SSAD Assignment 4 Report

Gleb Popov, B23-ISE-01

Innopolis University

Factory Method — Creational pattern

The **Factory Method** is usually used to create objects when a class cannot preface the types of objects it is supposed to create. In my case, the UserFactory class uses this pattern to create User objects, allowing you to easily add new types of users without changing the main code that creates users.

```
// FACTORY PATTERN
class UserFactory {
public:
    static User* createUser(const string& type, const string& name) {
        if (type == "premium") {
            return new User(true, name);
        } else {
            return new User(false, name);
        }
    }
};

int main() {
    ...
    } else if (command == "createUser") {
        string type;
        string name;
        cin >> type >> name;
        auto it = readers.find(name);
        if (it == readers.end()) {
            auto user = UserFactory::createUser(type, name);
            readers.emplace(name, *user);
        } else {
            cout << "User already exists" << endl;
        }
    }
    ...
}
```

Proxy — Structural pattern

The Proxy pattern is used to control access to an object by providing a substitute or placeholder for another object to control access to it. In my code, I used a Proxy to control access to books, I restrict access to some books based on the user's status.

```
class IBook {
public:
    virtual string getTitle() const = 0;
    virtual string getAuthor() const = 0;
    virtual void read(User* user) = 0;
    virtual void listen(User* user) = 0;
};

class BookProxy {
private:
    IBook* book;
    User* user;
public:
    BookProxy(IBook* b, User* u) : book(b), user(u) {}

    void readBook() {
        cout << user->getName() << " reading " << book->getTitle() << " by " << book-
>getAuthor() << endl;
    }

    void listenBook() {
        if (user->hasPremium()) {
            cout << user->getName() << " listening " << book->getTitle() << " by " <<
book->getAuthor() << endl;
        } else {
            cout << "No access" << endl;
        }
    }
};

class Book : public IBook {
    ...
    void read(User* user) {
        BookProxy proxy(this, user);
        proxy.readBook();
    }

    void listen(User* user) {
        BookProxy proxy(this, user);
        proxy.listenBook();
    }
};
```

Observer — Behavioral pattern

The **Observer** pattern is used to create a mechanism that allows one object (observer) to monitor changes in the state of another object (subject). In the context of online book store, I notify users about a change in the price of the book.

```
class Observer {
public:
    virtual void notify(const string& bookTitle, const string& newPrice) = 0;
};

class UserObserver : public Observer {
private:
    vector<const User*> users;
public:
    void notify(const string& bookTitle, const string& newPrice) override {
        for (auto& user : users) {
            cout << user->getName() << " notified about price update for " <<
bookTitle << " to " << newPrice << endl;
        }
    }
    void addUser(const User* user) {...}
    bool containsUser(const User* user) const {...}
    void removeUser(const User* user) {...}
};

class Book {
private:
    static UserObserver* obs;
public:
    static void subscribe(const User* user) {
        if (obs->containsUser(user)) {
            cout << "User already subscribed" << endl;
        } else {
            obs->addUser(user);
        }
    }
    static void unsubscribe(const User* user) {
        if (obs->containsUser(user)) {
            obs->removeUser(user);
        } else {
            cout << "User is not subscribed" << endl;
        }
    }
};

UserObserver* Book::obs = new UserObserver();
```