SSAD Assignment 3

PDF report

Gleb Popov, B23-ISE-01

Design patterns

Creational — Singleton

**Singleton** is used to ensure that a class has only one instance, and provides a global access point to that instance. In the context of the banking system, I wrote the Database class, which I use to store accounts and gain access to them, and the BankFacade class, which I use to manage the banking system. The structure designed through **Singleton** ensures that only one instances of the Database and BankFacade will be created.

class Database {

private:

static Database\* instance; // pointer to a single instance

map<string, Account> accounts; // map for storing accounts

Database() {} // private constructor to prohibit direct instantiation

public:

// a function for accessing a single instance

static Database\* getInstance() {

if (!instance) {

instance = new Database();

}

return instance;

}

// other functions

};

class BankFacade {

private:

static BankFacade\* instance; // pointer to a single instance

Database\* db = Database::getInstance();; // database (also singleton)

BankFacade() {} // private constructor to prohibit direct instantiation

public:

// a function for accessing a single instance

static BankFacade\* getInstance() {

if (!instance) {

instance = new BankFacade();

}

return instance;

}

// other functions

};

Structural — Façade

**Façade** provides a simplified interface to a complex class system, library, or framework. In the context of the banking system, I used **Façade** to create a simplified interface for performing banking operations (creating accounts, depositing, withdrawing, transferring, viewing, activating, deactivating).

class BankFacade {

private:

static BankFacade\* instance; // pointer to a single instance

Database\* db = Database::getInstance();; // database (also singleton)

BankFacade() {} // private constructor to prohibit direct instantiation

public:

// a function for accessing a single instance

static BankFacade\* getInstance() {

if (!instance) {

instance = new BankFacade();

}

return instance;

}

// creates a new account

void createAccount(const string& type, const string& name, const float& dep) {...}

// triggers crediting the deposit or returns nullptr if the account is not found

void deposit(const string& name, const float& dep) {...}

// triggers a withdrawal or returns nullptr if the account is not found

void withdraw(const string& name, const float& dep) {...}

// triggers a transfer or returns nullptr if at least one account is not found

void transfer(const string& from, const string& to, const float& dep) {...}

// triggers a view function or returns nullptr if the account is not found

void view(const string& name) const {...}

// triggers a deactivate function or returns nullptr if the account is not found

void deactivate(const string& name) {...}

// triggers an activate function or returns nullptr if the account is not found

void activate(const string& name) {...}

}

Thus, BankFacade implements two design patterns: **Singleton** and **Facade**. **Singleton** ensures that there is only one instance of the BankFacade class in the application. **Facade** provides a simplified interface to a complex class system, in this case to the Database and Account classes that manage bank accounts.

Behavioral — State

**State** is used to change the behavior of an object when its internal state changes. In the context of the banking system, I used this to work with two account states: active and inactive. Depending on the status of the active and inactive accounts, they perform different operations and display different messages. Thus, all the operations with an account excluding deposit (because it doesn’t depend on the account’s state) are performing in the **State** functions.

class AccountState {

public:

virtual void view(string name, string type, float balance, string tr) = 0;

virtual void deactivate(string name) = 0;

virtual void activate(string name) = 0;

virtual void withdraw(string name, float summ, float& balance, string type,  
 ostringstream& tr) = 0;

virtual float transfer(string name, string acc, float summ, float& balance, string  
 type, ostringstream& tr) = 0;

};

// implementation for active accounts

class ActiveState : public AccountState {

public:

void view(string name, string type, float balance, string tr) override {...};

void deactivate(string name) override {...}

void activate(string name) override {...}

void withdraw(string name, float summ, float& balance, string type,  
 ostringstream& tr) override {...}

float transfer(string name, string acc, float summ, float& balance, string  
 type, ostringstream& tr) override {...}

};

// implementation for inactive accounts

class InactiveState : public AccountState {

public:

void view(string name, string type, float balance, string tr) override {...}

void deactivate(string name) override {...}

void activate(string name) override {...}

void withdraw(string name, float summ, float& balance, string type,  
 ostringstream& tr) override {...}

float transfer(string name, string acc, float summ, float& balance, string  
 type, ostringstream& tr) override {...}

};

// objects for active and inactive states

ActiveState active;

InactiveState inactive;

class Account {

private:

const string type;

const string name;

float balance;

AccountState\* state = &active; // initial state is active

ostringstream transactions;

public:

// functions

}