Національний технічний університет України «Київський
політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

ЛАБОРАТОРНА РОБОТА №4

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Вступ до паттернів проектування»

Виконав:

студент групи ІА-33

Грицай Андрій

Перевірив:

асистент кафедри ІСТ

Мягкий Михайло Юрійович

Київ - 2025

Тема: Вступ до паттернів проектування
Мета: Вивчити структуру шаблонів «Singleton», «Iterator», «Proxy», «State», «Strategy» та навчитися застосовувати їх в реалізації програмної системи.
Тема лабораторного циклу:

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.

Теоретичні відомості

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.

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 роботі для реалізації функціоналу навігації по колекціях музичних треків було обрано патерн проєктування "Ітератор" (Iterator). Цей патерн був обраний як архітектурне рішення для вирішення фундаментальної проблеми: необхідно надати клієнтському коду (музичному плеєру) можливість послідовного доступу до елементів агрегатного об'єкта (бібліотеки або плейлиста) без розкриття його внутрішнього представлення.

У процесі роботи над проєктом стало очевидно, що "перехід до наступного треку" — це дія, яка може залежати від обраного режиму відтворення. Наприклад, логіка звичайного послідовного відтворення та логіка перемішування (Shuffle) є різними, але, по суті, виконують одне завдання — "надання наступного елемента колекції".

Без патерну "Ітератор" реалізація такого функціоналу призвела б до написання складної логіки управління індексами та численних умовних операторів (if/else) безпосередньо у класах Library або Playlist. Це зробило б код контейнерів перевантаженим, складним для тестування та важким для розширення новими алгоритмами обходу.

Патерн "Ітератор" вирішує цю проблему елегантно. Він дозволяє винести логіку перебору елементів в окремі класи, які реалізують спільний інтерфейс AudioIterator. Клієнт (у даному випадку, класи Library та Playlist) делегує виконання завдання навігації об'єкту-ітератору, не вникаючи в деталі алгоритму обходу.

Застосування цього патерну природним чином привело нас до визначення наступних конкретних ітераторів:

- LinearIterator — для стандартного послідовного відтворення.
- ShuffleIterator — для відтворення у випадковому порядку.

Реалізація класів

Реалізація шаблону “Iterator” вимагає створення окремих класів для реалізації кожного конкретного алгоритму.

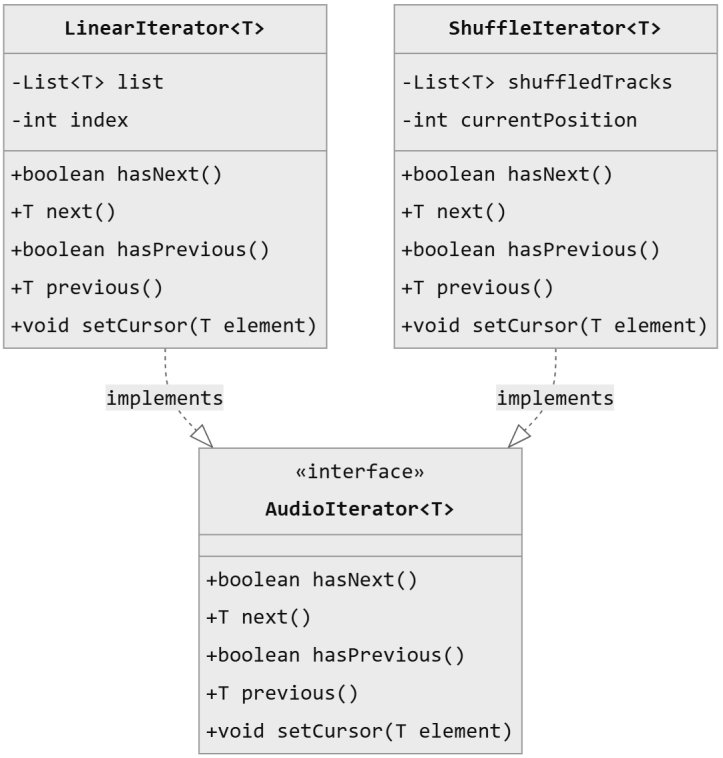


Рисунок 1. - Діаграма класів, яка представляє використання шаблону в реалізації системи

Опис створених об’єктів:

AudioIterator (інтерфейс). AudioIterator є абстрактною основою патерну. Він визначає загальний інтерфейс (контракт) для всіх конкретних алгоритмів обходу музичної колекції. Інтерфейс декларує методи для навігації (next, previous), перевірки наявності елементів (hasNext, hasPrevious) та встановлення курсору (setCursor), які мають бути реалізовані усіма конкретними ітераторами.

```
package com.example.labs.common.patterns.iterator;

public interface AudioIterator<T> {
    boolean hasNext();
    T next();
    boolean hasPrevious();
    T previous();
    void setCursor(T element);
}
```

LinearIterator (конкретний ітератор). Цей клас реалізує AudioIterator і представляє алгоритм "Послідовного відтворення". Його реалізація є стандартною для списків: він зберігає посилання на вихідний список та поточний індекс. Методи next() та previous() просто зміщують індекс вперед або назад по вихідному списку треків.

```
package com.example.labs.common.patterns.iterator;
import java.util.List;
```

```
public class LinearIterator<T> implements AudioIterator<T> {
    private List<T> list;
    private int index = -1;

    public LinearIterator(List<T> list) {
        this.list = list;
    }

    @Override
    public T next() {
        if (!hasNext()) return null;
        index++;
        return list.get(index);
    }
    // ... (інші методи: hasNext, previous, setCursor)
}
```

ShuffleIterator (конкретний ітератор). Цей клас реалізує AudioIterator для режиму "Випадкового відтворення" (Shuffle). На відміну від лінійного, при створенні він створює копію вихідного списку та перемішує її за допомогою Collections.shuffle(). Навігація відбувається вже по цьому перемішаному внутрішньому списку, забезпечуючи унікальний порядок відтворення без зміни основної колекції.

```
package com.example.labs.common.patterns.iterator;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ShuffleIterator<T> implements AudioIterator<T> {
    private List<T> shuffledTracks;
    private int currentPosition = -1;

    public ShuffleIterator(List<T> originalTracks) {
        this.shuffledTracks = new ArrayList<>(originalTracks);
        Collections.shuffle(this.shuffledTracks);
    }

    @Override
    public T next() {
        if (hasNext()) {
            currentPosition++;
            return shuffledTracks.get(currentPosition);
        }
        return null;
    }
    // ... (інші методи)
```

```
}
```

Library (Клас "Агрегат/Контекст"). Цей клас виступає "Контекстом" (або Агрегатом) для патерну. Він містить колекцію музичних треків (List<Track> tracks) та приватне поле AudioIterator<Track> iterator для зберігання посилання на поточний алгоритм обходу. Клас надає метод setShuffle(boolean enable), який динамічно змінює тип ітератора (між LinearIterator та ShuffleIterator). При виклику методів відтворення (next(), previous()), "Бібліотека" делегує виконання поточному об'єкту-ітератору.

```
package com.example.labs.common.model;
import com.example.labs.common.patterns.iterator.*;
import java.util.ArrayList;
import java.util.List;

public class Library implements Playable {
    private List<Track> tracks;
    private AudioIterator<Track> iterator;
    private boolean isShuffle = false;

    public Library() {
        this.tracks = new ArrayList<>();
        this.iterator = new LinearIterator<>(this.tracks);
    }

    private void updateIterator() {
        if (isShuffle) {
            this.iterator = new ShuffleIterator<>(this.tracks);
        } else {
            this.iterator = new LinearIterator<>(this.tracks);
        }
    }

    @Override
    public int next() {
        if (iterator.hasNext()) {
            Track t = iterator.next();
            return t != null ? t.getTrackID() : -1;
        }
        return -1;
    }
    // ... (інші методи)
}
```

Висновок: У ході лабораторної роботи я ознайомився з такими патернами, як Singleton, Ітератор, Проксі, Стан та Стратегія. Особливу увагу я приділив реалізації шаблону Iterator, 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

Контрольні запитання

1. Що таке шаблон проєктування?

Це формалізований, перевірений часом опис вдалого рішення типової проблеми, що часто зустрічається при проєктуванні програмних систем

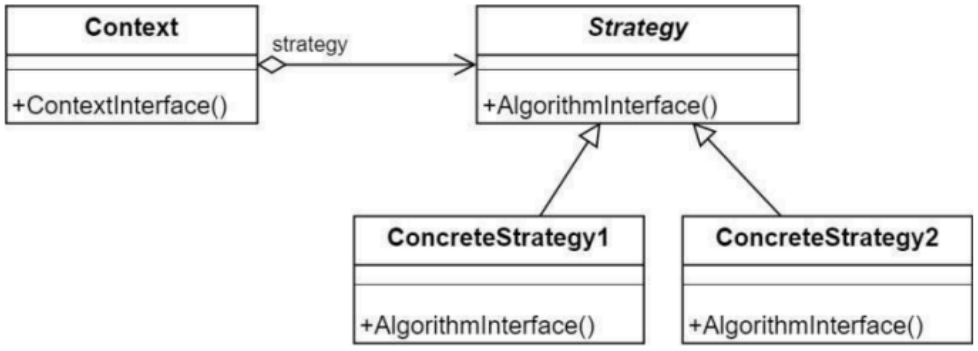
2. Навіщо використовувати шаблони проєктування?

Вони надають спільний словник для розробників, підвищують стійкість системи до змін, спрощують інтеграцію та дозволяють повторно використовувати перевірені архітектурні рішення.

3. Яке призначення шаблону «Стратегія»?

Дозволяє визначати сімейство алгоритмів, інкапсулювати кожен з них і робити їх взаємозамінними. Це дає змогу змінювати алгоритм незалежно від клієнтського коду, що його використовує

4. Нарисуйте структуру шаблону «Стратегія».



5. Які класи входять в шаблон «Стратегія», та яка між ними взаємодія?

Strategy (Інтерфейс): Оголошує загальний інтерфейс для всіх алгоритмів.

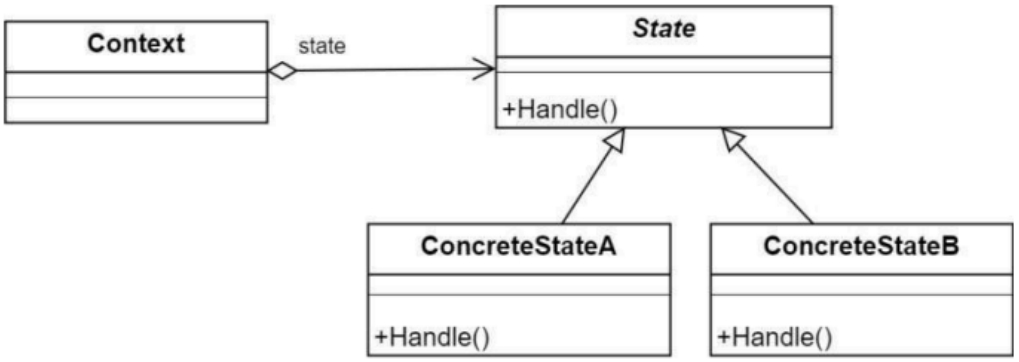
ConcreteStrategy (Класи): Реалізують конкретні алгоритми.

Context (Клас): Містить посилання на об'єкт-стратегію і взаємодіє з ним через загальний інтерфейс Strategy

6. Яке призначення шаблону «Стан»?

Дозволяє об'єкту змінювати свою поведінку при зміні його внутрішнього стану. Зовні це виглядає так, ніби об'єкт змінив свій клас.

7. Нарисуйте структуру шаблону «Стан».



8. Які класи входять в шаблон «Стан», та яка між ними взаємодія?

State (Інтерфейс): Оголошує інтерфейс для поведінки, пов'язаної зі станом.

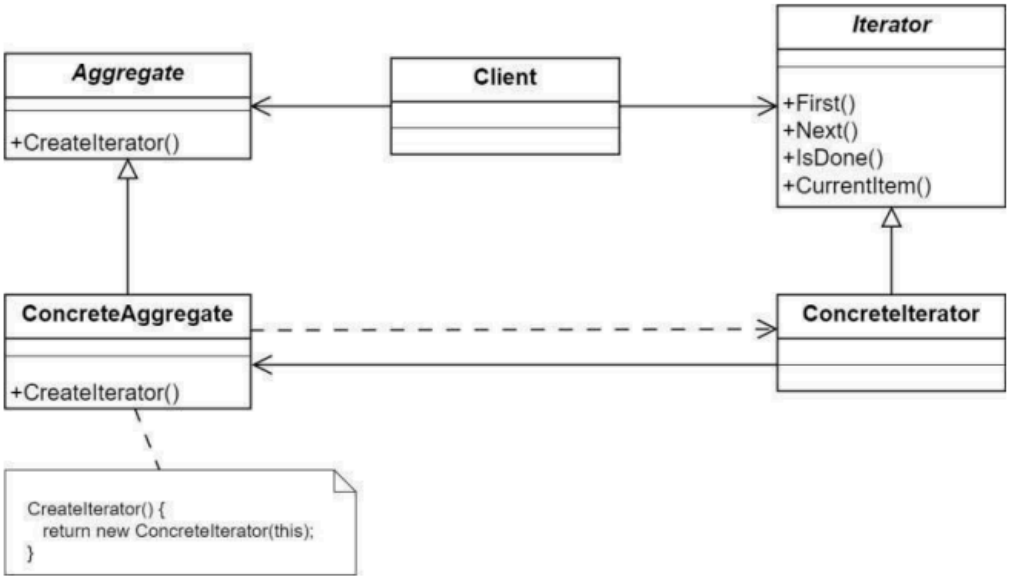
ConcreteState (Класи): Реалізують поведінку для конкретних станів.

Context (Клас): Зберігає посилання на поточний стан і делегує йому виконання роботи. Може змінювати свій стан.

9. Яке призначення шаблону «Ітератор»?

Надає уніфікований спосіб послідовного доступу до елементів колекції (агрегату), не розкриваючи її внутрішньої структури.

10. Нарисуйте структуру шаблону «Ітератор».



11. Які класи входять в шаблон «Ітератор», та яка між ними взаємодія?

Iterator (Інтерфейс): Визначає методи для обходу (hasNext, next).

ConcreteIterator (Клас): Реалізує алгоритм обходу і відстежує поточну позицію.

Aggregate (Інтерфейс): Визначає метод для створення ітератора.

ConcreteAggregate (Клас): Реалізує метод створення ітератора, повертаючи екземпляр ConcreteIterator

12. В чому полягає ідея шаблона «Одинак»? Гарантувати, що клас матиме лише один екземпляр (об'єкт), і надати глобальну точку доступу до цього екземпляра. Клас сам контролює створення екземпляра: приватний конструктор

забороняє створення нових об'єктів ззовні, а статичний метод getInstance() повертає той самий єдиний екземпляр при кожному виклику.

13. Чому шаблон «Одинак» вважають «анти-шаблоном»?

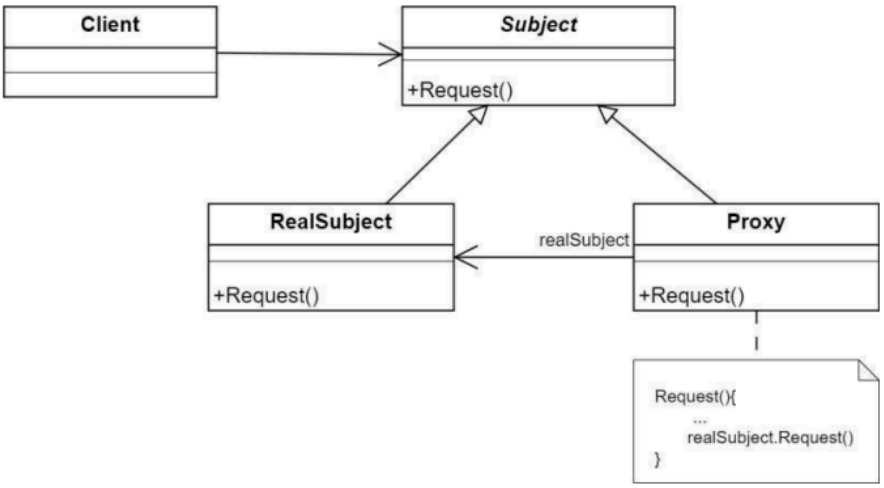
Singleton вважають анти-шаблоном через наступні проблеми:

1. Порушує Single Responsibility Principle – клас відповідає і за свою логіку, і за контроль кількості екземплярів.
2. Ускладнює тестування – складно підмінити Singleton mock-об'єктом для unit-тестів, неможливо створити окремі екземпляри для паралельних тестів.
3. Прихований глобальний стан – створює неявні залежності між класами, що знижує модульність коду.
4. Проблеми з багатопоточністю – потребує додаткової синхронізації для коректної роботи в багатопоточному середовищі.
5. Порушує Dependency Injection – класи безпосередньо звертаються до Singleton, замість отримувати залежності ззовні.

14. Яке призначення шаблону «Проксі»?

Надає об'єкт-замінник (сурогат), який контролює доступ до іншого об'єкта. Проксі має той самий інтерфейс, що й реальний об'єкт, тому клієнт може працювати з ним прозорим чином. Проксі може виконувати додаткову логіку до або після делегування виклику реальному об'єкту.

15. Нарисуйте структуру шаблону «Проксі».



16. Які класи входять в шаблон «Проксі», та яка між ними взаємодія?

Subject (Інтерфейс) – визначає загальний інтерфейс для RealSubject та Proxy. Завдяки йому Proxy може бути використаний замість RealSubject.

RealSubject (Клас) – реальний об'єкт, який виконує основну бізнес-логіку. Містить ресурсомісткі або захищені операції.

Proxy (Клас) – містить посилання на об'єкт RealSubject і реалізує той самий інтерфейс Subject. Перехоплює виклики клієнта, може виконувати додаткову логіку (перевірка прав, логування, кешування), а потім делегує виклик реальному об'єкту.

Взаємодія:

1. Клієнт звертається до Proxy через інтерфейс Subject.
2. Proxy виконує додаткову логіку (наприклад, перевірка прав доступу).
3. Proxy делегує виклик об'єкту RealSubject.
4. RealSubject виконує реальну роботу і повертає результат.
5. Proxy може обробити результат (кешувати, логувати) і повернути його клієнту.

Клієнт не знає, працює він з Proxy чи з RealSubject – обидва мають однаковий інтерфейс