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

з дисципліни «Технології розроблення програмного забезпечення»

Тема: “ШАБЛОНИ «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator»”

Варіант №5

Виконала:
студентка групи ІА-23
Архип’юк Катерина

Перевірив:
Мягкий Михайло Юрійович

Київ 2024

Зміст

Завдання

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.

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. Короткі теоретичні відомості

Принципи проєктування SOLID є основою для створення чистого, гнучкого та підтримуваного коду. Кожен з принципів націлений на зменшення зв'язаності між компонентами та збільшення можливості їх розширення без зміни існуючої логіки.

Принцип єдиного обов'язку (SRP) сприяє створенню класів, які виконують лише одну задачу, що робить код більш зрозумілим і знижує ймовірність помилок при його зміні. Принцип відкритості/закритості (OCP) дозволяє розширювати функціональність системи без зміни вже існуючого коду, що робить систему більш стійкою до змін і полегшує тестування. Принцип підстановки (LSP) забезпечує, щоб підкласи зберігали функціональність своїх батьківських класів, що дозволяє безпечне використання поліморфізму. Принцип розділення інтерфейсу (ISP) допомагає створювати компактніші і спеціалізовані інтерфейси, що спрощує підтримку та використання системи. Принцип інверсії залежностей (DIP) дозволяє знижувати залежність між модулями, використовуючи абстракції, що забезпечує більшу гнучкість і зручність у розширенні програмних систем.

Шаблон Абстрактна фабрика дозволяє створювати родини взаємопов'язаних або залежних об'єктів, не прив'язуючи код до конкретних класів. Це підхід для створення об'єктів, коли є кілька варіантів (родин) продуктів, але для їх використання не потрібно знати конкретні типи. Шаблон забезпечує інтерфейс для створення об'єктів, що належать до певної родини, без необхідності вказувати конкретні класи. Наприклад, для різних операційних систем можна створити окремі фабрики для створення вікон, кнопок, меню тощо.

Шаблон Фабричний метод надає інтерфейс для створення об'єктів, але дозволяє підкласам вирішувати, який клас створити. Це дозволяє делегувати створення об'єктів підкласам, а не викликати конкретні класи безпосередньо в коді. Фабричний метод дозволяє змінювати тип об'єкта, який створюється, без зміни коду клієнта, що робить систему гнучкішою.

Шаблон Мементо використовується для збереження і відновлення попередніх

станів об'єкта без порушення інкапсуляції. Він дозволяє зберігати внутрішній стан об'єкта в момент часу, щоб потім можна було відновити його, не розкриваючи деталей реалізації. Це корисно для реалізації функціональності відкату в програмах, таких як текстові редактори, де потрібно зберігати попередні стани документа.

Шаблон Спостерігач дозволяє створювати механізм, при якому один об'єкт (суб'єкт) сповіщає інші об'єкти (спостерігачі) про зміни свого стану. Це дозволяє ефективно реалізовувати події та реагування на зміни без жорсткої залежності між об'єктами. Наприклад, коли в одній частині програми змінюється стан, інші частини автоматично отримують оновлену інформацію (як у випадку з інтерфейсами користувача або системами, де кілька підсистем повинні отримувати сповіщення).

Шаблон Декоратор дозволяє додавати нову функціональність до об'єкта без зміни його структури. Це дозволяє гнучко розширювати можливості об'єкта, комбінуючи різні «декоратори». Шаблон особливо корисний, коли потрібно додавати функціональність об'єкту, не перевантажуючи його клас додатковими методами. Наприклад, якщо є об'єкт, що обробляє текст, можна застосувати декоратори для форматування тексту без зміни основної логіки обробки.

2. Шаблон Observer

2.1 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.

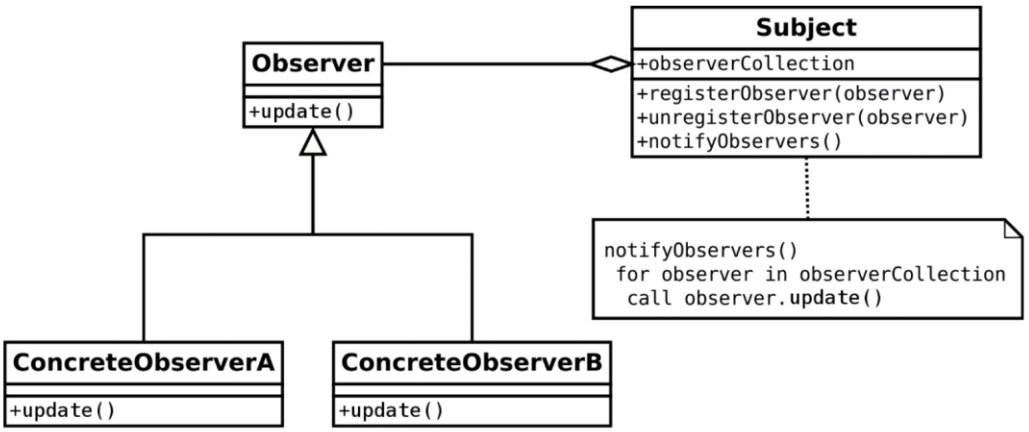

Рисунок 1. Загальна структура шаблону Observer

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.

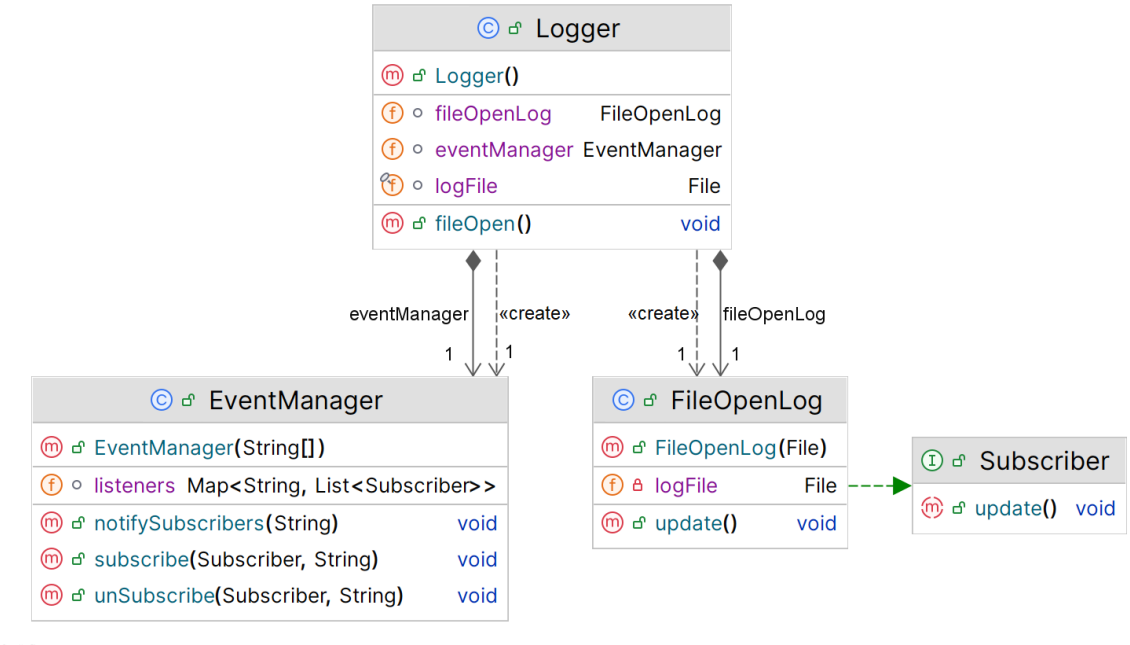

Рисунок 2. Структура класів реалізації шаблону Observer

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.

2.2 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();
}
```

Висновки: При виконанні цієї лабораторної роботи я закріпила навички застосування шаблонів проєктування. Шаблон Abstract Factory дозволяє створювати сімейства об'єктів без прив'язки до їх конкретних класів. Factory

Method дає можливість делегувати створення об'єктів підкласам. Memento використовується для збереження і відновлення стану об'єктів. Observer забезпечує автоматичне сповіщення про зміни в об'єктах. Decorator дозволяє додавати нову функціональність до об'єкта без зміни його класу.

Посилання на репозиторій з кодом проєкту:
https://github.com/KatiaArkhyp/AudioEditor