COMSATS UNIVERSITY ISLAMABAD

ABBOTTABAD CAMPUS

THEORY

SUBMITTED TO: Mr. MUKHTIAR ZAMIN

PREPARED BY: AHMED IRSHAD HUSSAIN (Group Leader)

SHAYAN MUGHAL

SUDAIS MURAD

HAFIZ KHIZAR ALI

MASHOOD AZAM

KAMRAN FIAZ

CLASS: BSE-5B

INCLUDES: USE CASE DIAGRAM, FULLY DRESSED USE CASE

SSD, SD, CLASS DIAGRAM & CODING STANDARD

COURSE: SOFTWARE DESIGN & ARCHITECTURE

Contents

INTRODUCTION

Project Title: FEE ACCOUNT SYSTEM

Project Description:

The **Fee Account System** is designed to streamline the management of student fee records in educational institutions. It ensures accuracy, transparency, and ease of tracking payments, dues, and financial reports. The system automates manual processes, supports real-time updates, and improves financial record-keeping efficiency.

Team Members:

- **Ahmed Irshad** – *Group Leader*
- **Sudais** – *Team Member*
- **Shayan** – *Team Member*
- **Khizar** – *Team Member*
- **Mashood** – *Team Member*
- **Kamran** – *Team Member*

USE CASES:

Ahmed Irshad (Registration)

Shayan Mughal (Submit Invoice)

Sudais Murad (Process Payment)

Hafiz Khizar (Generating Fee Receipt)

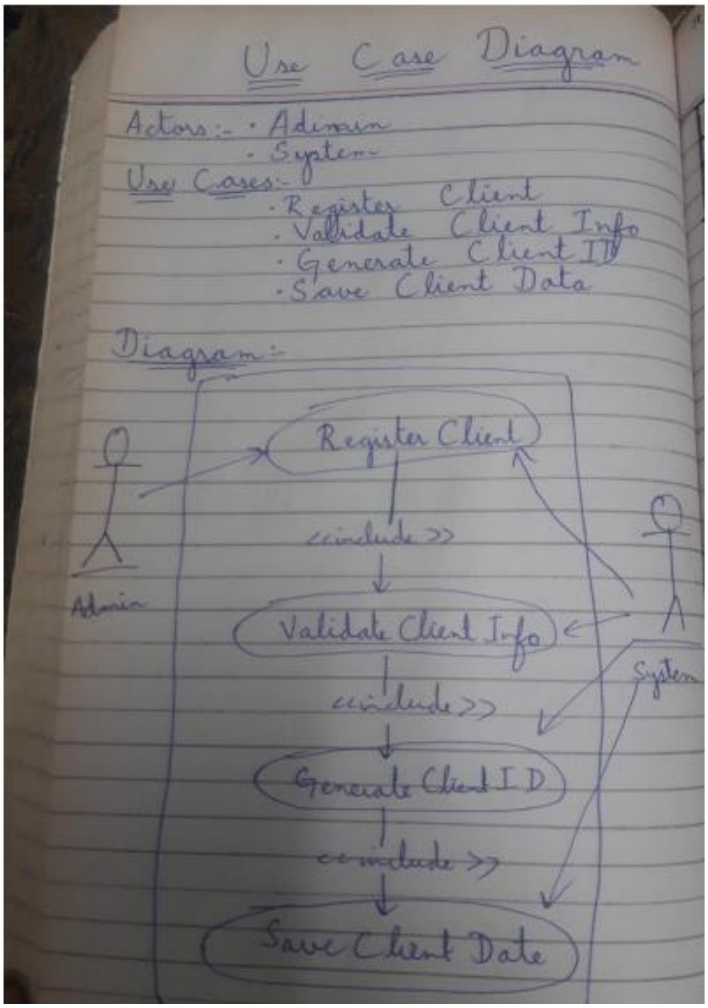
Mashood Azam (Send Payment Reminder)

Kamran Fiaz (View Payment History)

CHAPTER 1: USE CASE DIAGRAM

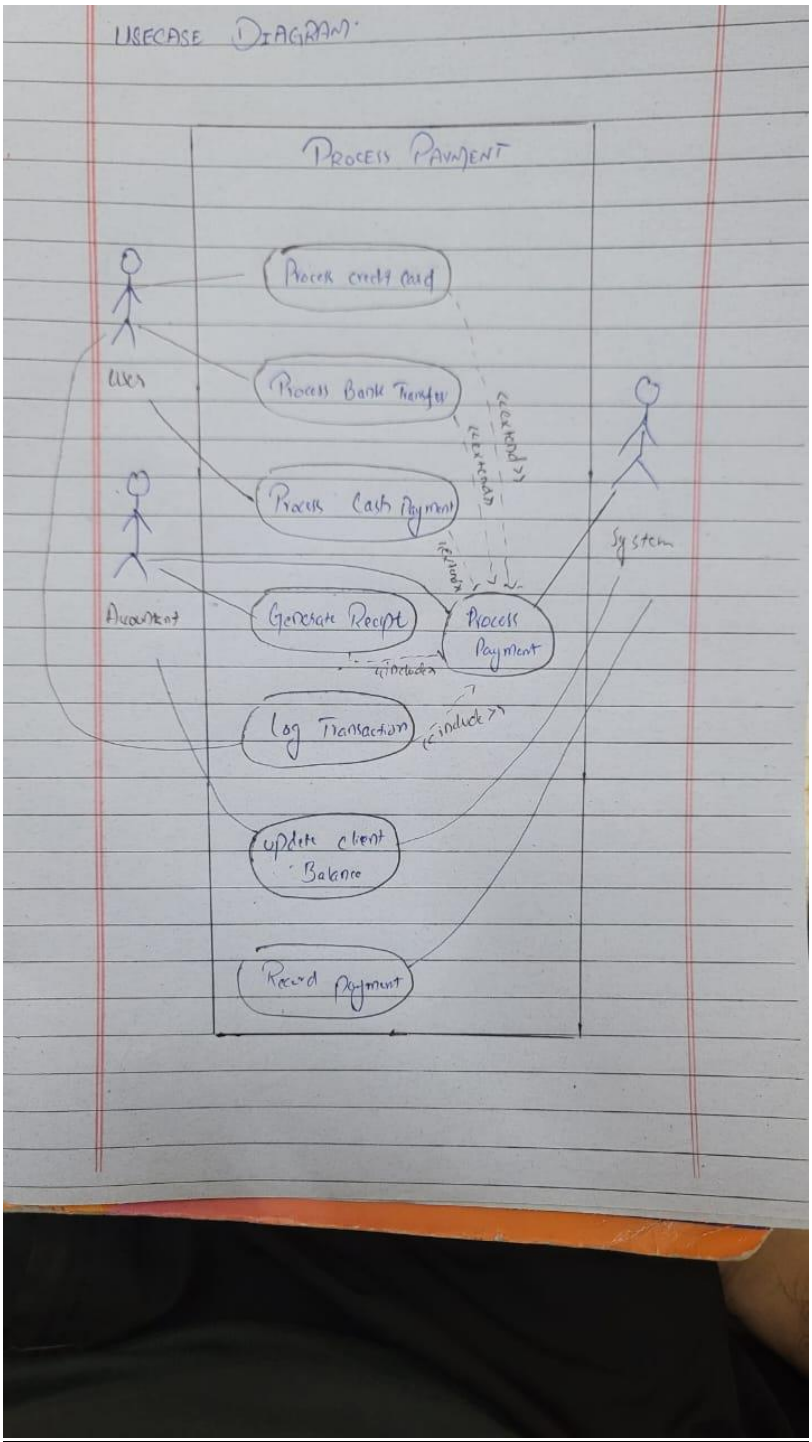
AHMED IRSHAD (USE CASE DIAGRAM):

Use Case: Registration



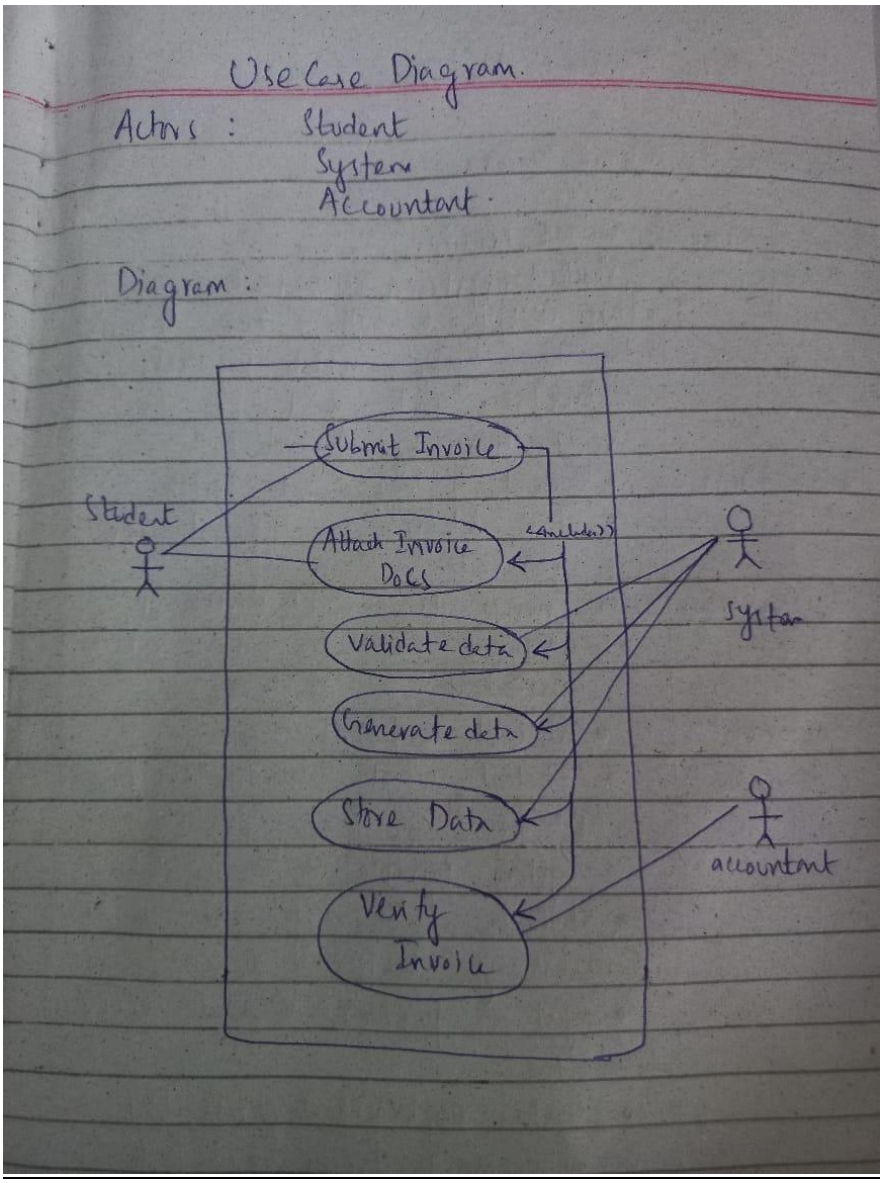
SUDAIS MURAD (USE CASE DIAGRAM):

Use Case: Process Payment



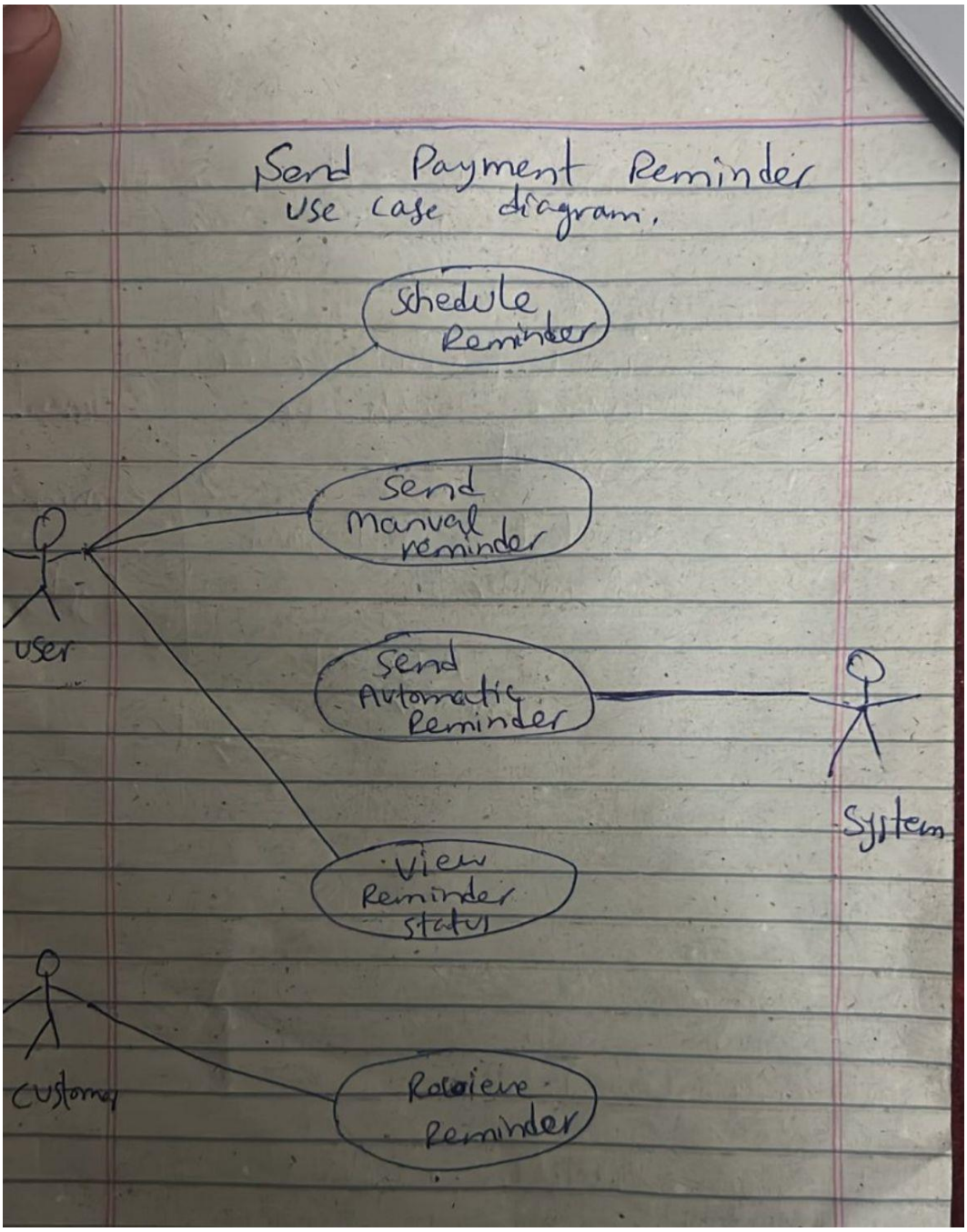
SHAYAN MUGHAL (USE CASE DIAGRAM):

Use case: Submit Invoice



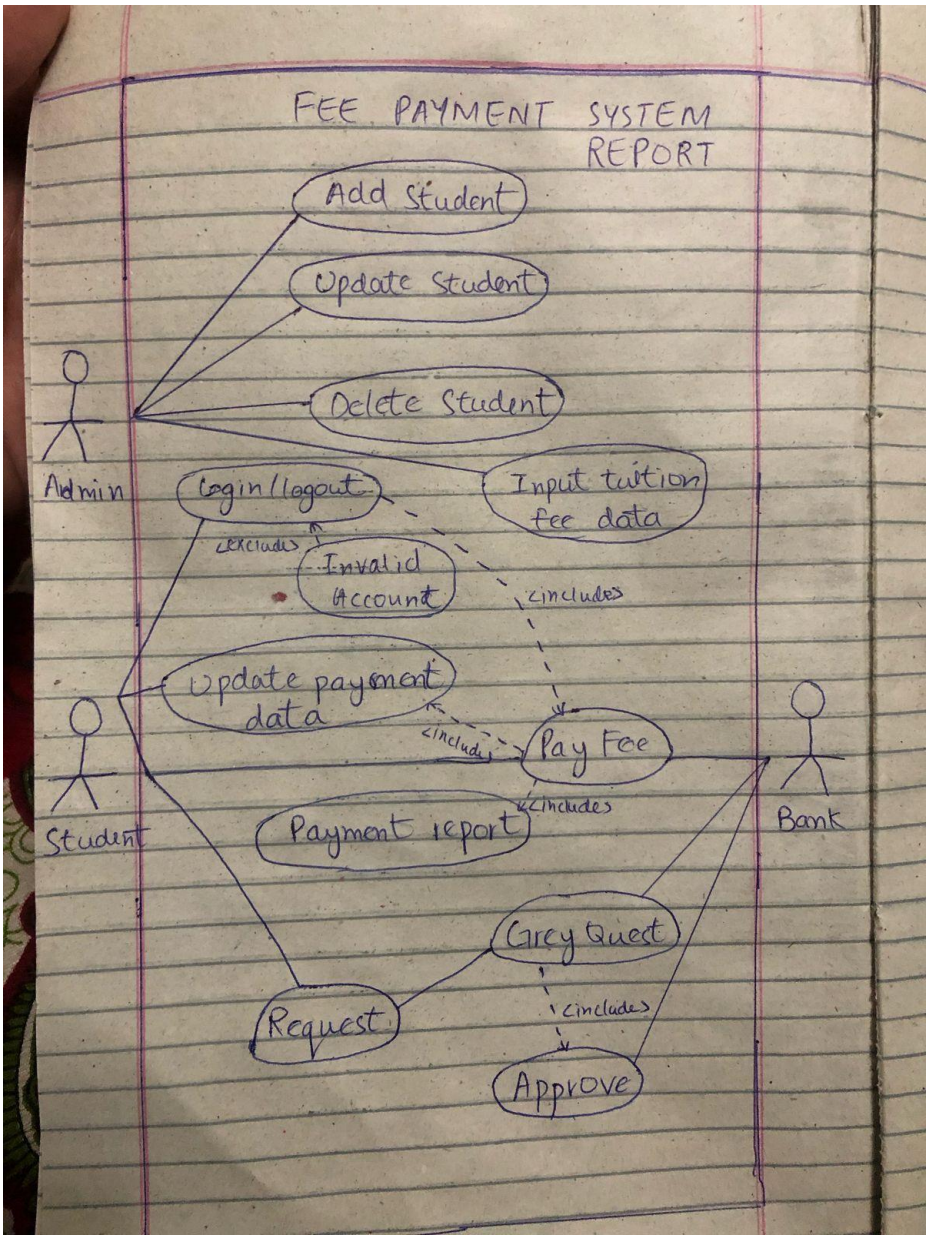
MASHOOD AZAM (USE CASE DIAGRAM):

Use Case: Send Payment Reminder

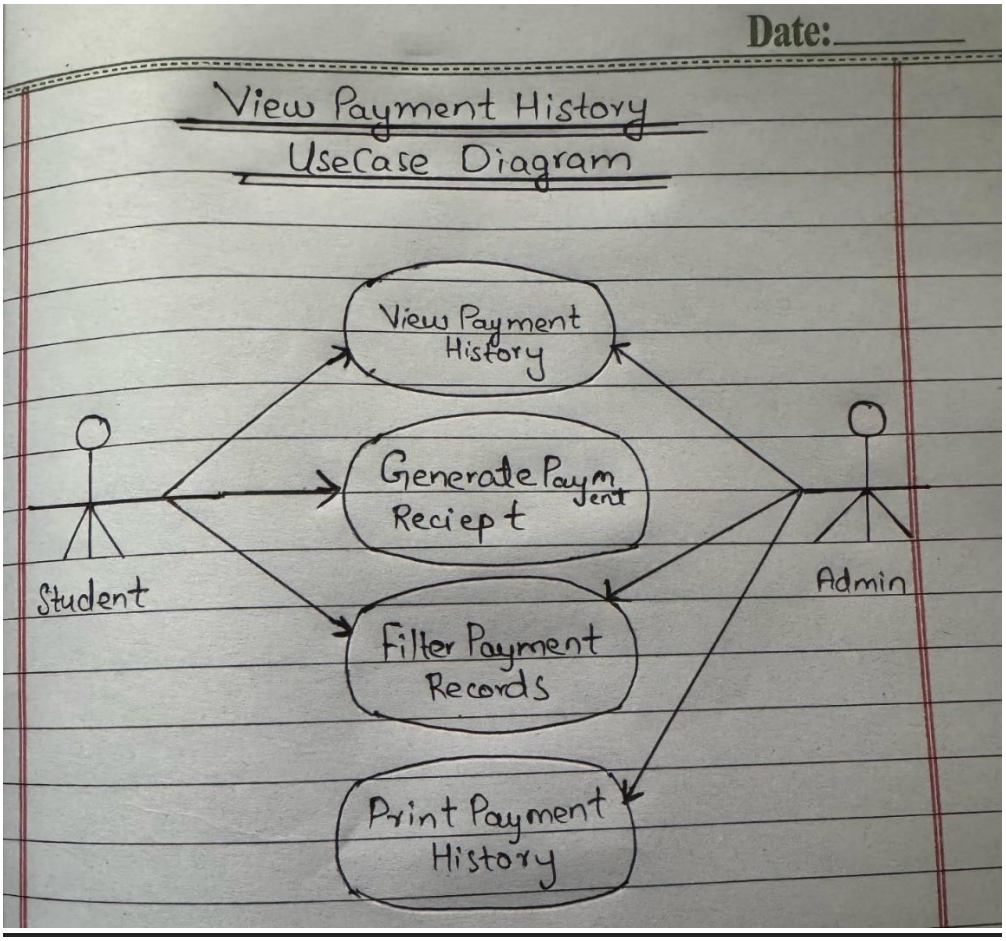


HAFIZ KHIZAR (USE CASE DIAGRAM):

Use Case: Generate Fee Receipt

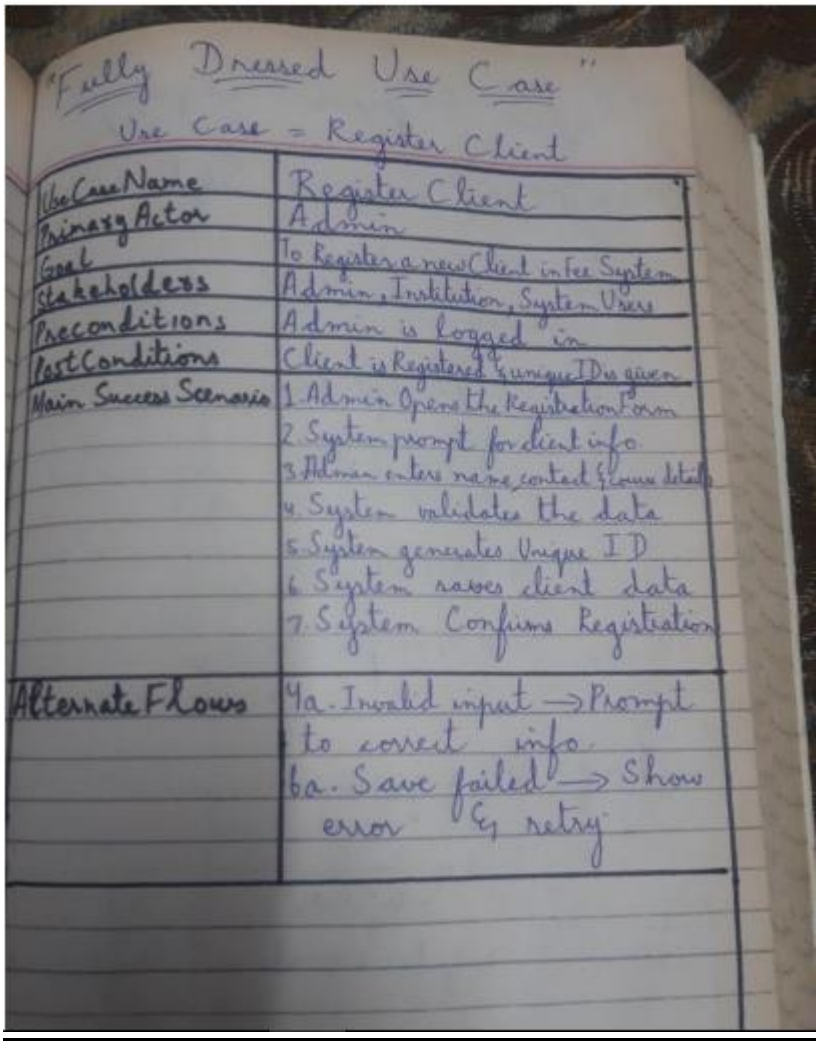


KAMRAN FIAZ (USE CASE DIAGRAM):



CHAPTER 2: FULLY DRESSED USE CASE

Ahmed Irshad:



"Fully Dressed Use Case"

Use Case = Register Client

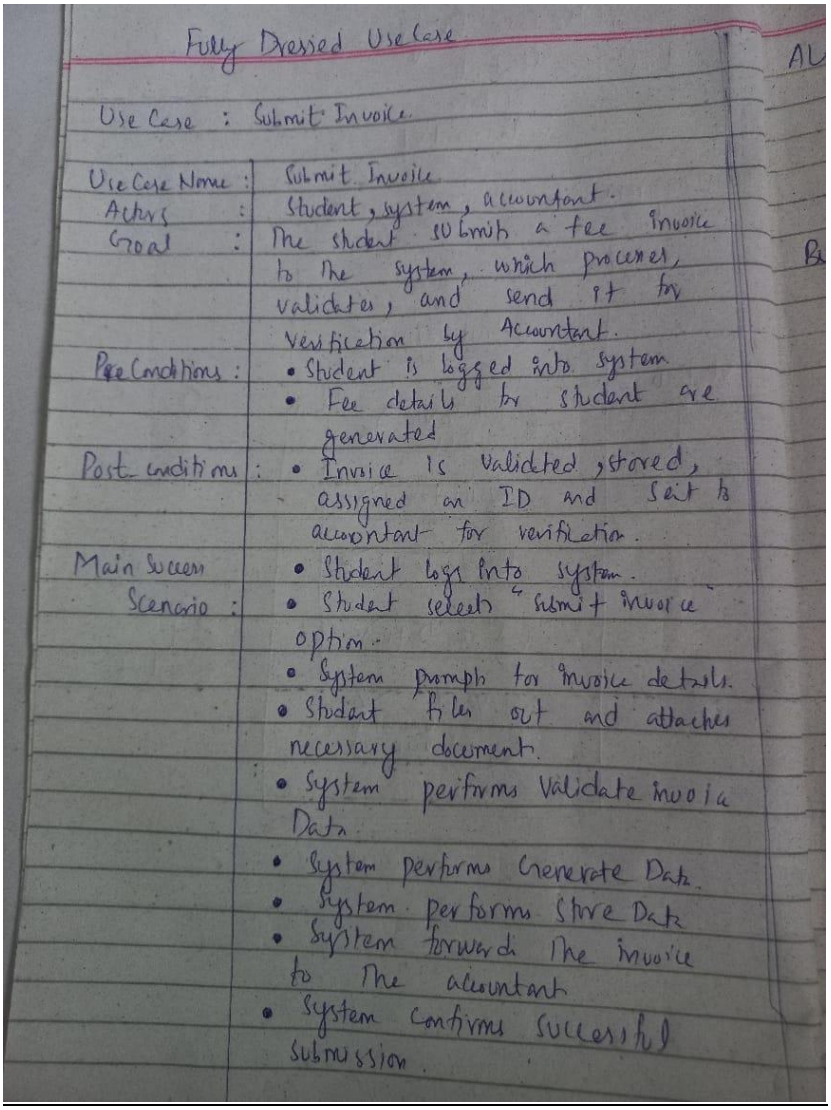
Use Case Name	Register Client
Primary Actor	Admin
Goal	To Register a new Client in Fee System
Stakeholders	Admin, Institution, System Users
Preconditions	Admin is logged in
Post Conditions	Client is Registered & unique ID is given
Main Success Scenario	1. Admin Opens the Registration Form 2. System prompt for client info 3. Admin enters name, contact & course details 4. System validates the data 5. System generates Unique ID 6. System saves client data 7. System Confirms Registration
Alternate Flows	4a. Invalid input → Prompt to correct info. 6a. Save failed → Show error & retry

SUDAIS MURAD:

Attribute	Details
Use Case Name	Process Payment
Use Case ID	UC-003
Primary Actor	Accountant, System (Automated)
Secondary Actors	Payment Gateway, User (Client)
Description	Records and validates a payment against an invoice, updates the client's balance, and generates a receipt.
Preconditions	1. Invoice exists and is approved. 2. Client account is active. 3. Payment details are valid.
Postconditions	1. Payment is recorded. 2. Client balance is updated. 3. Receipt is generated (if applicable).
Trigger	Accountant initiates payment or System detects a scheduled payment.
Main Success Scenario (Basic Flow)	1. Accountant/System selects the invoice for payment. 2. System validates invoice status and amount. 3. User/Payment Gateway provides payment details (credit card, bank transfer, etc.). 4. System processes payment and receives confirmation. 5. System records payment in the database. 6. System updates client balance. 7. System generates a receipt (optional). 8. Payment is marked as "Completed."
Alternative Flows	**E1: Invalid Payment** - 3a. Payment fails (e.g., insufficient funds). - 3b. System notifies Accountant/User and retries or cancels. **E2: Partial Payment** - 3a. User pays a partial amount. - 3b. System updates invoice status to "Partially Paid."

Attribute	Details
	E3: Offline Payment (Cash/Check) - 3a. Accountant manually marks payment as "Pending." - 3b. System updates status after manual verification.
Business Rules	1. Payments must match the invoice amount (± tolerance of 1%). 2. Overdue invoices may incur late fees. 3. Digital payments require gateway authentication.
Non-Functional Requirements	1. Payment processing time < 2 seconds. 2. PCI-DSS compliance for card payments. 3. Audit logs for all transactions.
Assumptions	1. Payment Gateway is always available. 2. Currency conversion is handled externally.

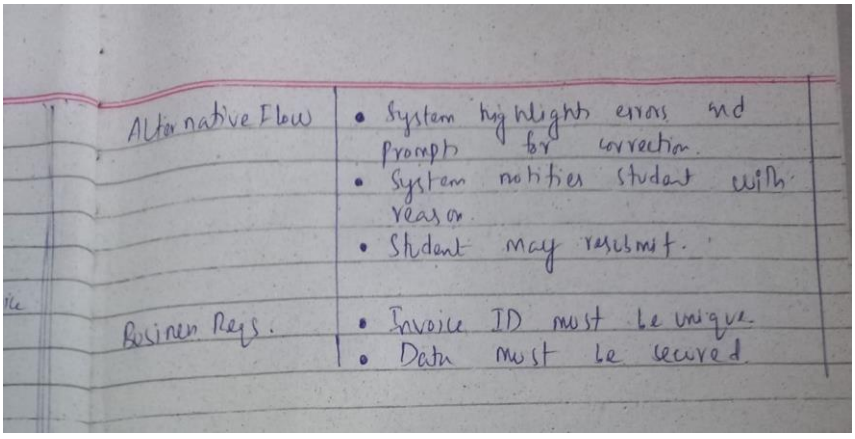
SHAYAN MUGHAL:



Fully Dressed Use Case

Use Case : Submit Invoice

Use Case Name :	Submit Invoice
Actors :	Student, system, accountant.
Goal :	The student submits a fee invoice to the system, which processes, validates, and send it for verification by Accountant.
Pre Conditions :	• Student is logged into system • Fee details for student are generated
Post conditions :	• Invoice is validated, stored, assigned an ID and sent to accountant for verification.
Main Success Scenario :	• Student logs into system. • Student selects "submit invoice" option. • System prompts for invoice details. • Student fills out and attaches necessary document. • System performs Validate invoice Data. • System performs Generate Data. • System performs Store Data • System forwards the invoice to the accountant • System confirms successful submission.



Alternative Flow	• System highlights errors and prompts for correction. • System notifies student with reason. • Student may resubmit.
Busines Reqs.	• Invoice ID must be unique. • Data must be secured.

MASHOOD AZAM:

Attribute	Details
Use Case Name	Send Payment Reminder
Scope	Payment Reminder System
Level	User Goal
Primary Actor	User (Admin or Staff)
Secondary Actor(s)	System, Customer
Stakeholders and Interests	- **User/Admin:** Wants to ensure customers are reminded of due payments. - **Customer:** Wants timely reminders to avoid penalties. - **System:** Automates reminders to reduce manual effort.
Preconditions	- The customer has a pending payment. - The due date is defined in the system. - Contact information (email/SMS) is available.
Postconditions	- The customer has received the reminder (email/SMS/notification). - Reminder status is updated in the system.
Main Success Scenario	1. User logs into the system. 2. User selects a customer or invoice with a due payment. 3. User selects 'Send Payment Reminder.' 4. The system fetches customer contact details. 5. The system sends a reminder via the configured channel (email/SMS). 6. The system logs the reminder in the reminder history. 7. Customer receives the reminder.
Extensions (Alternative Flows)	3a. Reminder is scheduled instead of sent immediately. 5a. Email/SMS server is unavailable → System retries or logs failure. 7a. Customer's contact info is invalid → System flags the error.
Special Requirements	- Reminder templates should be editable. - Multiple channels supported (Email, SMS, App).

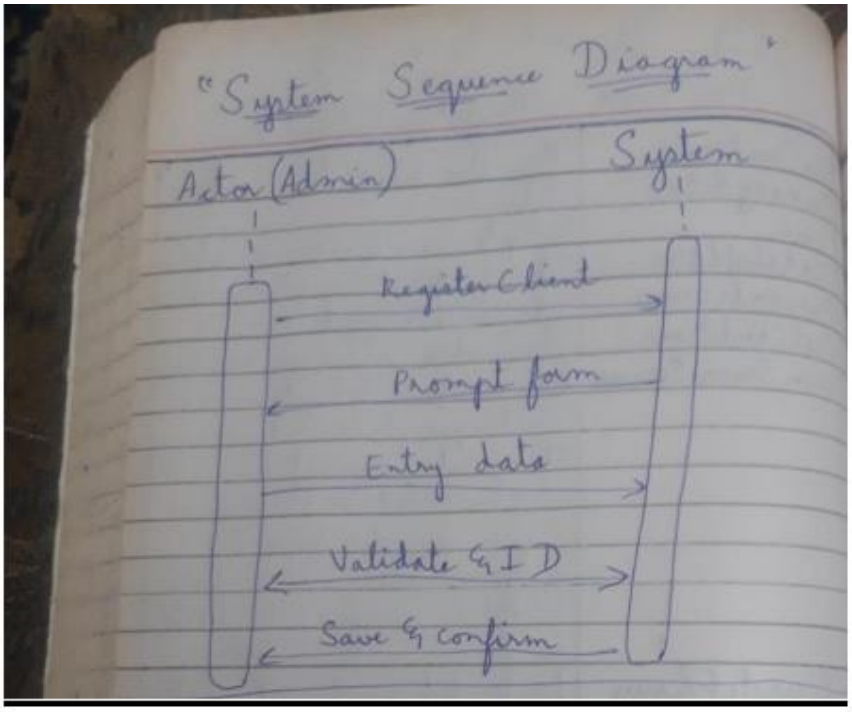
	- System must support scheduling and recurring reminders.
Frequency of Use	Daily or as configured.

HAFIZ KHIZAR:

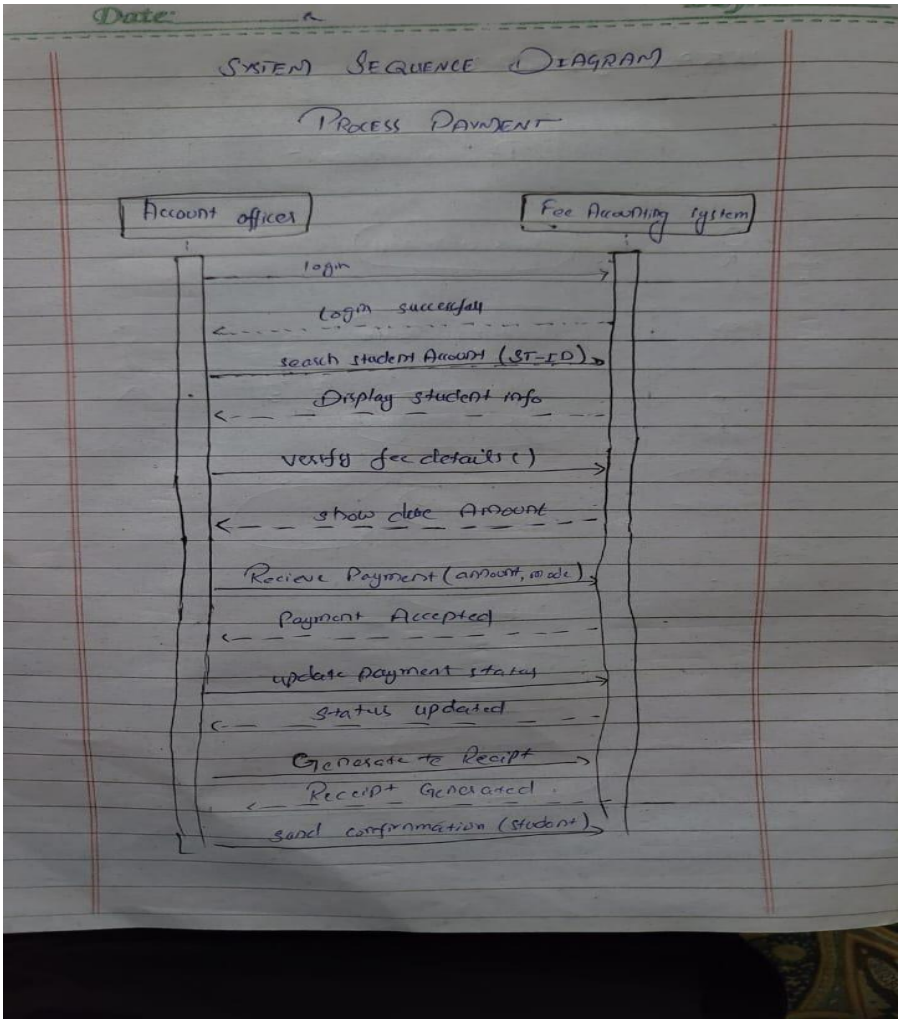
Use Case Name	Generate Fee Payment Report
Actor(s)	Admin, Accountant
Description	This use case allows the admin or accountant to generate a detailed report of student fee payments for record-keeping and analysis.
Trigger	The user selects the option to generate a fee report from the system interface.
Preconditions	The user must be logged into the system with appropriate privileges.
Postconditions	The system generates and displays/downloads the fee payment report.
Normal Flow	1. User navigates to the reports section. 2. User selects 'Generate Fee Payment Report'. 3. User sets the desired filters (date range, student name, class, etc.). 4. System fetches relevant payment data. 5. System displays or allows download of the report.
Alternate Scenarios	3a. No data available for selected filters: - System shows 'No data found' message.
Special Requirements	The report must be exportable in PDF and Excel formats.
Frequency of Use	As required by the admin or accountant, typically monthly or on demand.

CHAPTER 3: SSDs

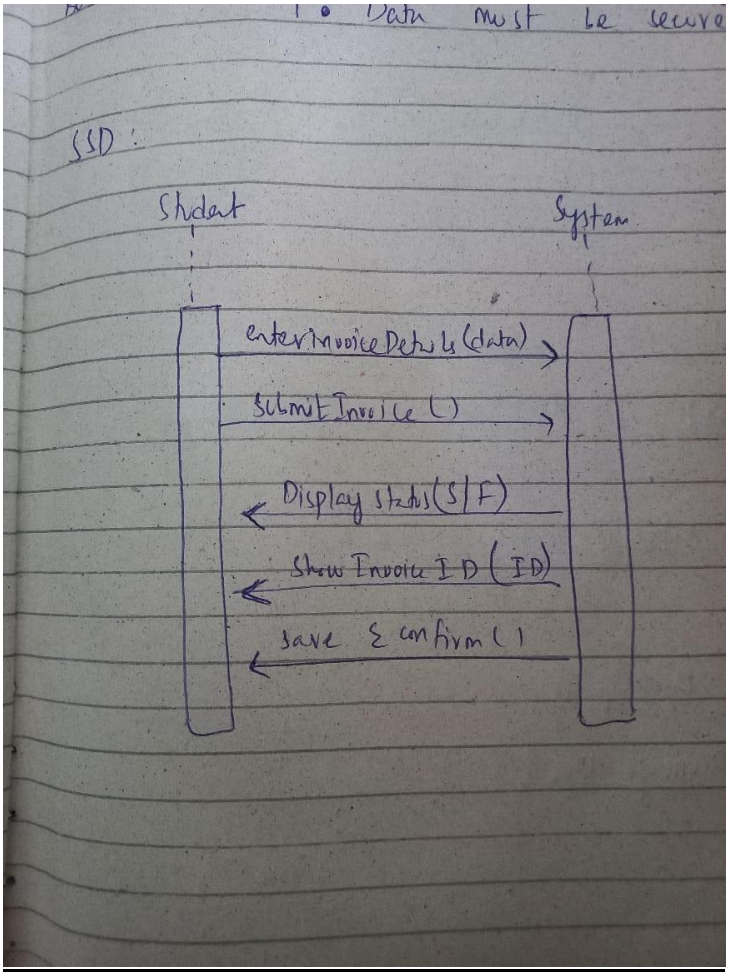
Ahmed Irshad:



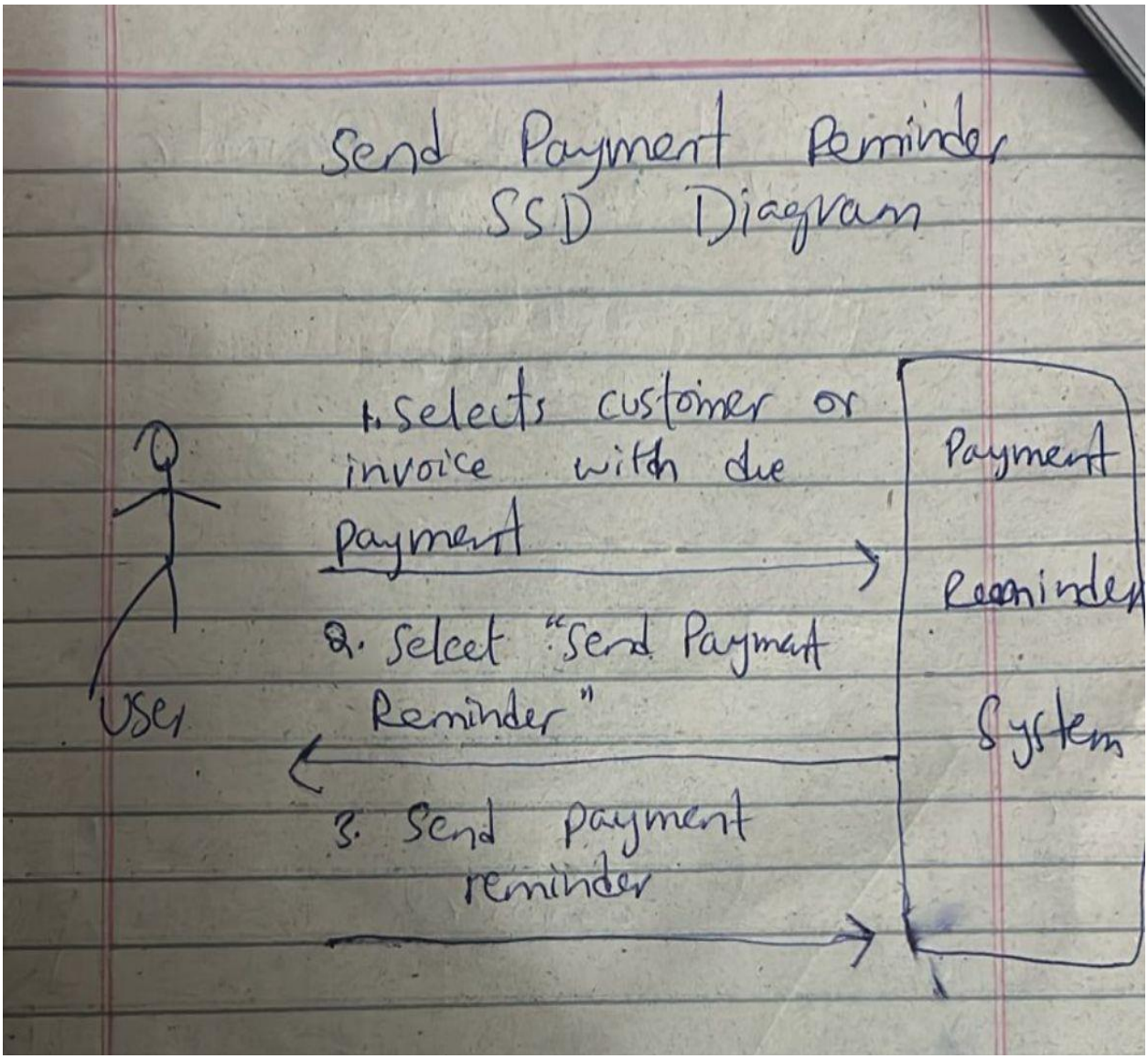
SUDAIS MURAD:



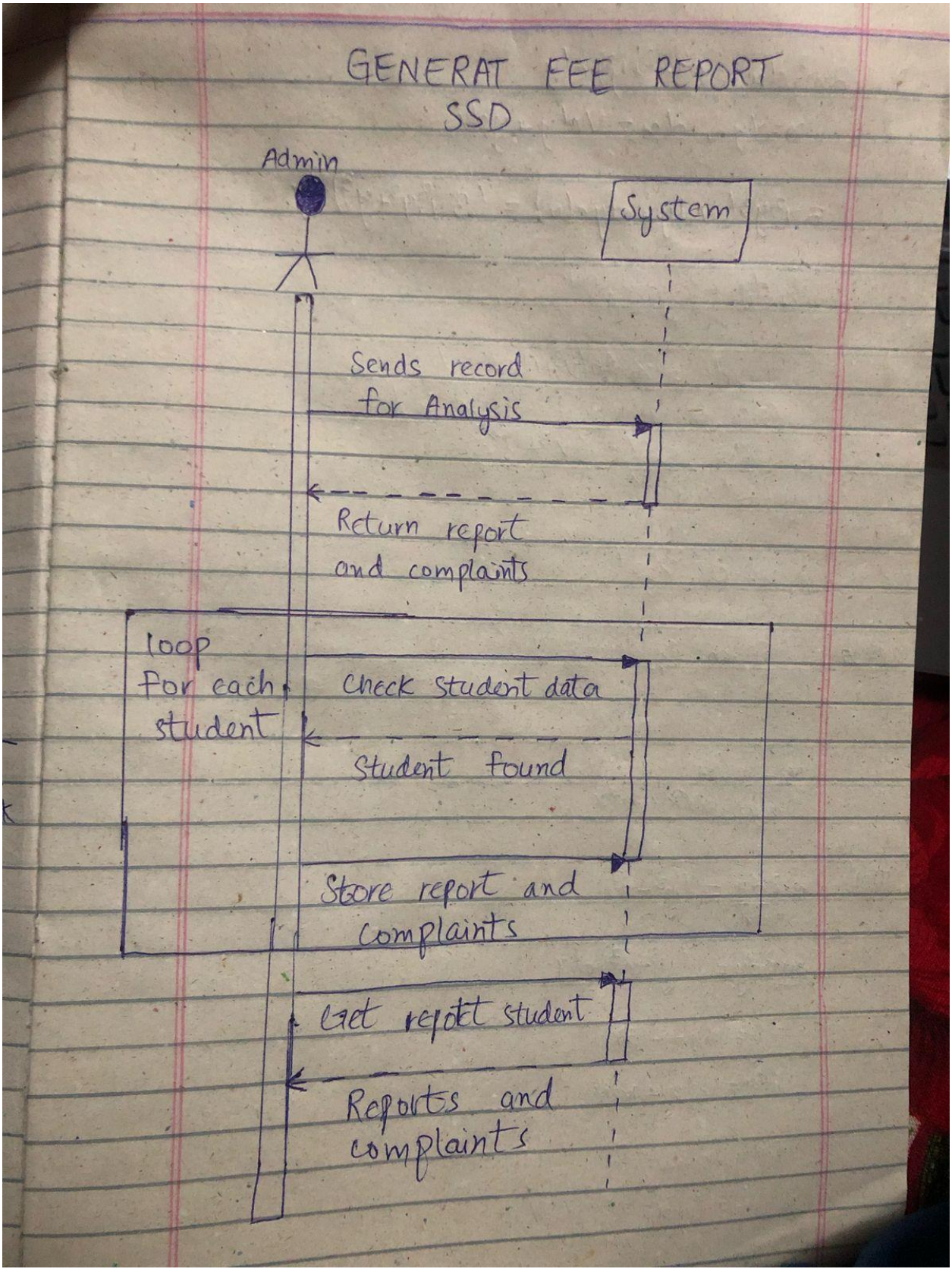
Shayan Mughal:



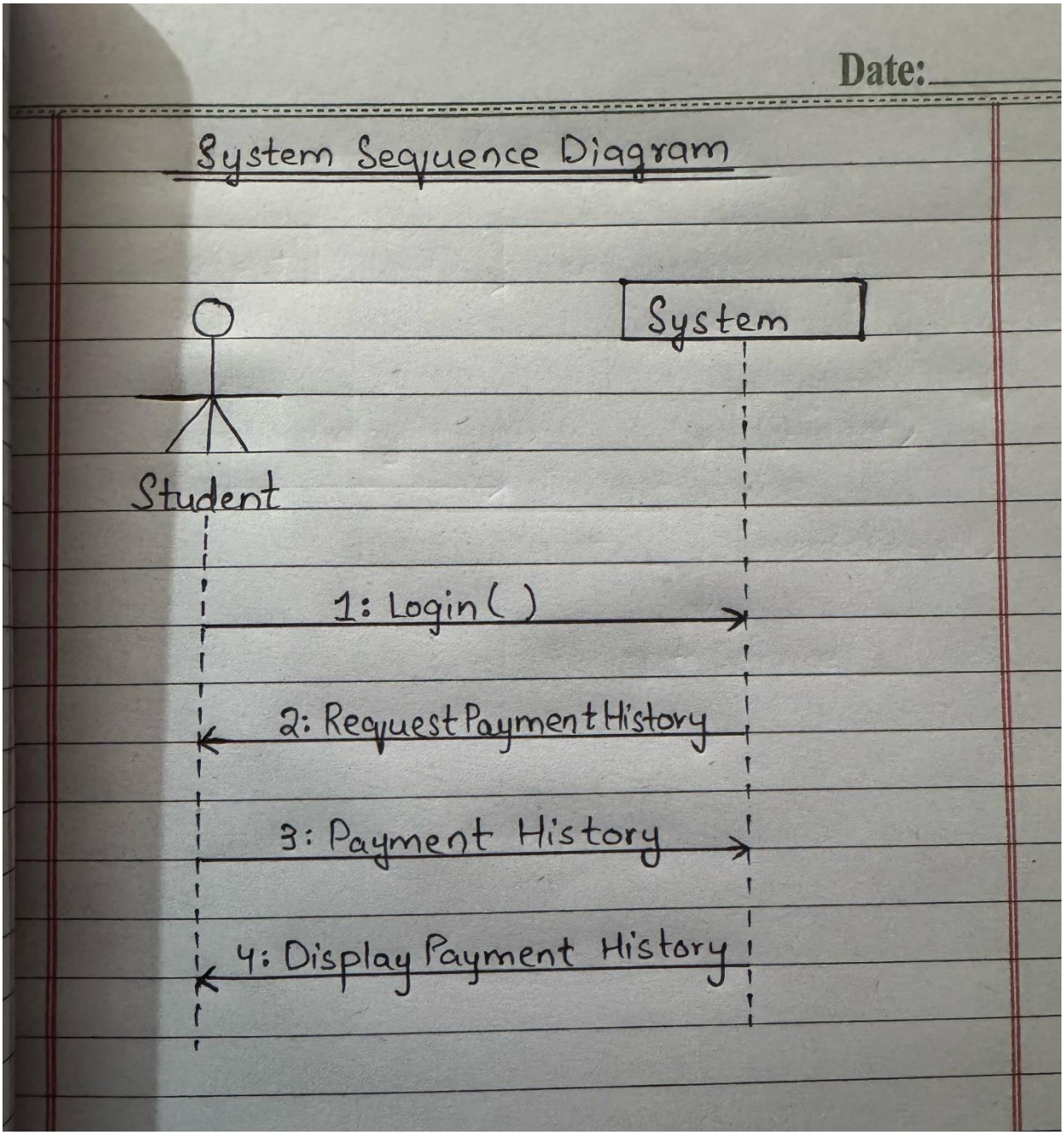
Mashood Azam:



HAFIZ KHIZAR:

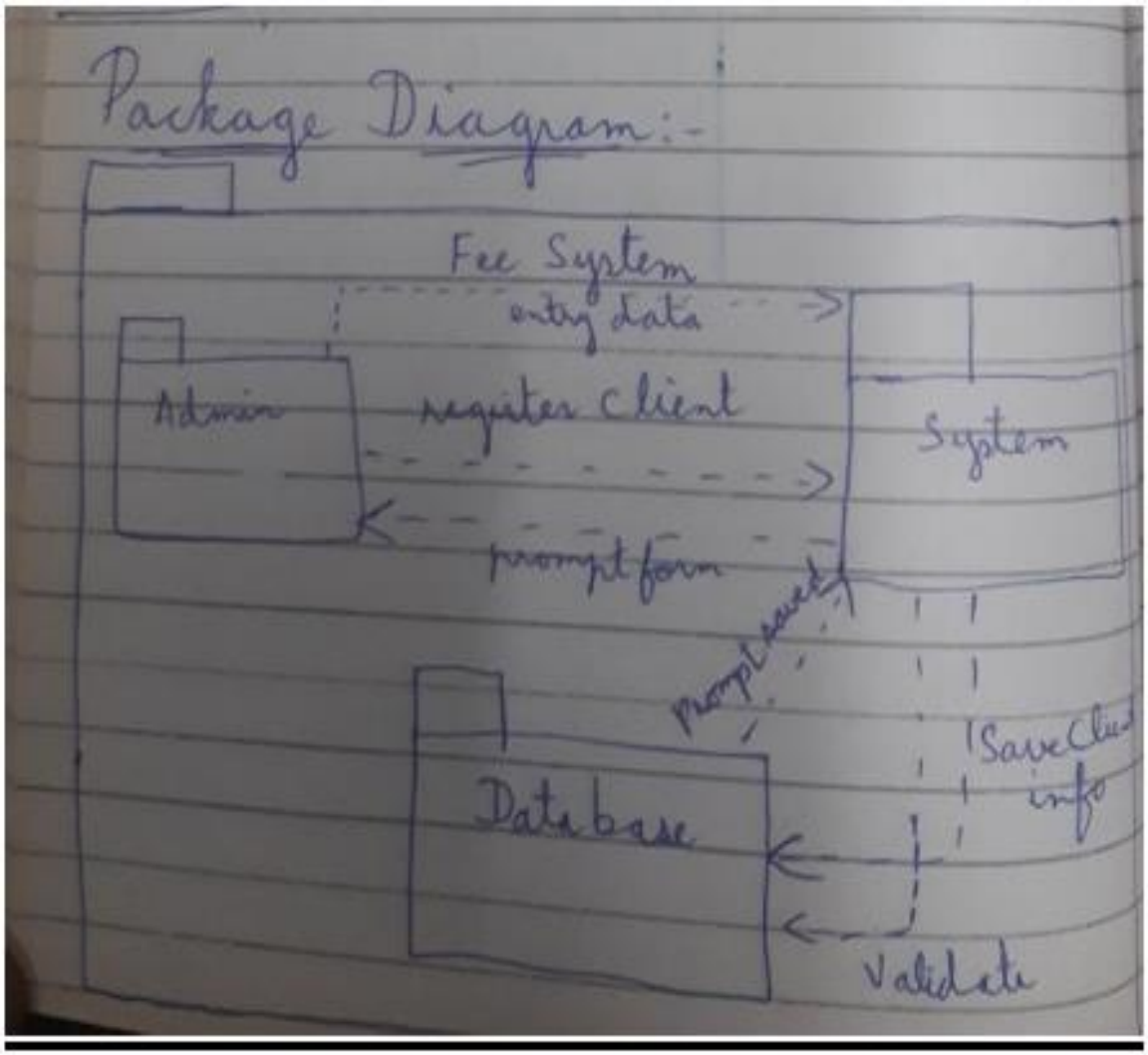


KAMRAN FIAZ:

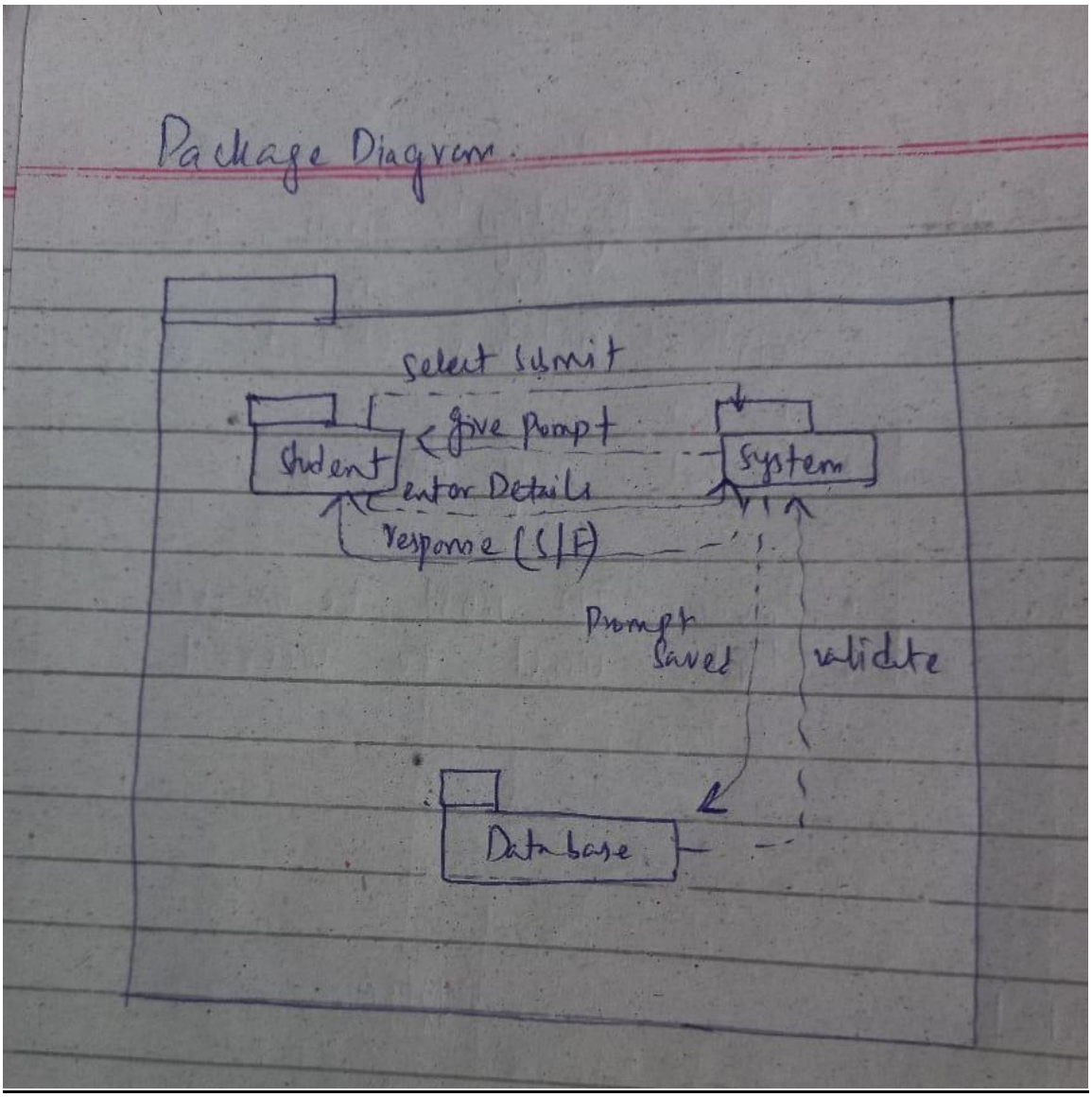

Date:
System Sequence Diagram
System
Student
1: Login( )
2: Request Payment History
3: Payment History
4: Display Payment History

CHAPTER 4: PACKAGE DIAGRAM

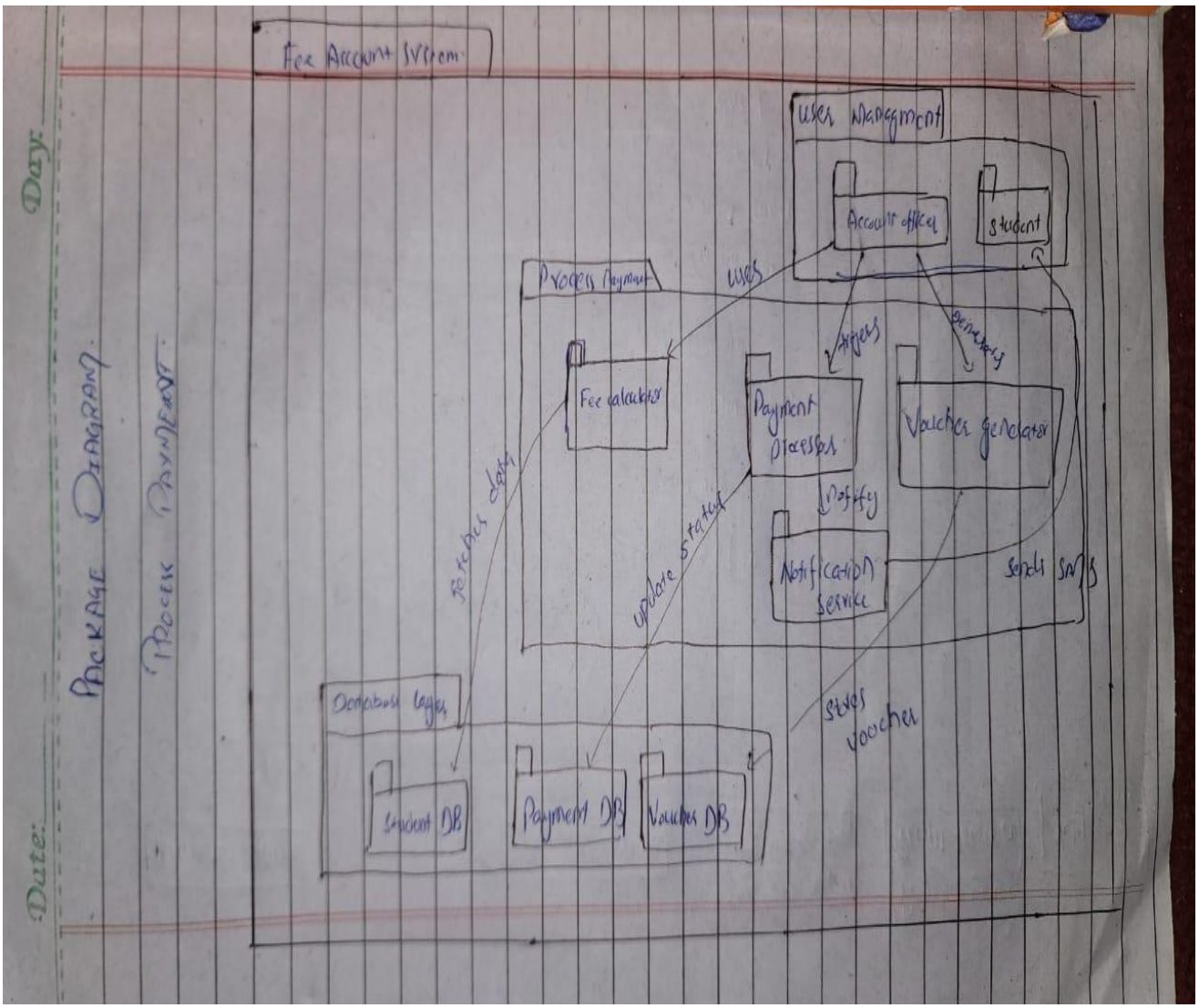
AHMED IRSHAD:



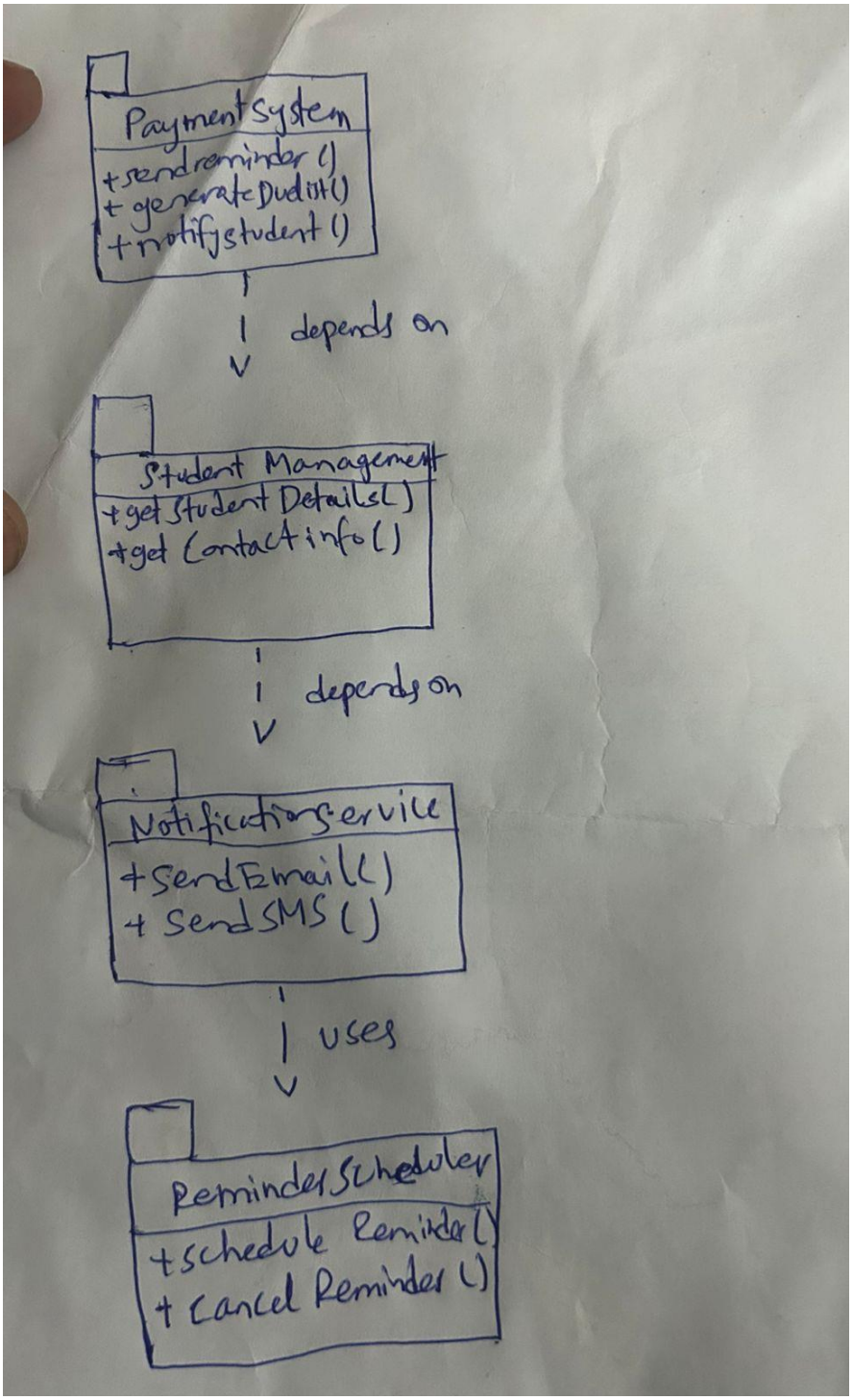
SHAYAN MUGHAL:



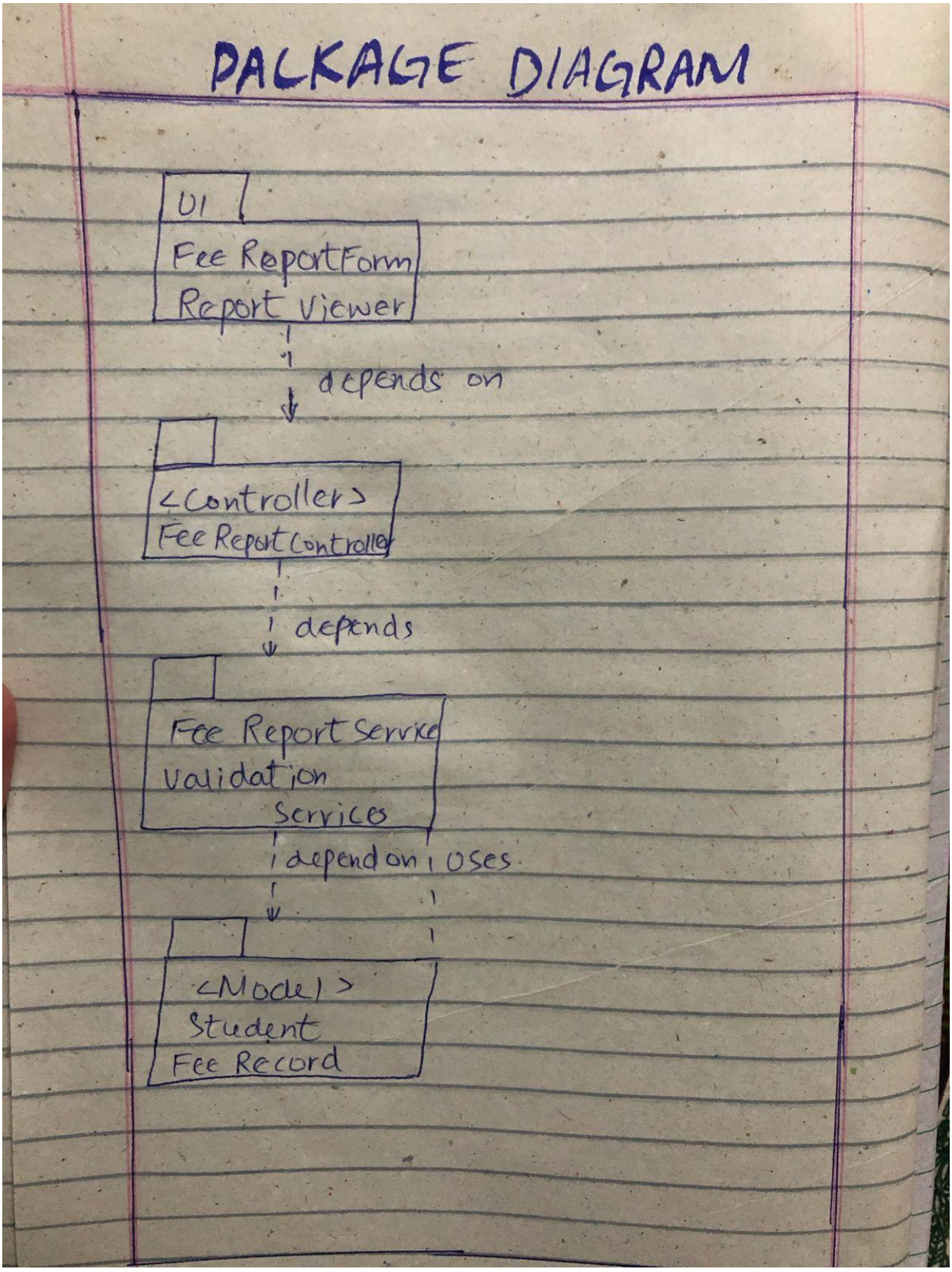
SUDAIS MURAD:



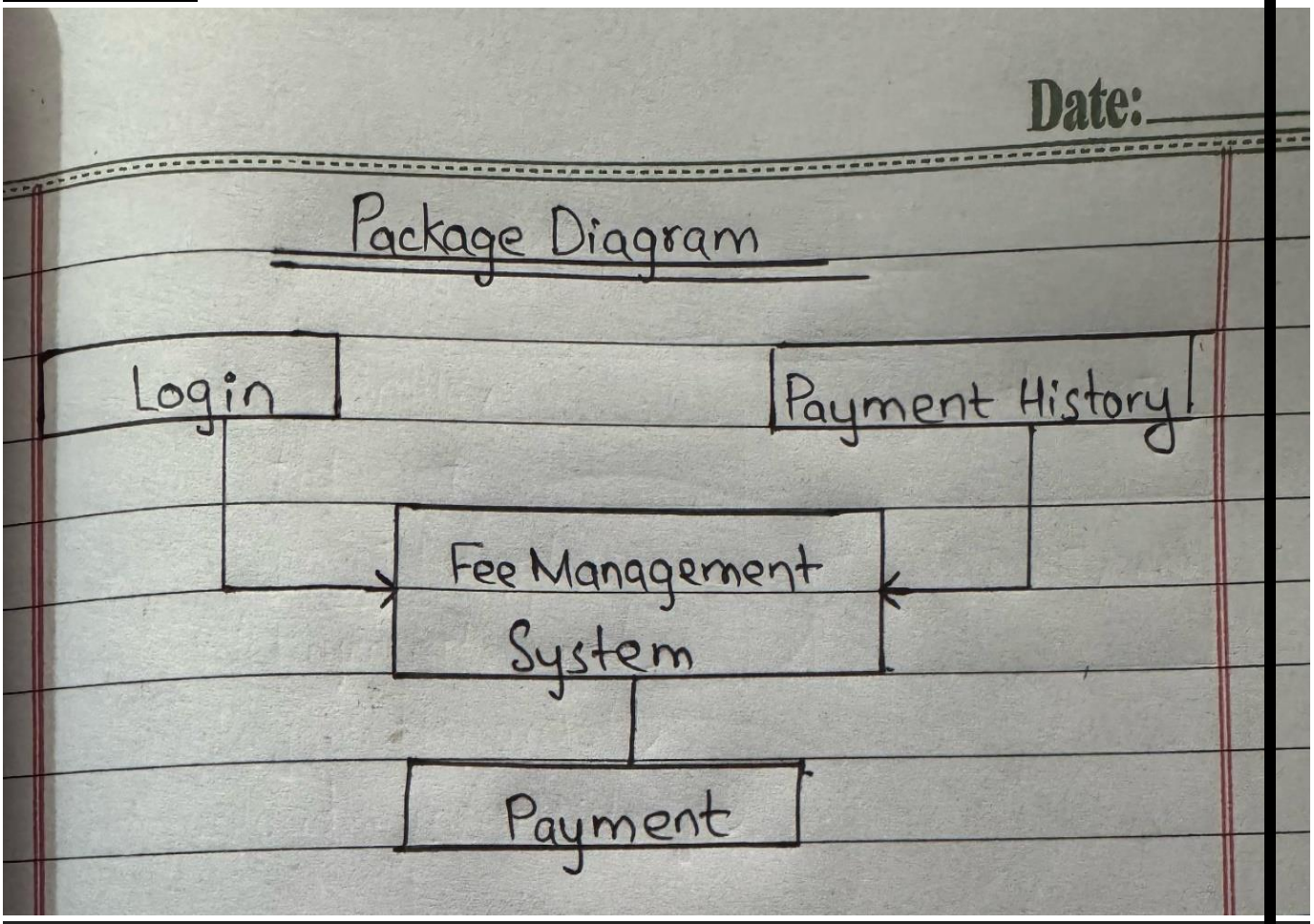
MASHOOD AZAM:



HAFIZ KHIZAR:

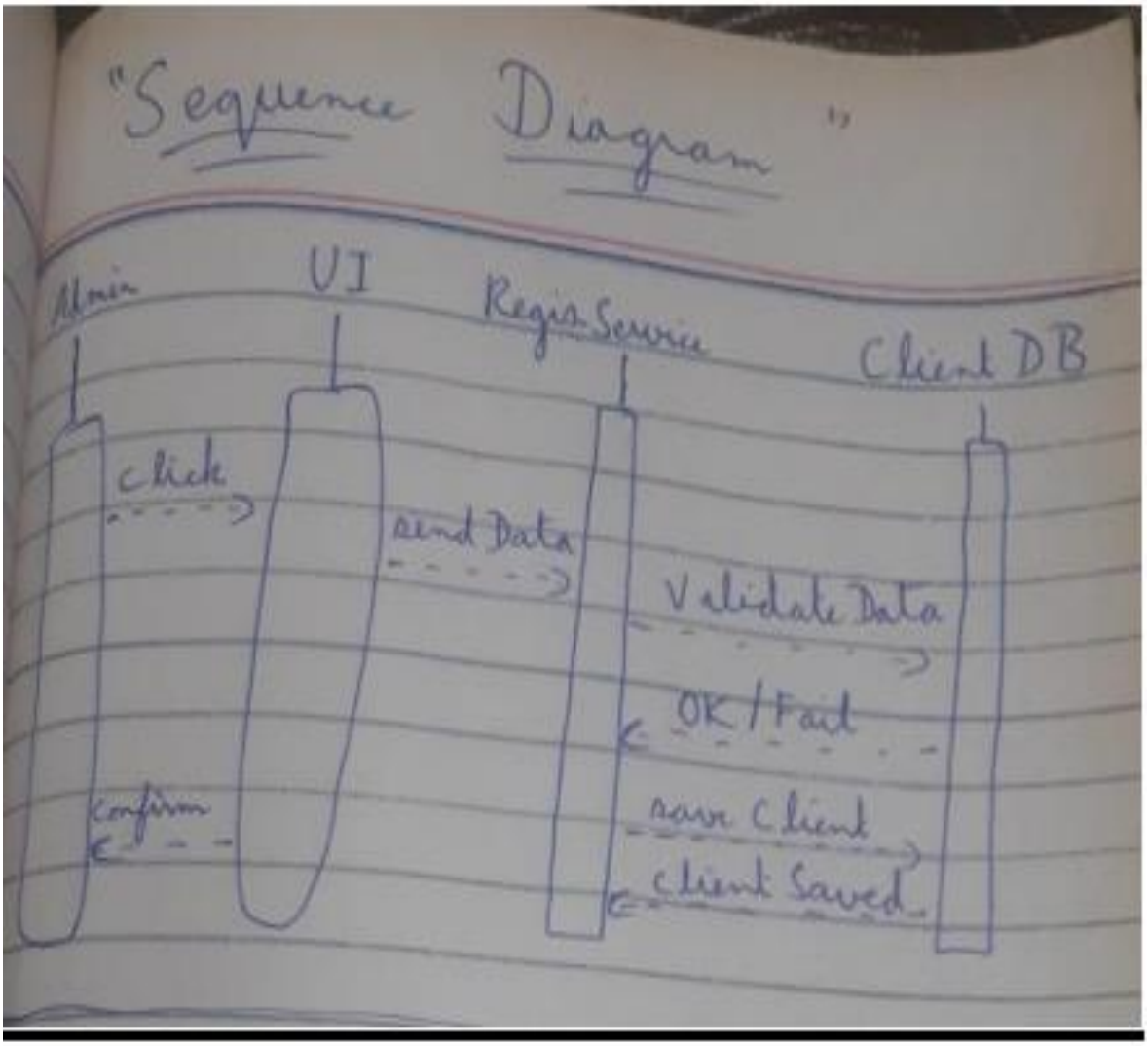


KAMRAN FIAZ:

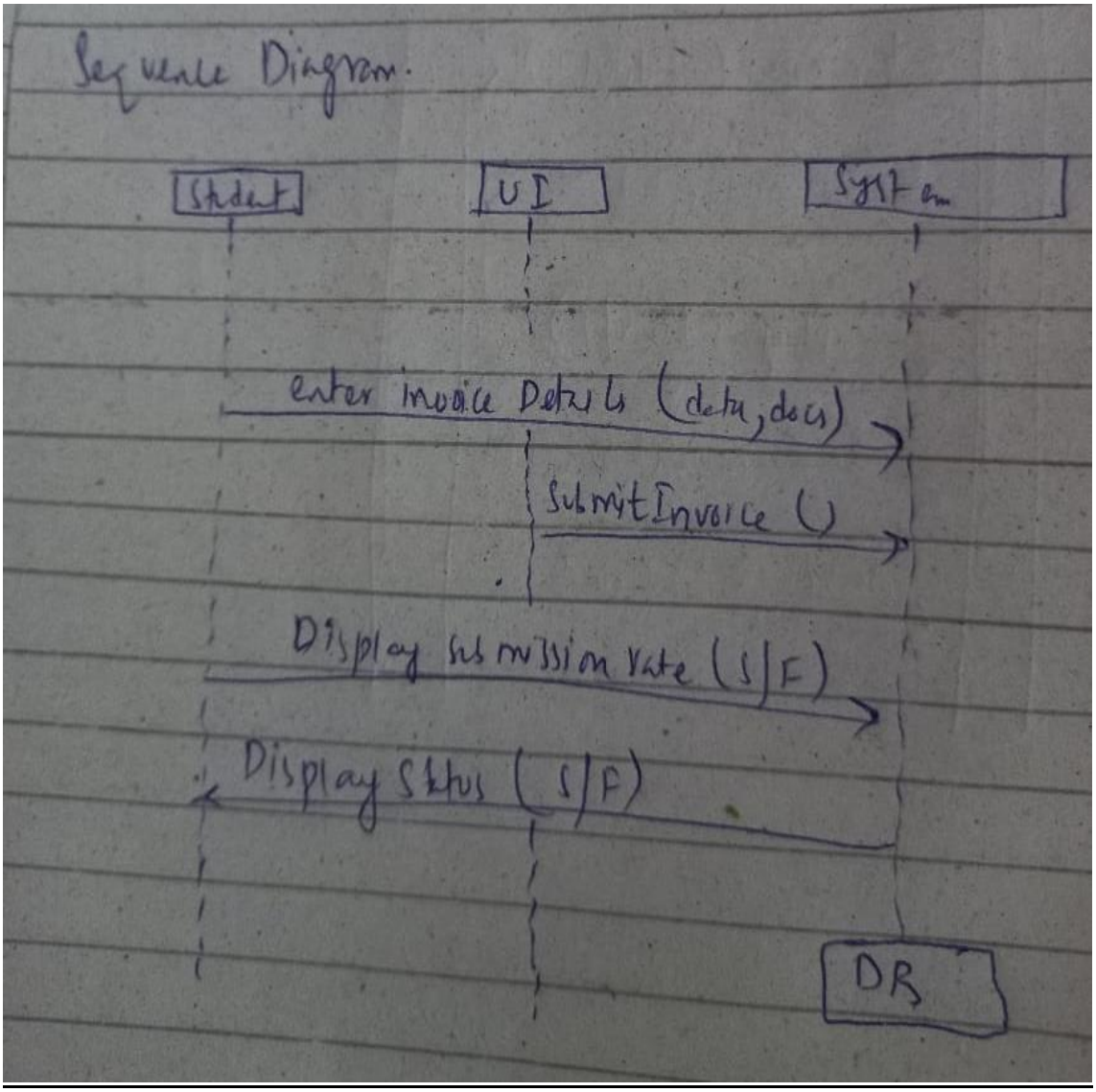

Date:
Package Diagram
Login
Payment History
Fee Management System
Payment

CHAPTER 5: COMMUNICATION DIAGRAM

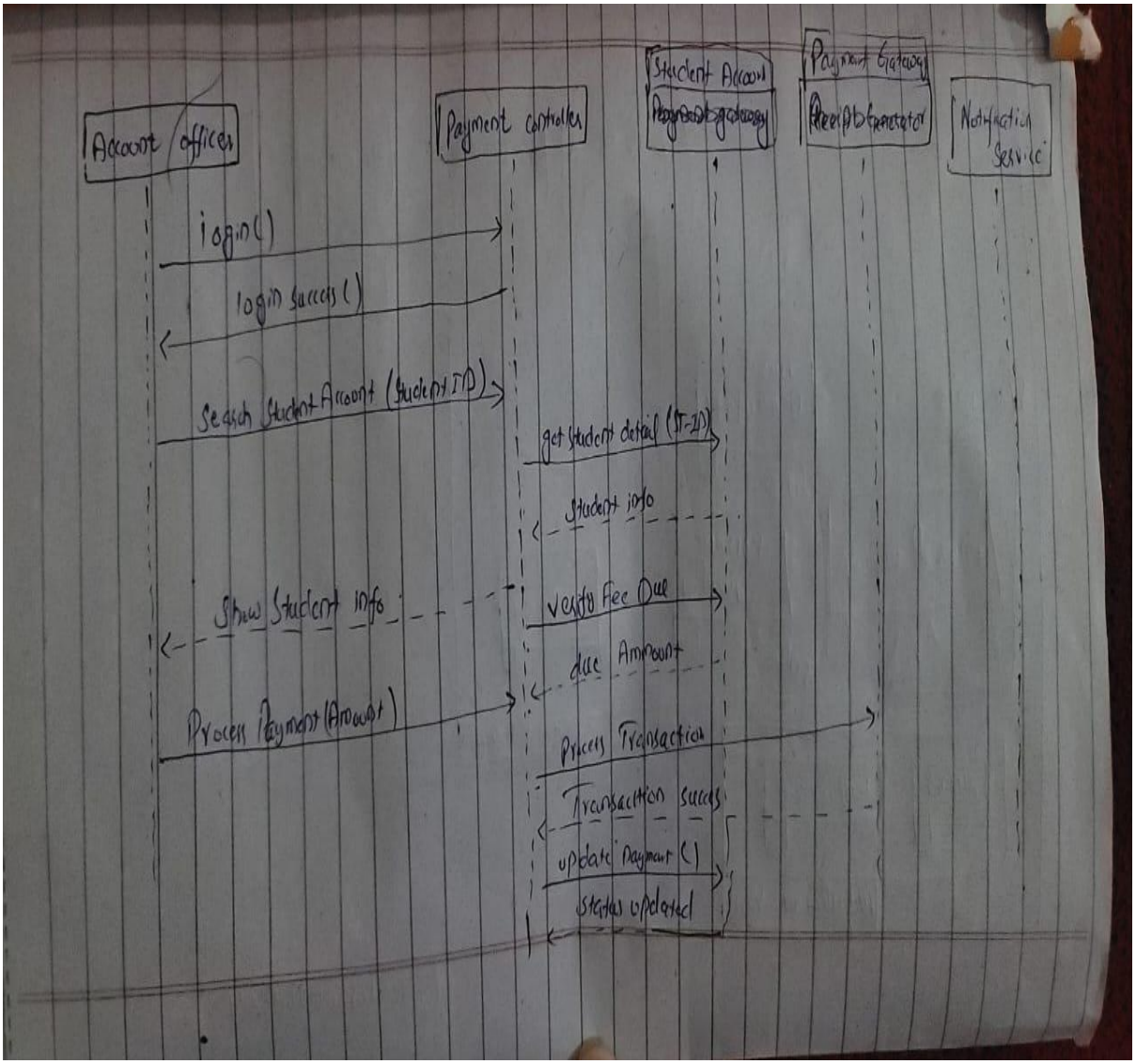
AHMED IRSHAD:



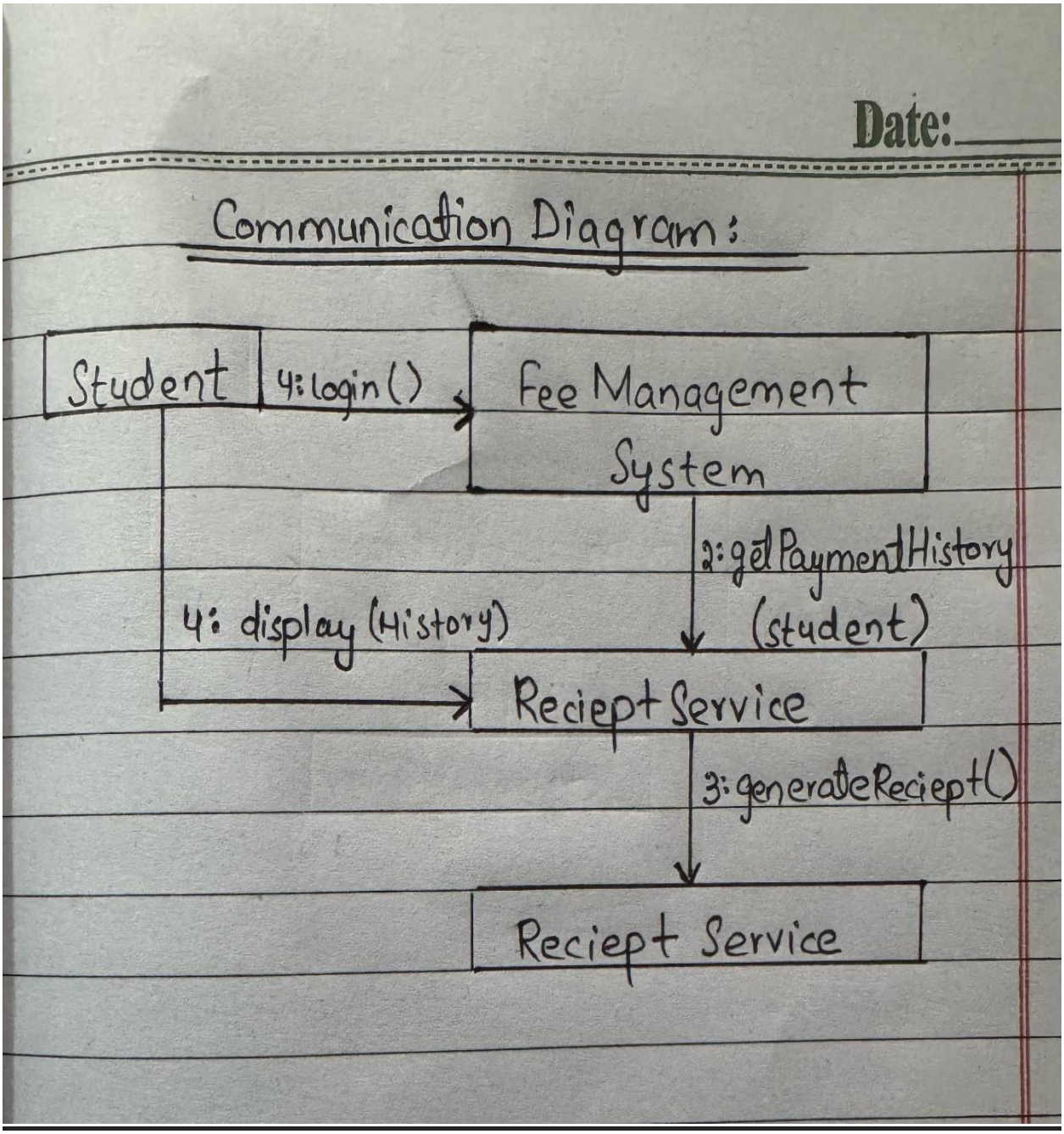
SHAYAN MUGHAL:



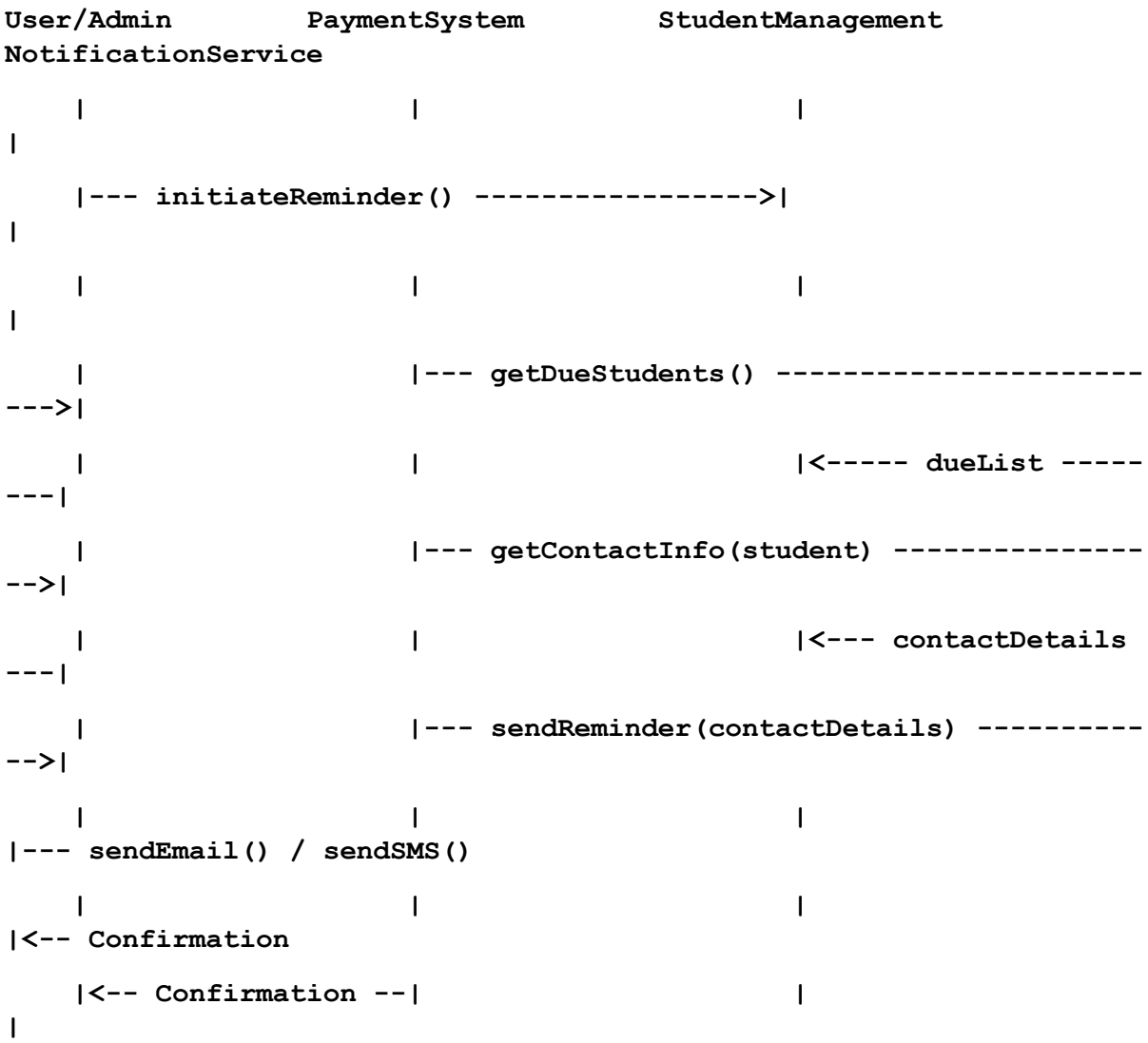
SUDAID MURAD:



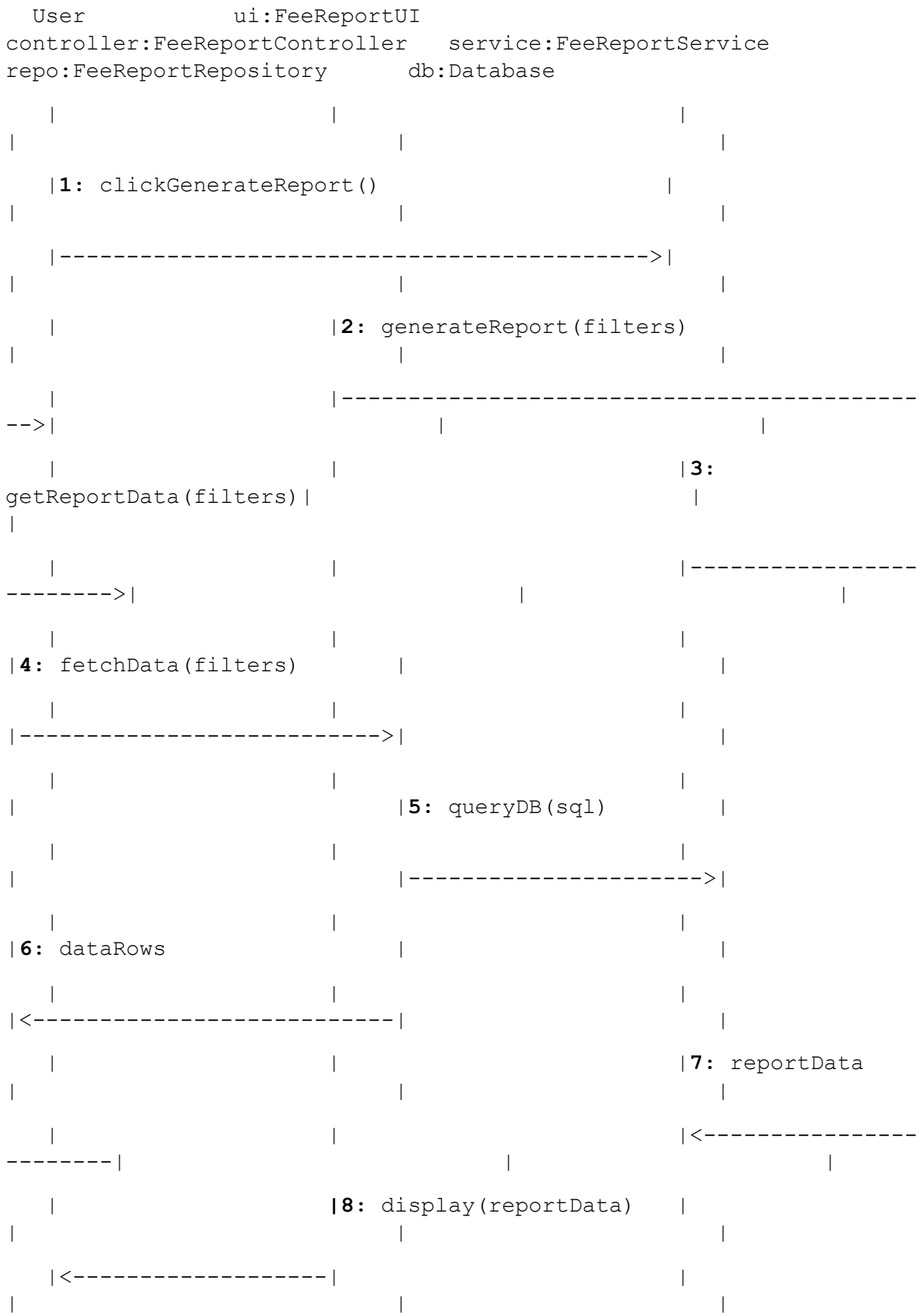
KAMRAN FIAZ:


Date:
Communication Diagram:
Student
4: login()
Fee Management System
2: getPaymentHistory (student)
4: display (History)
Reciept Service
3: generateReciept()
Reciept Service

MASHOOD AZAM:

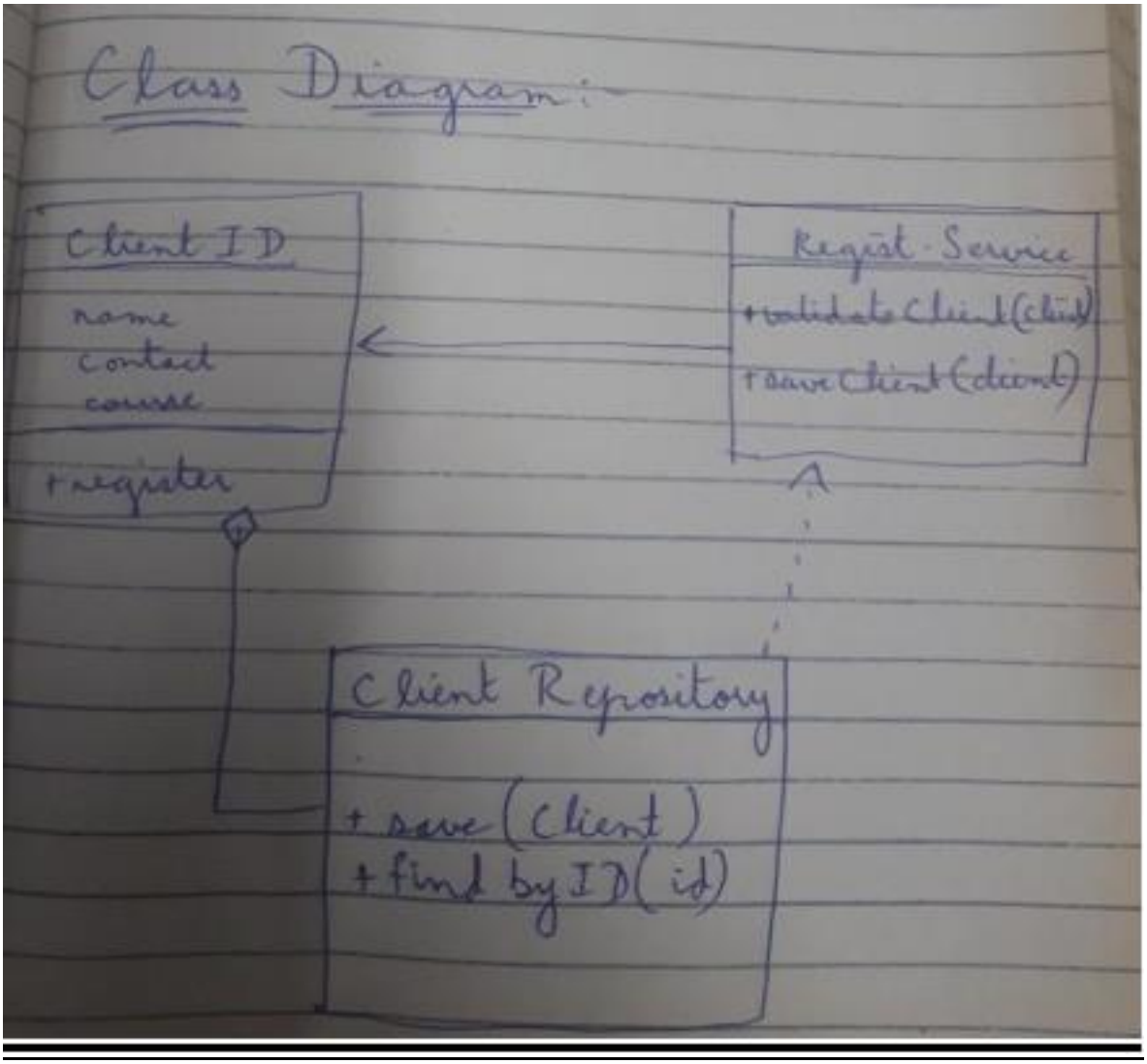


HAFIZ KHIZAR:

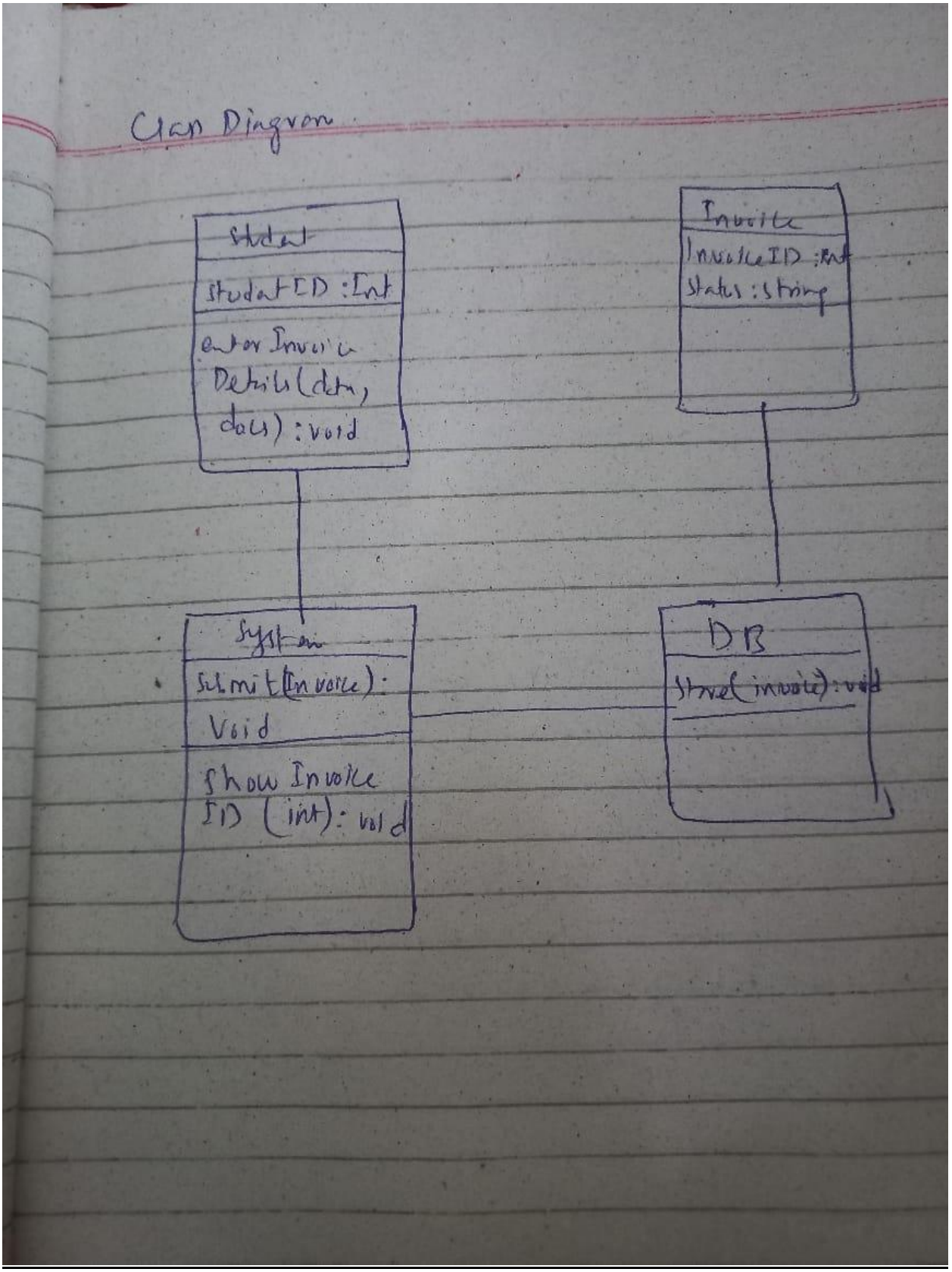


CHAPTER 6: CLASS DIAGRAMS:

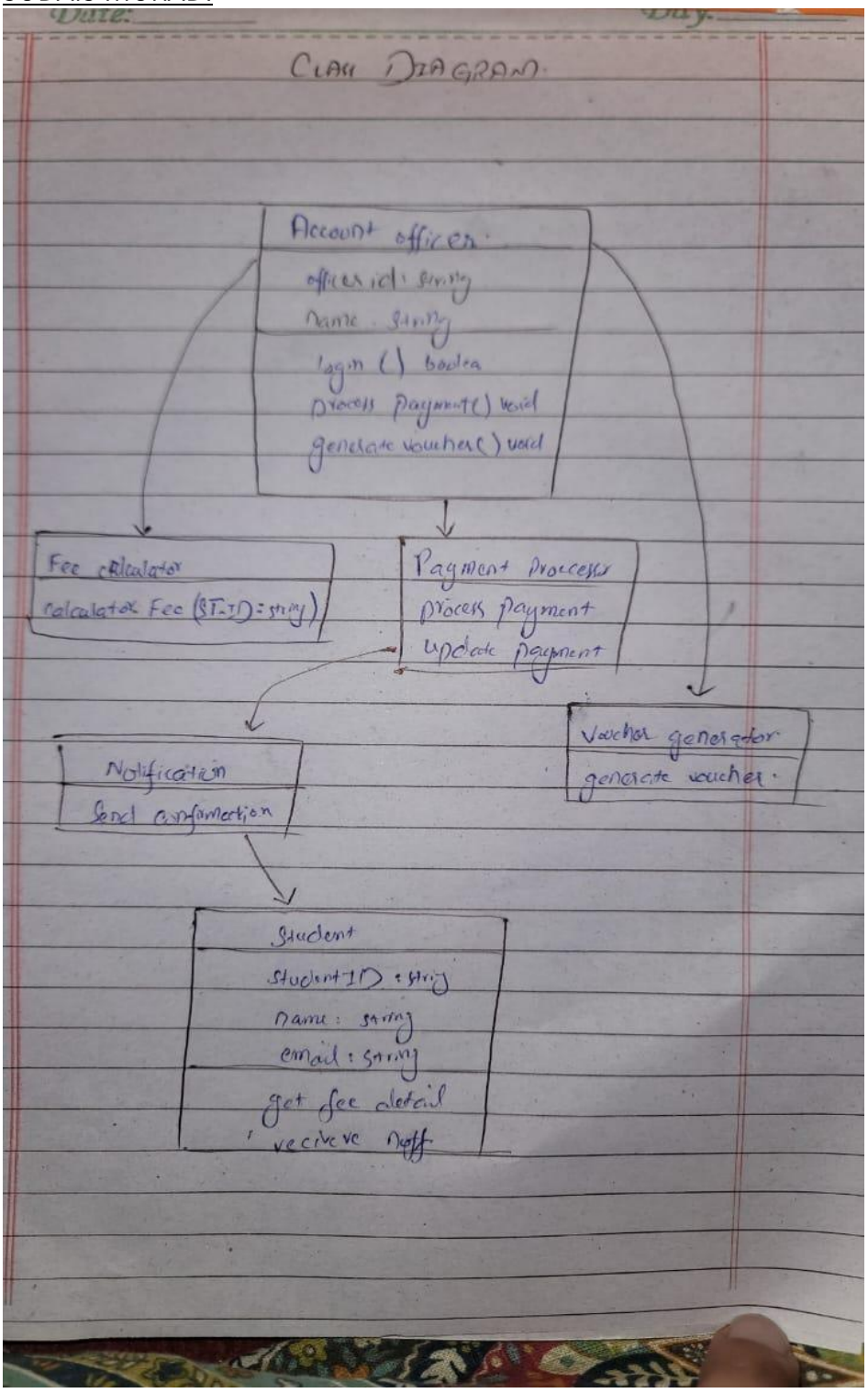
AHMED IRSHAD:



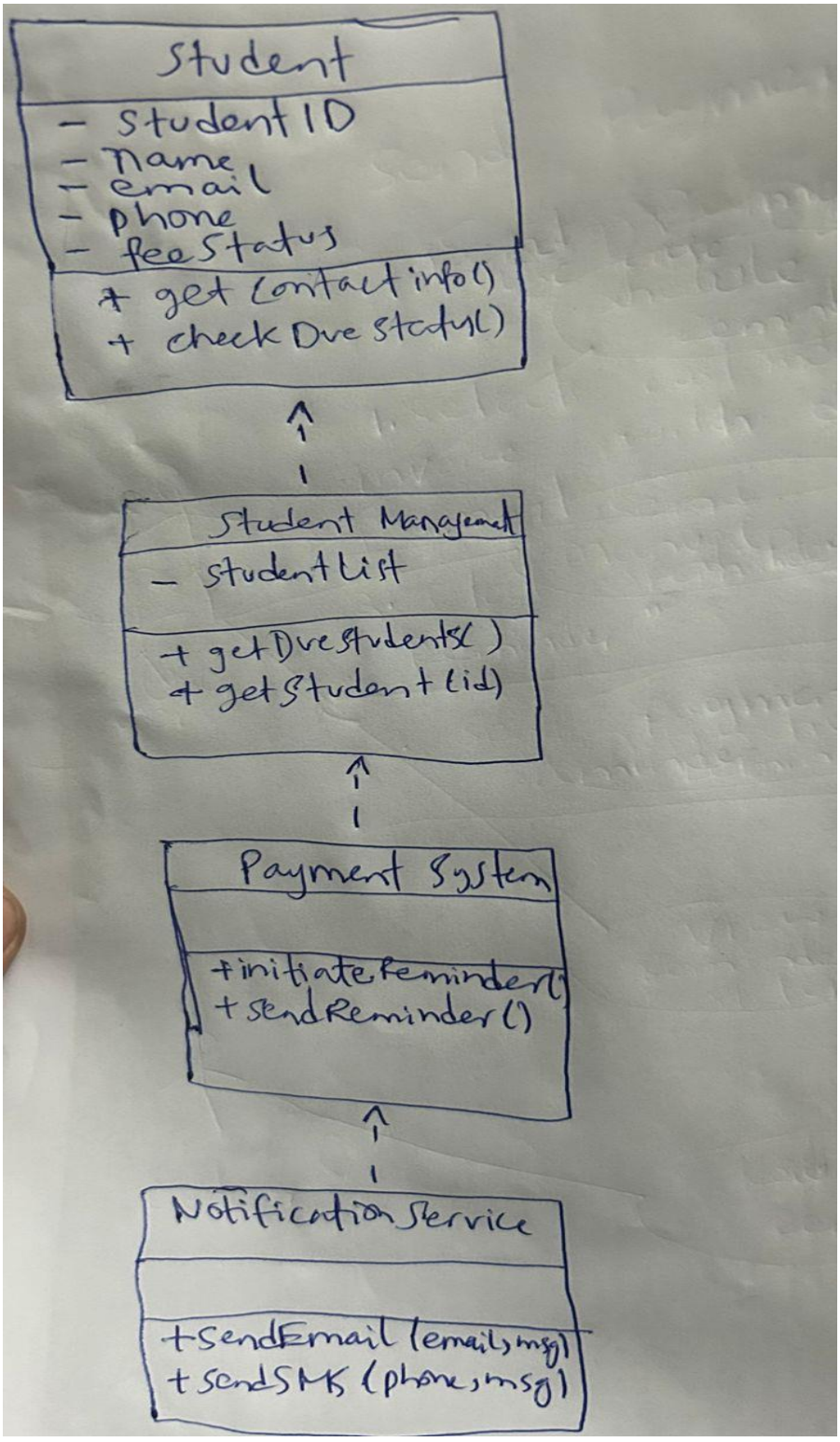
SHAYAN MUGHAL:



SUDAIS MURAD:

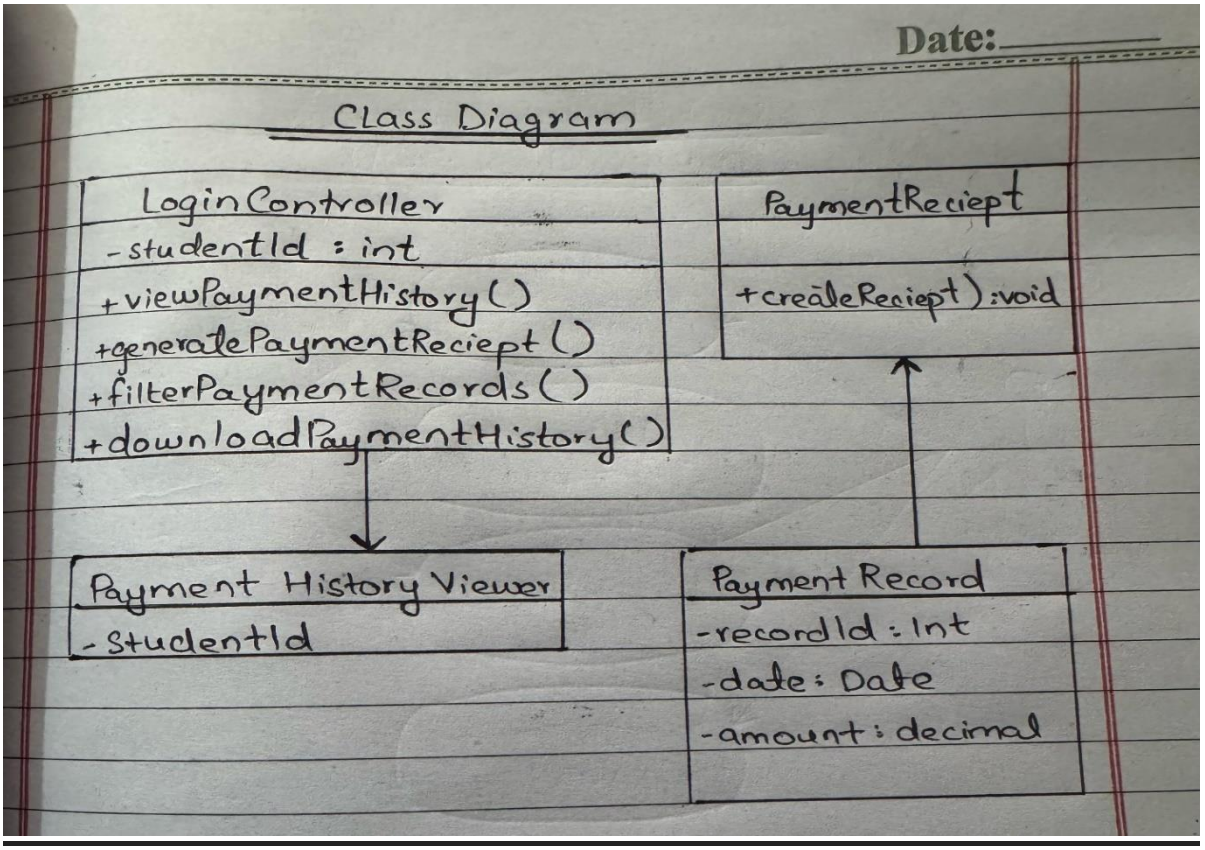


MASHOOD AZAM:



HAFIZ KHIZAR:

KAMRAN FIAZ:



CHAPTER 7: CODING STANDARDS

AHMED IRSHAD:

Element	Convention	Example
Project Name	PascalCase	`FeeSystem`
Package/Module	lowercase	`registration, data`
Class Names	PascalCase	`Client, FeeManager`
Method Names	camelCase	`registerClient()`
Variable Names	camelCase	`clientName, contactNumber`
Constants	UPPER_CASE	`MAX_CLIENTS`

---

2. Folder/Package Structure:

feesystem/

├── models/ # Data classes (e.g., Client)

├── services/ # Business logic (e.g., RegistrationService)

├── repository/ # Data handling (e.g., ClientRepository)

├── ui/ # Console or GUI interfaces

└── utils/ # Helper methods, validation, etc.

3. Code Formatting:

- Indentation: 4 spaces (no tabs)
- Line Length: Max 100 characters

- Braces: Open on the same line (Java style)

```
if (isValid) {
   registerClient(client);}
```

1. Error Handling:

```
try {
   clientRepository.save(client);
} catch (DatabaseException e) {
   System.err.println("Error saving client: " + e.getMessage());}
```

5. Validation and Security:

- Validate all user inputs (null checks, format checks, etc.).
- Sanitize inputs to avoid injection or corruption.
- Use encapsulation: keep class fields private, expose through getters/setters.

6. Testing Standards (optional for now):

- Write test cases for methods like validateClient(), saveClient().
- Use dummy/mock objects for testing repository behavior.

SUDAIS MURAD:

Folder/Package Structure

```
bash
CopyEdit
feesystem/
├── models/              # Data classes (e.g., Payment)
├── services/            # Business logic (e.g., PaymentProcessor)
├── repository/          # Data storage logic (e.g., PaymentRepository)
├── ui/                  # Interfaces for payment input/output
└── utils/               # Validation, currency formatting, etc.
```

1. Code Formatting

- **Indentation**: 4 spaces (no tabs)
- **Line Length**: Max 100 characters
- **Braces**: Open on same line

```
Java

if (paymentAmount > 0) {
    processPayment(studentId,
    paymentAmount);
}
```

2. Error Handling

Use `try-catch` for any payment failures (e.g., database, API):

```
java
CopyEdit
try {
    paymentRepository.save(payment);
} catch (PaymentException e) {
    System.err.println("Payment failed: " + e.getMessage());
}
```

3. Validation and Security

- Validate fields like `studentId`, `amount`, `paymentMode`
- Check for null, negative values, or invalid formats
- Sanitize all inputs before processing
- Use encapsulation:

```
java CopyEdit
private double paymentAmount;

public double getPaymentAmount() { return
    paymentAmount;
}
```

4. Testing Standards (optional for now)

- Write test cases for:
  - `processPayment()`
  - `validatePaymentDetails()`
  - `calculateLateFee()`
- Use mock repositories or services to isolate logic during testing

MASHOOD AZAM:

1. Package Naming

- Use all **lowercase** letters.
- Names should be short, meaningful, and based on functionality (e.g., `usermanagement`, `reporting`).

2. Class Naming

- Use **PascalCase** (each word starts with a capital letter).
- Class names should be **nouns** that reflect their responsibility (e.g., `ReminderScheduler`, `EmailSender`).

3. Method Naming

- Use **camelCase** (first word lowercase, following words capitalized).

- Method names should be **verbs** that describe the action (e.g., `sendReminder`, `generateReport`).

4. Variable Naming

- Use **camelCase**.
- Names should be meaningful and self-descriptive (e.g., `dueDate`, `customerEmail`).
- Avoid single-letter or vague variable names.

5. Access Modifiers

- Use appropriate access modifiers:
    - `private` for internal fields
    - `public` for externally accessible methods
    - Avoid `public` fields

6. Constants

- Use `final` and all **UPPER_SNAKE_CASE** for constants.
- Define constants in a separate utility class if needed.

7. Code Structure

- Follow this order in each class:
    1. Fields
    2. Constructors
    3. Public methods
    4. Private/helper methods

8. Documentation

- Use **JavaDoc** comments for:
    - Each public class and method
    - Parameters and return values

9. Exception Handling

- Use proper try-catch blocks where exceptions might occur (e.g., sending emails, file export).
- Don’t ignore exceptions silently.

10. Separation of Concerns

- Keep each class focused on a **single responsibility**.
- Avoid mixing UI, business logic, and data handling in the same class.

11. Code Readability

- Use proper **indentation and spacing**.
- Leave a blank line between methods.
- Avoid writing long methods; split into smaller helper methods if needed.

12. Modular Design

- Divide functionality logically across **packages** and **classes**.
- Ensure minimal coupling and high cohesion.

HAFIZ KHIZAR:

Aspect	Standard / Guideline
Naming Conventions	- Class names: `PascalCase` (e.g., `FeeReportService`) - Methods/variables: `camelCase`
Comments	- Use meaningful comments above classes and methods - Avoid redundant comments
Indentation	- 4 spaces per indentation level
Code Modularity	- Break down logic into small, reusable methods
Constants	- Use `final static` constants for fixed values
Error Handling	- Use try-catch blocks where necessary - Throw custom exceptions if needed
Layered Architecture	- Use layers: UI → Controller → Service → Repo

KAMRAN FIAZ:

1. General Coding Standards:

- Follow naming conventions:
- Variables/methods: camelCase (e.g., getPaymentHistory())
- Classes: PascalCase (e.g., PaymentHistoryController)
- Constants: UPPER_SNAKE_CASE
- Use meaningful names: Avoid temp, data, or xyz; instead use paymentList, studentId, etc.
- Keep functions short and single-responsibility
- Avoid hardcoding values; use config files or constants.

2. Backend (e.g., Java, Python, Node.js):

- Use layered architecture (Controller → Service → Repository)
- Example file structure (Java Spring Boot):

/controller/PaymentHistoryController.java

/service/PaymentHistoryService.java

/repository/PaymentRepository.java

/model/Payment.java

Handle exceptions with meaningful messages (e.g., PaymentHistoryNotFoundException)

- Use DTOs (Data Transfer Objects) to limit exposure of sensitive fields.
- REST API endpoint naming: /api/payments/history/{studentId}

3. Frontend (e.g., React, Angular, Vue):

- Component naming: PaymentHistoryComponent, PaymentRow
- Avoid inline styles, use CSS/SCSS files or CSS modules.
- Use state management (e.g., Redux, Vuex) where appropriate.
- Implement loading and error states
- Sanitize user input (especially filters and form data).

4. Database Design:

- Use normalized schema
- Table: payments
- payment_id (PK)
- student_id (FK)
- amount
- date
- payment_method
- status
- Use consistent naming, avoid abbreviations
- Use indexes on student_id, date for performance

5. Security Best Practices:

- Use parameterized queries to prevent SQL injection
- Mask sensitive info like card details in frontend/backend logs
- Authenticate and authorize access to payment history via tokens (e.g., JWT)

6. Documentation & Comments:

- Use doc comments (/** */ in Java, /// in Dart) to explain function purpose
- Keep inline comments concise and relevant
- Write README or API documentation using Swagger/OpenAPI

SHAYAN MUGHAL:

1. Naming Conventions

Element	Convention	Example
Project Name	PascalCase	InvoiceSystem
Package/Module	lowercase	invoice, services
Class Names	PascalCase	Invoice, InvoiceManager
Method Names	camelCase	submitInvoice(), validateInvoice()
Variable Names	camelCase	invoiceDate, clientId
Constants	UPPER_CASE	MAX_INVOICE_AMOUNT

---

2. Folder/Package Structure

```
bash
CopyEdit
invoicesystem/
├── models/              # Data classes (e.g., Invoice)
├── services/            # Business logic (e.g., InvoiceService)
├── repository/          # Data handling (e.g., InvoiceRepository)
├── ui/                  # Console/GUI interface (e.g., SubmitInvoiceForm)
└── utils/               # Helper methods (e.g., InvoiceValidator)
```

---

3. Code Formatting

- **Indentation**: 4 spaces (no tabs)
- **Line Length**: Maximum 100 characters
- **Braces**: Open on the same line (Java style)

```
java
CopyEdit
if (isInvoiceValid) {
    submitInvoice(invoice);
}
```

---

4. Error Handling

Use try-catch for all operations that may throw exceptions, especially during repository access.

```
java
CopyEdit
try {
    invoiceRepository.save(invoice);
} catch (DatabaseException e) {
```

```
    System.err.println("Error saving invoice: " + e.getMessage());
}
```

---

5. Validation and Security

- Validate all inputs: null checks, format checks (e.g., invoice date, amount).
- Sanitize input fields to avoid injection attacks.
- Keep class fields private; use getters/setters for access.

```
java
CopyEdit
public class Invoice {
    private String invoiceId;
    private double amount;

    public String getInvoiceId() {
        return invoiceId;
    }

    public void setInvoiceId(String invoiceId) {
        this.invoiceId = invoiceId;
    }
}
```

---

6. Testing Standards

- Write test cases for key methods: `validateInvoice()`, `submitInvoice()`.
- Use mock objects for testing repository operations.
