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

Лабораторна робота №6

із дисципліни «*ТРПЗ*»

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

Виконав:
Студент групи ІА-24
Бакалець А.І.

Перевірив:
Мягкий М.Ю.

Київ-2024

Мета: Навчитися використовувати шаблони abstract factory, factory method, memento, observer, decorator.

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:

..1 Музичний програвач (iterator, command, memento, facade, visitor, client-server)

Музичний програвач становить собою програму для програвання музичних файлів або відтворення потокової музики з можливістю створення, запам'ятовування і редагування списків програвання, перемішування/повторення (shuffle/repeat), розпізнавання різних аудіо-форматів, еквалайзер.

Github: https://github.com/Clasher3000/TRPZ_labs

Теоретичні відомості

Абстрактна фабрика (Abstract Factory)
Шаблон "Абстрактна фабрика" надає інтерфейс для створення групи взаємозалежних об'єктів без визначення їх конкретних класів. Використовується, коли система повинна працювати з кількома наборами об'єктів, які пов'язані між собою. Абстрактна фабрика визначає методи для створення кожного типу об'єктів, а конкретні реалізації фабрик створюють ці об'єкти. У Java цей шаблон зазвичай реалізується шляхом створення абстрактного класу або інтерфейсу фабрики, а потім конкретних фабрик, які створюють необхідні об'єкти. Це дозволяє легко змінювати цілі набори об'єктів, забезпечуючи незалежність клієнтського коду від конкретних реалізацій.

Фабричний метод (Factory Method)
Шаблон "Фабричний метод" визначає інтерфейс для створення об'єктів, але дозволяє підкласам вирішувати, який саме об'єкт буде створено. Він надає гнучкість у виборі класів, які потрібно створити, і сприяє відкритості до розширень. У Java цей шаблон часто реалізується за допомогою абстрактного класу або інтерфейсу з одним методом, який повертає створений об'єкт, а конкретні підкласи реалізують цей метод для створення потрібних об'єктів. Це корисно, коли система повинна створювати об'єкти, типи яких невідомі заздалегідь.

Збереження стану (Memento)
Шаблон "Memento" дозволяє зберігати і відновлювати внутрішній стан об'єкта без порушення його інкапсуляції. Це корисно для реалізації функцій "Скасувати" або "Повернутися до попереднього стану". У шаблоні є три учасники: "Одержувач" (об'єкт, стан якого зберігається), "Опікун" (керує збереженими станами) і "Сувенір" (зберігає стан). У Java це зазвичай реалізується через класи, де сувенір є внутрішнім класом об'єкта. Це дозволяє зберігати історію змін і повертатися до попереднього стану, забезпечуючи контроль за змінами об'єкта.

Спостерігач (Observer)
Шаблон "Спостерігач" визначає залежність "один-до-багатьох", коли зміна стану одного об'єкта повідомляє всім його підписникам (спостерігачам). Це забезпечує синхронізацію об'єктів і дозволяє автоматично оновлювати їх у разі змін. У Java цей шаблон реалізується через інтерфейси, такі як Observer і Observable, або за допомогою механізму слухачів (Listeners). Клас-спостережуваний повідомляє всіх зареєстрованих спостерігачів про зміни. Це корисно для систем, де дані змінюються динамічно, наприклад, GUI або обробка подій.

Декоратор (Decorator)
Шаблон "Декоратор" дозволяє динамічно додавати нові функції об'єкту, не змінюючи його структури. Він обгортає оригінальний об'єкт у додатковий об'єкт-декоратор, який реалізує ту ж саму базову функціональність, але додає нову поведінку. У Java цей шаблон часто реалізується за допомогою абстрактного класу або інтерфейсу, який реалізують і базовий клас, і декоратори. Це дозволяє створювати гнучкі системи, де об'єкти можуть бути розширені без множинного успадкування, зберігаючи прозорий інтерфейс для клієнта.

Хід роботи

У цій лабораторній роботі я реалізував шаблон Збереження стану (Memento), оскільки моя програма потребує можливості зберігати та відновлювати стан об'єктів. Цей шаблон спрощує збереження стану об'єкта в певний момент часу, дозволяючи повернутися до нього в майбутньому. Шаблон Memento допомагає інкапсулювати стан об'єкта таким чином, щоб він залишався недоступним для інших об'єктів, забезпечуючи інкапсуляцію.

Реалізація шаблону забезпечила можливість зберігати поточний стан відтворення музики, зокрема, активний трек і плейлист. Завдяки цьому, користувач може повернутися до попереднього стану відтворення, навіть якщо програма виконала декілька інших дій. Це підвищує зручність роботи, адже користувачеві не потрібно вручну вибирати трек або плейлист повторно.

Крім того, шаблон Memento дозволяє реалізувати функціональність "Відновлення стану", що є корисним для відтворення перерваного плейлиста або треку. Це також спрощує підтримку та розширення програми, оскільки механізм збереження стану відокремлений від основної логіки керування музичним плеєром.

У моїй програмі реалізація шаблону включає класи **Memento**, який зберігає стан, і **Caretaker**, який керує історією станів. Наприклад, при збереженні стану користувач може зберегти поточний трек і плейлист, а при відновленні — автоматично відтворити музику з того місця, де було зупинено. Це підвищує функціональність та гнучкість програми.

Структура шаблону Command:

```
package org.example.server.memento;

import org.example.entity.Playlist;
import org.example.entity.Track;

7 usages  Clasher3000
public class Memento {
    4 usages
    private Playlist playlist;

    3 usages
    private Track track;

    1 usage  Clasher3000
    public Memento(Playlist playlist, Track track) {
        this.playlist = playlist;
        this.track = track;
    }
    1 usage  Clasher3000
    public Memento(Track track) {
        playlist = null;
        this.track = track;
    }

    Clasher3000
    public Playlist getPlaylist() { return playlist; }

    Clasher3000
    public Track getTrack() { return track; }

    1 usage  Clasher3000
    public boolean isPlaylistBased() {
        return playlist != null;
    }
}
```

Рис.1 – Клас Memento

Клас **Memento** у нашій програмі відповідає за збереження стану важливих об'єктів, таких як **Track** і **Playlist**. Він дозволяє зафіксувати поточний стан цих об'єктів і відновити їх пізніше. Клас містить приватні поля для збереження даних про поточний трек і плейлист.

Конструктор класу **Memento** приймає об'єкти **Track** та **Playlist**, або тільки Track, що дозволяє зберігати їх у момент виклику. Ці дані не можна змінювати безпосередньо через сам клас, оскільки він лише інкапсулює стан, що робить клас **Memento** корисним для забезпечення відновлення стану без можливості його редагування.

Клас має методи для доступу до збережених об'єктів, але не дозволяє вносити зміни в них. Таким чином, він служить для збереження певного стану програми на момент виклику і забезпечує можливість відновлення цього стану в майбутньому.

```java
package org.example.server.memento;

import org.example.entity.Playlist;
import org.example.entity.Track;
import java.util.Stack;

3 usages  Clasher3000 *
public class Caretaker {
    4 usages
    private final Stack<Memento> mementoStack = new Stack<>();


    1 usage  Clasher3000
    public void save(Playlist playlist, Track track) {
        mementoStack.push(new Memento(playlist, track));
        System.out.println("State saved: Playlist - " +
                (playlist != null ? playlist.getName() : "null") +
                ", Track - " + track.getTitle());
    }

    1 usage  Clasher3000
    public void save(Track track) {
        mementoStack.push(new Memento(track));
        System.out.println("State saved: Individual Track - " + track.getTitle());
    }
```

```
1 usage   Clasher3000
public Memento undo() {
    if (!mementoStack.isEmpty()) {
        Memento previousState = mementoStack.pop();
        if (previousState.isPlaylistBased()) {
            System.out.println("State restored: Playlist - " +
                    previousState.getPlaylist().getName() +
                    ", Track - " + previousState.getTrack().getTitle());
        } else {
            System.out.println("State restored: Individual Track - " +
                    previousState.getTrack().getTitle());
        }
        return previousState;
    }
    System.out.println("No states to restore.");
    return null;
}


}
```

Рис.2 – Клас Caretaker

Клас **Caretaker** в нашій програмі виконує важливу роль у збереженні та відновленні станів об'єктів, зокрема **Track** і **Playlist**. Він діє як посередник між класами, які змінюють стан об'єктів, та класами, що зберігають ці стани.

Основною метою **Caretaker** є збереження об'єктів **Memento**, які містять зафіксовані стани. Коли користувач хоче зберегти поточний стан, наприклад, стан програваного треку або плейлиста, **Caretaker** отримує **Memento** і зберігає його. Це дозволяє відновити ці стани пізніше, якщо це необхідно.

Caretaker інкапсулює логіку збереження і відновлення станів, що дозволяє іншим класам не турбуватися про управління станами об'єктів. Вони можуть просто звертатися до **Caretaker**, який бере на себе функцію збереження та відновлення. Це спрощує обслуговування коду і дозволяє краще організувати взаємодію між різними частинами програми.

```java
1 usage  Clasher3000
public void saveState() {
    if (track != null) {
        if (playlist != null){
            caretaker.save(playlist, track);
            out.println("State saved: Playlist - " + playlist.getName() + ", Track - " + track.getTitle());
        }
        else{
            caretaker.save(track);
            out.println("State saved: Track - " + track.getTitle());
        }
    }
    else {
        out.println("Nothing is playing now.");
    }

}
```

```java
1 usage  Clasher3000
public void restoreState() {
    Memento memento = caretaker.undo();
    if (memento != null) {
        // Отримуємо трек і плейлист зі збереженого стану
        Playlist savedPlaylist = memento.getPlaylist();
        Track savedTrack = memento.getTrack();

        if (savedPlaylist != null && savedTrack != null) {
            // Відновлюємо плейлист і починаємо збережений трек
            playPlaylistOnTrack(savedPlaylist.getName(), savedTrack.getTitle());
        } else if (savedTrack != null) {
            // Якщо є тільки трек, граємо його
            playSong(savedTrack.getTitle());
        } else {
            out.println("No saved track or playlist to restore.");
        }
    } else {
        out.println("No saved state available.");
    }
}
```

Рис.3 – Використання методів класу Caretaker в MusicPlayer

У класі **MusicPlayer** методи класу **Caretaker** використовуються для збереження та відновлення станів плеєра. Ось як це працює:

1. **saveState()** — метод класу **MusicPlayer**, який відповідає за збереження поточного стану плеєра. Коли викликається цей метод, **MusicPlayer** перевіряє, чи є активний трек або плейлист. Якщо це так, він створює об'єкт **Memento**, що містить інформацію про поточний трек чи плейлист, і передає його в **Caretaker** через метод **save()**. Це зберігає поточний стан плеєра для можливості відновлення.
2. **restoreState()** — метод класу **MusicPlayer**, який використовується для відновлення збереженого стану. Він викликає метод **undo()** в класі **Caretaker**, щоб отримати останній збережений **Memento**. Далі **MusicPlayer** відновлює збережений стан, програючи трек або плейлист, який був збережений раніше.

Таким чином, **MusicPlayer** використовує **Caretaker** для збереження і відновлення станів плеєра, що дозволяє користувачеві повернутися до попереднього стану програвання, якщо це необхідно.

Висновок: В процесі виконання лабораторної роботи я ознайомився з такими патернами проектування, як Abstract Factory, Factory Method, Memento, Observer та Decorator. Особливу увагу я приділив шаблону Memento, який дозволив реалізувати функціональність збереження та відновлення стану музичного плеєра, зберігаючи стан треку та плейлиста для подальшого відновлення. Це забезпечило гнучкість та можливість відновлювати попередні стани, що значно полегшує підтримку і тестування коду. Застосування цих патернів підвищило модульність, читабельність і масштабованість програми, дозволяючи ефективно управляти об'єктами та операціями в проекті.