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 №4

із дисципліни «*ТРПЗ*»

Тема: «ШАБЛОНИ «SINGLETON», «ITERATOR», «PROXY», «STATE», «STRATEGY»**»**

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

Перевірив:
Мягкий М.Ю.

Київ-2024

Мета: Навчитися використовувати шаблони singleton, iterator, proxy, state, strategy

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

Патерни проектування

Патерн проектування — це типовий спосіб вирішення певної проблеми, що часто зустрічається при проектуванні архітектури програм.

На відміну від готових функцій чи бібліотек, патерн не можна просто взяти й скопіювати в програму. Патерн являє собою не якийсь конкретний код, а загальний принцип вирішення певної проблеми, який майже завжди треба підлаштовувати для потреб тієї чи іншої програми.

Патерни часто плутають з алгоритмами, адже обидва поняття описують типові рішення відомих проблем. Але якщо алгоритм — це чіткий набір дій, то патерн — це високорівневий опис рішення, реалізація якого може відрізнятися у двох різних програмах.

Якщо провести аналогії, то алгоритм — це кулінарний рецепт з чіткими кроками, а патерн — інженерне креслення, на якому намальовано рішення без конкретних кроків його отримання.

Найбільш низькорівневі та прості патерни — *ідіоми*. Вони не дуже універсальні, позаяк мають сенс лише в рамках однієї мови програмування.

Найбільш універсальні — *архітектурні патерни*, які можна реалізувати практично будь-якою мовою. Вони потрібні для проектування всієї програми, а не окремих її елементів.

Крім цього, патерни відрізняються і за призначенням. Існують три основні групи патернів:

- **Породжуючі патерни** піклуються про гнучке створення об’єктів без внесення в програму зайвих залежностей.
- **Структурні патерни** показують різні способи побудови зв’язків між об’єктами.
- **Поведінкові патерни** піклуються про ефективну комунікацію між об’єктами.

Одинак (Singleton)

Шаблон проєктування "Одинак" гарантує, що клас матиме лише один екземпляр, і забезпечує глобальну точку доступу до цього екземпляра. Це корисно, коли потрібно контролювати доступ до деяких спільних ресурсів, наприклад, підключення до бази даних або конфігураційного файлу. Основна ідея полягає у тому, щоб закрити доступ до конструктора класу і створити статичний метод, що повертає єдиний екземпляр цього класу. У мовах, які підтримують багатопоточність, також важливо синхронізувати метод доступу, щоб уникнути створення кількох екземплярів в різних потоках. Один із способів реалізації одинак у Java – використання статичної ініціалізації, яка автоматично забезпечує безпечність у багатопоточному середовищі. Деякі розробники вважають одинак антипатерном, оскільки він створює глобальний стан програми, що ускладнює тестування та підтримку. Використання цього шаблону має бути обґрунтованим і обмеженим певними обставинами.

Ітератор (Iterator)

Шаблон "Ітератор" надає спосіб послідовного доступу до елементів колекції без розкриття її внутрішньої структури. Він особливо корисний для обходу різних типів колекцій, таких як списки, множини або деревовидні структури, незалежно від того, як вони реалізовані. Ітератор інкапсулює поточний стан перебору, тому може зберігати інформацію про те, який елемент є наступним. Цей шаблон дозволяє відокремити логіку роботи з колекцією від логіки обходу, що робить код більш гнучким і модульним. У Java цей шаблон реалізований у вигляді інтерфейсу Iterator, який надає методи hasNext() та next() для послідовного перебору елементів. Ітератор також можна використовувати для видалення елементів під час обходу колекції. Завдяки цьому шаблону можна використовувати поліморфізм для однакового доступу до елементів різних колекцій.

Проксі (Proxy)

Шаблон "Проксі" створює замісник або посередника для іншого об’єкта, що контролює доступ до цього об’єкта. Проксі може виконувати додаткову роботу перед передачею викликів реальному об'єкту, як-от перевірку прав доступу або відкладену

ініціалізацію. Існує декілька типів проксі, серед яких захисний проксі, який контролює доступ, і віртуальний проксі, який затримує створення об'єкта, поки він не буде потрібен. У Java цей шаблон часто використовується для створення динамічних проксі за допомогою інтерфейсів, де проксі-клас реалізує той самий інтерфейс, що й реальний об’єкт. Проксі ефективний для оптимізації роботи з ресурсами або для контролю доступу до важких у створенні об'єктів. Це дозволяє зберігати оригінальний об'єкт захищеним і надає додатковий шар для маніпуляцій.

Стан (State)

Шаблон "Стан" дозволяє об'єкту змінювати свою поведінку при зміні внутрішнього стану, надаючи йому різні стани для різних контекстів. Він ефективно інкапсулює різні стани об'єкта як окремі класи і делегує дії поточному стану. Наприклад, кнопка може мати різні дії залежно від того, чи вона активована чи деактивована. Це дозволяє замість довгих умовних операторів використовувати поліморфізм, де кожен клас стану реалізує свою поведінку, визначену інтерфейсом стану. У Java цей шаблон може бути реалізований як клас з інтерфейсом для станів, де кожен стан є окремим підкласом, що відповідає за певну поведінку. Це полегшує масштабування та тестування, оскільки додавання нового стану не вимагає змін у вихідному коді.

Стратегія (Strategy)

Шаблон "Стратегія" дозволяє вибирати алгоритм або поведінку під час виконання, забезпечуючи взаємозамінність різних алгоритмів для конкретного завдання. Він передбачає інкапсуляцію різних варіантів поведінки в окремих класах, які реалізують один інтерфейс, що спрощує заміну і додавання алгоритмів. Клас контексту отримує об'єкт стратегії і викликає відповідні методи, не знаючи деталей реалізації конкретної стратегії. Наприклад, клас сортування може мати кілька стратегій: швидке сортування, сортування вставкою чи сортування вибором, і залежно від контексту обирається відповідний метод. У Java шаблон реалізується через інтерфейс стратегії, який мають різні класи конкретних стратегій.

Хід роботи

У цій лабораторній роботі я реалізував шаблон **Ітератор**, оскільки в моїй програмі використовується робота зі списками, а ітератор спрощує обробку цих колекцій. Ітератор надає зручний спосіб обходу елементів без потреби знати, як ці колекції реалізовані. Це дозволяє зосередитися на логіці програми, а не на деталях реалізації.

Завдяки ітератору я отримав уніфікований інтерфейс для обходу колекцій, таких як треки в плейлистах. Це підвищує узгодженість коду, адже я можу використовувати один і той же підхід для різних колекцій, не змінюючи логіку обробки.

Крім того, ітератор забезпечує захист даних, контролюючи доступ до елементів колекції. Це зменшує ймовірність помилок, пов'язаних із некоректним доступом до треків. Він також полегшує розширення програми — якщо потрібно буде додати нові типи колекцій, це можна зробити, не вносячи значних змін у вже написаний код.

У моїй програмі реалізація ітератора, наприклад, TrackIteratorImpl, дозволяє ітерувати через колекцію об'єктів Track, що є частиною плейлистів. Я передаю ітератору список треків, що дозволяє йому знати, з якою колекцією працювати. Метод hasNext() перевіряє, чи є ще елементи для обходу, а метод next() повертає наступний трек. Також в моїй роботі присутній метод hasPrevious(), який перевіряє чи є треки, які вже були програні та метод prev() який повертає попередній трек.

Структура шаблону Ітератор:

```
1   package org.example.server.iterator;
2
3   import org.example.entity.Track;
4
    3 usages  1 implementation  Clasher3000
5   public interface TrackIterator {
        2 usages  1 implementation  Clasher3000
6       boolean hasNext();
        1 usage  1 implementation  Clasher3000
7       Track next();
        2 usages  1 implementation  Clasher3000
8       boolean hasPrevious();
        1 usage  1 implementation  Clasher3000
9       Track previous();
10
11  }
```

Рис.1 - Інтерфейс ітератору треку

Інтерфейс, який задає базову структуру наших ітераторів, визначає основні методи, які будь-який конкретний ітератор повинен реалізувати. Основна мета цього інтерфейсу полягає в наданні уніфікованого способу обходу елементів колекції, незалежно від їхньої внутрішньої реалізації.

```java
public class TrackIteratorImpl implements TrackIterator {
    4 usages
    private List<Track> tracks;
    6 usages
    private int position = 0;

    1 usage  Clasher3000
    public TrackIteratorImpl(List<Track> tracks) {
        this.tracks = tracks;
    }

    2 usages  Clasher3000
    @Override
    public boolean hasNext() {
        return position < tracks.size();
    }

    1 usage  Clasher3000
    @Override
    public Track next() {
        return hasNext() ? tracks.get(position++) : null;
    }
    2 usages  Clasher3000
    @Override
    public boolean hasPrevious() {
        return position > 0;
    }

    1 usage  Clasher3000
    @Override
    public Track previous() {

        if(hasPrevious()){
            position = position - 1;
            Track track = tracks.get(position);
            return track;
        }
        else return null;

    }
```

Рис.2 - Реалізація ітератору треків

Клас, який реалізує інтерфейс ітератора, виконує функцію обходу списком треків, забезпечуючи можливість послідовного перегляду елементів,. Такий клас надає простий інтерфейс для роботи з колекцією треків, приховуючи складність структури даних від користувача.

```
public void playPlaylist(String playlistName) {
    try {
        Playlist playlist = playlistService.findByName(playlistName);
        if (playlist != null && playlist.getTracks() != null && !playlist.getTracks().isEmpty()) {
            trackIterator = new TrackIteratorImpl(playlist.getTracks());
            playNextTrackInPlaylist();
            out.println("Playlist: " + playlistName + " is playing");
        } else {
            out.println("Playlist is empty or does not exist.");
        }
    } catch (NoResultException e){
        out.println("Incorrect playlist name");
    }
}
3 usages  Clasher3000
public void playNextTrackInPlaylist() {
    if (trackIterator != null && trackIterator.hasNext()) {
        Track nextTrack = trackIterator.next();
        playSong(nextTrack.getTitle());
    } else {
        out.println("There are no more tracks.");
    }
}
1 usage  Clasher3000
public void playPreviousTrackInPlaylist() {
    if (trackIterator != null && trackIterator.hasPrevious()) {
        Track previousTrack = trackIterator.previous();
        playSong(previousTrack.getTitle());
    } else {
        out.println("There are no tracks before.");
    }
}
```

Рис3. - Клас, який використовує наш ітератор

У класі **MusicPlayer** ми реалізуємо методи, які використовують **TrackIterator** для управління відтворенням треків. Це дозволить зручно змінювати, додавати та відтворювати треки в плейлісті, спрощуючи процес навігації через колекцію треків. Використання ітератора забезпечить чистий та зрозумілий код, а також підвищить модульність програми, дозволяючи з легкістю реалізувати нові функції в майбутньому.

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

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

1 usage  1 implementation
public interface PlaylistIterator {
    1 usage  1 implementation
    boolean hasNext();
    no usages  1 implementation
    Playlist next();
}
```

Рис.4 - Інтерфейс для ітератору плейлистів

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

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

import java.util.List;

no usages
public class PlaylistIteratorImpl implements PlaylistIterator{
    3 usages
    private List<Playlist> playlists;
    2 usages
    private int position = 0;

    no usages
    public PlaylistIteratorImpl(List<Playlist> playlists) { this.playlists = playlists; }

    1 usage
    @Override
    public boolean hasNext() {
        return position < playlists.size();
    }

    no usages
    @Override
    public Playlist next() { return hasNext() ? playlists.get(position++) : null; }

}
```

Рис.5 - Клас, який реалізовує ітерацію плейлистів

Висновок: виконуючи цю лабораторну роботу, я ознайомився з такими патернами, як Singleton, Ітератор, Проксі, Стан та Стратегія. Особливу увагу я приділив реалізації шаблону Ітератор, детально описавши його основну логіку та функціональність у контексті мого проєкту. Завдяки цьому досвіду я краще зрозумів, як ці патерни можуть підвищити модульність, зручність використання та читабельність коду, а також як вони сприяють ефективному управлінню об'єктами в програмуванні.