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 №8

із дисципліни «*ТРПЗ*»

Тема: ШАБЛОНИ «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR»

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

Перевірив:
Мягкий М.Ю.

Київ-2024

Мета: Навчитися використовувати шаблони composite, flyweight, interpreter, visitor.

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:

..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

Компонувальник (Composite)

Шаблон "Компонувальник" дозволяє працювати з групами об'єктів так само, як з одним об'єктом, організовуючи їх у деревоподібну структуру. Використовується, коли потрібно представити ієрархію частина-ціле і надати єдиний інтерфейс для роботи з окремими об'єктами та їх групами. У Java цей шаблон реалізується шляхом створення базового інтерфейсу або абстрактного класу для всіх об'єктів у структурі та підкласів для

представлення як "листків" (окремих об'єктів), так і "гілок" (груп об'єктів). Це дозволяє клієнтському коду працювати з усією структурою без перевірок її деталей.

Легковаговик (Flyweight)
Шаблон "Легковаговик" оптимізує використання пам'яті шляхом спільного використання об'єктів, які поділяють однаковий стан. Використовується, коли система створює велику кількість однотипних об'єктів, що можуть розділяти загальний внутрішній стан. У Java цей шаблон зазвичай реалізується через фабрику, яка управляє пулом спільних об'єктів. Зовнішній стан об'єктів зберігається поза ними, щоб уникнути надмірного споживання пам'яті. Це дозволяє значно знизити накладні витрати в системах із великою кількістю об'єктів.

Інтерпретатор (Interpreter)
Шаблон "Інтерпретатор" визначає спосіб представлення граматики ієрархії мови і надає інтерпретатор для її виконання. Використовується для роботи з мовами, які мають чітко визначену граматику, наприклад, для створення калькуляторів або розбору виразів. У Java цей шаблон реалізується через створення класів, які представляють правила граматики, та методу interpret(), що виконує операції. Це забезпечує зручність у роботі з граматикою, проте може бути неефективним для складних мов через зростання кількості класів.

Відвідувач (Visitor)
Шаблон "Відвідувач" дозволяє визначати нові операції для об'єктів без зміни їхніх класів. Використовується, коли необхідно виконати кілька різних операцій над об'єктами складної структури, але їх класи не можна змінювати. У Java цей шаблон реалізується шляхом створення інтерфейсу Visitor, який має методи для кожного типу елементів, та їх реалізацій для виконання конкретних операцій. Об'єкти структури реалізують метод accept(Visitor visitor), який викликає відповідний метод відвідувача. Це дозволяє додавати нові операції, зберігаючи класи об'єктів незмінними.

.

Хід роботи

У цій лабораторній роботі я реалізував шаблон **Відвідувач (Visitor),** оскільки моя програма потребує виконання різних операцій над об'єктами складної структури без внесення змін у їхні класи. Цей шаблон дозволяє додавати нові операції, залишаючи незмінними класи елементів, що особливо зручно для систем із багатьма типами об'єктів.

Шаблон Visitor забезпечив чітке розділення операцій і даних у моїй програмі. Це дозволило уникнути дублювання коду та спростити підтримку, оскільки нові операції можна додавати у вигляді окремих відвідувачів, не змінюючи існуючу структуру елементів.

Реалізація шаблону Visitor у моїй програмі включає інтерфейс Visitor, який визначає методи для кожного типу об'єктів, зокрема для треків (TrackService) і плейлистів (PlaylistService). Конкретна реалізація відвідувача (FindVisitor) дозволяє виконувати

операції, такі як отримання списку треків або плейлистів із їхніми треками. Класи TrackService та PlaylistService реалізують метод accept(Visitor visitor), який забезпечує виклик відповідних методів відвідувача.

Шаблон Visitor значно спростив додавання нових функцій до програми. Наприклад, якщо знадобиться реалізувати нову операцію, як-от виведення статистики про треки або плейлисти, це можна зробити, додавши новий клас відвідувача, не змінюючи класи елементів.

Використання шаблону Visitor також сприяло підвищенню гнучкості програми. Наприклад, FindVisitor дозволяє легко знаходити всі треки або плейлисти та їхні треки, що підвищило зручність доступу до даних і їх обробки. Це забезпечило централізований спосіб управління операціями над різними об'єктами, зменшивши зв'язаність між компонентами програми.

Таким чином, реалізація шаблону Visitor зробила код моєї програми більш організованим і розширюваним. Цей підхід полегшив розробку нових функціональностей і забезпечив зручність роботи з системою, яка складається з різних типів об'єктів.

Структура шаблону Visitor

```
package org.example.visitor;

import org.example.service.PlaylistService;
import org.example.service.TrackService;

import java.util.List;

8 usages  1 implementation  Clasher3000
public interface Visitor {

    1 implementation  Clasher3000
    void findAll(Element... args);
    1 usage  1 implementation  Clasher3000
    List<String> visitTrack(TrackService trackService);
    1 usage  1 implementation  Clasher3000
    List<String> visitPlaylist(PlaylistService playlistService);
}
```

Рис.1 – Інтерфейс відвідувача

Цей рядок оголошує інтерфейс Visitor, який визначає методи, необхідні для реалізації шаблону "Відвідувач" (Visitor). Інтерфейс включає метод findAll(Element... args), що дозволяє виконувати операції над кількома об'єктами, переданими як параметри. Методи visitTrack(TrackService trackService) та visitPlaylist(PlaylistService playlistService) визначають логіку взаємодії з відповідними сервісами для обробки треків і плейлистів, повертаючи списки результатів у вигляді рядків.

```
public class FindVisitor implements Visitor{

    2 usages
    private PrintWriter out;

    1 usage   Clasher3000
    public FindVisitor(PrintWriter out) {
        this.out = out;
    }

    Clasher3000 *
    public void findAll(Element... args) {
        for (Element element : args) {
            List<String> result = element.accept( visitor: this);
            for (String item : result) {
                out.println(item);
            }
        }
    }
     Ctrl⇧L to Chat, Ctrl+I to Command
```

Рис.2 – Імплементація спостерігача та метод для пошуку всіх елементів

```java
@Override
public List<String> visitTrack(TrackService trackService) {
    List<Track> tracks = trackService.findAll();
    List<String> trackTitles = new ArrayList<>();
    for (Track track : tracks
         ) {
        trackTitles.add(track.getTitle() + " - Track");
    }
    return trackTitles;
}

1 usage  Clasher3000
@Override
public List<String> visitPlaylist(PlaylistService playlistService) {
    List<Playlist> playlists = playlistService.findAll();
    List<String> playlistsNames = new ArrayList<>();

    for (Playlist playlist : playlists) {
        // Створюємо список імен треків за допомогою стрімів
        String playlistTracks = playlist.getTracks().stream() Stream<Track>
                .map(Track::getTitle) Stream<String>
                .collect(Collectors.joining( delimiter: ", ", prefix: " [", suffix: "]"));

        // Додаємо назву плейлиста з треками до списку
        playlistsNames.add(playlist.getName() + playlistTracks);
    }
    return playlistsNames;
```

Рис.3 – Методи візіта в реалізації FindVisitor

Цей код реалізує клас **FindVisitor**, який є конкретним візитером у шаблоні "Візитор". Він використовується для роботи з об'єктами, що реалізують інтерфейс Element, забезпечуючи спосіб отримання даних із сервісів треків та плейлистів.

Метод **findAll** приймає кілька об'єктів типу Element, викликає їх метод accept і записує результати (список рядків) у вивід через PrintWriter.

Метод **visitTrack** отримує всі треки через TrackService, формує список назв треків із додаванням позначки " - Track" до кожного треку та повертає цей список.

Метод **visitPlaylist** отримує всі плейлисти через PlaylistService, формує список рядків, де кожен містить назву плейлиста і список треків, представлених у форматі назва плейлиста [трек1, трек2, ...], використовуючи Stream API для створення списку назв треків.

```java
package org.example.visitor;

import java.util.List;

11 usages  4 implementations  Clasher3000
public interface Element {
    1 usage  2 implementations  Clasher3000
    List<String> accept(Visitor visitor);
}
```

Рис.4 – Інтерфейс елемента

Цей код представляє інтерфейс **Element**, який є частиною реалізації шаблону **"Візитор"**.

Інтерфейс має єдиний метод **accept(Visitor visitor)**, який приймає об'єкт типу **Visitor** як параметр і повертає список рядків (**List<String>**).

Метод **accept** визначає контракт для взаємодії між класами, які реалізують цей інтерфейс, і класами-візитерами. Його реалізація в конкретних класах дозволяє візитеру виконувати специфічні операції над об'єктами цих класів.

```
package org.example.service;

import org.example.entity.Playlist;
import org.example.visitor.Element;

import java.util.List;

16 usages  1 implementation  Clasher3000 *
public interface PlaylistService extends Element {

    3 usages  1 implementation  Clasher3000
    Playlist findByName(String name);

    1 implementation  Clasher3000
    void create(String name);

    1 implementation  Clasher3000
    List<Playlist> findAll();

}
```

Рис.5 – Інтерфейс PlaylistService, який розширює Element

Інтерфейс **PlaylistService** визначає контракт для сервісу, який працює з об'єктами типу **Playlist**. Він успадковує інтерфейс **Element**, що дозволяє реалізувати шаблон **"Візитор"**. У межах цього інтерфейсу визначені методи для пошуку плейлистів за назвою, створення нових плейлистів і отримання всіх плейлистів у вигляді списку. Оскільки **PlaylistService** успадковує **Element**, цей інтерфейс також містить метод **accept**, який дозволяє застосовувати шаблон **"Візитор"** для виконання операцій з плейлистами через візитора.

```java
public class PlayListServiceImpl implements PlaylistService {

    1 usage  Clasher3000
    @Override
    public List<String> accept(Visitor visitor) { return visitor.visitPlaylist( playlistService: this); }

    4 usages
    private PlaylistRepository playlistRepository;

    3 usages  Clasher3000
    public PlayListServiceImpl() { this.playlistRepository = new PlaylistRepository(); }

    3 usages  Clasher3000
    @Override
    public Playlist findByName(String name) { return playlistRepository.getPlaylistByName(name); }

    Clasher3000
    @Override
    public void create(String name) { playlistRepository.createPlaylist(name); }

    Clasher3000
    @Override
    public List<Playlist> findAll() { return playlistRepository.findAll(); }
}
```

Рис.6 – Реалізація інтерфейсу PlayListService

Цей клас **PlayListServiceImpl** є реалізацією інтерфейсу **PlaylistService**. Він надає конкретну реалізацію методів для роботи з плейлистами. Клас використовує **PlaylistRepository** для доступу до даних про плейлисти. У методі **accept** реалізовано виклик методу візитора, що дозволяє застосовувати шаблон **"Візитор"** для взаємодії з плейлистами. Метод **findByName** шукає плейлист за його назвою, **create** створює новий плейлист, а **findAll** повертає всі плейлисти з репозиторію.

```java
package org.example.service;

import org.example.entity.Track;
import org.example.visitor.Element;
import org.example.visitor.Visitor;

import java.util.List;

18 usages  1 implementation  Clasher3000
public interface TrackService extends Element{

    1 usage  1 implementation  Clasher3000
    Track findByTitle(String title);

    1 usage  1 implementation  Clasher3000
    void addTrack(String title, String path);

    1 usage  1 implementation  Clasher3000
    void addTrackToPlaylist(String playlistName, String trackTitle);

    1 implementation  Clasher3000
    List<Track> findAll();
}
```

Рис.7 – Інтерфейс TrackService, який розширює Element

Цей інтерфейс **TrackService** описує операції для роботи з треками в системі. Він розширює інтерфейс **Element**, що дозволяє використовувати шаблон "Візитор" для взаємодії з треками. Інтерфейс містить методи для пошуку трека за його назвою (**findByTitle**), додавання трека з вказаними параметрами (**addTrack**), додавання трека до певного плейлиста (**addTrackToPlaylist**), а також для отримання всіх треків (**findAll**).

```java
1 usage  Clasher3000
public void findAllTracks(Element... args){
    FindVisitor findVisitor = new FindVisitor(out);
    findVisitor.findAll(args);
}
```

Рис.8 – Метод в MusicPlayer, який створює візітор і знаходить всі існуючі треки і списки відтворення в програмі.

Цей метод **findAllTracks** викликає метод **findAll** з класу **FindVisitor** для обробки переданих елементів, які є варіантами для пошуку (наприклад, плейлисти чи треки). Він передає ці елементи в об'єкт візитора, і в результаті кожен елемент приймає цей візит і

повертає відповідну інформацію. Результати відображаються через об'єкт **PrintWriter**, який виводить інформацію на консоль або в інший потік виводу.

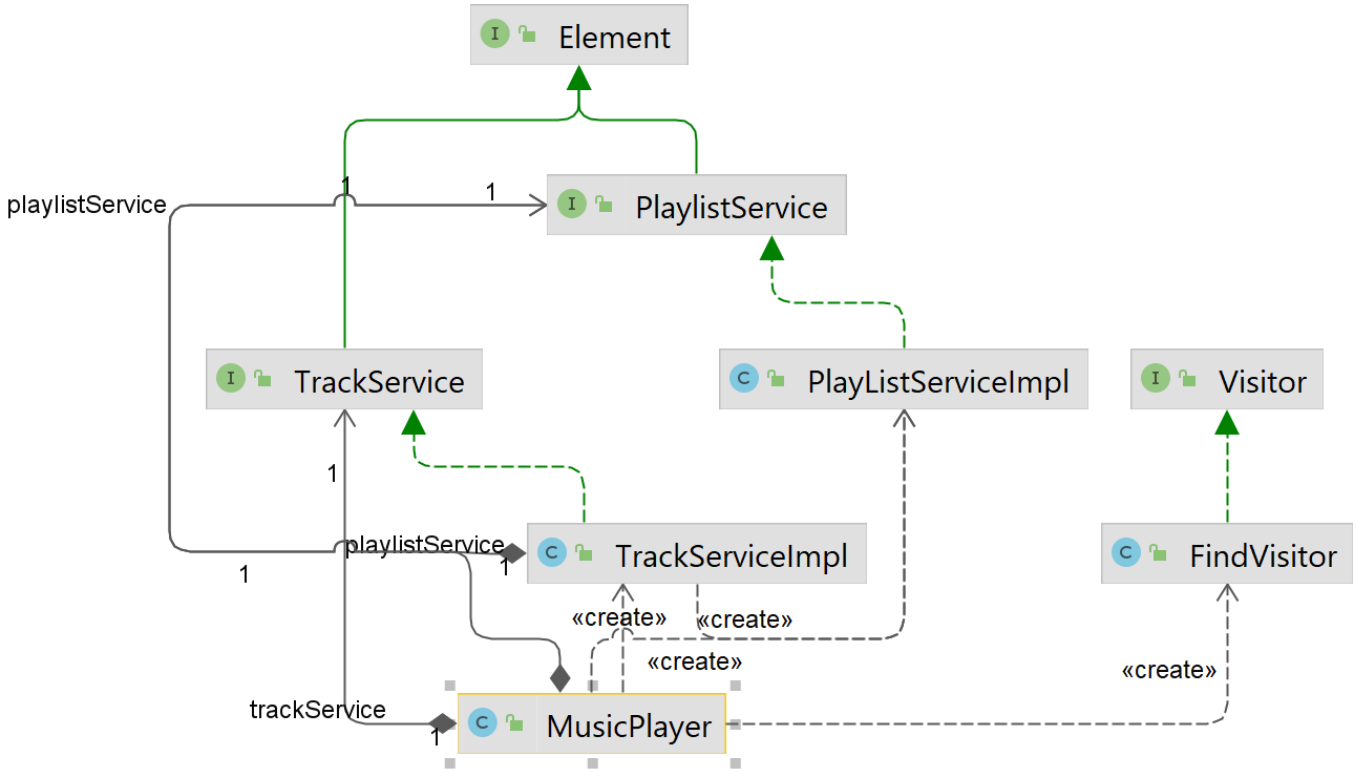

Рис.9 – Схема компонентів, використаних в шаблоні Visitor

Демонстрація виконання

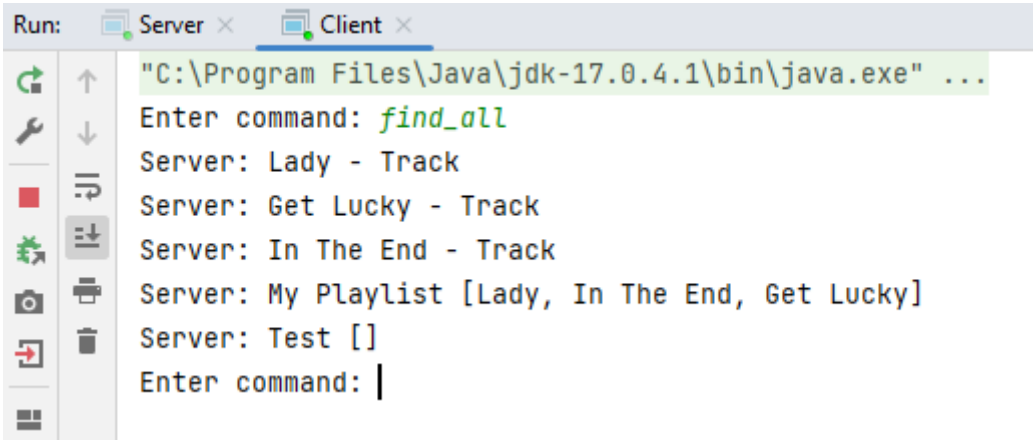

Рис.10 – Результат виклику команди find_all

Коли користувач вводить команду find_all, сервер обробляє її, використовуючи патерн "Відвідувач". Сервер передає всі елементи (треки та плейлисти) на обробку через

Відвідувача. Відвідувач викликає відповідні методи для кожного елемента: для треків — метод visitTrack(), а для плейлистів — метод visitPlaylist(). Ці методи збирають інформацію про всі треки та плейлисти, додають їх у результати й передають їх назад на сервер. У результаті сервер виводить список треків і плейлистів. Це дозволяє зручно отримати повний список всіх доступних треків і плейлистів за допомогою централізованої обробки через Відвідувача.

Висновок:

У процесі виконання лабораторної роботи я ознайомився з кількома патернами проектування, такими як *Composite*, *Flyweight*, *Interpreter* і *Visitor*. Кожен з них має свою специфіку та корисність для вирішення різних завдань в програмуванні, підвищуючи гнучкість і масштабованість систем.

Особливу увагу я приділив патерну *Visitor*, який я використав для реалізації програми. Завдяки цьому патерну, я зміг реалізувати обробку елементів без зміни їхніх класів, що значно спростило розширення функціоналу та підвищило зручність підтримки коду. Використання *Visitor* дозволило організувати чітку ієрархію елементів і зробити програму більш гнучкою для майбутніх змін або додавання нових типів обробки.