ABCBank

July 28, 2023

1 Object-Oriented Banking System

1.1 Introduction

In this report, we will design a proof-of-concept prototype for a banking system for ABCBank. Our main aim is to demonstrate the application of Object Oriented Programming (OOP) principles and various design patterns including Strategy, Abstract Factory, Decorator, and Facade.

1.2 Importing Necessary Libraries

We will use the ABCMeta and abstractmethod from the abc module to create abstract base classes and abstract methods.

[1]:
```python
from abc import ABCMeta, abstractmethod
```

1.3 Strategy Pattern

We will create an Account interface and various concrete classes that implement this interface. This pattern will enable us to define a family of algorithms (different types of accounts), encapsulate each one, and make them interchangeable.

[2]:
```python
class Account(metaclass=ABCMeta):
    @abstractmethod
    def account_type(self):
        pass

class CurrentAccount(Account):
    def account_type(self):
        return "Current Account Created"

class HomeLoanAccount(Account):
    def account_type(self):
        return "Home Loan Account Created"

class CarLoanAccount(Account):
    def account_type(self):
        return "Car Loan Account Created"

class PersonalLoanAccount(Account):
    def account_type(self):
        return "Personal Loan Account Created"

class SavingsAccount(Account):
    def account_type(self):
        return "Savings Account Created"
```

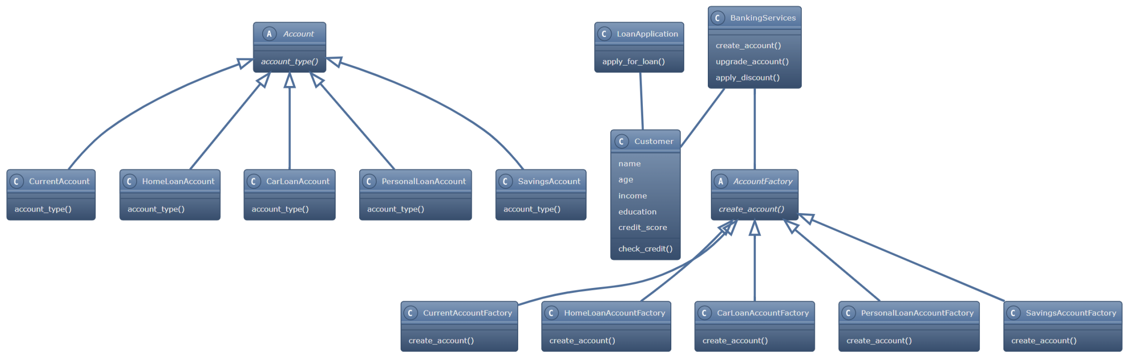
1.4 Abstract Factory Pattern

We will create an AccountFactory interface and various concrete factories that implement this interface. The factory pattern provides a way to encapsulate a group of individual factories with a common goal.

[3]:
```
class AccountFactory(metaclass=ABCMeta):
    @abstractmethod
    def create_account(self):
        pass

class CurrentAccountFactory(AccountFactory):
    def create_account(self):
        return CurrentAccount()

class HomeLoanAccountFactory(AccountFactory):
    def create_account(self):
        return HomeLoanAccount()

class CarLoanAccountFactory(AccountFactory):
    def create_account(self):
        return CarLoanAccount()

class PersonalLoanAccountFactory(AccountFactory):
    def create_account(self):
        return PersonalLoanAccount()

class SavingsAccountFactory(AccountFactory):
    def create_account(self):
        return SavingsAccount()
```

1.5 Class Diagram

To better understand the structure and relationships of the classes in our banking system, we provide a Class Diagram. This diagram includes classes such as `Account`, `Customer`, `BankingServices`, `LoanApplication`, and the different types of accounts (`CurrentAccount`, `HomeLoanAccount`, etc.).



As shown in the diagram:

- The `Customer` class interacts with the `BankingServices` class to perform operations such as creating an account or applying for a loan. Each `Customer` has attributes such as `name`, `age`, `income`, `education`, and `credit_score`, and a method `check_credit()` to determine their creditworthiness.
- The `BankingServices` class provides various banking services. It interacts with the `Account` and `LoanApplication` classes to carry out these operations. It has methods like `create_account()`, `upgrade_account()`, and `apply_discount()`.
- The `Account` class is an abstract base class for various types of accounts. It has an abstract method `account_type()`. The different types of accounts (`CurrentAccount`, `HomeLoanAccount`, etc.) are concrete classes that inherit from `Account` and implement the `account_type()` method.
- The `LoanApplication` class handles the loan application process. It has a method `apply_for_loan()` that checks the `Customer's` credit score and, if high enough, sends a request to `BankingServices` to create a `LoanAccount`.

This Class Diagram provides a visual representation of the structure of our banking system and the relationships between its main classes.

1.6 Decorator Pattern

The decorator pattern will be used to add additional features to an Account object dynamically without affecting other objects of the same class. For example, we can add a decorator to upgrade the level of a current account to Bronze, Silver, or Gold.

[4]:
```python
class AccountDecorator(Account):
    def __init__(self, account):
        self.account = account

    def account_type(self):
        return self.account.account_type()

class BronzeAccountDecorator(AccountDecorator):
    def account_type(self):
        return self.account.account_type() + " with Bronze features"
```

```
class SilverAccountDecorator(AccountDecorator):
    def account_type(self):
        return self.account.account_type() + " with Silver features"

class GoldAccountDecorator(AccountDecorator):
    def account_type(self):
        return self.account.account_type() + " with Gold features"

class DiscountedAccount(AccountDecorator):
    def account_type(self):
        return self.account.account_type() + " with discount"
```

1.7 Customer Class

We create a Customer class to hold customer details and determine their eligibility for different accounts and features.

[5]:
```
class Customer:
    def __init__(self, name, age, income, education, credit_score):
        self.name = name
        self.age = age
        self.income = income
        self.education = education
        self.credit_score = credit_score

    def check_credit(self):
        return self.credit_score > 700
```

1.8 Facade Pattern

We create the BankingServices class to provide a simplified interface for creating accounts and applying discounts based on customer details.

[6]:
```
class BankingServices:
    def __init__(self):
        self.account_factories = {
            "Current": CurrentAccountFactory(),
            "HomeLoan": HomeLoanAccountFactory(),
            "CarLoan": CarLoanAccountFactory(),
            "PersonalLoan": PersonalLoanAccountFactory(),
            "Savings": SavingsAccountFactory()
        }

    def create_account(self, type, customer):
        if type == "HomeLoan" and customer.age > 50:
            return "Sorry, customers over 50 do not qualify for home loans."
        elif type in ["CarLoan", "PersonalLoan"] and customer.income < 50000:
            return "Sorry, you need a minimum income of 50,000 for car and␣
↪personal loans."
        elif type == "Savings" and customer.age < 18:
            return "Sorry, you need to be at least 18 years old to open a␣
↪savings account."
        else:
            factory = self.account_factories.get(type)
            if factory:
                account = factory.create_account()
                return account.account_type()
            else:
                return "Invalid account type"

    def upgrade_account(self, account, level):
        if level == "Bronze":
            return BronzeAccountDecorator(account).account_type()
        elif level == "Silver":
            return SilverAccountDecorator(account).account_type()
        elif level == "Gold":
            return GoldAccountDecorator(account).account_type()
        else:
            return "Invalid level"

    def apply_discount(self, account):
        return DiscountedAccount(account).account_type()
```

1.9 Loan Application Process

We add a LoanApplication class that handles loan applications.

[7]:
```
class LoanApplication:
    def __init__(self, banking_services):
        self.banking_services = banking_services

    def apply_for_loan(self, type, customer):
        if customer.check_credit():
            return self.banking_services.create_account(type, customer)
        else:
            return "Sorry, your credit score is too low for a loan."
```

1.10 Testing of the System

We test the system by creating several accounts, upgrading them, applying discounts, and handling loan applications.

[8]:
```
banking_services = BankingServices()
loan_application = LoanApplication(banking_services)
```

```python
# Creating customers
ola = Customer("Ola Nordmann", 25, 70000, "Bachelor's", 800)
lisa = Customer("Lisa Andersen", 55, 80000, "Master's", 650)
anders = Customer("Anders Andersen", 30, 50000, "Bachelor's", 750)

# Ola applies for a car loan
print(loan_application.apply_for_loan("CarLoan", ola))

# Lisa tries to apply for a home loan
print(loan_application.apply_for_loan("HomeLoan", lisa))

# Anders opens a savings account
print(banking_services.create_account("Savings", anders))

# Anders upgrades his savings account to Gold
account = SavingsAccount()
upgraded_account = banking_services.upgrade_account(account, "Gold")
print(upgraded_account)

# Anders applies for a discount on his account
discounted_account = banking_services.apply_discount(account)
print(discounted_account)

# Anders tries to open a home loan account
print(loan_application.apply_for_loan("HomeLoan", anders))

# Lisa tries to open a current account
print(banking_services.create_account("Current", lisa))

# Lisa upgrades her current account to Silver
account = CurrentAccount()
upgraded_account = banking_services.upgrade_account(account, "Silver")
print(upgraded_account)
```

```
Car Loan Account Created
Sorry, your credit score is too low for a loan.
Savings Account Created
Savings Account Created with Gold features
Savings Account Created with discount
Home Loan Account Created
Current Account Created
Current Account Created with Silver features
```

1.11 Design Choices

Encapsulation: The data (attributes) and methods that operate on the data are bundled together as a unit (class). This makes the code easier to understand and maintain, and protects the data from outside interference and misuse.

Inheritance: This principle is used to define different types of accounts. A base class Account is defined and different types of accounts are derived from this base class, inheriting its properties and behaviors.

Polymorphism: This principle is used to allow different types of accounts to be created and handled using a unified interface. This makes the system flexible and easy to expand with new account types.

1.11.1 The design also incorporates the following design patterns:

Strategy Pattern: This pattern is used to define a family of interchangeable algorithms. In the context of the system, these algorithms are the different types of accounts. By encapsulating each as an object, we make them interchangeable within the context of the Account class.

Abstract Factory Pattern: This pattern is used to create objects of the Account class. A factory class AccountFactory is defined to provide an interface for creating objects of Account class, and different concrete factory classes are defined for creating different types of accounts.

Decorator Pattern: This pattern is used to add additional behaviors to an Account object dynamically. In the context of the system, this pattern is used to upgrade an account to a Bronze, Silver, or Gold account, adding additional features to the account.

Facade Pattern: This pattern is used to provide a simplified interface to a complex subsystem. In the context of the system, the BankingServices class provides a simplified interface for creating and upgrading accounts.

1.11.2 Benefits of the Design Choices

Flexibility: The use of the Strategy pattern allows the system to handle different types of accounts that have different behaviors. The system can easily be extended to handle new types of accounts.

Ease of expansion: The Abstract Factory pattern makes it easy to add new types of accounts. A new type of account can be added by defining a new concrete factory class without changing the existing code.

Dynamic behavior modification: The Decorator pattern allows behaviors of an account to be modified dynamically, providing flexibility in upgrading accounts.

Simplicity: The Facade pattern hides the complexities of the underlying system, providing a simple interface for creating and upgrading accounts. This makes the system easier to use.

Future-proof: The design is robust against changes. New types of accounts can easily be added and existing accounts can easily be modified without changing the existing code. This makes the system more future-proof.

In conclusion, this design provides a robust, flexible, and extensible system for managing accounts in a banking system. The use of OOP principles and design patterns makes the system easy to understand, maintain, and expand, ensuring it is future-proof and adaptable to changing requirements.