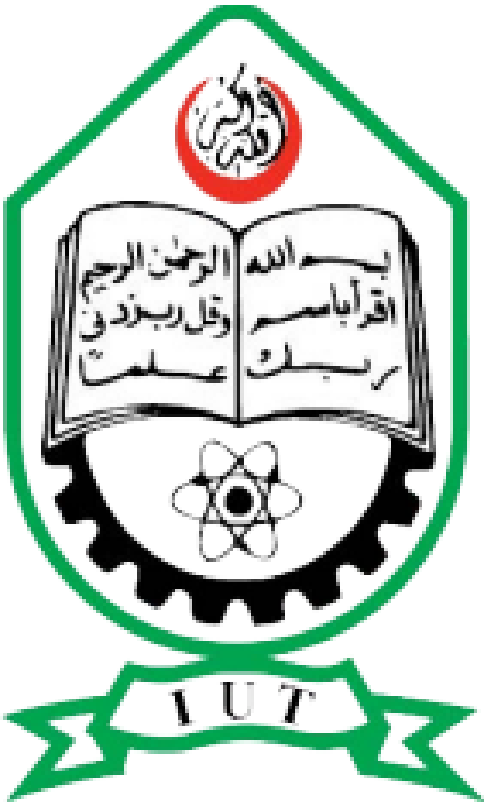
Assignment 1



CSE 4851: Design Pattern

Tasnimul Hasnat
190041113

February 14, 2024

Financial Institution Account Management System

In a financial institution, there's a need for an account management system that can handle various types of accounts such as savings accounts and checking accounts. Each account may have different interest calculation methods and additional features. Let's explore how each design pattern is applied in this scenario:

1. Strategy Pattern:

Scenario: Different interest calculation methods are required for different types of accounts. For instance, savings accounts might have a simple interest calculation method, while checking accounts might have a compound interest calculation method.

Application: The Strategy pattern is applied by defining an interface `InterestCalculationStrategy` that declares a method `calculate_interest()`. Concrete implementations such as `SimpleInterestStrategy` and `CompoundInterestStrategy` are then created, each implementing the `calculate_interest()` method differently. This allows the system to encapsulate these algorithms independently of the clients that use them.

2. Factory Pattern:

Scenario: The system needs to create different types of accounts without exposing the instantiation logic directly to the client code. Each type of account may have different initialization procedures or dependencies.

Application: The Factory pattern is used to provide an interface `AccountFactory` with a method `create_account()`. Concrete factory implementations like `SavingsAccountFactory` and `CheckingAccountFactory` are responsible for creating instances of specific account types, such as savings accounts or checking accounts. This decouples the client code from the actual object creation process and allows for easier extension when new types of accounts are introduced.

3. Decorator Pattern:

Scenario: Additional features or bonuses need to be added to accounts dynamically without modifying their structure. For instance, a bonus might be applied to a savings account, or an extra fee might be added to a checking account.

Application: The Decorator pattern is employed by defining a base class `Account` representing the core functionality of an account. Concrete account implementations like `SavingsAccount` and `CheckingAccount` extend this base class. Additionally, decorators such as `BonusDecorator` and `InterestRateDecorator` are created to add extra functionalities dynamically to account objects. This allows for flexible composition of behaviors at runtime without altering the existing codebase.

Taking User Choice

To take user choices, I implemented a simple console-based input mechanism where users can select the type of account they want to set up, the interest calculation method, and any additional customization options.

Implementation

Interface for different types of accounts

```
interface Account {
    void deposit(double amount);
    void withdraw(double amount);
    double getBalance();
}
```

Concrete implementation of a savings account

```
class SavingsAccount implements Account {
    private double balance;

    @Override
    public void deposit(double amount) {
        balance += amount;
    }

    @Override
    public void withdraw(double amount) {
        balance -= amount;
    }

    @Override
    public double getBalance() {
        return balance;
    }

    public void calculateInterest() {
        System.out.println("Interest calculated for savings account.");
    }
}
```

Concrete implementation of a checking account

```
class CheckingAccount implements Account {
    private double balance;

    @Override
    public void deposit(double amount) {
        balance += amount;
    }

    @Override
    public void withdraw(double amount) {
        balance -= amount;
    }

    @Override
    public double getBalance() {
        return balance;
    }

    public void processChecks() {
         System.out.println("Checks processed for checking account.");
    }
}
```

Interface for different interaction channels

```
interface BankingChannel {
    void login();
    void viewBalance();
    void transferFunds();
}
```

Concrete implementation of online banking

```
class OnlineBanking implements BankingChannel {
    @Override
    public void login() {
        System.out.println("Logged into online banking.");
    }

    @Override
    public void viewBalance() {
        System.out.println("Viewing balance in online banking.");
    }
```

```
    @Override
    public void transferFunds() {
        System.out.println("Transferring funds in online banking.");
    }

    public void payBills() {
        System.out.println("Paying bills in online banking.");
    }
}
```

Concrete implementation of mobile banking

```
class MobileBanking implements BankingChannel {
    @Override
    public void login() {
        System.out.println("Logged into mobile banking.");
    }

    @Override
    public void viewBalance() {
        /System.out.println("Viewing balance in mobile banking.");
    }

    @Override
    public void transferFunds() {
        System.out.println("Transferring funds in mobile banking.");
    }

    public void depositCheck() {
        System.out.println("Depositing a check in mobile banking.")
    }
}
```

Concrete implementation of automated phone system

```
class AutomatedPhoneSystem implements BankingChannel {
    @Override
    public void login() {
       System.out.println("Authenticated through automated phone system.");
    }

    @Override
    public void viewBalance() {
      System.out.println("Viewing balance through automated phone system.");
    }
```

```
    @Override
    public void transferFunds() {
           System.out.println("Transferring funds through automated phone
system.");
    }
}
```

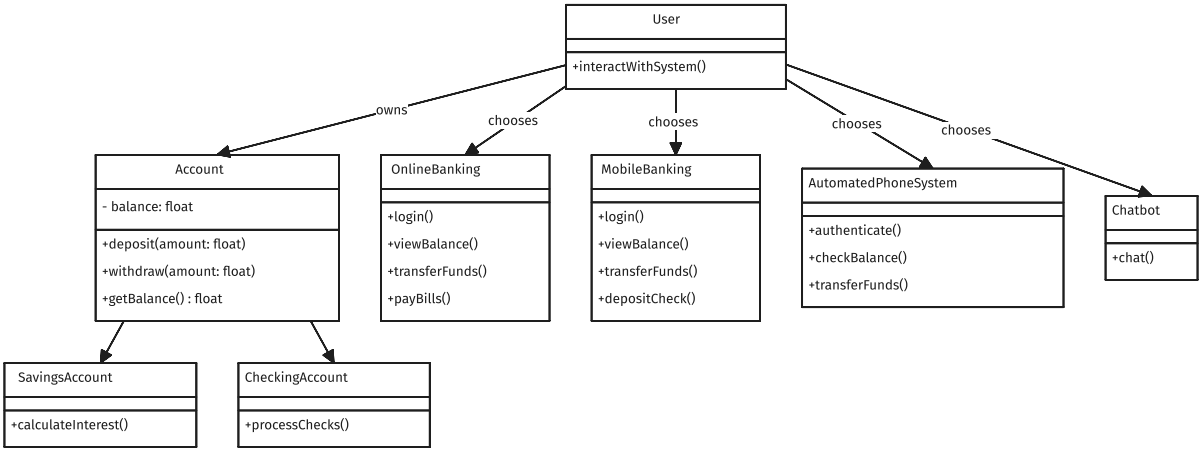
Concrete implementation of a chatbot

```
class Chatbot implements BankingChannel {
    @Override
    public void login() {
        System.out.println("Chat session started with the chatbot.");
    }

    @Override
    public void viewBalance() {
        System.out.println("Viewing balance through the chatbot.");
    }

    @Override
    public void transferFunds() {
        System.out.println("Transferring funds through the chatbot.");
    }
}
```

UML Diagram



In this diagram:

- `User` interacts with the system through various channels such as `OnlineBanking`, `MobileBanking`, `AutomatedPhoneSystem`, or `Chatbot`.
- `Account` represents the core functionality of bank accounts, with `SavingsAccount` and `CheckingAccount` as specific types of accounts.
- `OnlineBanking`, `MobileBanking`, `AutomatedPhoneSystem`, and `Chatbot` are different interaction channels available to users for managing their accounts.

Conclusion

The implemented financial institution account management system showcases a well-structured Java application, leveraging interfaces and classes to model diverse account types and interaction channels. Through interfaces like Account and BankingChannel, the system achieves encapsulation and abstraction, hiding implementation details while exposing essential functionalities. The flexibility and modularity of the design enable seamless extension and customization, allowing for the addition of new account types or interaction channels without disrupting existing code. With its emphasis on scalability, maintainability, and adherence to software design principles, the system serves as a robust foundation for building sophisticated banking applications capable of meeting the evolving needs of users in a dynamic financial landscape.