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

Технології розроблення програмного забезпечення

Лабораторна робота №6
«ШАБЛОНИ «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator».»

Виконала:
Студентка групи ІА-22
Микитенко Ірина

Перевірив:
Мягкий Михайло Юрійович

Київ 2024

Зміст

Тема: шаблони «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator».

Мета: ознайомитися з шаблонами проектування «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator» та набути практичних навичок їх застосування. Реалізувати частину функціоналу програми за допомогою одного з розглянутих шаблонів для досягнення конкретних функціональних можливостей та забезпечення ефективної взаємодії між класами.

Хід роботи

..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.

1. Короткі теоритичні відомості

Принципи SOLID, запропоновані Робертом Мартіном, визначають правила розробки класів у системах.

Кожен клас має виконувати лише одну чітко визначену задачу.
Приклад: Якщо служба звітів інтегрована зі службою друку в одному класі, зміни в одній службі можуть порушити роботу іншої. Відокремлення обов'язків знижує зв'язаність і підвищує цілісність системи.

Класи мають бути відкритими для розширення, але закритими для зміни.
Реалізація: Використання інтерфейсів або абстрактних класів дозволяє змінювати поведінку без модифікації вихідного коду, наприклад, для додавання нових функцій чи тестування.

Підкласи повинні зберігати поведінку батьківських класів.
Приклад: Якщо в дочірньому класі метод "сума" реалізований через множення, це порушує принцип. Програма має працювати коректно як із батьківським, так і з дочірнім класом.

Багато вузьких інтерфейсів краще, ніж один універсальний.

Суть: Кожен інтерфейс має відповідати за конкретну функцію. Це спрощує підтримку й використання системи, дозволяючи взаємодіяти лише з необхідними частинами.

Залежності повинні будуватися на основі абстракцій, а не деталей. *Реалізація*: Верхньорівневі модулі посилаються на інтерфейси, які реалізуються нижньорівневими модулями. Це зменшує зв’язаність і спрощує модифікацію системи.

SOLID-принципи допомагають створювати гнучкі, підтримувані та масштабовані програмні системи.

Шаблон «Abstract Factory»

Призначення:
Створює сімейства пов'язаних об'єктів без вказівки їх конкретних класів. Застосовується, коли необхідно забезпечити сумісність продуктів одного сімейства.

Структура:

: загальний інтерфейс для створення продуктів.

: реалізації для різних сімейств продуктів.

: загальний інтерфейс для кожного продукту.

: конкретні реалізації продуктів.

: працює з фабриками та продуктами через загальні інтерфейси.

Проблема:
Наприклад, потрібно створювати меблі (крісла, дивани, столики) у різних стилях (Ар-деко, Вікторіанський, Модерн), які пасуватимуть одне до одного.

Рішення:

иділяються загальні інтерфейси для кожного продукту.

ля кожного сімейства продуктів створюються свої фабрики, що реалізують інтерфейс фабрики.

лієнт працює лише із загальними інтерфейсами, що забезпечує гнучкість і сумісність.

Приклад:
Уявіть симулятор меблевого магазину, який створює меблі в заданих стилях, щоб уникнути їх несумісності.

Переваги:

- Забезпечує сумісність продуктів.
- Звільняє код від залежності від конкретних класів.
- Реалізує принцип відкритості/закритості.

Недоліки:

- Складність через велику кількість класів.
- Обов’язкова наявність усіх типів продуктів у кожній варіації.

Шаблон «Factory Method»

Призначення:
Визначає інтерфейс для створення об'єктів певного базового типу, дозволяючи підкласи заміняти об'єкти на їх підтипи. Це спрощує підтримку коду та розширення функціоналу без зміни базового коду.

Структура:

- **Creator:** базовий клас із фабричним методом.
- **ConcreteCreator:** підкласи, що реалізують фабричний метод для створення конкретних об'єктів.
- **Product:** базовий інтерфейс для об'єктів.
- **ConcreteProduct:** реалізації продуктів.

Проблема:
Програма для вантажних перевезень спочатку працює лише з вантажівками. Для додавання інших видів транспорту (судна, літаки) потрібно змінювати весь код, що ускладнює його підтримку.

Рішення:
Замінити пряме створення об’єктів (new) викликом фабричного методу.
Кожен вид транспорту (Вантажівка, Судно) реалізує спільний інтерфейс **Транспорт**, а підкласи логістики (Дорожня, Морська) повертають відповідні об’єкти через фабричний метод.

Приклад:
Менеджер із найму делегує процес співбесіди різним спеціалістам залежно від вакансії, зберігаючи спільну логіку найму.

Переваги:

- Усунення прив’язки до конкретних класів продуктів.
- Спрощення підтримки коду.
- Легке додавання нових продуктів.

Недоліки:

- Може створити великі паралельні ієрархії класів.

Шаблон «Memento»

Призначення:
Шаблон дозволяє зберігати та відновлювати стан об'єкта без порушення інкапсуляції. Об'єкт-знімок (Memento) використовується виключно для збереження стану. Початковий об'єкт (Originator) може створювати та відновлювати свій стан із знімка, тоді як зовнішні об'єкти, такі як Caretaker, лише зберігають або передають знімки.

Проблема:
Наприклад, у текстовому редакторі потрібно реалізувати функцію скасування дій. Клас редактора повинен зберігати свій стан перед кожною зміною. Прямий доступ до полів об'єкта порушує інкапсуляцію та створює труднощі при зміні структури класу.

Рішення:
Шаблон «Memento» передбачає, що об'єкт сам створює знімки свого стану й надає обмежений доступ до них іншим об'єктам. Caretaker зберігає ці знімки та передає їх назад для відновлення стану.

Приклад з життя:
Клас історії у текстовому редакторі може зберігати список знімків із назвами та датами. Коли користувач виконує скасування, історія передає останній знімок редактору для відновлення.

Переваги:

- Не порушує інкапсуляцію.
- Спрощує вихідний об'єкт, прибираючи необхідність зберігати історію станів.

Недоліки:

- Вимагає багато пам’яті при частому створенні знімків.
- Може перевантажувати пам’ять, якщо застарілі знімки не видаляються.

Шаблон «Observer»

Призначення:
Шаблон визначає залежність "один-до-багатьох", дозволяючи об'єктам-спостерігачам автоматично отримувати сповіщення про зміни стану об'єкта-видавця.

Проблема:
Постійне ручне перевіряння змін (як покупець, що щодня перевіряє товар у магазині) неефективне, а масова розсилка повідомлень може викликати незадоволення у користувачів.

Рішення:

- Видавець зберігає список підписників.
- Підписники додають або видаляють себе зі списку за допомогою методів видавця.
- При зміні стану видавець повідомляє підписників, викликаючи їх методи.
- Загальний інтерфейс дозволяє підписникам працювати з різними видавцями.

Приклад:
Підписка на журнал: видавництво надсилає нові номери без додаткових запитів із вашого боку.

Переваги:

- Динамічна підписка/відписка.
- Слабке зв’язування між видавцем і підписниками.
- Відповідає принципу відкритості/закритості.

Недоліки:

- Невизначена послідовність сповіщення підписників.

Шаблон «Decorator»

Призначення:

Динамічно додає функціональність об'єкту під час виконання програми, зберігаючи його базову поведінку. Декоратор обертає об'єкт, додаючи додаткові функції, що забезпечує більшу гнучкість, ніж спадкування.

Приклад:

Декоратор для бібліотечних елементів, що додає можливість оренди:

```
class Borrowable : Decorator {
   protected List<string> borrowers = new List<string>();
   public Borrowable(LibraryItem item) : base(item) { }
   public void BorrowItem(string name) {
      borrowers.Add(name); libraryItem.NumCopies--;
```

```
    }
    public void ReturnItem(string name) {
        borrowers.Remove(name); libraryItem.NumCopies++;
    }
    public override void Display() {
        base.Display();
        foreach (string borrower in borrowers) Console.WriteLine("Borrower: " +
borrower);
    }
}
```

Базові функції елементів (наприклад, відображення даних) залишаються незмінними, а оренда додається динамічно.

Проблема:

Уявіть систему сповіщень, яка надсилає повідомлення різними каналами (email, SMS, Slack). Спадкування для кожної комбінації каналів значно роздуває код.

Рішення:

Замінити спадкування композицією. Декоратор обгортає об'єкт, додаючи функції без зміни вихідного класу. Наприклад, Notifier можна обгорнути декораторами для SMS і Slack, зберігаючи email-сповіщення.

Приклад із життя:

Одяг — аналог декоратора. Ви вдягаєте светр для тепла, а плащ — для захисту від дощу, не змінюючи свою сутність.

Переваги:

- Додає функціональність «на льоту».
- Гнучкість у порівнянні зі спадкуванням.
- Відповідає принципу відкритості/закритості.

Недоліки:

- Велика кількість дрібних класів.
- Складність налаштування при багатошаровій обгортці.

2. Реалізація не менше 3-х класів відповідно до обраної теми.

Структура проекту з реалізованими класами зображена на рисунку 1

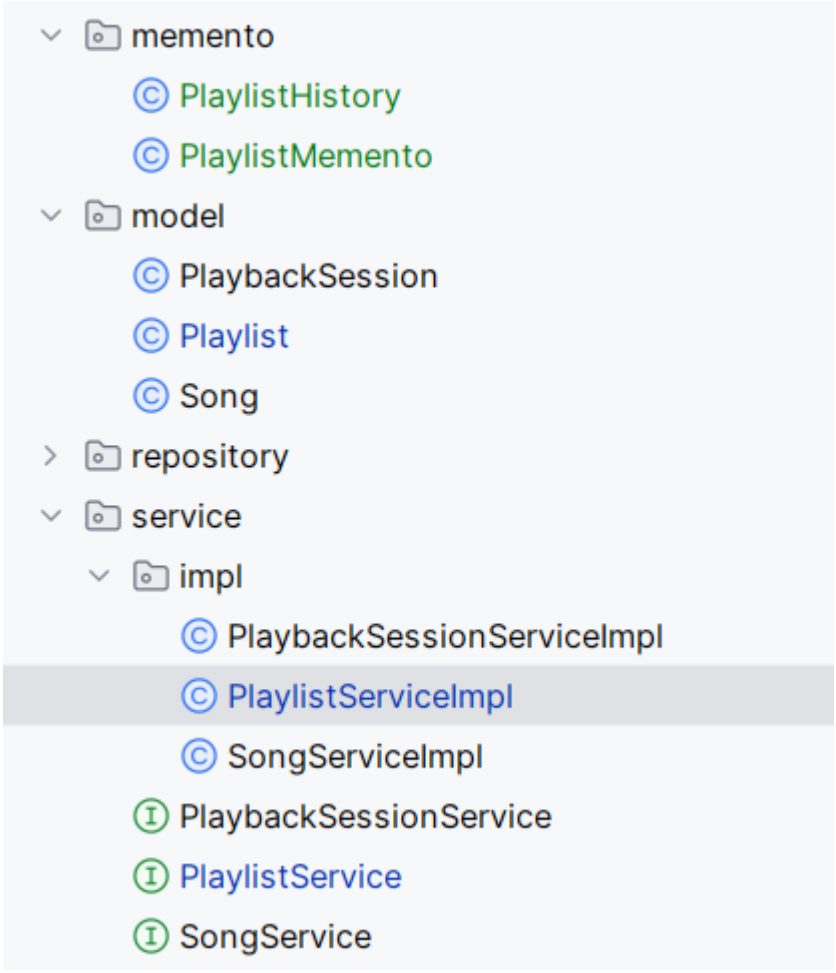

Рисунок 1 – структура проекту

У ході лабораторної роботи була реалізована система музичного плеєра, де основним завданням було використання патерна **Memento** для збереження та відновлення стану плейлистів. В результаті виконано такі кроки:

1. **Клас Playlist (Originator)**:
    - Цей клас відповідає за зберігання поточного стану плейлиста, включаючи колекцію пісень у плейлисті. Основне завдання цього класу — надання можливості створення м'якого стану (Memento) та відновлення його.
    - У класі Playlist є метод createMemento(), який створює новий об'єкт PlaylistMemento, що зберігає поточний стан пісень. Також є метод restoreFromMemento(PlaylistMemento memento), який дозволяє відновити стан плейлиста зі збереженого стану.
2. **Клас PlaylistMemento (Memento)**:
    - Цей клас є представником м'якого стану, в якому зберігаються всі необхідні дані для відновлення стану плейлиста.

- Основним атрибутом є список пісень (List<Song> songs), який зберігається на момент створення стану. Мета класу — забезпечити доступ до цих даних, не даючи можливості змінювати їх ззовні.

3. **Клас PlaylistHistory (Caretaker)**:
   - Цей клас виконує роль «сторожа» станів (Mementos). Він зберігає історію всіх змін плейлиста і дозволяє отримати збережений стан за індексом.
   - Основним атрибутом є список List<PlaylistMemento> mementos, в якому зберігаються стани. Клас має методи для додавання нових станів (addMemento()) та отримання збережених станів за індексом (getMemento()).
4. **Клас PlaylistService (Клієнт)**:
   - PlaylistService виступає клієнтом для патерна **Memento**, працюючи з історією змін плейлистів та використовуючи збережені стани для відновлення попередніх версій плейлиста.
   - Основні функції цього сервісу:
     1. **Збереження стану плейлиста**: Кожного разу після зміни пісень у плейлисті (додавання або видалення), сервіс викликає метод savePlaylistState(), який зберігає поточний стан плейлиста через createMemento(). Це дозволяє відновити його в майбутньому.
     2. **Відновлення стану плейлиста**: Сервіс надає метод для відновлення стану плейлиста за допомогою методу restorePlaylistState(), вказавши індекс збереженого стану. Це дозволяє користувачу повернутися до попереднього стану плейлиста.
5. **Інтеграція з контролером**:
   - Контролер взаємодіє з сервісом для виконання операцій із плейлистами. Він надає кінцеві точки для збереження та відновлення стану плейлистів через API.

3. Реалізація шаблону за обраною темою

У ході виконання лабораторної роботи ми застосували патерн **Memento** для збереження та відновлення стану плейлистів у музичному плеєрі. Цей патерн дозволяє зберігати стан об'єкта (плейлиста) в певний момент часу та відновлювати його при необхідності. Ось кроки реалізації цього патерна:

PlaylistMemento (Memento):

- Клас PlaylistMemento є реалізацією патерна **Memento**. Його мета — зберігати стан плейлиста, тобто список пісень, які знаходяться в плейлисті на момент збереження.
- **Атрибути**:
  - List<Song> songs: зберігає список пісень, який є частиною стану плейлиста.
- **Методи**:
  - Конструктор PlaylistMemento(List<Song> songs): зберігає список пісень в момент створення.
  - getSavedSongs(): повертає збережений список пісень, що дозволяє відновити їх у майбутньому.

Даний клас представлено на рисунку 2.

```
package com.example.music_player.memento;

import com.example.music_player.model.Song;

import java.util.List;

public class PlaylistMemento {  14 usages  new *
    private final List<Song> songs;  2 usages

    public PlaylistMemento(List<Song> songs) {  1 usage  new *
        this.songs = songs;
    }

    public List<Song> getSavedSongs() {  1 usage  new *
        return this.songs;
    }
}
```

Рисунок 2- PlaylistMemento

PlaylistHistory (Caretaker):

- Клас PlaylistHistory зберігає історію всіх збережених м'яких станів плейлиста (через об'єкти типу PlaylistMemento). Цей клас не змінює стан плейлиста, а лише зберігає його в історії та надає можливість відновлення.
- **Атрибути**:
  - List<PlaylistMemento> mementos: колекція, що містить збережені стани плейлистів.

- **Методи**:
  - addMemento(PlaylistMemento memento): додає новий м'який стан у історію.
  - getMemento(int index): отримує збережений стан за індексом для відновлення.

Даний клас представлено на рисунку 3.

```
import java.util.ArrayList;
import java.util.List;

public class PlaylistHistory {  3 usages  new *
    private final List<PlaylistMemento> mementos;  3 usages

    public PlaylistHistory() {  no usages  new *
        this.mementos = new ArrayList<>();
    }

    public void addMemento(PlaylistMemento memento) {  1 usage  new *
        mementos.add(memento);
    }

    public PlaylistMemento getMemento(int index) {  1 usage  new *
        return this.mementos.get(index);
    }
}
```

Рисунок 3 - PlaylistHistory

Playlist (Originator):

- Клас Playlist є **Originator** у контексті патерна **Memento**. Це клас, чий стан зберігається та відновлюється через стани (PlaylistMemento). Клас має список пісень і відповідає за створення станів, а також відновлення стану плейлиста з збережених станів.
- **Атрибути**:
  - List<Song> songs: колекція пісень, що знаходяться в плейлисті.
- **Методи**:
  - createMemento(): створює та повертає стан (PlaylistMemento) поточного стану плейлиста.
  - restoreFromMemento(PlaylistMemento memento): відновлює стан плейлиста з переданого стану.

Дані методи представлено на рисунку 4.

```
public PlaylistMemento createMemento() {  1 usage  new *
    return new PlaylistMemento(this.songs);
}

public void restoreFromMemento(PlaylistMemento memento) {
    this.songs = memento.getSavedSongs();
}
```

Рисунок 4 – методи createMemento() та restoreFromMemento(PlaylistMemento memento)

4. Діаграма класів для паттерну Memento

Діаграма класів, які реалізовують паттерн Memento зображена на рисунку 5

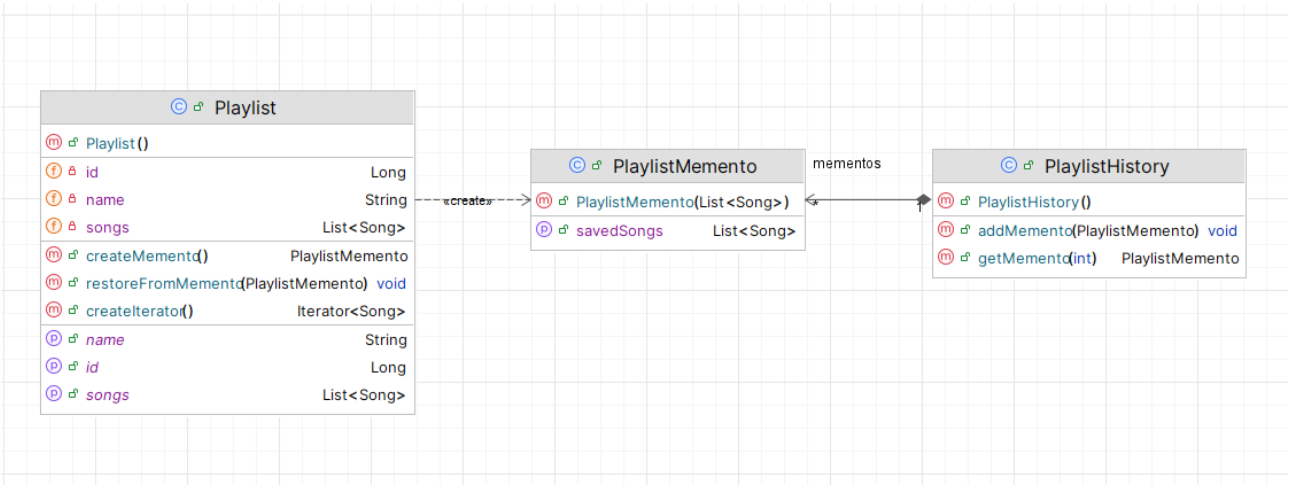

Рисунок 5- діаграма класів

Проблема, яку допоміг вирішити шаблон Memento

До використання патерна **Memento**, збереження та відновлення стану плейлиста було складним завданням. Кожного разу, коли потрібно було зберегти поточний стан колекції пісень, необхідно було вручну зберігати та відновлювати елементи колекції. Це призводило до дублювання коду, а також до труднощів при додаванні нових функціональностей, оскільки потрібно було кожного разу імплементувати логіку збереження та відновлення стану.

Використовуючи патерн **Memento**, вдалося:

- **Інкапсулювати логіку збереження та відновлення стану** плейлиста в окремий клас мементо, що зробило код чистішим та зручнішим для підтримки.
- **Зберегти стан об'єкта** без необхідності розкриття його внутрішньої структури, що дозволило реалізувати механізм збереження та відновлення стійких версій плейлиста.

- **Полегшити додавання нових можливостей** для збереження і відновлення станів плейлистів без втручання в основний код програми.

Переваги застосування патерну Memento

Інкапсуляція логіки збереження та відновлення стану: Патерн дозволяє винести всю логіку збереження стану в окремий клас мементо. Це дозволяє розділити відповідальності та робить код більш організованим і чистим.

Полегшена підтримка та розширення: Додавати нові способи збереження та відновлення стану можна без змін у основній логіці програми. Клас мементо відповідає тільки за збереження стану, що дозволяє розширювати функціональність без змін у коді плейлиста чи інших частин програми.

Збереження історії станів: Патерн дає можливість створювати безпечне збереження різних версій стану плейлиста, що дозволяє зручно відновлювати попередні стани у разі потреби.

Гнучкість у відновленні станів: Патерн дозволяє відновлювати стан плейлиста на будь-якому етапі, навіть якщо структура даних змінилася. Можна зберігати стани на різних етапах і при необхідності повернутися до потрібного.

Висновок: У даній лабораторній роботі я реалізувала патерн проектування Memento для музичного плеєра, що дозволило створити ефективний механізм збереження та відновлення стану плейлиста. Завдяки цьому патерну, стан пісень у плейлисті зберігається в окремих об'єктах мементо, що забезпечує легкість підтримки та розширення функціональності. Переваги застосування Memento включають інкапсуляцію логіки збереження стану, зручність в управлінні історією змін та гнучкість у відновленні стану. Реалізація цього патерну зробила систему більш організованою, масштабованою та простішою для подальшого розвитку.