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

ЛАБОРАТОРНА РОБОТА №2

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Online radio station»

Виконав
Перевірив:

студент групи ІА–33:
Мягкий М.Ю.

Якименко Ярослав

Київ 2025

Зміст

Вступ

У сучасній розробці програмного забезпечення використання патернів проєктування (Design Patterns) є ключовим фактором для створення гнучких, масштабованих та зрозумілих систем. Патерни надають уніфіковані рішення для типових архітектурних проблем. У даній лабораторній роботі розглядаються структурні та поведінкові патерни, такі як Singleton, Proxy, Strategy, State та Iterator. Основна увага приділяється практичному застосуванню патерну Strategy в рамках проєкту «Онлайн-радіостанція» для реалізації гнучкого механізму обробки аудіофайлів.

Мета роботи

Вивчити структуру та принципи роботи шаблонів проєктування «Singleton», «Iterator», «Proxy», «State», «Strategy» та набути практичних навичок їх застосування при реалізації програмної системи.

Завдання роботи

1. Опрацювати теоретичні відомості щодо обраних патернів проєктування.
2. Спроєктувати архітектуру модуля обробки даних, використовуючи принципи об'єктно-орієнтованого програмування.
3. Реалізувати патерн «Strategy» для системи обробки медіафайлів (валідація, вилучення метаданих, обробка обкладинок).
4. Розробити не менше трьох класів (стратегій), що імплементують спільний інтерфейс.
5. Підготувати звіт, що містить діаграму класів реалізованого патерну та лістинг програмного коду.

Теоретичні відомості

Патерн проєктування — це формалізоване та багаторазово використовуване рішення для типових проблем, що виникають у процесі розробки ПЗ. Він являє собою не готовий код, а опис взаємодії класів та об'єктів. Використання патернів дозволяє уніфікувати термінологію серед розробників та підвищити адаптивність системи до змін.

Розглянемо ключові патерни, що вивчаються в роботі:

1. Шаблон «Singleton» (Одинак) Цей породжуючий патерн гарантує, що клас матиме лише один екземпляр, і надає до нього глобальну точку доступу.

- Застосування: Контроль доступу до спільних ресурсів (наприклад, підключення до бази даних або файл конфігурації).
- Особливість: Порушує принцип єдиної відповідальності (SRP), оскільки клас сам контролює створення свого екземпляра.'

2. Шаблон «Iterator» (Ітератор) Поведінковий патерн, який дає змогу послідовно обходити елементи складної колекції, не розкриваючи її внутрішньої реалізації (список, стек, дерево).

- Принцип: Логіка перебору виноситься в окремий об'єкт-ітератор, що

дозволяє мати кілька варіантів обходу однієї колекції одночасно.

3. Шаблон «Proxy» (Замісник) Структурний патерн, що надає об'єкт-замінник для керування доступом до іншого об'єкта. Проксі перехоплює виклики до оригінального об'єкта, дозволяючи виконати щось до або після передачі запиту (наприклад, логування, перевірка прав доступу, кешування).

- Приклад: "Ліниве" завантаження важких зображень на веб-сторінках (відображення заглушки, поки вантажиться оригінал).

4. Шаблон «State» (Стан) Поведінковий патерн, що дозволяє об'єкту змінювати свою поведінку при зміні внутрішнього стану. Ззовні це виглядає так, ніби об'єкт змінив свій клас.

- Реалізація: Поведінка, залежна від стану, виноситься в окремі класи, що реалізують спільний інтерфейс. Контекст просто зберігає посилання на поточний об'єкт-стан і делегує йому роботу.

5. Шаблон «Strategy» (Стратегія) Поведінковий патерн, який визначає сімейство схожих алгоритмів, поміщає кожен з них у власний клас і робить їх взаємозамінними.

- Призначення: Дозволяє обирати або змінювати алгоритм роботи "на льоту" залежно від умов, не змінюючи код, який використовує цей алгоритм.

- Відмінність від State: Стратегія використовується, коли є різні способи зробити одну й ту саму дію (наприклад, різні алгоритми сортування або способи обробки файлу), і вибір залежить від зовнішніх факторів, а не від внутрішнього стану об'єкта.

Хід роботи

У ході виконання лабораторної роботи було реалізовано патерн проєктування Strategy для системи обробки медіафайлів. Метою

використання цього патерну є послідовна обробка об'єктів TrackEntity різними алгоритмами (стадіями) без жорсткої прив'язки клієнтського коду до конкретної реалізації обробника. Такий підхід забезпечує гнучкість системи: ми можемо легко додавати нові етапи аналізу (наприклад, нормалізацію гучності або визначення BPM) без модифікації основного коду сервісу

1. Ознайомлення з теорією та застосування

Патерн Strategy дозволяє визначити сімейство алгоритмів, інкапсулювати кожен із них в окремий клас і зробити їх взаємозамінними. Клієнтський код взаємодіє з об'єктами через спільний інтерфейс, не знаючи деталей реалізації.

У системі «Онлайн-радіостанція» патерн Strategy застосовано для реалізації різних етапів обробки завантаженого файлу:

1. FormatValidationStrategy — перевірка бінарної структури файлу (чи дійсно це MP3, чи не пошкоджений заголовок).
2. MetadataExtractionStrategy — вилучення технічних даних (тривалість, бітрейт, ID3-теги).

Кожна стратегія реалізує спільний інтерфейс ProcessingStrategy. Це дозволяє побудувати конвеєр обробки, де кожен етап є незалежним модулем.

2. Реалізація класів

Для імплементації патерну було розроблено наступну структуру класів :

1. ProcessingStrategy — інтерфейс для всіх стратегій обробки. Містить метод execute(TrackEntity track), який визначає контракт для виконання логіки.
2. FormatValidationStrategy — конкретна реалізація стратегії. Відповідає за валідацію формату (аналог етапу "Build" у CI/CD).
3. MetadataExtractionStrategy — конкретна реалізація стратегії. Відповідає за аналіз вмісту файлу (аналог етапу "Test").

4. StrategyResolver — допоміжний сервіс, який отримує зі Spring-контейнера потрібну реалізацію стратегії за ключем (типом задачі) і передає її виконавцю.
5. ProcessingContext — контекст, що зберігає посилання на поточну стратегію та делегує їй виконання роботи.
6. TrackProcessor — головний сервіс-оркестратор. Він отримує повідомлення про новий трек з черги RabbitMQ, визначає необхідні кроки (Validation → Metadata) і послідовно запускає їх через контекст.

3. Послідовність виконання

Логіка роботи системи з використанням патерну виглядає наступним чином:

1. Трек завантажується, отримує статус PROCESSING і його ID потрапляє в чергу.
2. TrackProcessor отримує задачу.
3. Спочатку викликається FormatValidationStrategy. Якщо файл пошкоджено — процес зупиняється, статус змінюється на ERROR.
4. Якщо валідація успішна, контекст перемикається на MetadataExtractionStrategy. Відбувається зчитування тривалості та бітрейту.
5. Після успішного завершення всіх стратегій трек отримує статус READY і стає доступним для ефіру.

4. Використані класи та взаємодія

Детальний опис ролей основних класів у реалізованій архітектурі :

1. TrackEntity Модель даних, що представляє музичний трек. Містить поля для метаданих та статус (PROCESSING, READY, ERROR). Цей об'єкт передається між стратегіями, і кожна з них може зчитувати його дані

або оновлювати їх (наприклад, записувати знайдену тривалість).

2. ProcessingStrategy (Interface) Визначає єдиний метод execute. Це дозволяє ProcessingContext працювати з будь-якою реалізацією (чи то валідація, чи аналіз), дотримуючись принципу поліморфізму.
3. Concrete Strategies Класи FormatValidationStrategy та MetadataExtractionStrategy містять інкапсульовану бізнес-логіку. Вони не знають один про одного, що зменшує зв'язність коду.
4. ProcessingContext Клас, що грає роль "Контексту" в термінології патерну. Він дозволяє динамічно підміняти активну стратегію під час виконання програми, не створюючи нових об'єктів сервісів.

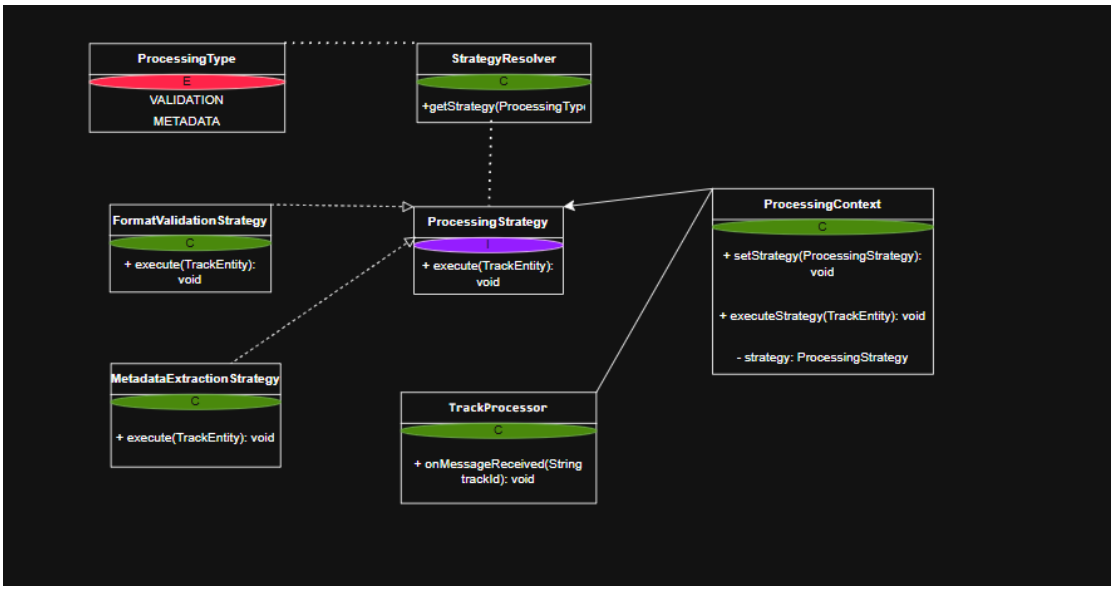

Рис 1.1 Діаграмма класів системи

5. Код системи який реалізує паттерн Strategy

ProcessingStrategy.java Інтерфейс (Strategy), що визначає спільний контракт для всіх алгоритмів обробки.

```java
package com.example.radio.service.processor.strategy;

import com.example.radio.model.TrackEntity;

public interface ProcessingStrategy {
    void execute(TrackEntity track);
}
```

FormatValidationStrategy.java Конкретна стратегія (ConcreteStrategy A). Відповідає за перевірку формату файлу.

```java
package com.example.radio.service.processor.strategy;

import com.example.radio.model.TrackEntity;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;

@Slf4j
@Component("VALIDATION")
public class FormatValidationStrategy implements ProcessingStrategy {

    @Override
    public void execute(TrackEntity track) {
        log.info("Strategy: Validating format for track {}", track.getId());

        // Симуляція перевірки формату
        boolean isValid = true; // У реальності тут перевірка заголовків файлу

        if (isValid) {
            log.info("Format validation passed.");
        } else {
            log.error("Invalid format.");
            track.setStatus("ERROR");
            throw new RuntimeException("Validation failed");
        }
    }
}
```

MetadataExtractionStrategy.java Конкретна стратегія (ConcreteStrategy B). Відповідає за витягування метаданих (тривалість, бітрейт).

```java
package com.example.radio.service.processor.strategy;

import com.example.radio.model.TrackEntity;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;

@Slf4j
@Component("METADATA")
public class MetadataExtractionStrategy implements ProcessingStrategy {

    @Override
    public void execute(TrackEntity track) {
        log.info("Strategy: Extracting metadata for track {}", track.getId());

        // Симуляція аналізу файлу
        int duration = 345; // Наприклад, 5 хвилин 45 секунд
        String bitrate = "320kbps";

        track.setDuration(duration);
        log.info("Metadata extracted: Duration={}, Bitrate={}", duration, bitrate);
    }
}
```

ProcessingContext.java Контекст (Context). Зберігає посилання на поточну стратегію і делегує їй виконання роботи.

```java
package com.example.radio.service.processor;

import com.example.radio.model.TrackEntity;
import com.example.radio.service.processor.strategy.ProcessingStrategy;
import lombok.Setter;
import org.springframework.stereotype.Service;

@Setter
@Service
public class ProcessingContext {

    private ProcessingStrategy strategy;

    public void executeStrategy(TrackEntity track) {
        if (strategy == null) {
            throw new IllegalStateException("Strategy not set");
        }
        strategy.execute(track);
    }
}
```

StrategyResolver.java Допоміжний клас для отримання потрібної стратегії зі

Spring Context за типом.

```java
package com.example.radio.service.processor;

import com.example.radio.service.processor.strategy.ProcessingStrategy;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import java.util.Map;

@Service
@RequiredArgsConstructor
public class StrategyResolver {

    // Spring автоматично ін'єкує всі біни, що імплементують інтерфейс, у Map
    private final Map<String, ProcessingStrategy> strategies;

    public ProcessingStrategy getStrategy(ProcessingType type) {
        return strategies.get(type.name());
    }
}
```

ProcessingType.java Перерахування типів стратегій.

```java
package com.example.radio.service.processor;

public enum ProcessingType {
    VALIDATION,
    METADATA
}
```

TrackProcessor.java Клас-клієнт, який використовує контекст і резолвер для виконання послідовності дій.

```java
package com.example.radio.service.processor;

import com.example.radio.config.RabbitConfig;
import com.example.radio.model.TrackEntity;
import com.example.radio.service.TrackService;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Service;

@Slf4j
@Service
@RequiredArgsConstructor
public class TrackProcessor {

    private final TrackService trackService;
    private final ProcessingContext context;
    private final StrategyResolver resolver;

    @RabbitListener(queues = RabbitConfig.TRACK_PROCESSING_QUEUE)
    public void onMessageReceived(String trackId) {
        TrackEntity track = trackService.findById(trackId);

        try {
            // Крок 1: Встановлюємо стратегію валідації
            context.setStrategy(resolver.getStrategy(ProcessingType.VALIDATION));
            context.executeStrategy(track);

            // Крок 2: Встановлюємо стратегію метаданих
            context.setStrategy(resolver.getStrategy(ProcessingType.METADATA));
            context.executeStrategy(track);

            // Фіналізація
            track.setStatus("READY");
            trackService.save(track);
            log.info("Track processing completed successfully.");

        } catch (Exception e) {
            log.error("Processing failed", e);
            track.setStatus("ERROR");
            trackService.save(track);
        }
    }
}
```

6. Питання до лабораторної роботи

1. Що таке шаблон проєктування? Шаблон проєктування (Design Pattern) — це перевірене часом рішення типових проблем у розробці програмного забезпечення. Він описує структуру класів та їхню взаємодію для досягнення певної функціональності, але не є готовим кодом, а скоріше «ескізом» архітектурного рішення.

2. Навіщо використовувати шаблони проєктування? Використання патернів дозволяє створювати гнучкі та масштабовані системи, зменшувати

дублювання коду та підвищувати його повторне використання. Також це уніфікує термінологію між розробниками, що полегшує командну роботу та підтримку проєкту.

3. Призначення шаблону «Стратегія» Цей шаблон дозволяє винести набір алгоритмів у власні класи та робити їх взаємозамінними. Це дає змогу динамічно змінювати поведінку об'єкта (алгоритм) під час виконання програми, не змінюючи код самого класу, який цей алгоритм використовує .

4. Структура шаблону «Стратегія» Основна структура включає Контекст (Context), який взаємодіє з інтерфейсом Стратегії (Strategy). Конкретні алгоритми реалізуються у класах ConcreteStrategy. Контекст делегує виконання роботи об'єкту стратегії, не знаючи деталей її реалізації .

5. Класи та взаємодія («Стратегія»)

- Strategy — інтерфейс, що визначає метод для виконання алгоритму.
- ConcreteStrategy — конкретна реалізація алгоритму (у нашому випадку FormatValidationStrategy, MetadataExtractionStrategy).
- Context — клас, що зберігає посилання на стратегію і викликає її методи (ProcessingContext) .

6. Призначення шаблону «Стан» Дозволяє об'єкту змінювати свою поведінку залежно від його внутрішнього стану. Ззовні здається, що об'єкт змінив свій клас. Це допомагає уникнути великих умовних операторів (if-else, switch) у коді .

7. Структура шаблону «Стан» Контекст зберігає посилання на поточний об'єкт-стан (State). Усі залежні від стану методи делегуються цьому об'єкту. Для зміни стану контекст просто замінює об'єкт-стан на інший.

8. Класи та взаємодія («Стан»)

- State — інтерфейс стану.
- ConcreteState — класи, що реалізують поведінку для конкретного стану.
- Context — об'єкт, поведінка якого змінюється динамічно.

9. Призначення шаблону «Ітератор» Цей патерн дає змогу послідовно обходити елементи складної колекції (списку, дерева, графа) без розкриття її

внутрішньої структури клієнту .

10. Структура шаблону «Ітератор» Колекція (Aggregate) має метод для створення Ітератора. Сам Ітератор визначає інтерфейс для переміщення по елементах (next, hasNext).

11. Класи та взаємодія («Ітератор»)

- terator — інтерфейс для доступу та обходу елементів.
- ConcreteIterator — реалізує логіку обходу конкретної колекції.
- Aggregate — інтерфейс для створення ітератора.
- ConcreteAggregate — конкретна колекція, що повертає свій ітератор.

12. Ідея шаблону «Одинак» (Singleton) Гарантує, що клас має лише один екземпляр, і надає глобальну точку доступу до нього. Використовується для спільних ресурсів, таких як підключення до БД або конфігурація.

13. Чому «Одинак» вважають «анти-шаблоном»? Він вводить глобальний стан у програму, що ускладнює тестування (моки), порушує принцип єдиної відповідальності та створює сильну зв'язність між компонентами.

14. Призначення шаблону «Проксі» Надає об'єкт-замінник для іншого об'єкта, щоб контролювати доступ до нього. Це корисно для "лінивої" ініціалізації, логування, кешування або перевірки прав доступу .

15. Структура шаблону «Проксі» Клієнт працює з інтерфейсом Subject. Proxy та RealSubject імплементують цей інтерфейс. Проксі отримує запит від клієнта, виконує проміжні дії і передає запит Реальному Суб'єкту .

16. Класи та взаємодія («Проксі»)

- Subject — спільний інтерфейс.
- RealSubject — реальний об'єкт, що виконує корисну роботу.
- Proxy — контролює доступ до RealSubject