Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №6

із дисципліни «**Технології розроблення програмного забезпечення**»

“ШАБЛОНИ «ABSTRACT FACTORY», «FACTORY METHOD», «MEMENTO», «OBSERVER», «DECORATOR»”

Варіант №5. Аудіоредактор

Виконав
студент групи ІА–24
Криворучек В. С.

Перевірив
викладач
Мягкий М. Ю.

Київ 2024

Зміст

Мета: Вивчення основних принципів застосування шаблонів «ABSTRACT FACTORY», «FACTORY METHOD», «MEMENTO», «OBSERVER», «DECORATOR» при створенні проєкту за індивідуальним варіантом

Тема (Варіант №5)

5 Аудіо редактор (singleton, adapter, observer, mediator, composite, client-server) Аудіо редактор повинен володіти наступним функціоналом: представлення аудіо даних будь-якого формату в WAVE-формі, вибір і подальші операції копіювання / вставки / вирізання / деформації по сегменту аудіозапису, можливість роботи з декількома звуковими доріжками, кодування в найбільш поширених форматах (ogg, flac, mp3).

Хід роботи

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
3. Застосування одного з розглянутих шаблонів при реалізації програми.

Теоретичні відомості

Принципи SOLID – це набір рекомендацій для проєктування програмного забезпечення, які допомагають створювати код, що легко підтримується, розширюється та тестується. Вони спрямовані на те, щоб зробити систему більш модульною і стійкою до змін. SOLID – це абревіатура, яка об'єднує п'ять основних принципів.

1. Принцип єдиного обов'язку (Single Responsibility Principle, SRP)

Цей принцип стверджує, що кожен клас має виконувати лише одну логічну задачу або відповідати за одну частину функціональності. У класу має бути лише одна причина для змін. Наприклад, якщо у вас є клас, що обробляє дані і одночасно відповідає за їх відображення, це порушує SRP. Краще розділити його на два класи: один для обробки даних, а інший – для їх відображення.

2. Принцип відкритості/закритості (Open/Closed Principle, OCP)

Програма має бути відкритою для розширення, але закритою для змін. Це означає, що ви повинні мати змогу додавати нову функціональність, не змінюючи вже існуючий код. Цього можна досягти через використання абстракцій, інтерфейсів або базових класів. Наприклад, замість того щоб модифікувати метод сортування в існуючому класі, можна створити новий клас, який реалізує цей метод, використовуючи спільний інтерфейс.

3. Принцип підстановки Барбари Лісков (Liskov Substitution Principle, LSP)

Кожен підклас або похідний клас має мати можливість замінити базовий клас без порушення функціональності програми. Іншими словами, класи-нащадки повинні підтримувати поведінку, визначену базовим класом. Наприклад, якщо у вас є клас "Фігура" з методом площа(), підкласи, такі як "Круг" або "Прямокутник", повинні коректно реалізовувати цей метод і поводитися передбачувано. Якщо підклас змінює поведінку базового класу так, що програма починає працювати некоректно, LSP порушується.

4. Принцип розділення інтерфейсу (Interface Segregation Principle, ISP)

Інтерфейси мають бути невеликими і спеціалізованими. Клієнти не повинні залежати від методів, які вони не використовують. Наприклад, якщо у вас є великий інтерфейс, який включає методи для малювання, збереження та друку, різні класи, що реалізують цей інтерфейс, можуть бути змушені реалізовувати методи, які їм не потрібні. Замість цього краще розбити інтерфейс на декілька менших, кожен з яких відповідає окремій задачі.

5. Принцип інверсії залежностей (Dependency Inversion Principle, DIP)

Високоуровневі модулі не повинні залежати від низькорівневих модулів. Обидва повинні залежати від абстракцій. Крім того, абстракції не повинні залежати від деталей, а деталі повинні залежати від абстракцій. Це означає, що залежності між класами слід організовувати через інтерфейси або абстрактні класи, а не через конкретні реалізації. Наприклад, замість того щоб клас "Робот" напряму використовував клас "Лазер", він повинен працювати через інтерфейс "Зброя". Це дозволяє легко замінити лазер на будь-яку іншу зброю, не змінюючи код класу "Робот".

Ці принципи тісно пов’язані між собою і часто використовуються разом. Їх застосування дозволяє уникнути проблем із залежностями, дублюванням коду та ускладненнями при внесенні змін до програмного забезпечення.

Abstract Factory (Абстрактна фабрика)

Абстрактна фабрика допомагає створювати сімейства взаємопов'язаних об'єктів без прив'язки до їхніх конкретних класів. Наприклад, якщо ваша програма підтримує кілька операційних систем, цей шаблон дозволить створювати специфічні елементи інтерфейсу, такі як кнопки та вікна, для кожної ОС, залишаючи клієнтський код незмінним. Завдяки цьому можна легко адаптувати програму до нових платформ.

Factory Method (Фабричний метод)

Фабричний метод визначає інтерфейс для створення об'єктів, але дозволяє підкласам самостійно вирішувати, який саме об'єкт створити. Наприклад, у грі

можна мати клас зброї, де метод createProjectile() створює відповідні снаряди. Клас «Лук» може створювати стріли, а клас «Гармата» – ядра. Це дозволяє змінювати типи об’єктів, не змінюючи загальної логіки програми.

Memento (Збереження)

Цей шаблон дозволяє зберігати та відновлювати стан об'єкта без порушення його інкапсуляції. Наприклад, у текстовому редакторі можна реалізувати функцію «Скасувати» за допомогою збереження стану документа в об'єкти-збереження. Коли користувач хоче повернутися до попередньої версії, програма просто відновлює потрібний стан.

Observer (Спостерігач)

Шаблон «Спостерігач» використовується для автоматичного сповіщення об'єктів про зміну стану іншого об'єкта. Наприклад, у мобільному додатку для новин, коли сервер оновлює список новин, усі підписані компоненти (зокрема, інтерфейс користувача) автоматично отримують оновлену інформацію. Це дозволяє підтримувати синхронізацію без прямого зв’язку між об'єктами.

Decorator (Декоратор)

Декоратор дозволяє динамічно додавати нову функціональність об'єкту, не змінюючи його структуру. Наприклад, у програмі для оформлення замовлення кави базовий клас напою можна доповнювати такими декораторами, як додавання молока, вершків або сиропу. Таким чином, можна створювати різні комбінації функцій без роздування кількості класів.

Ці шаблони проєктування допомагають створювати код, який легко адаптується до змін і забезпечує гнучкість у розробці. Вони знижують зв’язність між компонентами системи та спрощують розширення її функціональності.

Шаблон Observer

Структура та обгрунтування вибору.

Ознайомившись з теоретичними матеріалами, було прийнято рішення реалізувати шаблон Observer. Цей поведінковий шаблон дає можливість реалізувати поведінку об’єкта, котрий повинен стежити за станом іншого. За допомогою шаблону Observer можна надавати такий функціонал декільком об’єктам.

Для розробки desktop застосунку, де присутня логіка виконання команд, тобто застосунок повинен реагувати на певні дії користувача, наприклад: натиснути на екранну кнопку, натиснути на фізичну кнопку, зробити дію мишкою і т.д., такий шаблон буде значно облегшувати розробку.

Загальну структуру графічно описано на Рисунку 1.

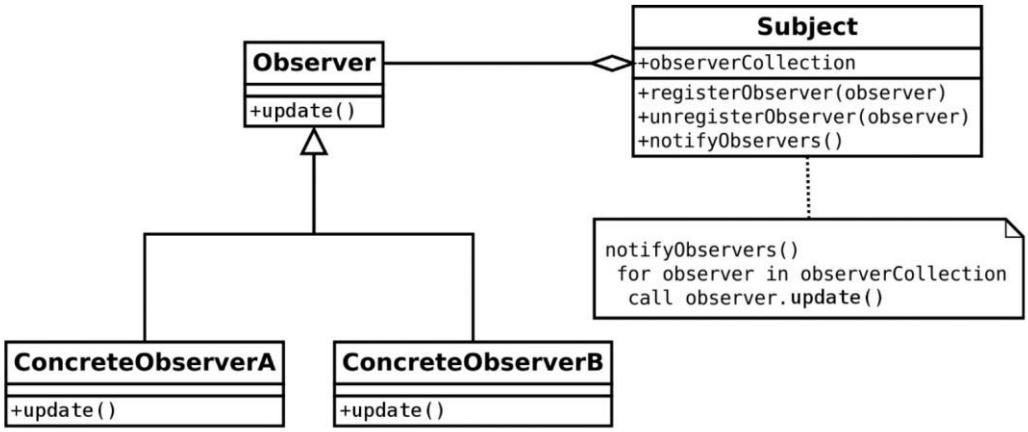

Рисунок 1. Загальна структура шаблону Observer

Оскільки застосунок розробляється на мові програмування Java, графічний інтерфейс базується на Swing, який в свою чергу вже має різні реалізовані типи шаблону observer. Тому, щоб повністю не переписувати код бібліотеки, для деяких випадків буде застосований вбудований у Swing шаблон Observer. Однак, Swing передбачає роботу виключно з графічним інтерфейсом. Для решти функціоналу застосунку було розроблено власну реалізацію шаблону Observer.

Потребу ведення журналу дій користувача цілком можна вирішити через шаблон observer.

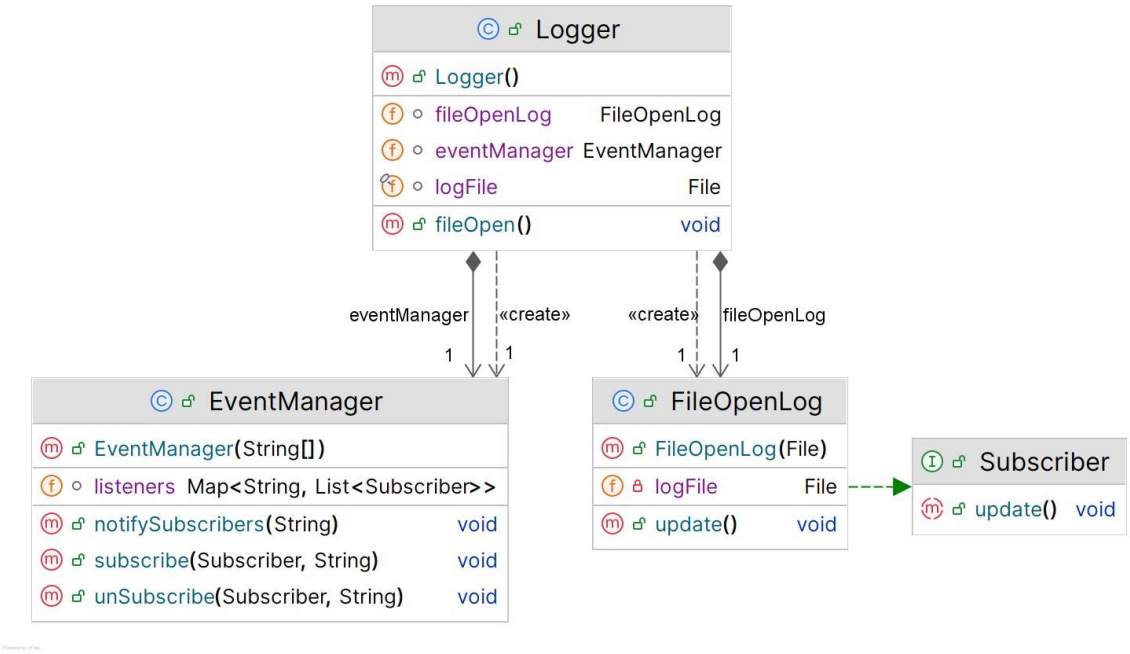

Рисунок 2. Структура класів реалізації шаблону Observer

Клас спостерігача EventManager має три методи: subscribe() та unSubscribe() – підписують та відписують об’єкт інтерфейсу Subscriber на певну подію event, notifySubscribers() – повідомляє про подію event всіх її підписників. Це класична реалізація шаблону observer.

Клас FileOpenLog імплементує інтерфейс Subscriber, відповідно перевизначає метод update(), який містить логіку запису у журнал дії користувача і час коли вона була виконана. У цьому конкретному випадку дія – це відкриття файлу. Тобто кожен разколи користувач відкриває файл відповідний запис буде фіксований у журналі дій.

Клас Logger є компонованою реалізацію логіки запису дій. Він має агрегацію EventManager, а також класи запису певних подій. У конструкторі класу створюються видавник з певною множиною подій на які підписуються їх класи запису. Наприклад, fileOpenLog підписується на подію “openFile”, коли метод Logger.fileOpen() буде викликаний, усі підписники події “openFile”, а отже і fileOpenLog будуть повідомленні.

Реалізація функціоналу у коді з використанням шаблону

class EventManager

```
public class EventManager {
    Map<String, List<Subscriber>> listeners = new HashMap<>();

    public EventManager(String... operations)
{         for (String operation : operations)
{
            this.listeners.put(operation, new ArrayList<>());
        }
    }      public void subscribe(Subscriber subscriber,
String event) {
        List<Subscriber> users = listeners.get(event);
users.add(subscriber);

}
    public void unSubscribe(Subscriber subscriber, String event) {
        List<Subscriber> users = listeners.get(event);
users.remove(subscriber);

}
    public void notifySubscribers(String event) {
        List<Subscriber> users =
listeners.get(event);         for (Subscriber
listener : users) {             listener.update();
        }
    }
}
```

class FileOpenLog

```
public class FileOpenLog implements Subscriber {
private File logFile;

    public FileOpenLog(File logFile) {
this.logFile = logFile;
    }

    @Override
    public void update() {
        try (FileWriter writer = new FileWriter(logFile, true)) {
writer.write("File was opened at " + LocalTime.now() + " " +
LocalDate.now() + "\n");
        } catch (IOException e) {
throw new RuntimeException(e);
        }

}
}
```

class Logger

```
public class Logger {
    EventManager eventManager;
    final File logFile = new File("logs/log.txt");
    FileOpenLog fileOpenLog;

    // ...
    public Logger() {
```

```
        eventManager = new EventManager("openFile");

        fileOpenLog = new FileOpenLog(logFile);
eventManager.subscribe(fileOpenLog, "openFile");

        //...

}
    public void fileOpen() {
        eventManager.notifySubscribers("openFile");

}
}
```

interface Subscriber

```
public interface Subscriber {
public void update();
}
```

Висновок

У ході виконання лабораторної роботи було досліджено п’ять популярних шаблонів проєктування: «Abstract Factory», «Factory Method», «Memento», «Observer» та «Decorator». Ми ознайомилися з їхньою сутністю, призначенням та сферами застосування, а також реалізували приклади використання.

1. Abstract Factory забезпечує створення сімейств взаємопов’язаних об’єктів без залежності від їх конкретних класів, що підвищує гнучкість системи та полегшує адаптацію до нових умов.
2. Factory Method дозволяє делегувати створення об’єктів підкласам, що спрощує розширення функціоналу та підтримує принцип відкритості/закритості (OCP).
3. Memento допомагає зберігати стан об’єкта без порушення інкапсуляції, що є корисним для реалізації функціоналу скасування змін чи відкату до попереднього стану.
4. Observer створює ефективну залежність «один-до-багатьох», яка забезпечує автоматичне інформування об’єктів про зміни, що знижує зв’язність компонентів системи.

5. Decorator дозволяє динамічно додавати нові функції об’єктам, не змінюючи їхньої структури, що полегшує створення гнучких і розширюваних рішень.

Робота з шаблонами проєктування показала, як ці інструменти допомагають створювати масштабовані, зрозумілі та підтримувані програми. Використання шаблонів дозволяє ефективно організувати архітектуру проєкту, зменшити зв’язність між компонентами та підвищити зручність унесення змін до програмного забезпечення.