SSAD Report - 3rd Assignment

Description and Reasoning for the used Design patterns:

The provided code implements a simple banking system with various functionalities such as creating accounts, depositing, withdrawing, transferring money between accounts, viewing account details, activating, and deactivating accounts. The code employs several design patterns:

Prototype Pattern:(Creational Design Pattern)

The clone() method in the Account class and its concrete subclasses (SavingsAccount, CheckingAccount, BusinessAccount) facilitates the creation of new account instances by cloning existing prototypes.
This design pattern was chosen to facilitate the creation of new account instances with varying types without tightly coupling the client code to specific account classes. It also enables the bank to create new accounts dynamically based on existing prototypes, which promotes code reuse and scalability.

Facade Pattern:(Structural Design Pattern)

The Facade pattern is Implemented through the BankFacade class, which provides a simplified interface to access the complex banking system. It encapsulates the complexities of the banking operations and provides a unified interface for clients to interact with the bank functionalities
This pattern was used because it simplifies the usage of the banking system by providing a high-level interface, hiding the complexities of the underlying subsystems (account creation, and transaction management).

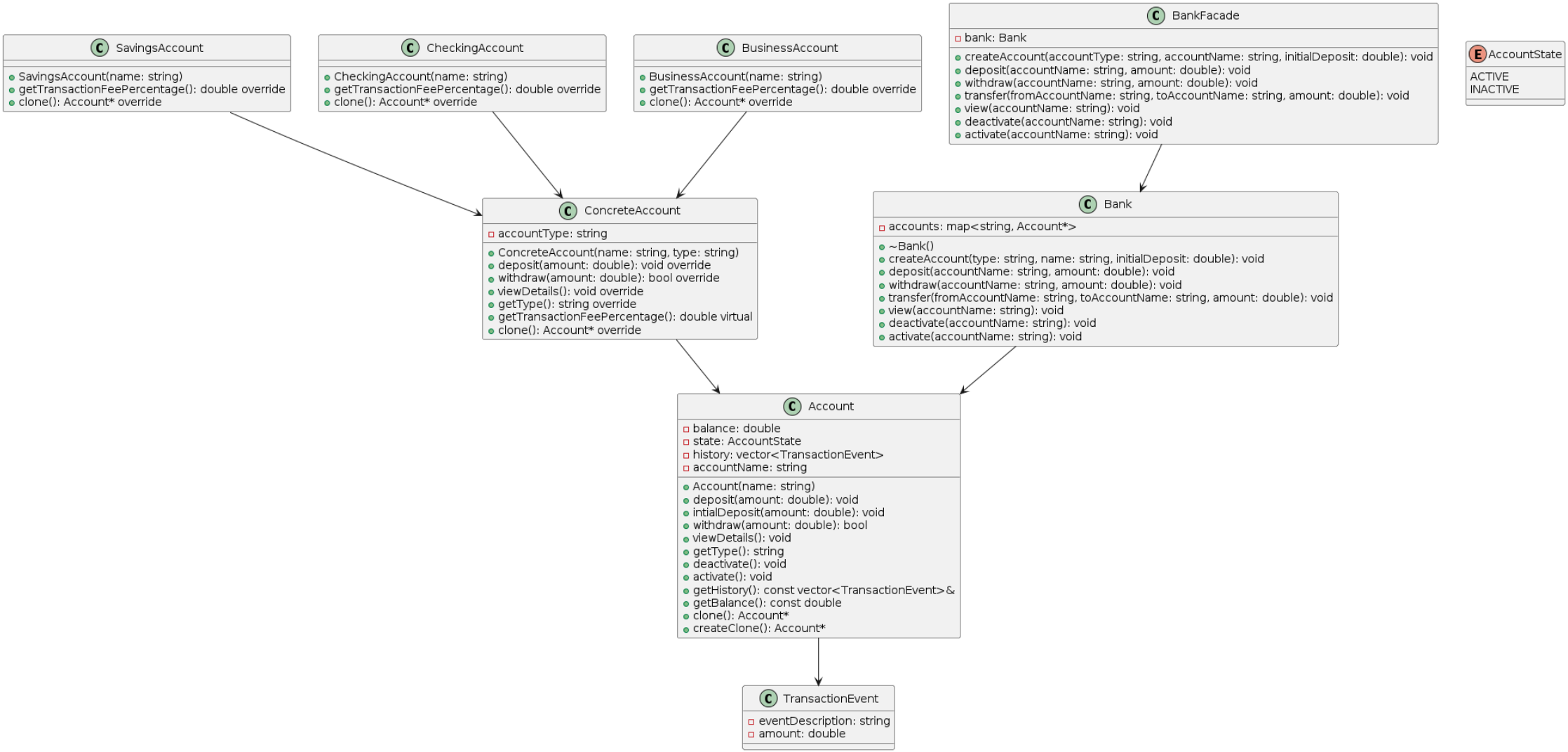
Strategy Pattern:(Behavioral Design Pattern)

The Strategy Pattern is Implemented in the ConcreteAccount subclasses (SavingsAccount, CheckingAccount, BusinessAccount), using the getTransactionFeePercentage() method.
Allows the calculation of transaction fees based on different strategies (transaction fee percentages) for each type of account.
It was used because it accommodates varying transaction fee policies for different account types in a flexible and modular form. This approach facilitates easy extension and modification of transaction fee strategies, enhancing the code's adaptability to evolving requirements and diverse account needs.

State Pattern:(Behavioral Design Pattern)

The State pattern is implemented through the AccountState enum and its usage in the Account class, the state pattern facilitates dynamic behavior changes based on the account's current state (ACTIVE or INACTIVE). With the activate() and deactivate() methods transitioning the account between states, this pattern eliminates the need for cluttering conditional statements in the code. By encapsulating state-specific behavior within state classes (AccountState), it simplifies the addition or modification of behaviors associated with different states, promoting code encapsulation and maintainability.

The State Pattern is used because it manages account behavior based on the activation state, reducing clutter and maintaining the code. It encapsulates state-specific behavior in separate classes, enhancing readability and facilitating new or modified states without significantly impacting the existing codebase.


SavingsAccount
+ SavingsAccount(name: string)
+ getTransactionFeePercentage(): double override
+ clone(): Account* override
CheckingAccount
+ CheckingAccount(name: string)
+ getTransactionFeePercentage(): double override
+ clone(): Account* override
BusinessAccount
+ BusinessAccount(name: string)
+ getTransactionFeePercentage(): double override
+ clone(): Account* override
BankFacade
- bank: Bank
+ createAccount(accountType: string, accountName: string, initialDeposit: double): void
+ deposit(accountName: string, amount: double): void
+ withdraw(accountName: string, amount: double): void
+ transfer(fromAccountName: string, toAccountName: string, amount: double): void
+ view(accountName: string): void
+ deactivate(accountName: string): void
+ activate(accountName: string): void
AccountState
ACTIVE
INACTIVE
ConcreteAccount
- accountType: string
+ ConcreteAccount(name: string, type: string)
+ deposit(amount: double): void override
+ withdraw(amount: double): bool override
+ viewDetails(): void override
+ getType(): string override
+ getTransactionFeePercentage(): double virtual
+ clone(): Account* override
Bank
- accounts: map<string, Account*>
+ ~Bank()
+ createAccount(type: string, name: string, initialDeposit: double): void
+ deposit(accountName: string, amount: double): void
+ withdraw(accountName: string, amount: double): void
+ transfer(fromAccountName: string, toAccountName: string, amount: double): void
+ view(accountName: string): void
+ deactivate(accountName: string): void
+ activate(accountName: string): void
Account
- balance: double
- state: AccountState
- history: vector<TransactionEvent>
- accountName: string
+ Account(name: string)
+ deposit(amount: double): void
+ intialDeposit(amount: double): void
+ withdraw(amount: double): bool
+ viewDetails(): void
+ getType(): string
+ deactivate(): void
+ activate(): void
+ getHistory(): const vector<TransactionEvent>&
+ getBalance(): const double
+ clone(): Account*
+ createClone(): Account*
TransactionEvent
- eventDescription: string
- amount: double