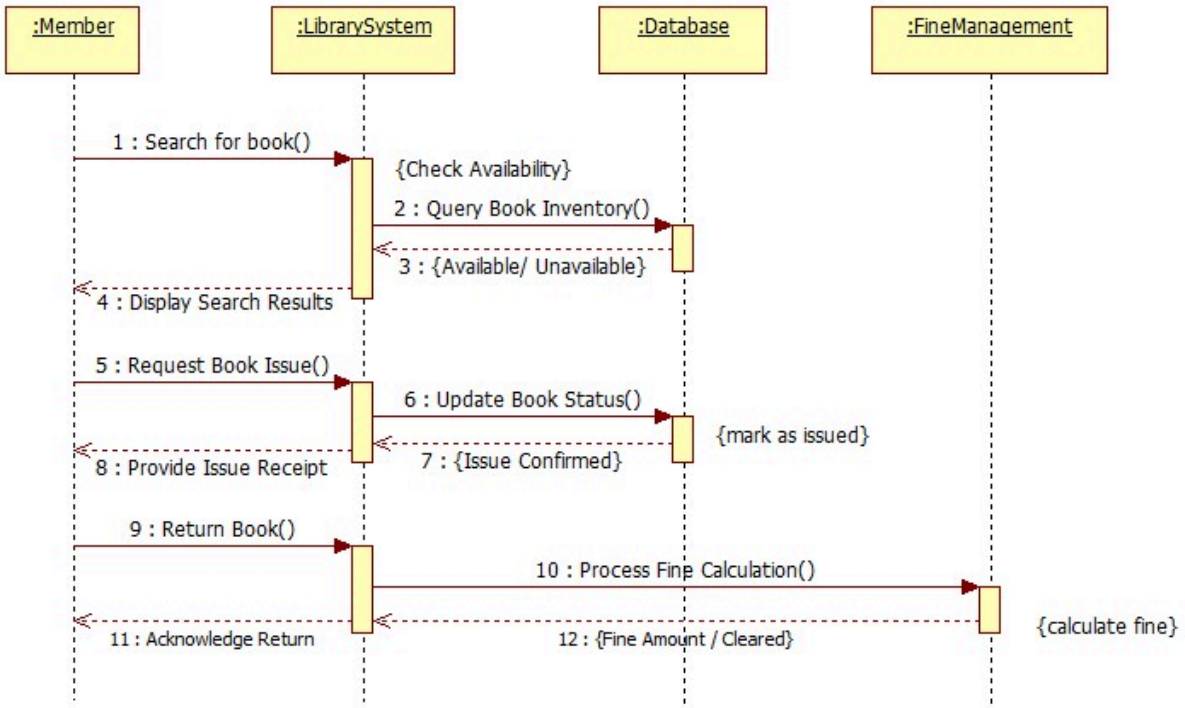


Fig 3.7

The basic sequence diagram describes the steps involved when a member searches and issues a book.

The **Member** searches for a book, the **Library System** queries the **Database**, and availability status is returned. If available, the member requests issue. The system updates book status, confirms issue, and later processes fine calculation during book return through the **Fine Management** component. Finally, the system acknowledges the return and updates records. This sequence captures a complete borrow–return cycle.

3.10 Advanced Sequence Diagram

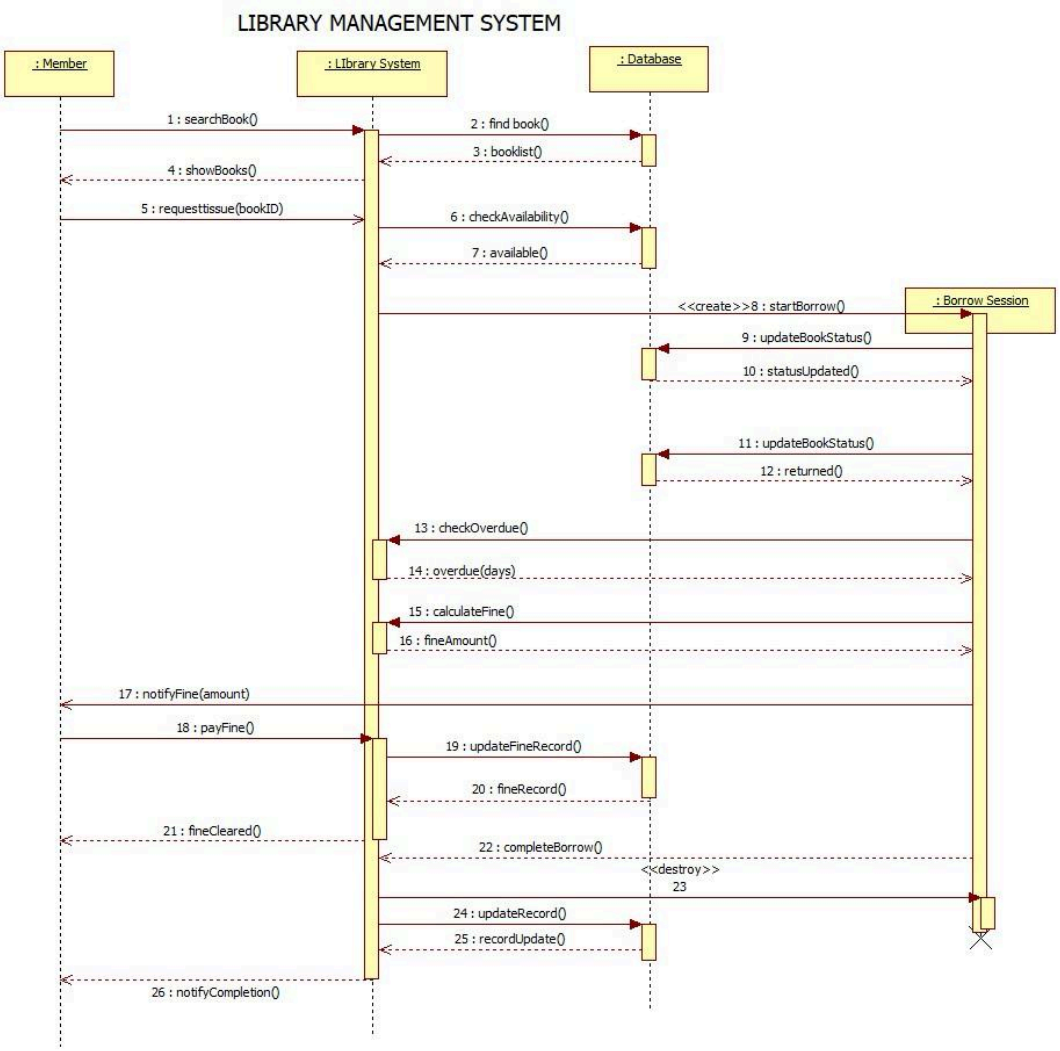


Fig 3.8

The advanced sequence diagram provides detailed interactions for issuing and returning books. The **Library System** interacts with the **Database** to find the book, check availability, and update status. A new **Borrow Session** is created to track loan duration. When the book is returned, overdue days are calculated, fines are computed, and fine records are updated. After fine clearance, the borrow session ends, and the library database updates the final return status. This detailed flow shows complete handling of search, issue, renew, return, and fine processing.

3.11 Activity Diagram

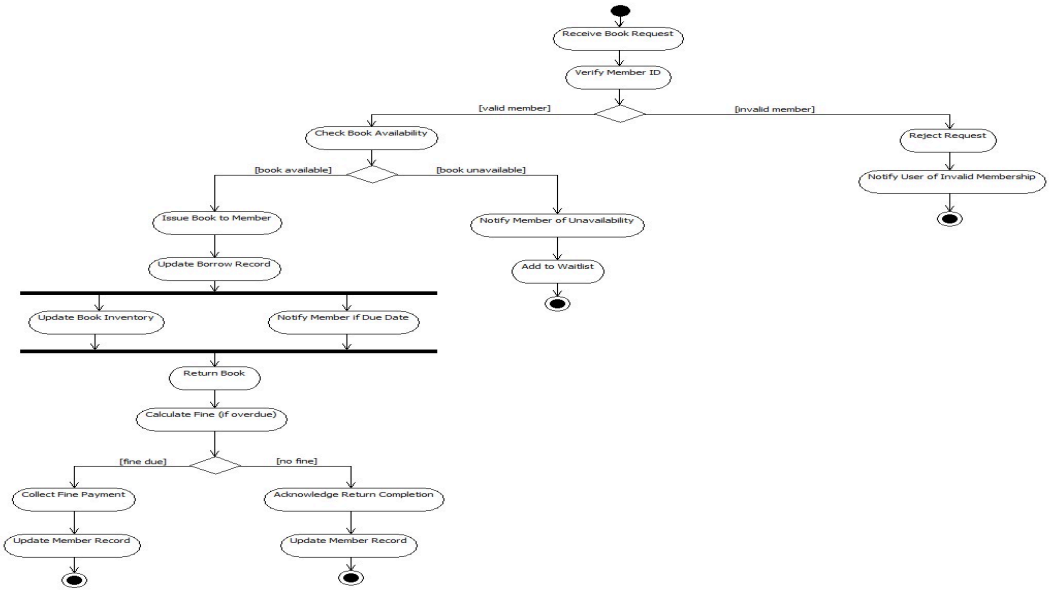


Fig 3.9

The activity diagram shows user actions and system responses during the book borrowing process.

The flow starts with a member choosing a book and going to the front desk. The librarian verifies the member ID. If invalid, the member is informed, and the process stops. If valid, book availability is checked. If unavailable, the member is notified. If available, the book is issued, and a borrow record is created. After reading, the member returns the book. The system checks for overdue fines, processes fine payments if needed, and finally reshelves the book. This diagram covers the complete operational workflow.

3.12 Advanced Activity Diagram

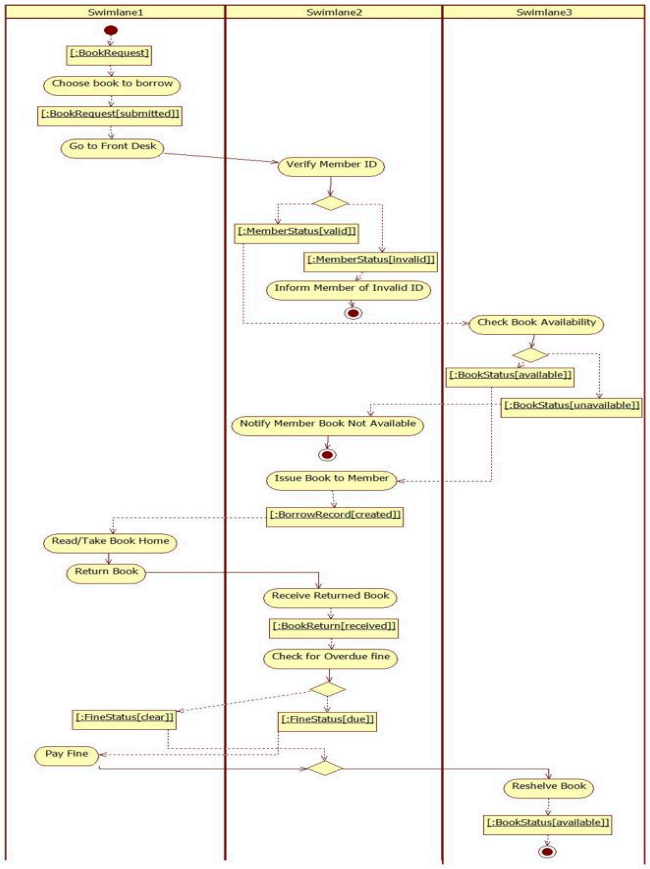


Fig 3.10

The advanced activity diagram represents the complete workflow of borrowing, returning, and renewing books in the library. The process begins when a member searches for a book, after which the system checks its availability. If the book is not available, the member is added to a waitlist. If available, the book is issued and the borrow record is created.

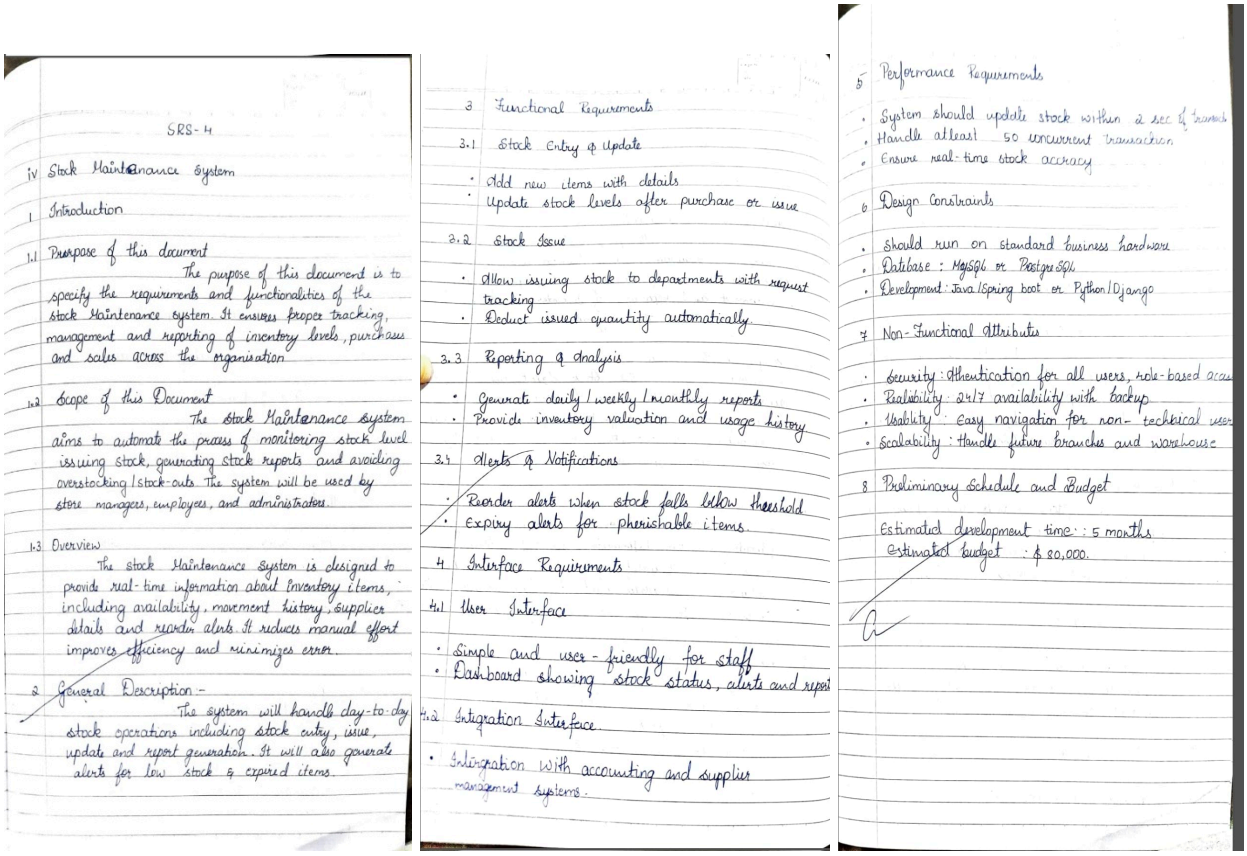
During the borrowing period, the system monitors the due date. Members may request renewal, after which the due date is extended. When the book is returned, the system checks for overdue days, calculates any fine, processes fine payment if applicable, and updates member and inventory records. The workflow ends by reshelving the book and marking it as available again. This activity diagram captures all detailed operations involved in issuing, borrowing, renewing, and returning books efficiently.

4.Stock Maintenance System

4.1Problem Statement

The Stock Maintenance System is designed to manage all operations related to inventory, including adding new stock, updating quantities, handling customer orders, generating reports, and coordinating with suppliers. The system ensures accurate tracking of stock levels, validates item availability during requests, updates records when stock is added or deducted, and generates reports for decision-making. It also supports order placement, stock movement management, supplier coordination, and warehouse operations.

4.2 SRS-Software Requirements Specification



SRS-4

iv Stock Maintenance System

1 Introduction

1.1 Purpose of this document

The purpose of this document is to specify the requirements and functionalities of the stock Maintenance system. It ensures proper tracking, management and reporting of inventory levels, purchase and sales across the organisation.

1.2 Scope of this Document

The stock Maintenance system aims to automate the process of monitoring stock level issuing stock, generating stock reports and avoiding overstocking/stock-outs. The system will be used by store managers, employees, and administrators.

1.3 Overview

The stock Maintenance system is designed to provide real-time information about inventory items, including availability, movement history, supplier details and reorder alerts. It reduces manual effort improves efficiency and minimizes error.

2 General Description -

The system will handle day-to-day stock operations including stock entry, issue, update and report generation. It will also generate alerts for low stock & expired items.

3 Functional Requirements

3.1 Stock Entry & Update

- Add new items with details
- Update stock levels after purchase or issue

3.2 Stock Issue

- Allow issuing stock to departments with request tracking
- Deduct issued quantity automatically

3.3 Reporting & Analysis

- Generate daily/weekly/monthly reports
- Provide inventory valuation and usage history

3.4 Alerts & Notifications

- Reorder alerts when stock falls below threshold
- Expiry alerts for perishable items.

4 Interface Requirements

4.1 User Interface

- Simple and user-friendly for staff
- Dashboard showing stock status, alerts and reports

4.2 Integration Interface

- Integration with accounting and supplier management systems.

5 Performance Requirements

- System should update stock within 2 sec of transaction
- Handle atleast 50 concurrent transaction
- Ensure real-time stock accuracy

6 Design Constraints

- Should run on standard business hardware
- Database : MySQL or PostgreSQL
- Development: Java/Spring boot or Python/Django

7 Non-Functional Attributes

- Security: Athentication for all users, role-based access
- Realability: 24/7 availability with backup
- Usability: Easy navigation for non-technical users
- Scalability: Handle future branches and warehouse

8 Preliminary Schedule and Budget

Estimated development time: 5 months
Estimated budget : $ 80,000.

4.3 Class Diagram

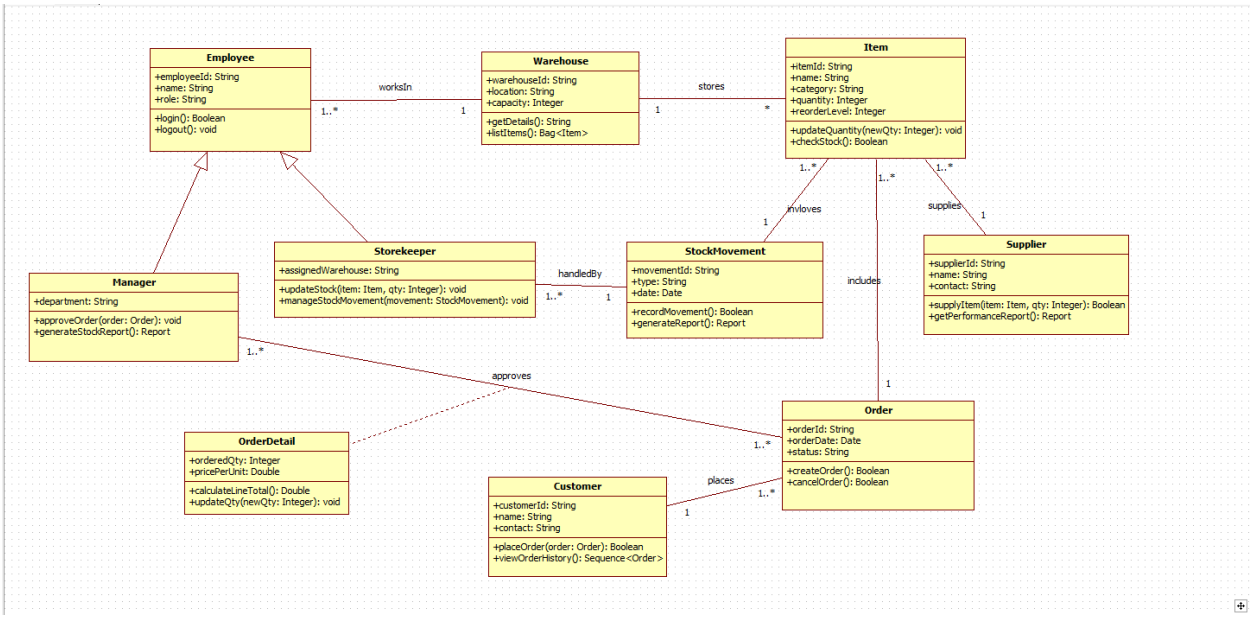


Fig 4.1

The simple class diagram shows the main classes involved in the Stock Maintenance System. The **Item** class stores details such as item name, category, quantity, and reorder level. The **Supplier** provides items to the system and is linked to the Item class. Customers place orders through the **Order** class, which contains information like order ID, date, and status. Each order includes details stored in the **OrderDetail** class.

The **Warehouse** class stores multiple items and maintains item lists. The **Employee** class represents workers in the system, and it has two specialized roles: **Manager**, who approves orders and generates reports, and **Storekeeper**, who updates stock and handles stock movements. Stock changes are recorded in the **StockMovement** class, which stores the type of movement and date.

Overall, the simple class diagram outlines how items, orders, employees, suppliers, and warehouses are connected to maintain stock efficiently.

4.4 Advanced Class Diagram

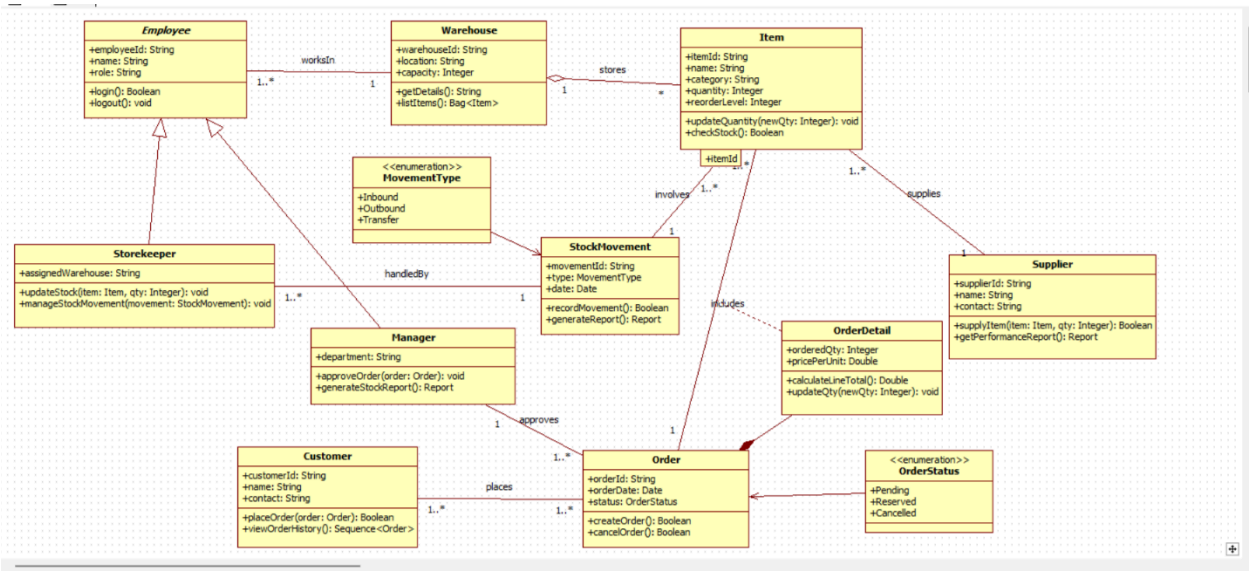


Fig 4.2

The advanced class diagram displays the structural model of the Stock Maintenance System.

- **Employee**, **Storekeeper**, and **Manager** manage stock operations and approvals.
- **Warehouse** stores multiple items.
- **Item** contains details like quantity, category, reorder level, and provides stock-checking methods.
- **Order** represents customer orders and is linked to **OrderDetail** for item quantity and pricing.
- **Supplier** provides items and maintains performance reports.
- **StockMovement** records inbound, outbound, and transfer movements using **MovementType** enumeration.
  Relationships show how orders are placed, approved, supplied, and stored, forming a complete inventory management structure.

4.5 State Diagram

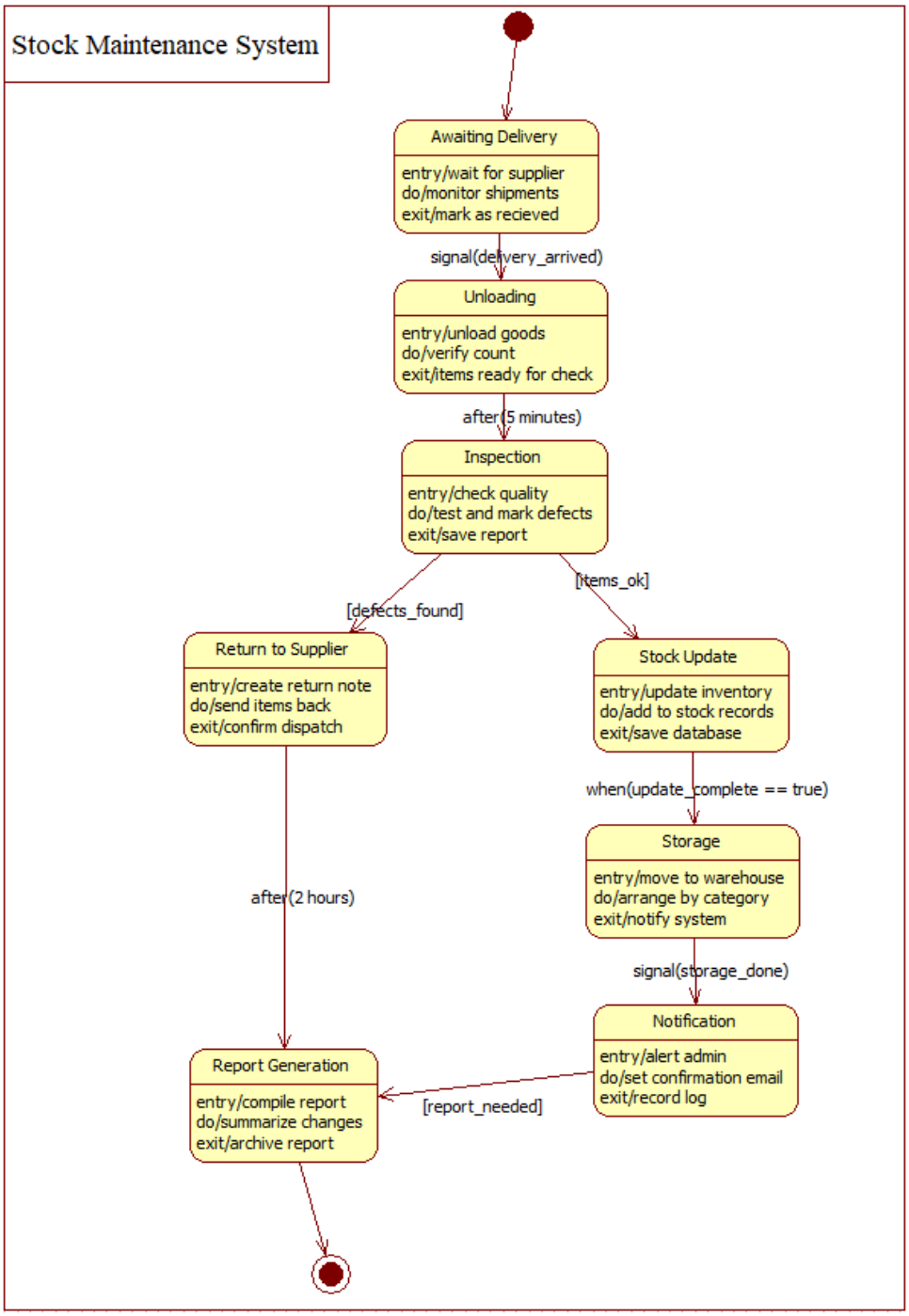


Fig 4.3

The state diagram models stock lifecycle. Items move through states like Awaiting Delivery, Unloading, Inspection, Returning to Supplier (if defects found), Stock Update, Storage, and Notification. Each state has entry, do, and exit actions such as verifying count, checking quality, updating records, and notifying admin. The flow ensures that stock is properly received, validated, updated, and stored.

4.6 Advanced State Diagram

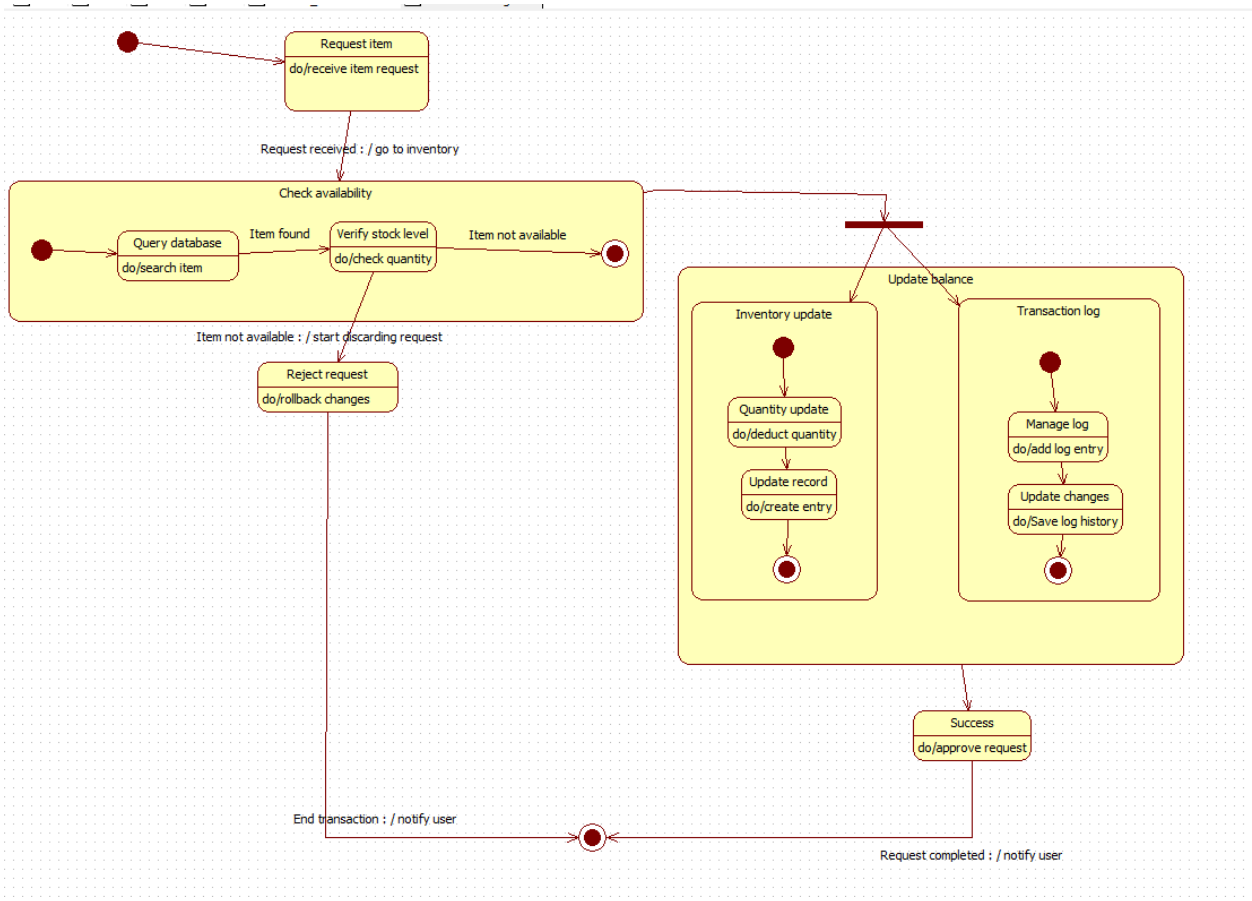


Fig 4.4

The advanced state diagram shows how stock items move through different stages during the maintenance process. The system begins in the **Awaiting Delivery** state, where it waits for items to arrive from the supplier. Once delivered, the items move to the **Unloading** state, where they are checked and counted.

Next, the items enter the **Inspection** state, where quality and defects are verified.

- If defects are found, the system transitions to **Return to Supplier**, where items are sent back.
- If the items are fine, they move to **Stock Update**, where inventory records are updated.

After updating, the items enter the **Storage** state, where they are arranged and stored properly. The system then moves to the **Notification** state to alert the admin and finally ends in **Report Generation**, where a summary report is prepared.

This diagram clearly represents the flow of stock from arrival to inspection, update, storage, and reporting.

4.7 Use Case Diagram

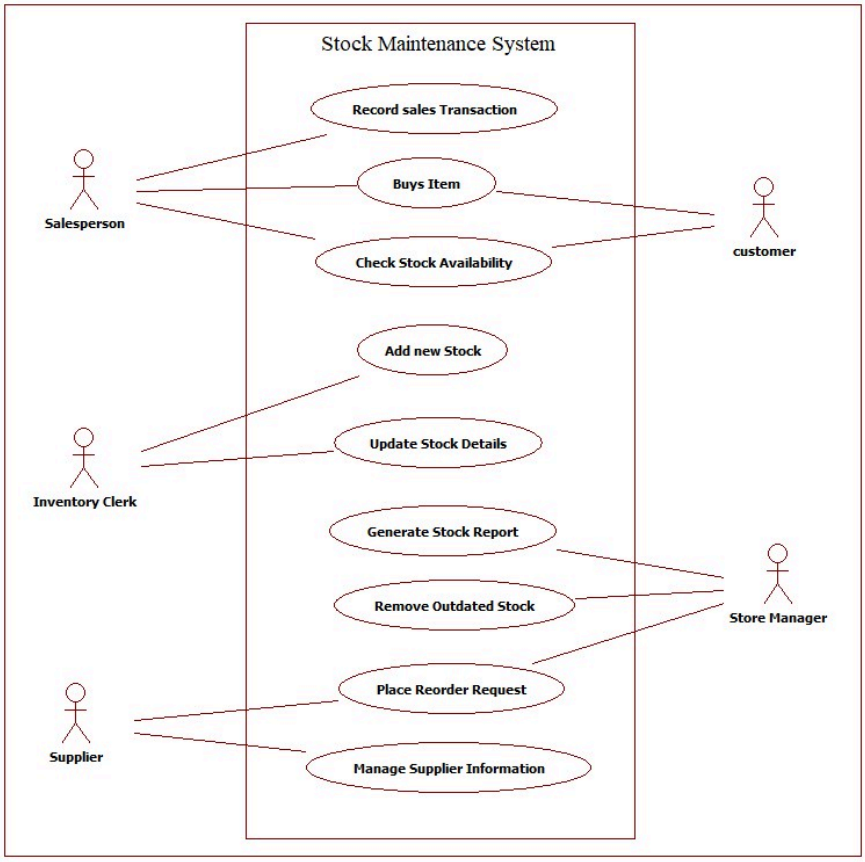


Fig 4.5

This version focuses on key roles: Salesperson records sales and checks item availability; Inventory Clerk adds and updates stock; Store Manager removes outdated stock, generates

reports, and places reorder requests; Suppliers manage supplier information. Customers can purchase items and view availability.

Overall, this diagram highlights external interactions with the stock system.

4.8 Advanced Use Case Diagram

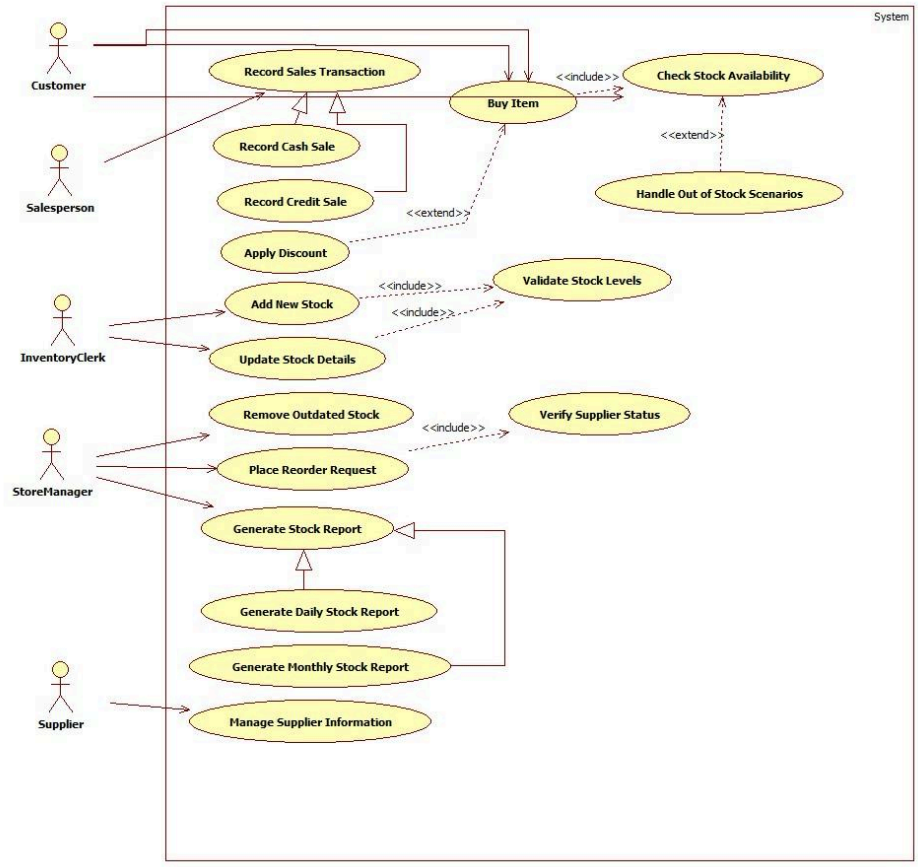


Fig 4.6

The advanced use case diagram shows all major actors and their interactions with the Stock Maintenance System. **Salespersons** buy and record sales transactions while checking stock availability. **Inventory Clerks** manage stock by adding new items, updating stock details, and removing outdated stock. **Store Managers** generate daily and monthly stock reports and place reorder requests when inventory is low. **Suppliers** provide stock and manage supplier information for replenishment.

Some use cases include or extend others—for example, validating stock levels is included in both adding and updating stock, and handling out-of-stock situations extends the “Buy Item” use

case. Overall, the diagram captures how different users collaborate to maintain accurate inventory, ensure stock availability, and support efficient stock operations.

4.9 Sequence Diagram

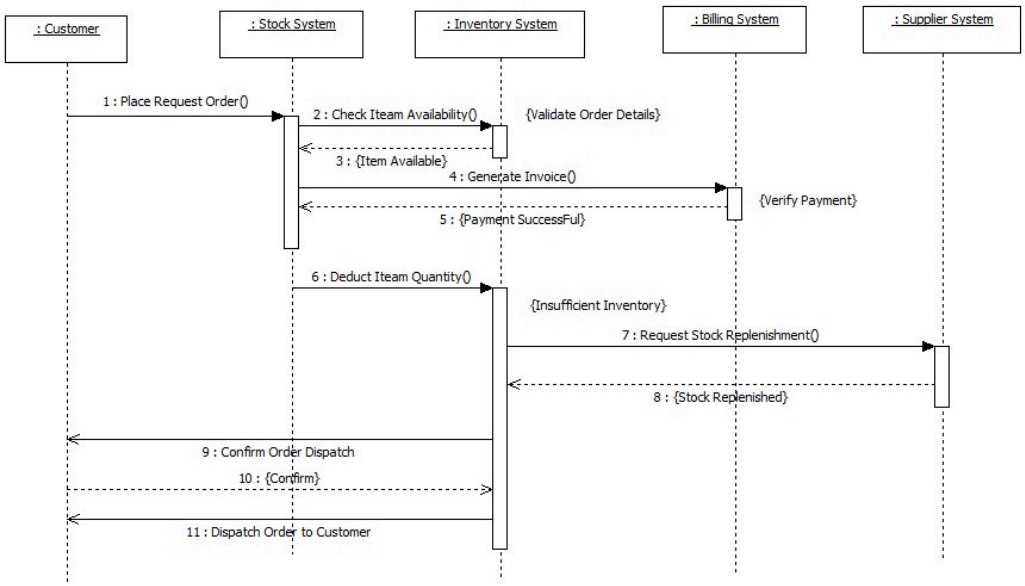


Fig 4.7

The sequence diagram shows how the Admin interacts with the Stock Management System to add or update stock. When stock is added, the system inserts the details into the inventory database and returns the confirmation. During stock update, the system deducts quantity, updates the database, and sends back the updated status. A report object is then created and used to generate final stock reports before being destroyed. This diagram highlights communication between Admin, Stock System, and Database.

4.10 Advanced Sequence Diagram

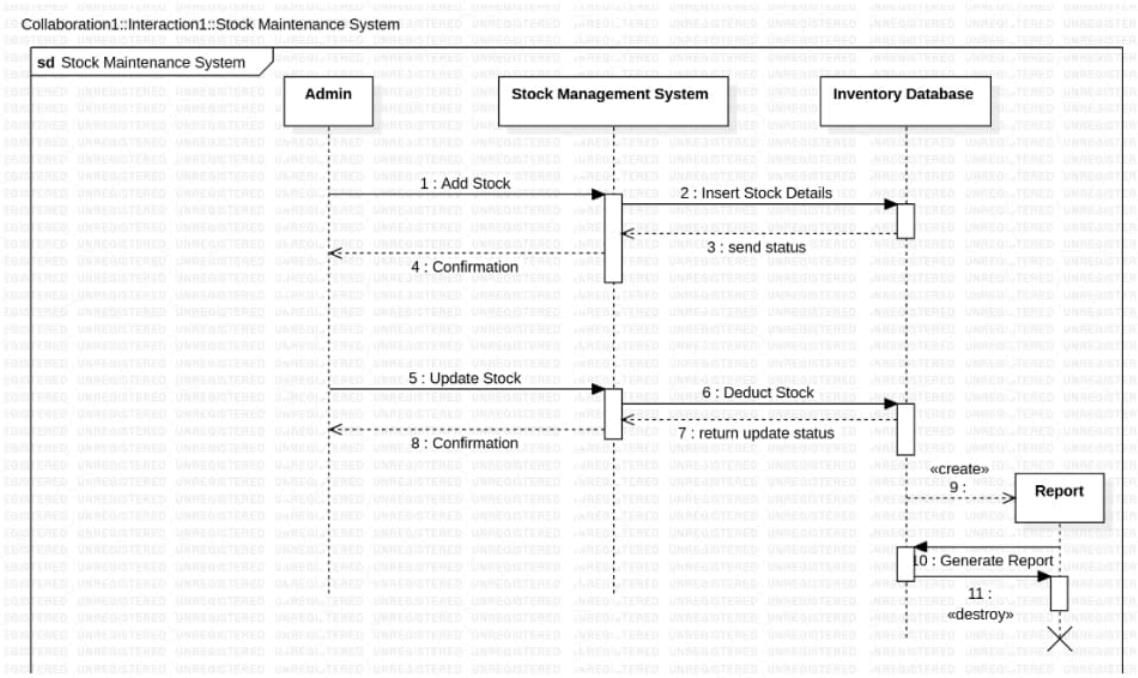


Fig 4.8

The sequence diagram illustrates how the Stock Maintenance System processes stock-related operations. First, the Admin sends an “Add Stock” request to the Stock Management System, which then inserts the new stock details into the Inventory Database. The database returns a status message, and the system sends a confirmation back to the Admin. Similarly, when the Admin sends an “Update Stock” request, the system forwards a stock deduction request to the database, receives the updated status, and confirms the update to the Admin. After these operations, a Report object is created, and the system generates a detailed stock report before destroying the report object. Overall, the diagram clearly shows the step-by-step communication flow between the Admin, Stock Management System, and Inventory Database during stock

addition, update, and report generation.

4.11 Activity Diagram

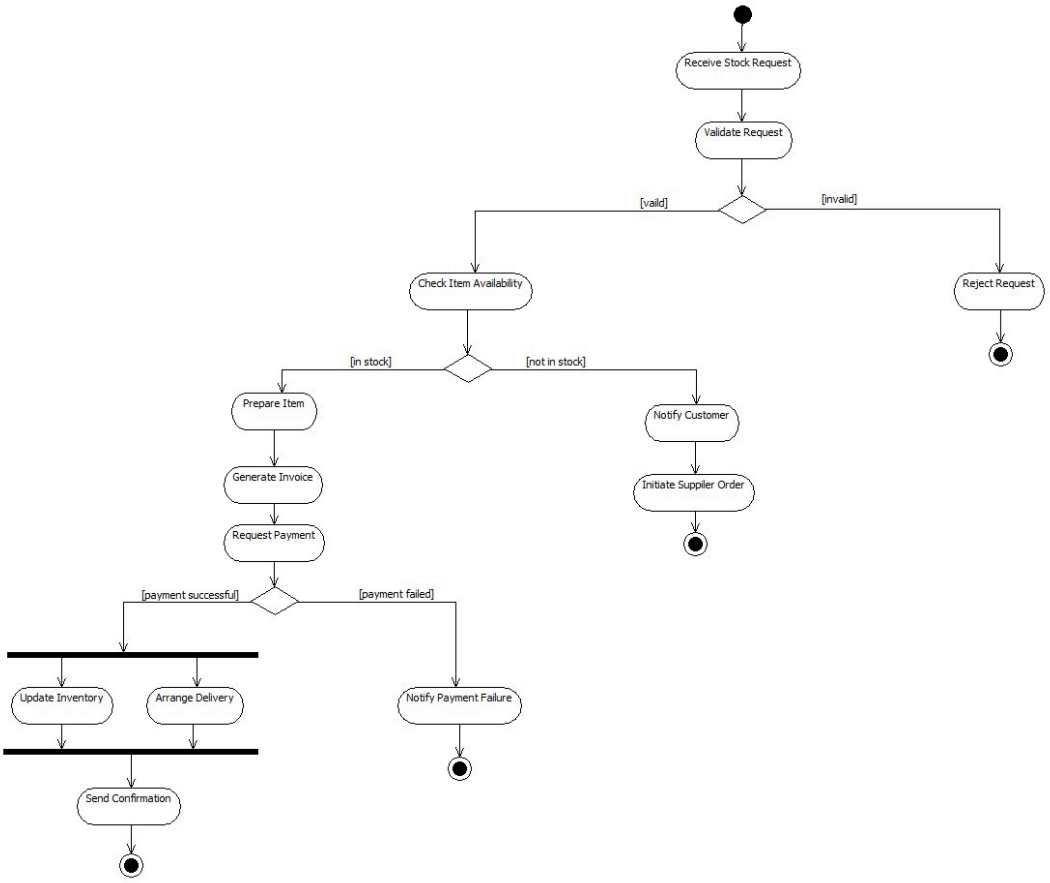


Fig 4.9

The activity diagram explains the workflow of a stock request process. The request is validated, item availability is checked, and based on availability, the system either prepares the item or initiates a supplier order. Next, invoice is generated, payment is requested, and depending on success or failure, the system updates inventory, arranges delivery, or notifies payment failure. The process ends after sending confirmation to the requester.

4.12 Advanced Activity Diagram

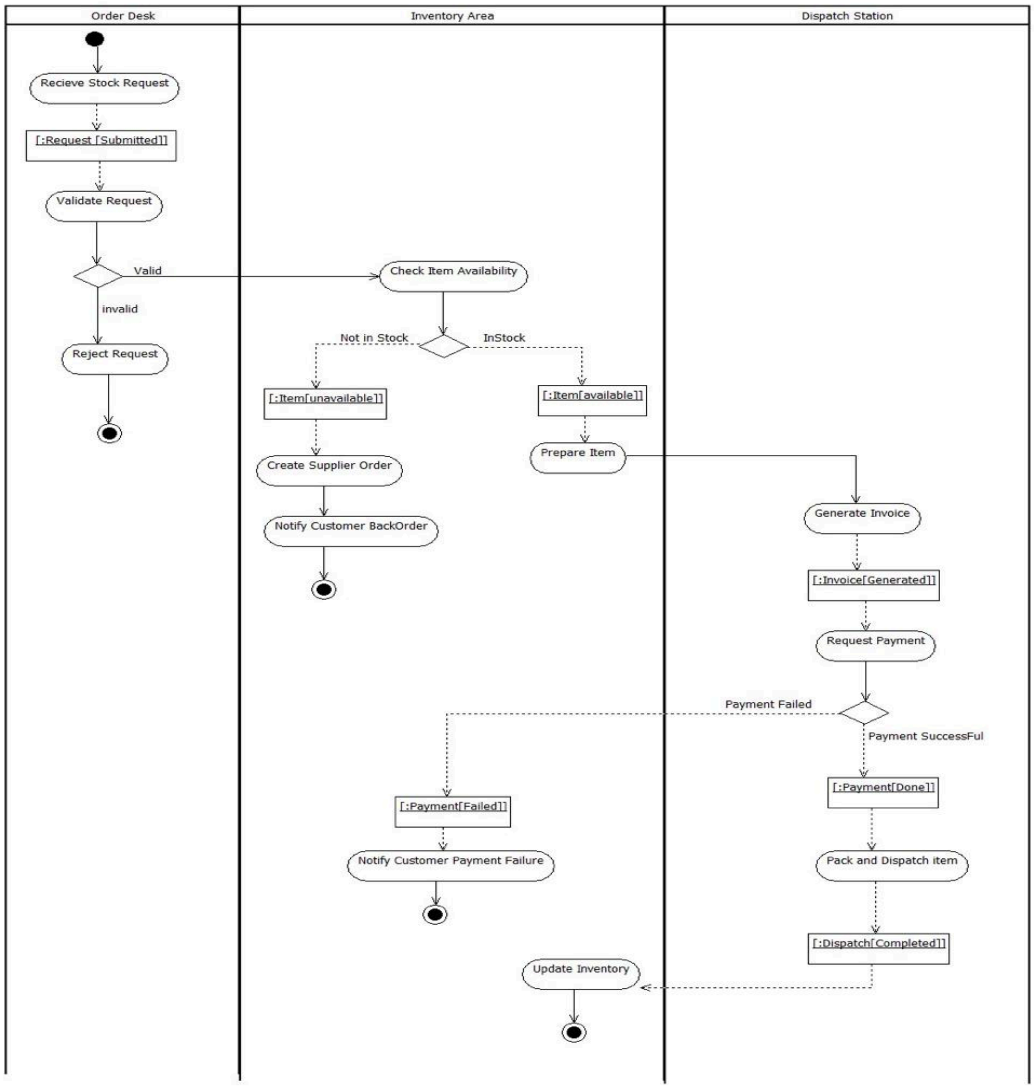


Fig 4.10

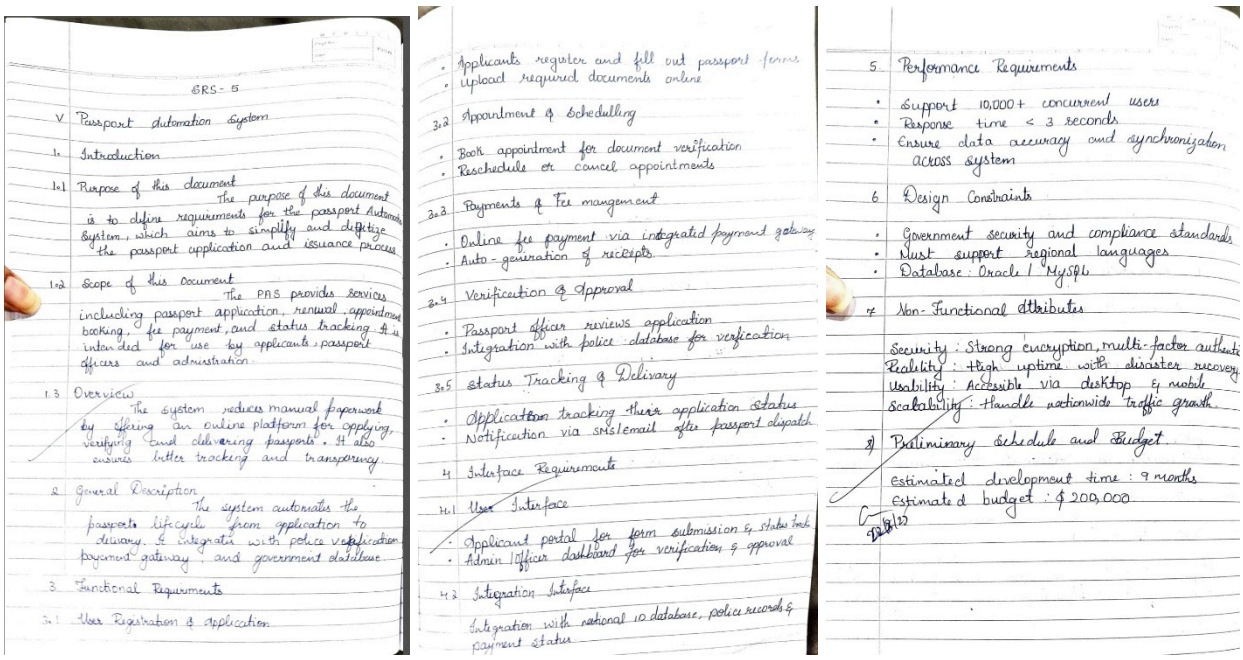
The activity diagram explains the workflow of a stock request process. The request is validated, item availability is checked, and based on availability, the system either prepares the item or initiates a supplier order. Next, invoice is generated, payment is requested, and depending on success or failure, the system updates inventory, arranges delivery, or notifies payment failure. The process ends after sending confirmation to the requester.

5. Passport Automation System

5.1 Problem Statement

The Passport Automation System aims to simplify and automate the entire passport application process, starting from submitting an application to the final issuance of the passport. The system allows applicants to apply online, schedule appointments, upload documents, and track their application status. Passport officers verify documents, conduct biometric checks, update status, and process approvals or rejections. The system also coordinates with the Police Department to perform background verification. By integrating all stakeholders—Applicant, Passport Office, Verification Officer, and Police Department—the system ensures faster processing, reduced manual work, secure document handling, and complete transparency for the applicant.

5.2 SRS-Software Requirements Specification



5.3 Class Diagram

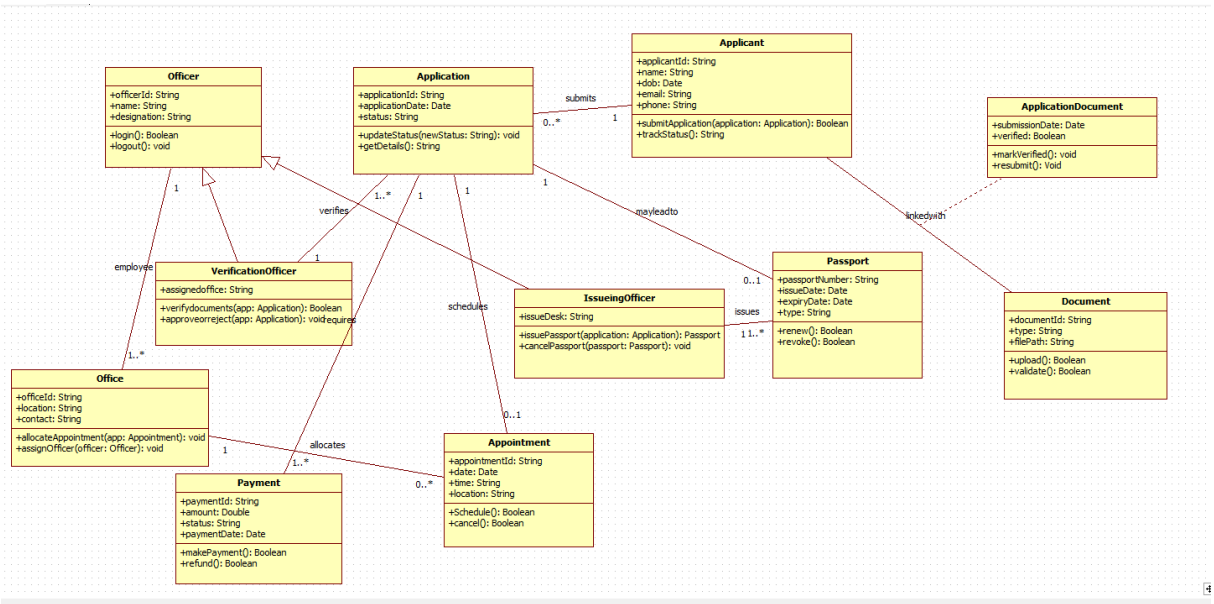


Fig 5.1

The class diagrams show the structure of the Passport Automation System. Important classes include Applicant, Application, Passport, Officer, Appointment, Payment, and Document. Each class has attributes and functions, and the relationships show how they are connected—for example, applicants submit applications, officers verify them, and documents belong to applications.

5.4 Advanced Class Diagram

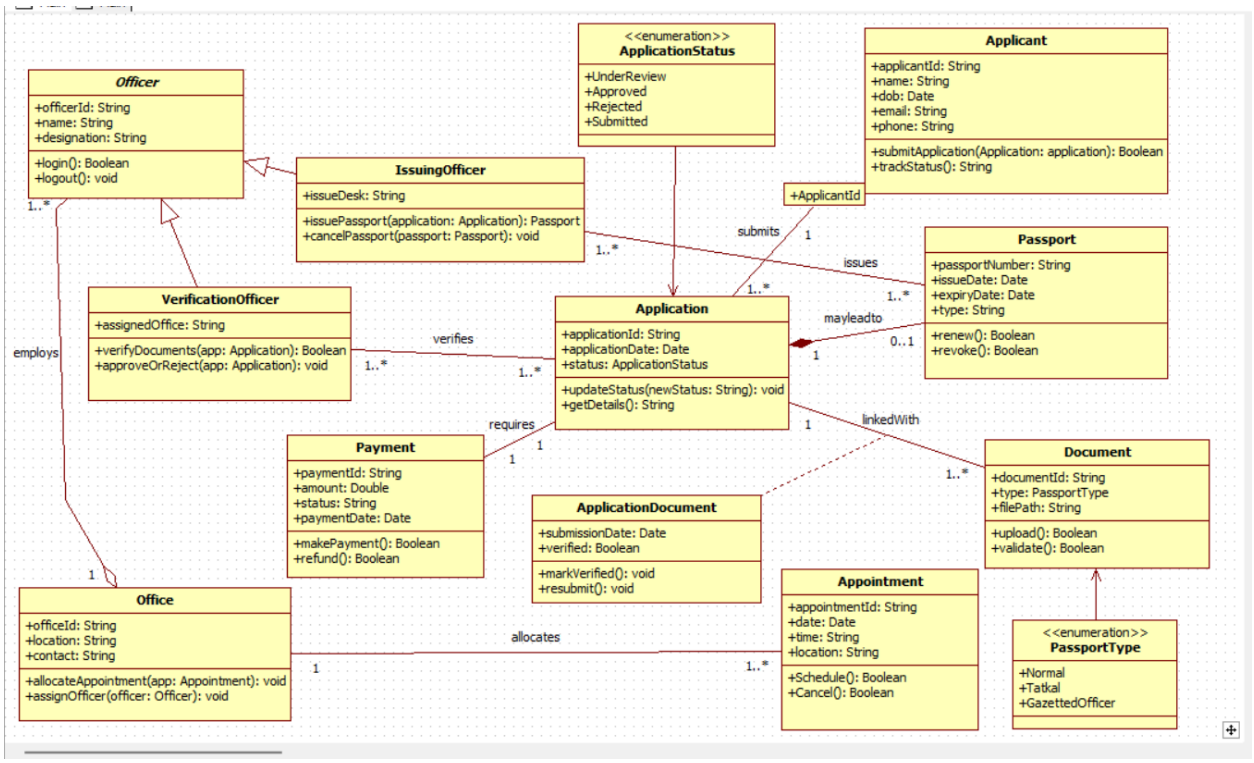


Fig 5.2

The advanced class diagrams define the complete object-oriented architecture of the system. They capture relationships such as associations, aggregations, compositions, multiplicities, and enumerations. The Application class links to Applicant, Documents, Payment, Appointment, and Passport, forming the core of the system. Specialized roles like VerificationOfficer and IssuingOfficer extend the Officer class through inheritance. Enumerations such as ApplicationStatus and PassportType provide controlled value sets. Method signatures describe behavior like verifyDocuments(), approveOrReject(), allocateAppointment(), and issuePassport(). Together, the diagrams establish the system's static design blueprint.

5.5 State Diagram

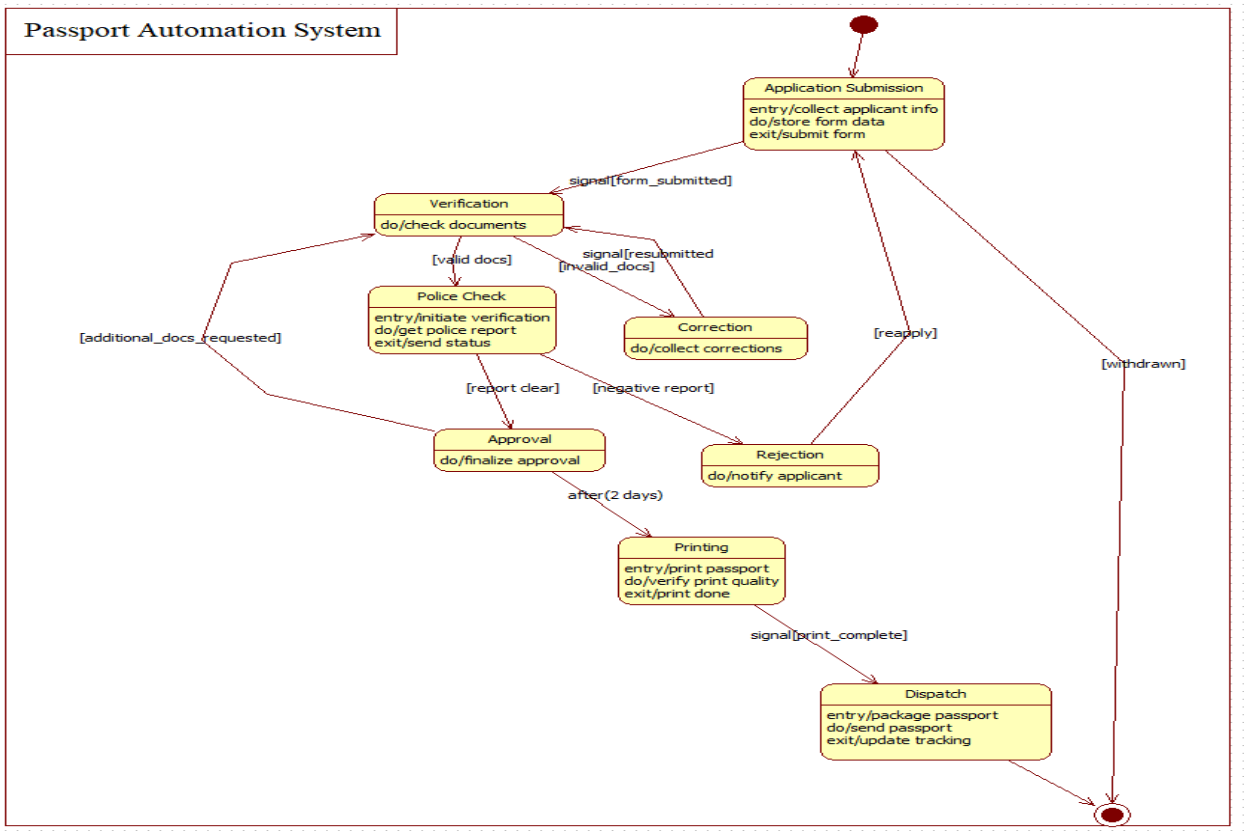


Fig 5.3

The state diagrams show how an application moves from one stage to another. It starts at "Application Submitted", goes through document verification, biometric verification, police verification, approval, printing, and finally passport dispatch. Each state represents the current position of the application in the process.

5.6 Advanced State Diagram

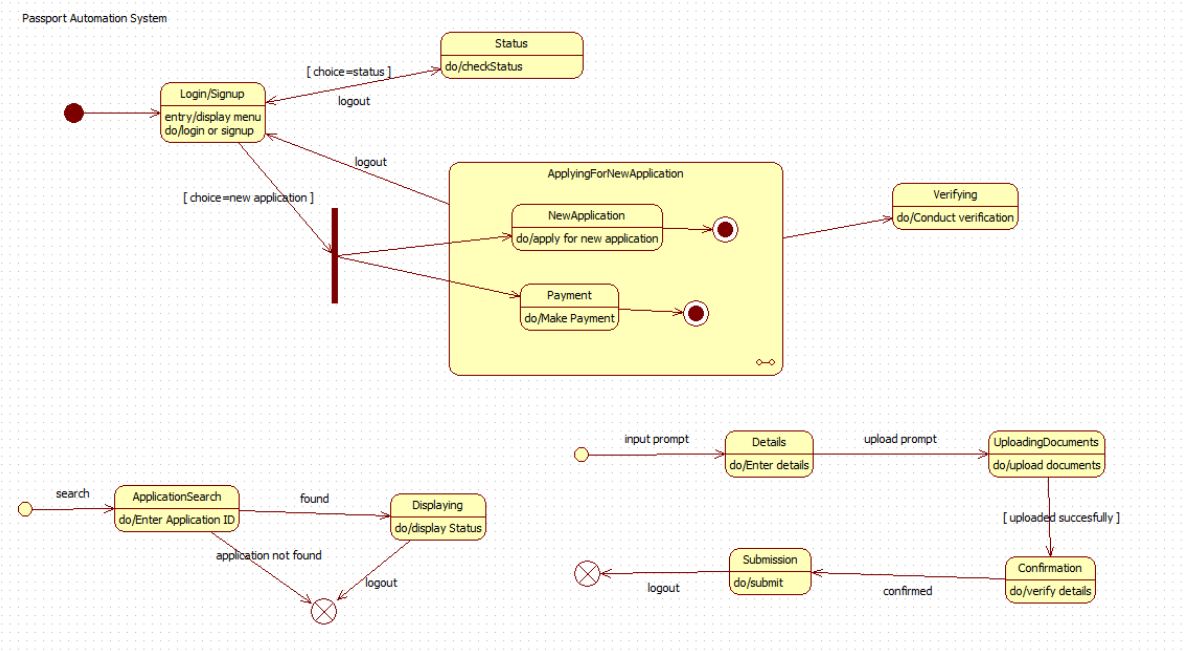


Fig 5.4

TThe advanced state machine diagrams detail the dynamic lifecycle of a passport application. Each state contains entry/do/exit actions that specify internal operations such as storing form data, validating documents, initiating police checks, performing corrections, or printing passports. Transition triggers like form_submitted, invalid_docs, report_clear, and print_complete control the progression between states. Special conditions handle resubmission, withdrawal, and rejection. The model demonstrates reactive system behavior based on events, delays, and signals, providing a complete state-based representation of the application process.

5.7 Use Case Diagram

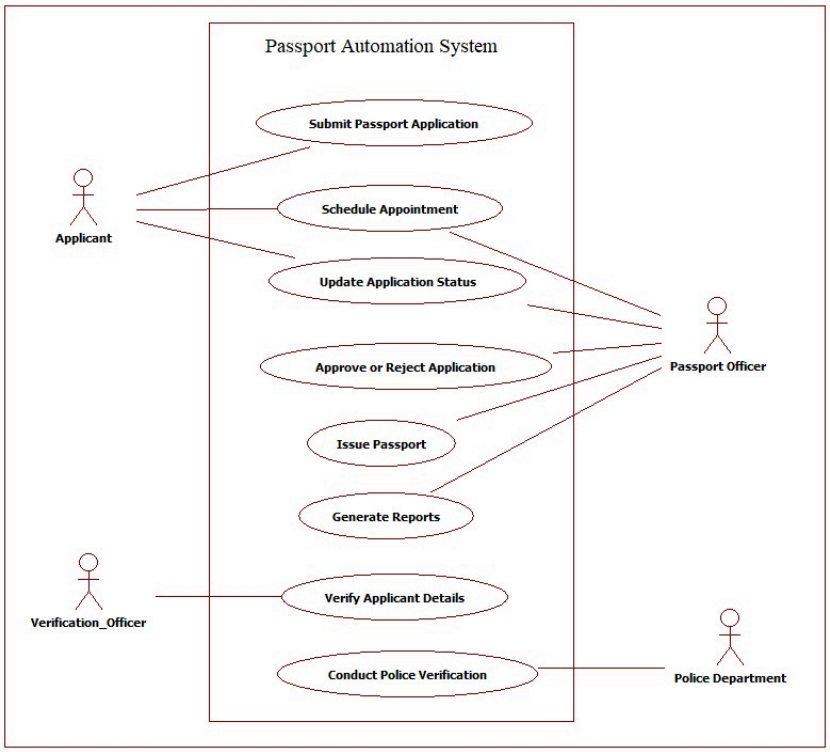


Fig 5.5

The use case diagrams show the different actions that users can perform in the Passport Automation System. Applicants can submit applications, schedule or reschedule appointments, upload documents, and track their status. Passport officers verify the application, approve or reject it, and issue passports. Verification officers check documents and biometrics, while the police department performs background verification. The diagram simply shows how each actor interacts with the system.

5.8 Advanced Use Case Diagram

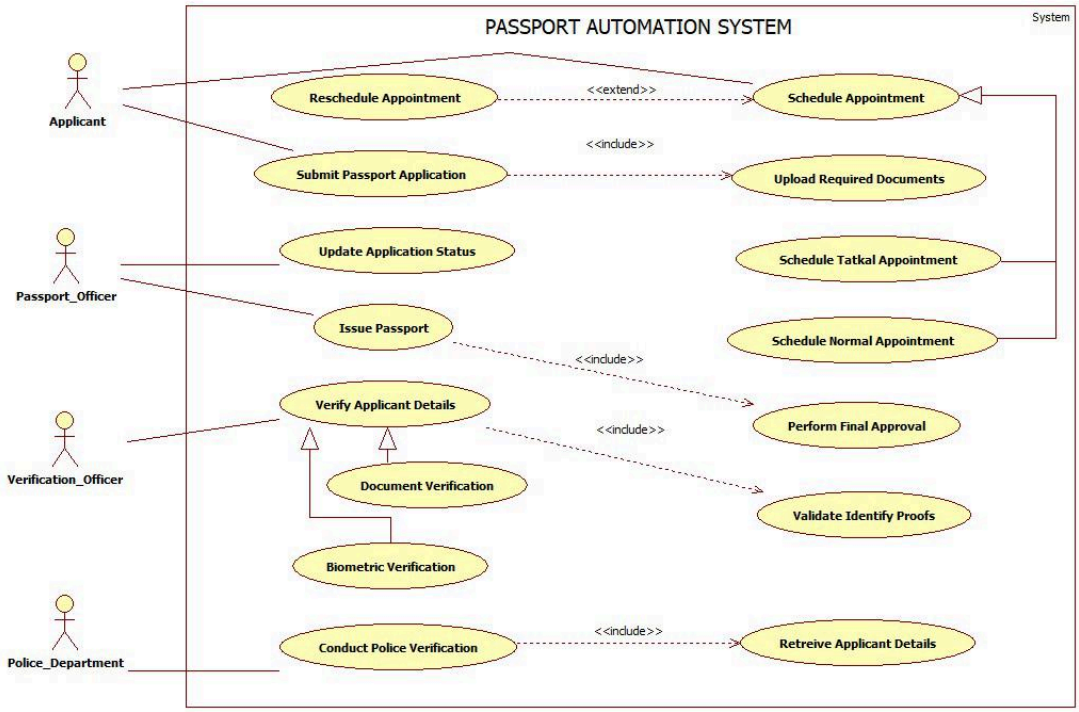


Fig 5.6

The advanced use case diagrams present detailed functional interactions between system actors and the passport automation workflow. They illustrate mandatory processes such as document submission, biometric enrollment, identity verification, police background check, and final approval. <<include>> and <<extend>> relationships highlight modularity—for example, biometric verification and police verification are mandatory sub-processes under applicant details verification. The diagrams also differentiate between normal and Tatkal appointments, extended rescheduling flows, and system-driven processes such as retrieving applicant details and validating identity proofs. Overall, they model the complete functional scope of the passport system.

5.9 SEQUENCE DIAGRAMS

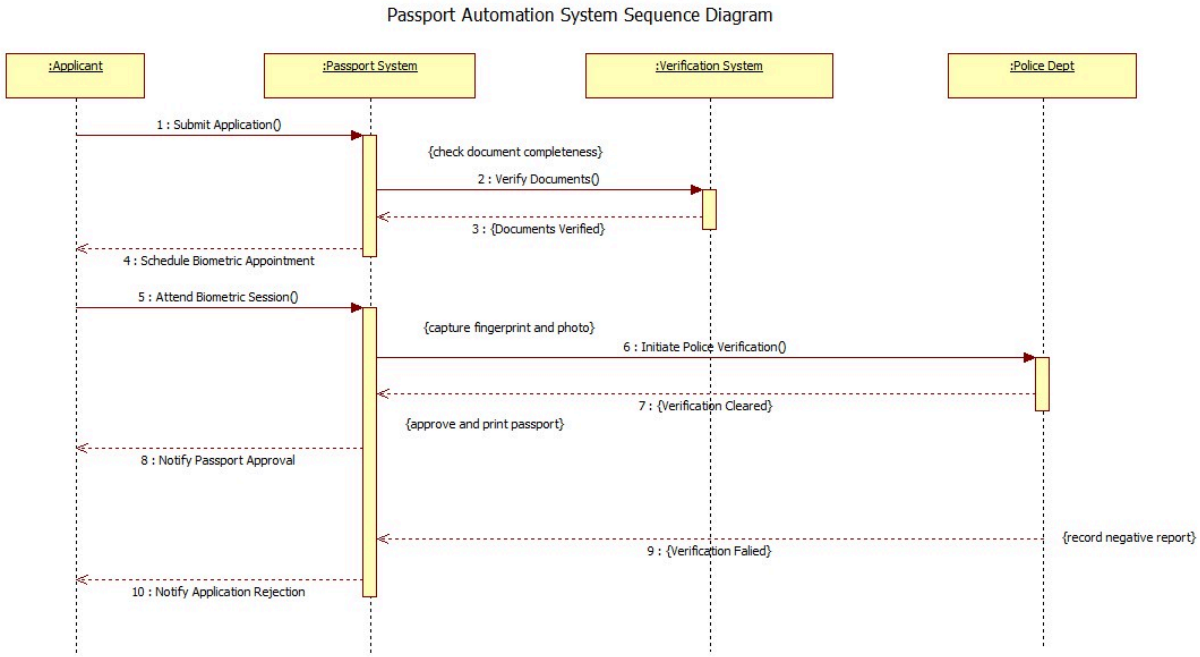


Fig 5.7

The sequence diagrams show how the passport application process happens step-by-step. First, the applicant submits the application, then the passport system checks documents, schedules biometrics, and sends the request to verification and police departments. After verification is completed, the system approves or rejects the application, and the applicant is notified.

5.10 Advanced SEQUENCE DIAGRAMS

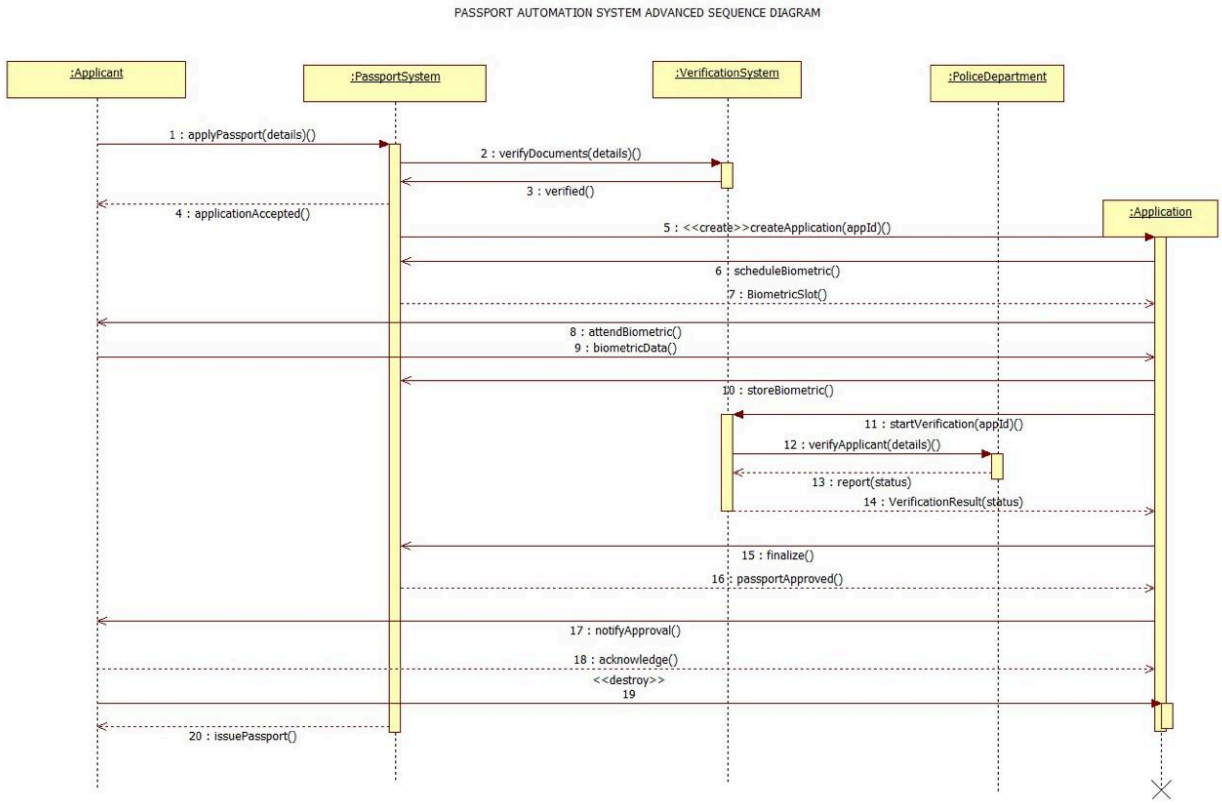


Fig 5.8

The advanced sequence diagrams illustrate the chronological interaction between system components: Applicant, Passport System, Verification System, and Police Department. They capture message passing such as document verification, biometric scheduling, biometric storage, police verification requests, status reporting, and application finalization. The diagrams also show object creation (like Application object instantiation), conditional flows, and system notifications. This demonstrates how control moves across lifelines, ensuring process synchronization from submission to passport issuance and closure.

5.11 ACTIVITY DIAGRAMS

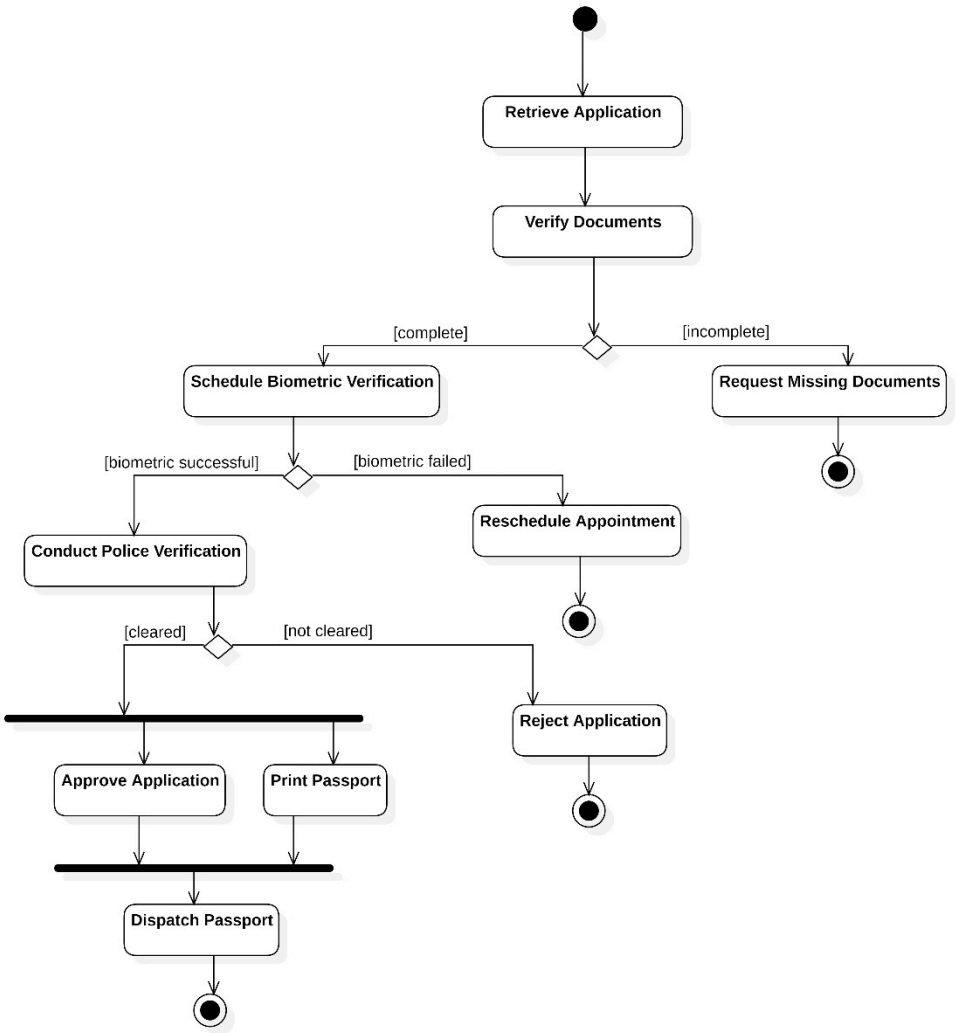


Fig 5.9

The activity diagrams show the overall workflow of the passport process. It starts with the applicant submitting the application. The office then verifies the documents and schedules biometrics. After biometrics, the police department performs verification. Based on the results, the application is either approved, printed, and dispatched or rejected.

5.12 Advanced ACTIVITY DIAGRAMS

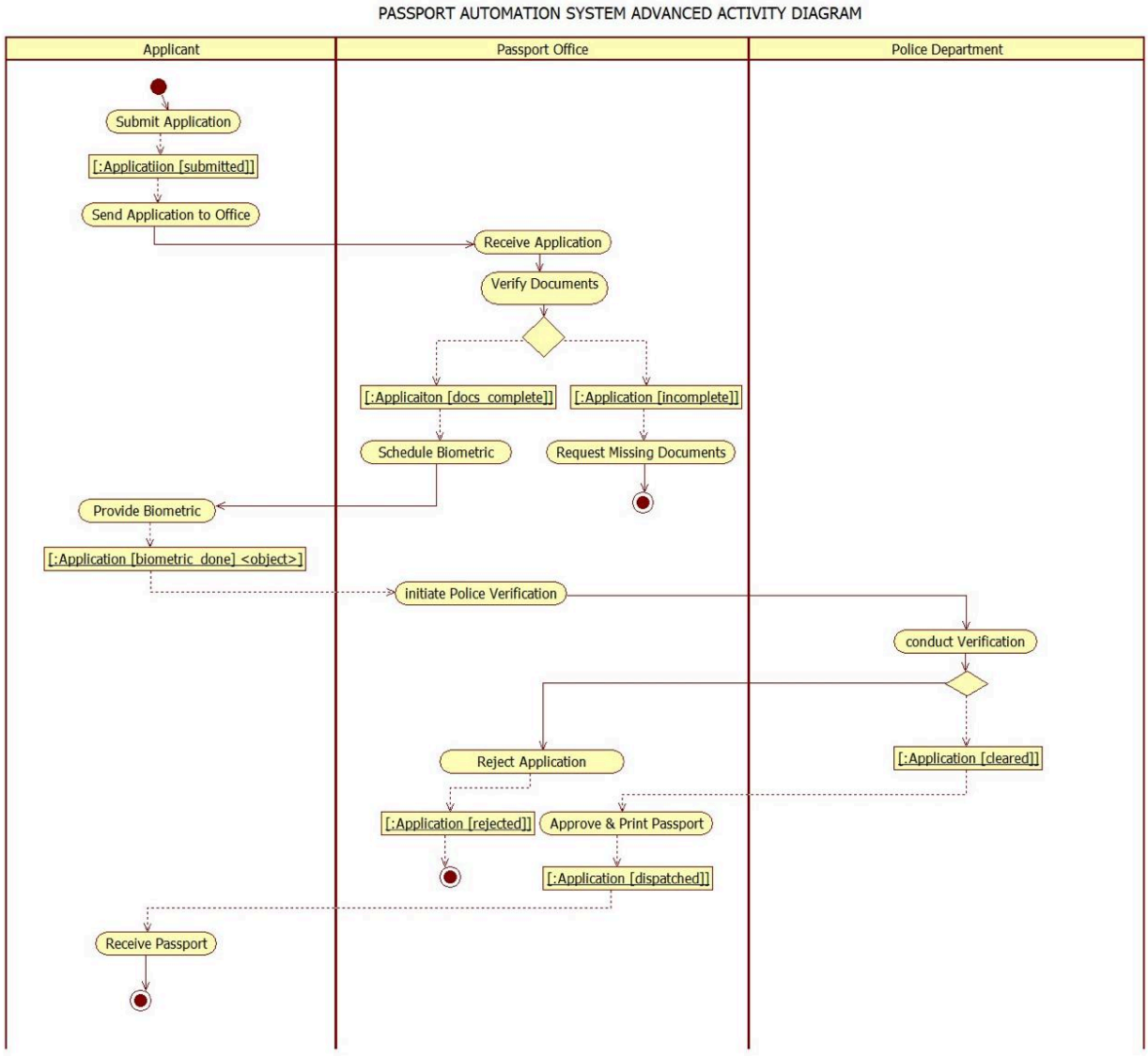


Fig 5.10

The advanced activity diagrams highlight parallel actions, decisions, and cross-department workflows through swimlanes. They illustrate branching conditions such as incomplete documents, biometric failure, or negative police verification. Synchronization bars show concurrent actions like approval and printing before dispatch. The diagrams emphasize

event-driven transitions (like applicationSubmitted, biometric_done, application_cleared) and exception flows involving missing documents or rejection. This provides a complete behavioral representation of the system's operational process.