KHULNA UNIVERSITY

Project on Banking Management System

Course Title : **Advanced Programming Laboratory**

Course Code : **0714 02 CSE 2100**

Submitted To

Dr. Engr. Kazi Masudul Alam
Professor
CSE Discipline
Khulna University,
Khulna-9208

Submitted By

Name : **MD. Ishrak Dewan**
Student ID : **230212**
Year : **2nd**
Term : **1st**
Session : **2023-2024**
Discipline : **CSE**

Name : **Tamal Paul**
Student ID : **230213**
Year : **2nd**
Term : **1st**
Session : **2023-2024**
Discipline : **CSE**

➢ Introduction :

The **Banking Management System (BMS)** is a software solution designed to streamline and automate various banking operations, providing customers and staff with a seamless experience. This system adheres to the **Model-View-Controller (MVC)** architecture, ensuring modularity, scalability, and maintainability. The BMS supports account management, investments, loans, notifications, transactions, and personnel management, enabling comprehensive financial operations under a unified platform.

➢ Objectives :

1. **Efficiency in Banking Operations:** Automate and simplify tasks such as account management, loan processing, and investment tracking.
2. **Customer-Centric Approach:** Provide customers with tailored solutions for deposits, withdrawals, and loan or investment services.
3. **Scalable Design:** Create a system architecture that can evolve with expanding business needs and functionalities.
4. **Security and Reliability:** Ensure that customer data and financial transactions are secure and accurately processed.
5. **User-Friendly Interface:** Develop intuitive views for users to interact with different system functionalities effectively.

➢ Documentation :

This is a Banking Management System implemented using the **Model-View-Controller (MVC)** design pattern, which separates the business logic, user interface, and control flow. The system is organized into different modules representing various aspects of a banking application, including **accounts**, **investments**, **loans**, **transactions**, **notifications**, and **person management**. Here is a detailed description of the code:

1. Models

Each module has a model class that encapsulates the business logic related to that module. The models interact with underlying data, manage state, and perform necessary operations.

- **AccountModel**: Manages various types of accounts such as Savings, Fixed Deposit (FDR), Deposit Plan (DPS), and Current accounts. It includes methods for creating accounts, depositing, withdrawing, and retrieving account details.
- **InvestmentModel**: Handles investment operations, including adding new investments, retrieving investments for a customer, and removing an investment.

- **LoanModel**: Deals with loan operations such as adding a new loan, retrieving loan details by loan ID, displaying all loans, and removing a loan.
- **NotificationModel**: Responsible for sending notifications using a specific notification service and switching notification services when needed.
- **PersonModel**: Manages the details of both customers and employees. It allows for displaying their information and working with their personal details.
- **TransactionModel**: Manages the logic behind transactions, particularly deposits and withdrawals. It processes transaction execution (e.g., updating account balances and sending notifications).

2. Views

The view classes are responsible for displaying information to the user and getting user input. They interact with the user and provide necessary data to the controllers for processing.

- **AccountView**: Displays the menu for account operations, gets account creation details from the user, and allows deposits, withdrawals, and displays account details.
- **InvestmentView**: Displays investment-related options, collects investment details from the user, retrieves investment details, and displays investments.
- **LoanView**: Shows loan options, collects loan-related details from the user, retrieves loan details, and displays all loans.
- **NotificationView**: Displays options for notifications, collects details for sending notifications, and shows a confirmation after sending them.
- **PersonView**: Displays user options for managing people (both customers and employees), collects details for adding new people, and displays person details.
- **TransactionView**: Displays transaction menus, collects deposit and withdrawal details from the user, and shows a list of transactions.

3. Controllers

Each controller acts as an intermediary between the models and views. The controllers contain the logic for handling user input, processing that input by invoking the necessary methods on the models, and then updating the views accordingly.

- **AccountController**: Handles user input related to accounts, including creating new accounts, deposits, withdrawals, and displaying account details.
- **InvestmentController**: Manages the process of adding, removing, and displaying investments, as well as displaying investments by customers.

- **LoanController**: Handles loan-related operations such as adding loans, retrieving loan details, displaying all loans, and removing loans.
- **NotificationController:** Manages sending notifications and changing the notification service.
- **PersonController**: Manages customer and employee data, displaying personal details, adding new people, and displaying a list of people.
- **TransactionController**: Manages deposit and withdrawal transactions, processes transactions, and displays a list of transactions.

4. Main Class (Main.java)

Main Class: The Main class serves as the entry point to the application. It initializes the system by creating instances of models, views, and controllers for various entities (Account, Investment, Loan, etc.). It also handles the logic for navigating the system and displaying appropriate menus.

Key Methods in Main Class

- **displayMainMenu()**: Displays the main menu for the user to navigate the system.
- **displayWelcomeMessage()**: Greets the user and welcomes them to the banking system.
- **exitSystem()**: Handles the logic for exiting the system, such as saving the state or logging out.
- **resetSystemState()**: Resets the system to its default state.

5. System Overview

- **Account Management**:
  - Users can create different types of accounts, such as **Savings**, **FDR**, **DPS**, or **Current** accounts. The **AccountModel** contains logic for creating and managing these accounts, while the **AccountController** facilitates the user interaction for account creation and operations like deposit and withdrawal.
- **Investment Management**:
  - The **InvestmentModel** is used to add and manage investments, such as stocks or bonds .The **InvestmentController** handles adding investments and retrieving them by customer. It also allows for removing investments.

- **Loan Management**:
  - The **LoanModel** handles loan operations like adding, retrieving, and removing loans. It interacts with the **LoanManager** to manage the loan portfolio. The **LoanController** is responsible for taking user input for loan details and displaying information about loans.
- **Notification System**:
  - The **NotificationModel** allows the system to send notifications (e.g., alerts for account balance, loan payments) to users. The **NotificationController** handles the process of sending notifications and switching notification services dynamically (e.g., email, SMS).
- **Person Management**:
  - The **PersonModel** represents a **Person** (either a **Customer** or **Employee**) in the system. The **PersonController** is responsible for adding, displaying, and managing people, while the **PersonView** handles user input and displays person details.
- **Transaction Management**:
  - The **TransactionModel** processes financial transactions like deposits and withdrawals. The **TransactionController** handles user input related to transactions, processes the operations, and updates the system state accordingly.

6. System Flow

1. **Start**: The system initializes with a welcome message.
2. **User Interactions**: The user navigates through menus for creating accounts, managing investments, applying for loans, making transactions, etc.
3. **Controllers**: Each user action triggers a corresponding controller method, which interacts with the model (e.g., creating a new account or processing a transaction).
4. **View Updates**: Based on the controller logic, the view displays the updated information to the user.
5. **Exit**: When the user chooses to exit, the system will save the state (if necessary) and terminate.

This design provides a clear separation of concerns:

- **Model**: Handles the data and business rules.
- **View**: Manages user interaction and displays information.
- **Controller**: Coordinates the system flow and processes user inputs.

Summary

The Banking Management System:

- Implements core banking functionalities such as account operations, investment tracking, and loan management.
- Integrates with notification services for enhanced communication with customers and staff.
- Uses MVC architecture for modular and maintainable code, ensuring a robust development framework.
- Facilitates easy scalability to meet the evolving needs of banks and financial institutions.

The modular design allows future integration with additional services, such as online banking and advanced analytics, positioning the system for growth.

Conclusion :

In conclusion, this **Banking Management System** design follows the **Model-View-Controller (MVC)** pattern, ensuring a clear separation of concerns between data management, user interaction, and application logic. By organizing the system into distinct **Model**, **View**, and **Controller** components, it becomes easier to maintain and extend the system in the future.

- The **Model** layer encapsulates the core business logic and data management related to accounts, transactions, loans, investments, notifications, and user management.
- The **View** layer handles user interactions, displaying information and capturing user input to drive the system's operations.
- The **Controller** layer acts as an intermediary between the **Model** and **View**, processing user requests and coordinating actions to ensure the system functions correctly.

This approach not only improves the modularity and maintainability of the code but also provides flexibility for future enhancements, such as adding new account types, integrating third-party services, or improving the user interface.

However, while the structural foundation has been laid, the next steps involve implementing the missing business logic and user interface details. This includes methods for processing transactions, managing account balances, handling loan approvals, and generating notifications. Once completed, the system will provide a fully functional, user-friendly banking management solution that can be deployed and used in real-world applications.