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)** developed in Java aims to handle various aspects of banking, such as account management, transactions, customer details, loan management, investments, and notifications. This system follows Object-Oriented Programming (OOP) principles such as **Abstraction**, **Encapsulation**, **Inheritance**, and **Polymorphism** to ensure modularity and maintainability. Additionally, it adheres to **SOLID** principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion) for maintaining code quality, scalability, and flexibility.

The **Banking Management System (BMS)** allows users to perform essential banking operations like creating accounts, making deposits and withdrawals, applying for loans, managing investments, and receiving notifications for transactions. The goal is to create a system that is easy to maintain, secure, and adaptable to future changes.

Objectives :

- **Account Management :** To support various types of accounts (e.g., Checking, Savings, DPS, FDR, etc.) with functionalities like deposit, withdrawal, and balance retrieval.
- **Transaction Handling :** To facilitate deposit and withdrawal transactions, ensuring proper validation and security at each step.
- **Customer Information Management :** To manage customer details, including personal information, contact details, and customer-specific services such as loans and investments.
- **Loan and Investment Management :** To provide tools for loan calculations (including interest and repayment), as well as tracking investments made by customers.
- **Notification System :** To notify users about the status of their transactions and account activities, ensuring transparency.
- **Adherence to SOLID Principles :** The system is designed to follow the SOLID principles to improve maintainability, scalability, and flexibility. The system is modular, with each class performing a single responsibility (Single Responsibility Principle), and it is open for extension while closed for modification (Open/Closed Principle).
- **Security and Data Integrity:** Through proper validation, encapsulation, and abstraction, the system ensures secure and error-free banking transactions and data management.

Description :

The code is organized in a way that each major component or feature has its own dedicated class or interface. This ensures that the system remains maintainable and scalable as new features or banking products are added over time. Below, we explore the key components and design principles in greater detail :

1.Account Management :

The **Account** class serves as the base class for various types of accounts, including Checking, Savings, Current, DPS, and FDR accounts. It provides common properties, such as :

- **accountNumber :** A unique identifier for the account.
- **balance :** The balance of the account, which can be modified by deposit and withdrawal operations.
- **accountholder :** The name of the person who owns the account.

Each account type extends the Account class and implements its own specific logic for withdrawal and deposit operations. For example :

- **CheckingAccount :** Allows withdrawals up to a specified overdraft limit, ensuring that the account can go into negative balance but only up to a certain amount.
- **DPSAccount (Fixed Deposit) :** Restricts withdrawals before the account matures, ensuring that customers cannot access their funds until the deposit term has passed.
- **FDRAccount (Fixed Deposit Receipt)** : Similar to DPS, but with additional features, such as interest calculation upon maturity.

The deposit and withdraw methods are enforced across all account types to ensure valid transactions. If the withdrawal amount exceeds the available balance or violates the rules of a specific account (e.g., DPS or FDR), the system throws an exception. This helps to maintain data integrity and prevent errors in banking operations.

2. Transactions :

Transactions in the system are abstracted by the Transaction class, which includes essential properties like transactionId, accountNumber, and amount. The class is abstract, meaning it cannot be instantiated directly, and must be extended by specific transaction types such as DepositTransaction and WithdrawalTransaction**.**

- **DepositTransaction :** This class handles the deposit operation. Upon execution, the specified amount is added to the balance of the account. After completing the transaction, a notification is sent to the customer via a provided NotificationService (either SMS or Email).
- **WithdrawalTransaction :** This class handles the withdrawal operation. It ensures that the requested amount is within the available balance and follows the specific rules for

each account type. For instance, a Checking Account can support overdrafts, while a Fixed Deposit account cannot allow withdrawals before maturity.

Both transaction types follow the Single Responsibility Principle (SRP) by focusing only on executing the respective transaction and notifying the customer. This makes the code more modular and easy to maintain.

3. Loan Management :

The system includes the management of loans through the Loan class, which models a loan entity by storing information such as :

- **loanId :** A unique identifier for the loan.
- **amount :** The principal amount of the loan.
- **interestRate :** The rate at which interest accrues.
- **duration :** The loan duration, which determines how long the borrower has to repay the loan.

The LoanManager class manages all loans in the system, allowing for adding, removing, and retrieving loans. The Loan class implements the LoanType interface, which defines methods for calculating the total interest (calculateTotalInterest) and the total repayment (calculateTotalRepayment).

This structure follows the Dependency Inversion Principle (DIP) because the high-level module (LoanManager) depends on the abstraction (LoanType), not on the concrete Loan implementation. This makes it easy to extend the system with new types of loans in the future (e.g., mortgage loans, personal loans) without modifying the LoanManager class.

4. Investment Management :

The Investment class models a customer's investment, capturing key details such as :

- **investmentId :** A unique identifier for the investment.
- **Amount :** The amount of money invested.
- **investmentType :** The type of investment (e.g., stocks, bonds, fixed deposits).
- **Customer :** The customer who made the investment.

The InvestmentManager class is responsible for managing the investments, such as adding new investments, retrieving investments associated with a customer, and removing investments. The displayInvestments method prints out all investments, providing a comprehensive view of the customer's portfolio.

The system's design follows SRP by having dedicated classes for managing specific aspects of investment-related operations, ensuring the responsibility for investment management is clearly defined and not mixed with other parts of the banking system.

5. Customer and Employee Management :

- **Customer :** The Customer class holds personal information such as customerId, name, address, contact, and email. This class implements the Person interface, ensuring that both customers and employees share common methods for retrieving their id, name, contact, and email.
- **Employee :** Similar to the Customer class, the Employee class holds employee-specific information, such as employeeId, position, salary, and contact. The Employee class also implements the Person interface, which ensures that the bank can manage both customers and employees in a uniform way.

This structure allows for a clear separation between customers (who have accounts and investments) and employees (who handle the operations and business logic).

6. Notification Services :

To notify customers about various events (e.g., deposits, withdrawals), the system includes **NotificationService** interface implementations for different communication methods :

- **EmailNotificationService** : Sends notifications via email.
- **SMSNotificationService** : Sends notifications via SMS.

Both services implement the sendNotification method, adhering to the Dependency Inversion Principle (DIP), allowing the system to add new types of notification services (e.g., push notifications) without altering core transaction or account logic.

7. Design Principles :

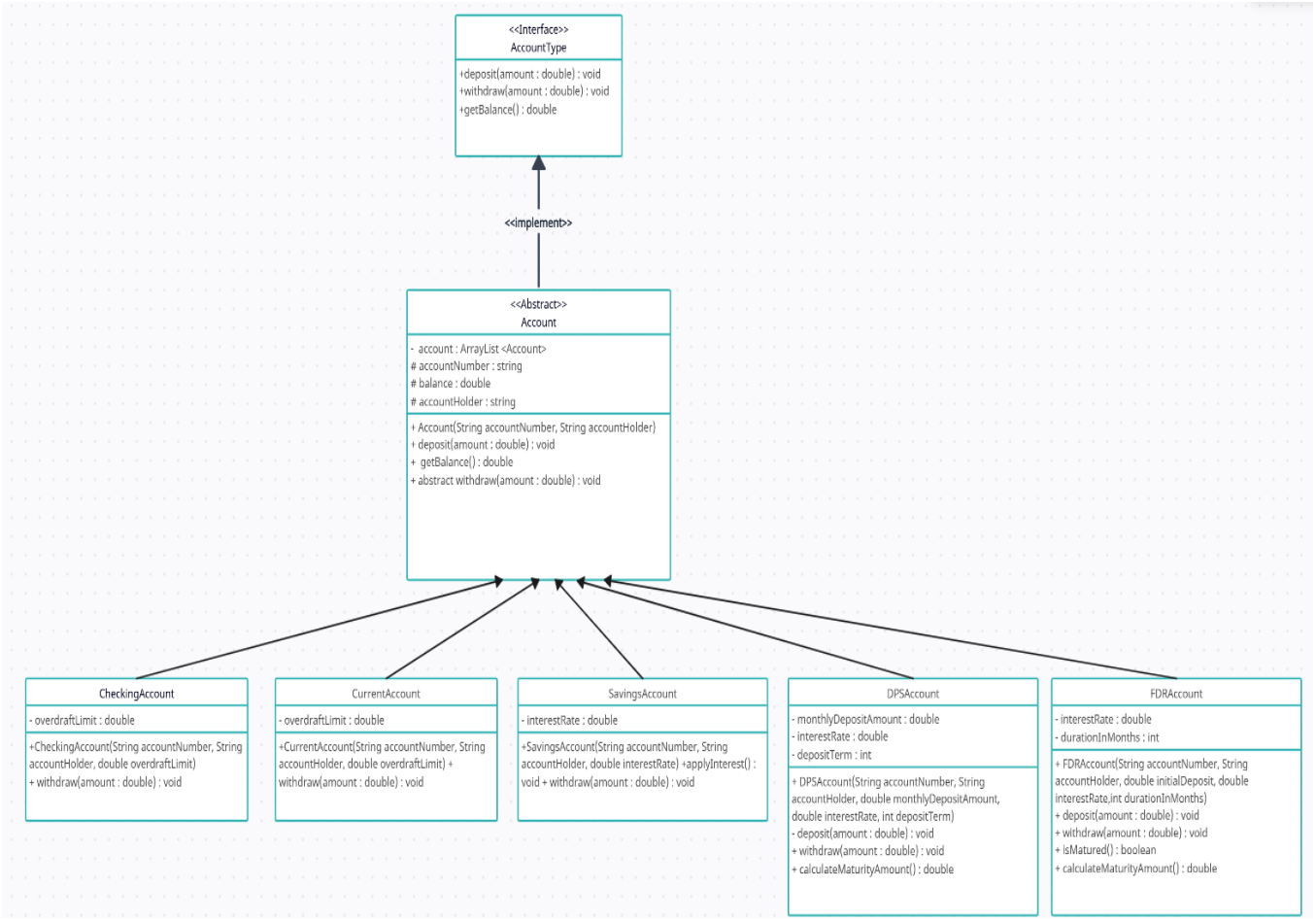
- **Single Responsibility Principle (SRP) :** The system is designed to ensure that each class has a single responsibility. For example, the LoanManager class only manages loans, while the InvestmentManager class handles investments. This leads to better organization and maintainability.
- **Open/Closed Principle (OCP) :** The system can be easily extended with new functionality without changing existing code. For example, new account types (e.g.,

BusinessAccount) or transaction types (e.g., TransferTransaction) can be added by simply creating new classes that extend base classes like Account or Transaction.

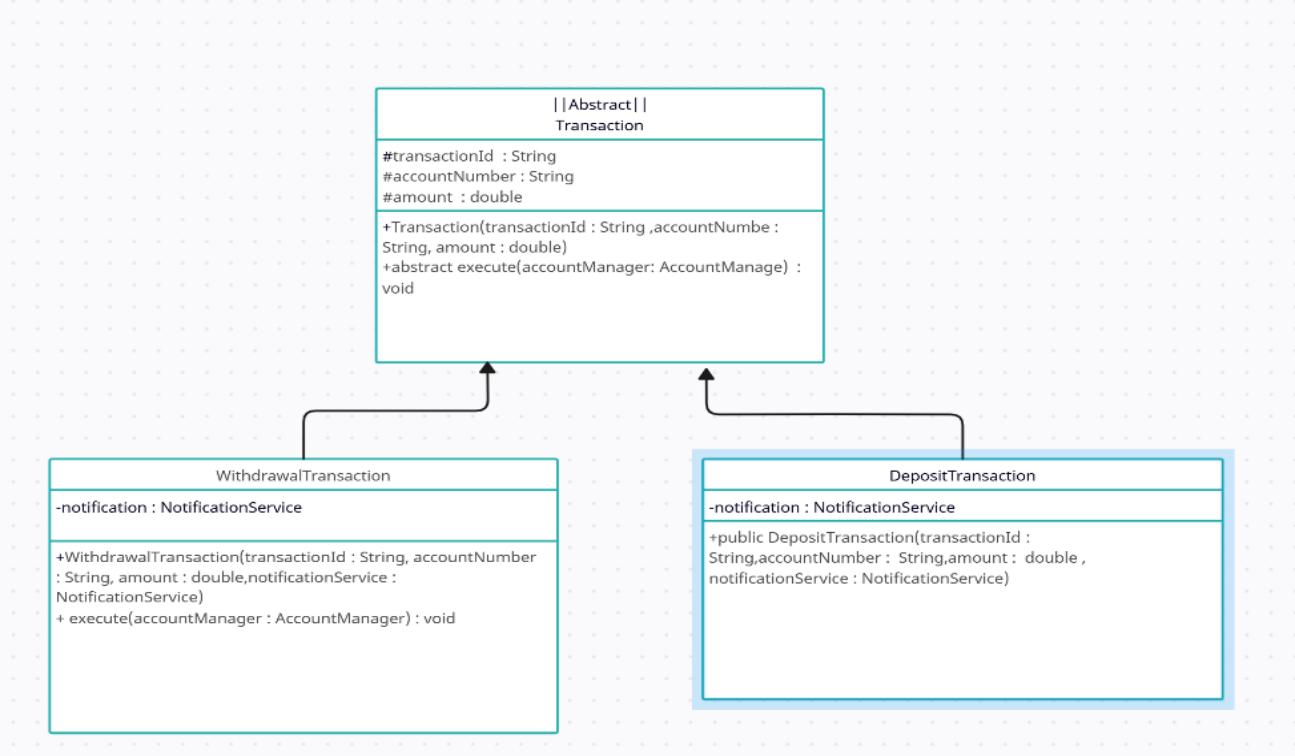
- **Liskov Substitution Principle (LSP) :** All derived classes (e.g., SavingsAccount, CheckingAccount) can be substituted in place of the Account class without altering the system's behavior. This ensures that polymorphism is applied correctly.
- **Interface Segregation Principle (ISP) :** The system uses small, specific interfaces like LoanType and NotificationService to ensure that clients only interact with the methods they need. This prevents the system from becoming bloated with unnecessary functionality.
- **Dependency Inversion Principle (DIP) :** High-level modules, like Transaction and LoanManager, depend on abstractions (e.g., LoanType, NotificationService) rather than concrete implementations. This ensures flexibility and allows easy future extensions.

⟹ **UML Diagrams:**

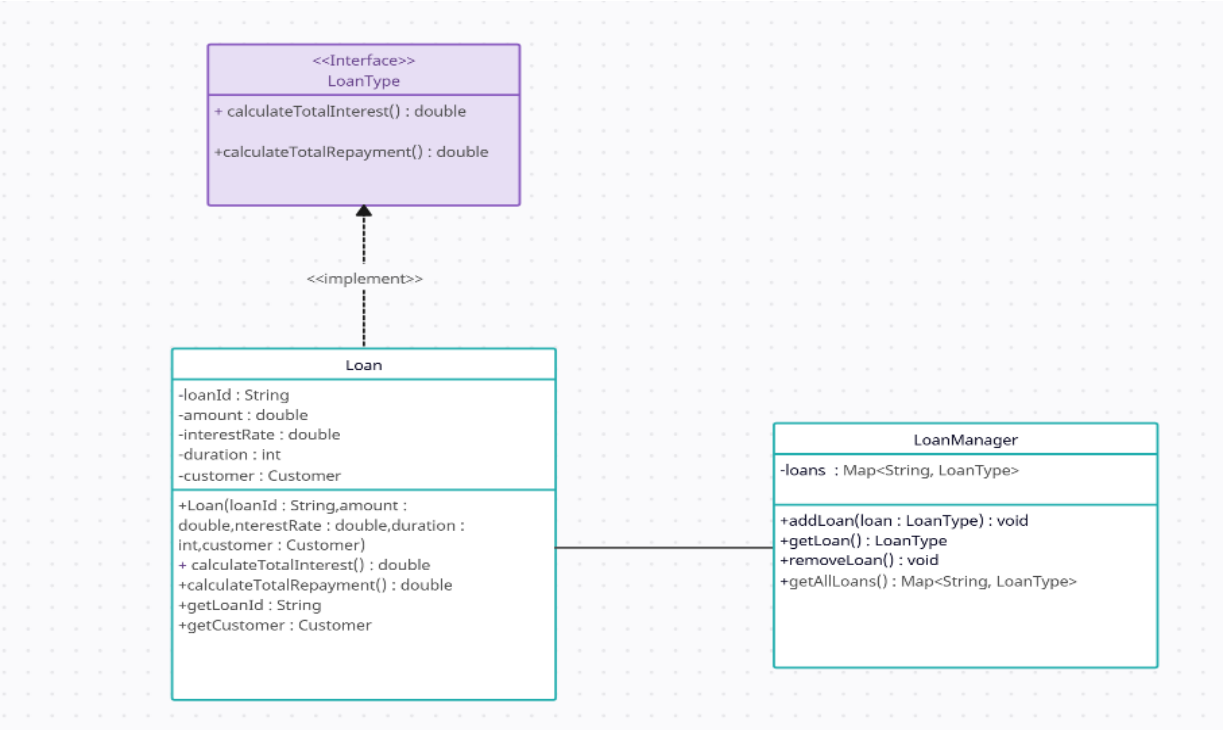
Account Management :



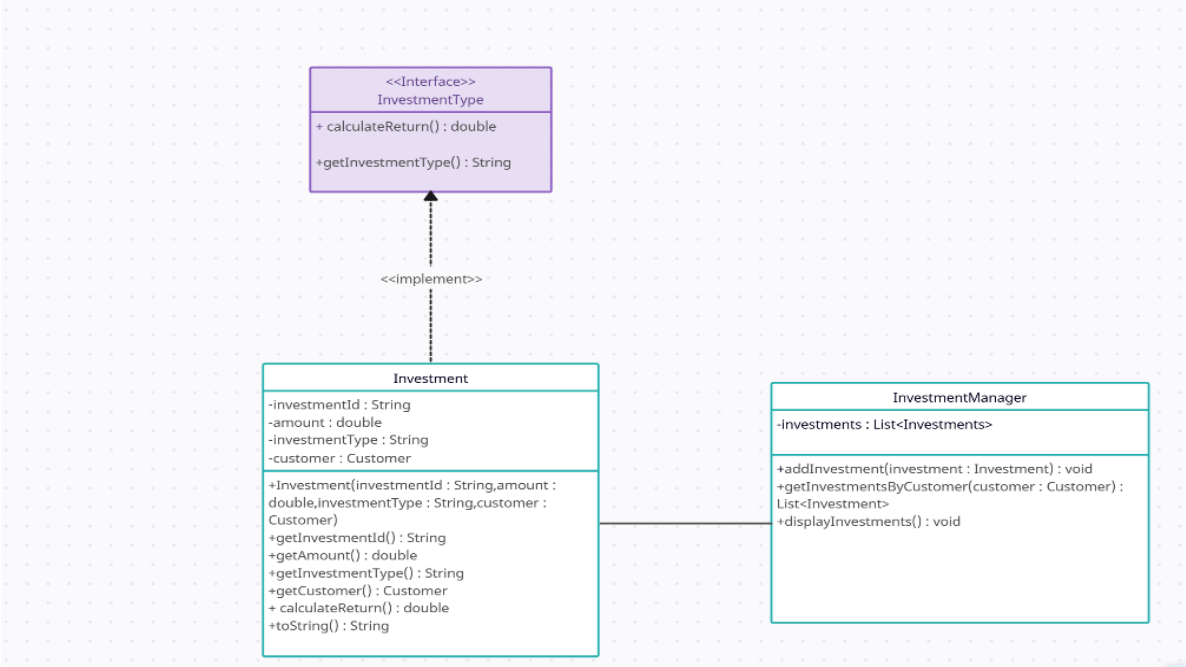
Transaction Management :



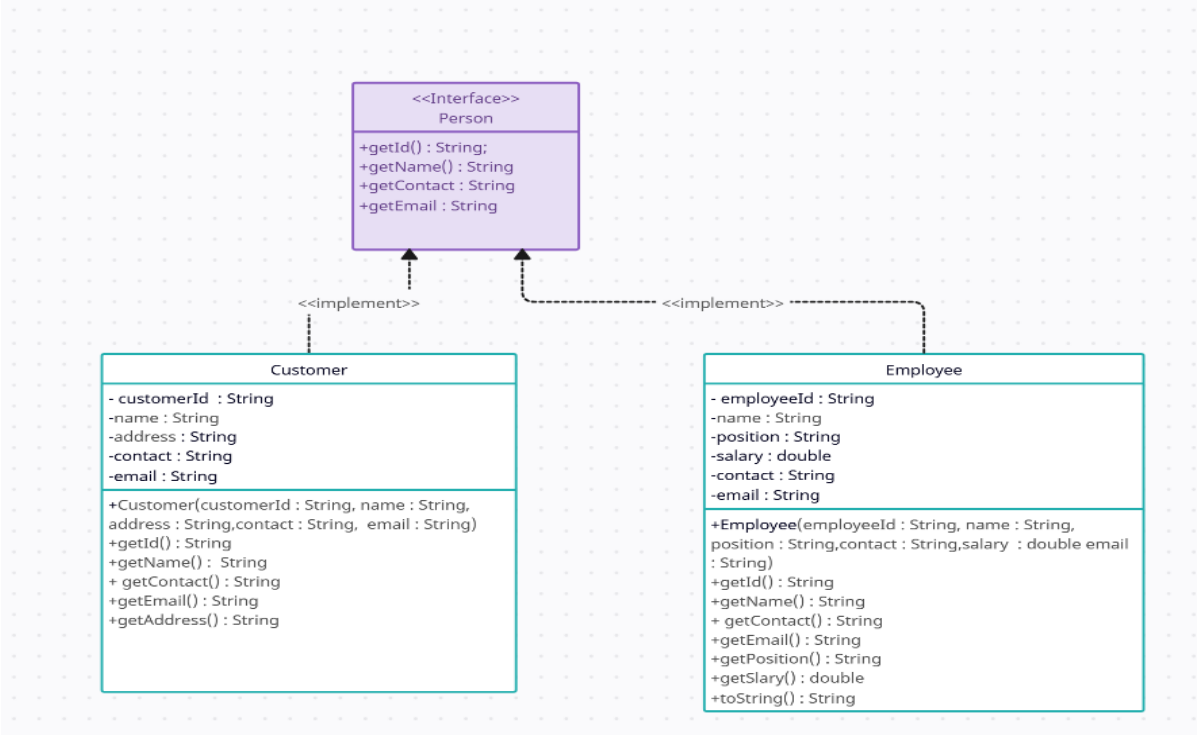
Loan Management :



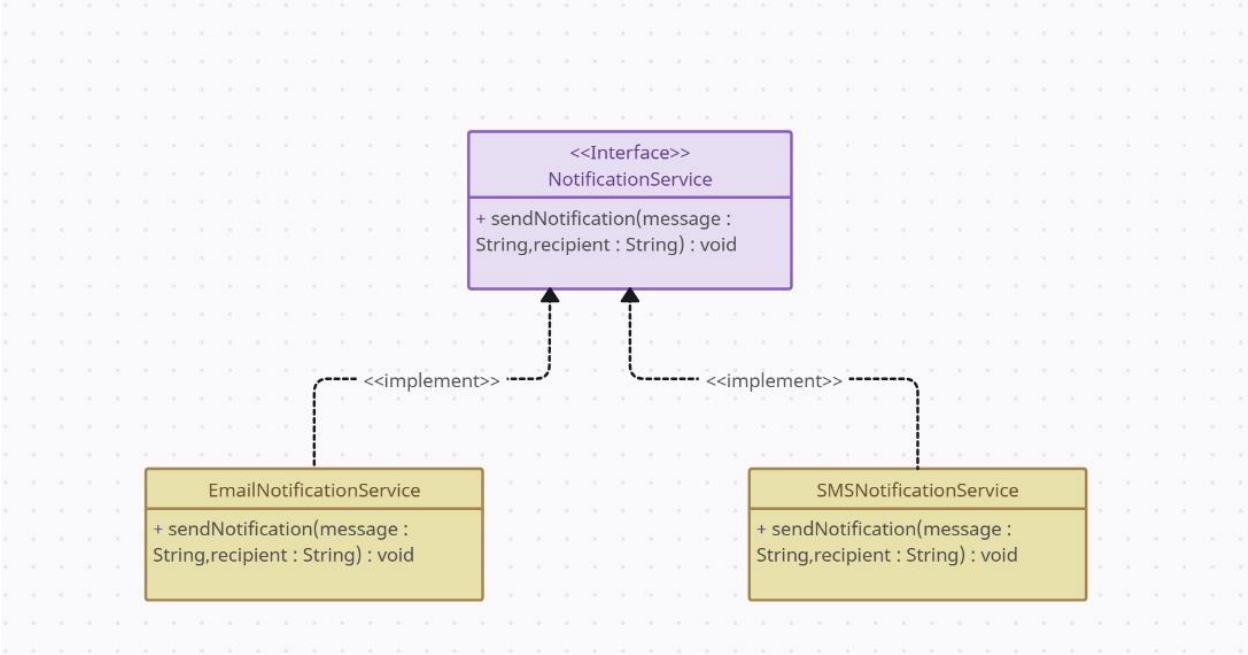
Investment Management :



Person :



Notification :



Conclusion :

The Banking Management System provides a robust solution for managing banking operations, incorporating essential features such as account management, transaction processing, customer services, and notifications. By **adhering** to the **SOLID principles**, the system is designed to be maintainable, extensible, and flexible, ensuring it can accommodate new features and changes in the future without breaking existing functionality.

This architecture supports secure financial operations and promotes scalability, which is critical in real-world banking systems where flexibility and robust performance are key. By using interfaces and abstract classes, the system is designed for ease of extension and modification, making it adaptable to future requirements, such as integrating new account types, transaction methods, or even regulatory changes.

Overall, the Banking Management System, through its design and implementation, **provides** a secure, efficient, and extensible foundation for managing banking operations and customer services, ensuring high-quality software and user satisfaction.