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 №4

«Вступ до паттернів проектування»

Виконав:
студент групи ІА–32
Лось Ярослав

Київ 2025

Мета: Вивчити структуру шаблонів «Singleton», «Iterator», «Proxy», «State», «Strategy» та навчитися застосовувати їх в реалізації програмної системи.

Зміст

Теоретичні відомості

Будь-який патерн проєктування, використовуваний при розробці інформаційних систем, являє собою формалізований опис, який часто зустрічається в завданнях проєктування, вдале рішення даної задачі, а також рекомендації по застосуванню цього рішення в різних ситуаціях [5]. Крім того, патерн проєктування обов’язково має загальновживане найменування. Правильно сформульований патерн проєктування дозволяє, відшукавши одного разу вдале рішення, користуватися ним знову і знову. Варто підкреслити, що важливим початковим етапом при роботі з патернами є адекватне моделювання розглянутої предметної області. Це є необхідним як для отримання належним чином формалізованої постановки задачі, так і для вибору відповідних патернів проєктування.

Застосування патернів проєктування підвищує стійкість системи до зміни вимог та спрощує неминуче подальше доопрацювання системи. Крім того, важко переоцінити роль використання патернів при інтеграції інформаційних систем організації. Також слід зазначити, що сукупність патернів проєктування, по суті, являє собою єдиний словник проєктування, який, будучи уніфікованим засобом, незамінний для спілкування розробників один одним. Таким чином шаблони представляють собою, підтверджені роками розробок в різних компаніях і на різних проєктах, «ескізи» архітектурних рішень, які зручно застосовувати у відповідних обставинах.

Шаблон «Iterator»

Призначення: «Iterator» (Ітератор) являє собою шаблон реалізації об'єкта доступу до набору (колекції, агрегату) елементів без розкриття внутрішніх механізмів реалізації. Ітератор виносить функціональність перебору колекції елементів з самої колекції, таким чином досягається розподіл обов'язків: колекція відповідає за зберігання даних, ітератор – за прохід по колекції

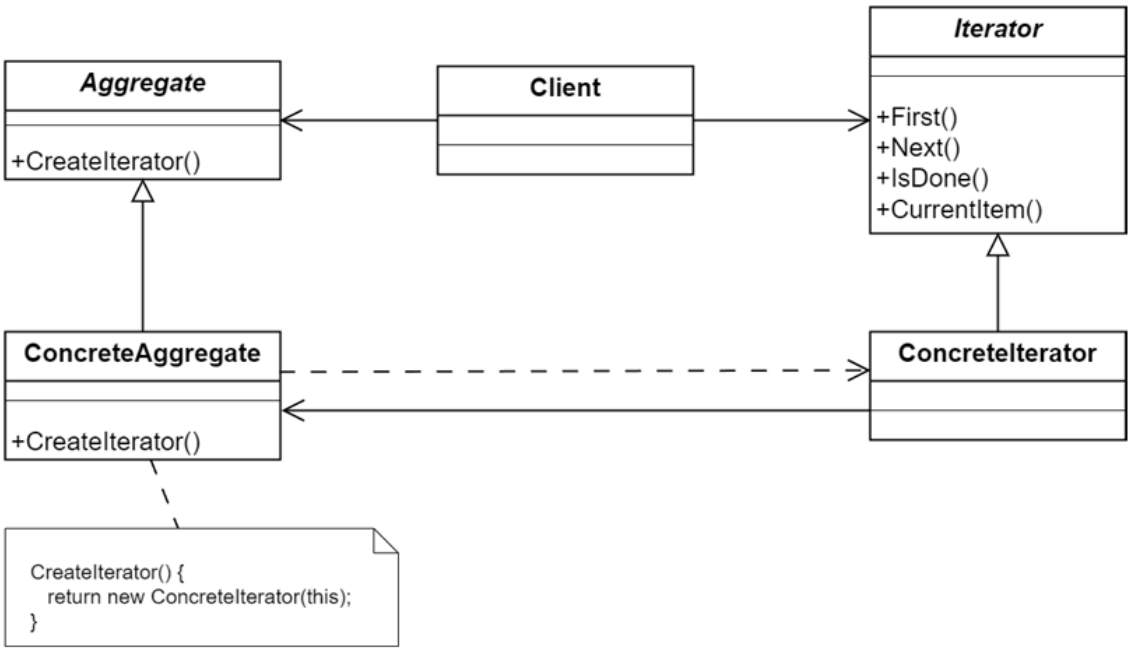


Рисунок 4.2. Структура патерну Ітератор

При цьому алгоритм ітератора може змінюватися – при необхідності пройти в зворотньому порядку використовується інший ітератор. Можливо також написання такого ітератора, який проходить список спочатку по парних позиціях (2,4,6-й елементи і т.д.), потім по непарних. Тобто, шаблон ітератор дозволяє реалізовувати різноманітні способи проходження по колекції незалежно від виду і способу представлення даних в колекції.

Проблема: Більшість колекцій виглядають як звичайний список елементів.

Але є й екзотичні колекції, побудовані на основі дерев, графів та інших складних структур даних.

Але як би не була структурована колекція, користувач повинен мати можливість послідовно обходити її елементи, щоб виробляти з ними якісь дії. Але яким способом слід переміщатися по складній структурі даних? Наприклад, сьогодні може бути достатнім обхід дерева в глибину, але завтра буде потрібно можливість переміщатися по дереву в ширину. А на наступному тижні і того гірше – знадобиться обхід колекції у випадковому порядку.

Додаючи все нові алгоритми в код колекції, ви потроху розмиваєте її основне завдання, яке полягає в ефективному зберіганні даних.

Рішення: Ідея патерна Ітератор полягає в тому, щоб винести поведінку обходу колекції з самої колекції в окремий клас. Об'єкт-ітератор буде відстежувати стан обходу, поточну позицію в колекції і скільки елементів ще залишилося обійти. Одну і ту ж колекцію зможуть одночасно обходити різні ітератори, а сама колекція не буде навіть знати про це. До того ж, якщо вам знадобиться додати новий спосіб обходу, ви зможете створити окремий клас ітератора, не змінюючи існуючий код колекції.

Шаблонний ітератор містить:

• First() – установка покажчика перебору на перший елемент колекції;

• Next() – установка покажчика перебору на наступний елемент колекції;

• IsDone – булевське поле, яке встановлюється як true коли покажчик перебору досяг кінця колекції;

• CurrentItem – поточний об'єкт колекції.

Переваги та недоліки: Цей шаблон дозволяє уніфікувати операції проходження по наборам об'єктів для всіх наборів. Тобто, незалежно від реалізації (масив, зв'язаний список, незв'язаний список, дерево та ін.), кожен з наборів може використовувати будь-який з реалізованих ітераторів.

+ Дозволяє реалізувати різні способи обходу структури даних.

+ Спрощує класи зберігання даних.

- Не виправданий, якщо можна обійтися простим циклом.

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 аудіоформатів, еквалайзер.

1. Загальний опис роботи

Основна мета полягає в тому, щоб реалізувати механізм послідовного (Sequential) та довільного (Shuffle) обходу колекції музичних треків без розкриття внутрішньої структури цієї колекції.

В контексті нашої теми, використаємо паттерн **Iterator**, який дозволить винести логіку перебору елементів (пісень) з класу-зберігача (Playlist) в окремі класи-ітератори.

Такий підхід дозволить мати єдину колекцію даних, але різні способи її обходу (по порядку, випадково), не змінюючи код самого плейлиста.

Програмна система складається з наступних компонентів:

- **Інтерфейс ітератора (MusicIterator)**: визначає стандартні методи для обходу (first, next, isDone, currentItem).
- **Інтерфейс агрегату (Aggregate)**: визначає контракт для створення ітератора.
- **Клас даних (Song)**: представляє окремий музичний трек.
- **Конкретний агрегат (Playlist)**: зберігає список пісень та створює екземпляри конкретних ітераторів.
- **Конкретні ітератори (SequentialIterator, ShuffleIterator)**: реалізують специфічні алгоритми обходу.
- **Клієнт (MusicPlayerClient)**: демонструє роботу системи.

2. Опис класів програмної системи

Клас Song та інтерфейси

Опис:

Клас **Song** є базовим елементом даних. Інтерфейс **MusicIterator** визначає контракт, який мають реалізувати всі варіанти обходу (послідовний, випадковий), забезпечуючи поліморфізм для клієнтського коду.

Характеристики:

- Song містить поля title та artist, а також перевизначений метод toString() для зручного виводу.
- MusicIterator оголошує методи навігації: first() (старт), next() (крок), isDone() (перевірка кінця), currentItem() (отримання об'єкта).
- Aggregate декларує фабричний метод createIterator().

Код класу Song.java

```
public class Song {
    private String title;
    private String artist;

    public Song(String title, String artist) {
        this.title = title;
        this.artist = artist;
    }

    public String getTitle() {
        return title;
    }

    @Override
    public String toString() {
        return title + " - " + artist;
    }
}
```

Код інтерфейсу MusicIterator

```
public interface MusicIterator {
    void first();
    void next();
    boolean isDone();
    Song currentItem();
}
```

Код інтерфейсу Aggregate

```
public interface Aggregate {
    MusicIterator createIterator();
}
```

Клас Playlist

Опис:

Клас **Playlist** виступає в ролі конкретного агрегату (ConcreteAggregate). Він відповідає за зберігання об'єктів пісень, але не містить логіки їх перебору.

Характеристики:

- Використовує ArrayList<Song> для внутрішнього зберігання даних.
- Реалізує методи управління колекцією: addSong(), getSong(), size().
- Виступає фабрикою ітераторів:
  - createIterator() повертає стандартний SequentialIterator.
  - createShuffleIterator() повертає ShuffleIterator для випадкового відтворення.

Код класу Playlist

```
public class Playlist implements Aggregate {
    private List<Song> songs = new ArrayList<>();

    public void addSong(Song song) {
        songs.add(song);
    }

    public int size() {
        return songs.size();
    }

    public Song getSong(int index) {
        return songs.get(index);
    }

    @Override
    public MusicIterator createIterator() {
        return new SequentialIterator(this);
    }

    public MusicIterator createShuffleIterator() {
        return new ShuffleIterator(this);
    }
}
```

Клас SequentialIterator

Опис:

Це класична реалізація ітератора, що проходить по списку від першого до останнього елемента.

Характеристики:

- Зберігає посилання на екземпляр Playlist та поточний індекс current.
- Метод first() скидає індекс на 0.
- Метод next() інкрементує індекс на одиницю.
- Забезпечує прямий доступ до елементів колекції в порядку їх додавання.

Код класу SequentialIterator

```
public class SequentialIterator implements MusicIterator {
    private Playlist playlist;
    private int current;

    public SequentialIterator(Playlist playlist) {
        this.playlist = playlist;
        this.current = 0;
    }

    @Override
    public void first() {
        current = 0;
    }

    @Override
    public void next() {
        current++;
    }

    @Override
    public boolean isDone() {
        return current >= playlist.size();
    }

    @Override
    public Song currentItem() {
        if (isDone()) return null;
        return playlist.getSong(current);
    }
}
```

Клас ShuffleIterator

Опис:

Цей клас реалізує логіку перемішування (Shuffle). Головна особливість — він не змінює реальний порядок пісень у плейлисті, а створює "віртуальний" перемішаний порядок.

Ключові характеристики:

- При ініціалізації (first()) створює допоміжний список цілих чисел shuffledIndices, що відповідають індексам пісень.
- Використовує Collections.shuffle() для рандомізації цього допоміжного списку.
- Метод currentItem() використовує не прямий індекс, а значення з перемішаного масиву: playlist.getSong(shuffledIndices.get(current)).
- Це дозволяє обходити одну й ту ж колекцію різними ітераторами одночасно, не ламаючи порядок для інших.

Код класу ShuffleIterator

```
public class ShuffleIterator implements MusicIterator {
    private Playlist playlist;
    private int current;
    private List<Integer> shuffledIndices;

    public ShuffleIterator(Playlist playlist) {
        this.playlist = playlist;
        first();
    }

    @Override
    public void first() {
        current = 0;
        shuffledIndices = new ArrayList<>();
        for (int i = 0; i < playlist.size(); i++) {
            shuffledIndices.add(i);
        }
        Collections.shuffle(shuffledIndices);
    }

    @Override
    public void next() {
        current++;
    }
```

```
    @Override
    public boolean isDone() {
        return current >= shuffledIndices.size();
    }

    @Override
    public Song currentItem() {
        if (isDone()) return null;
        int realIndex = shuffledIndices.get(current);
        return playlist.getSong(realIndex);
    }
}
```

Клас MusicPlayerClient

Опис:

Клас виступає в ролі Клієнта. Його задача продемонструвати роботу патерну, створивши плейлист та запустивши його програвання у двох різних режимах.

Характеристики:

- Створює об'єкт Playlist і наповнює його тестовими даними.
- Спочатку отримує стандартний ітератор через createIterator() і виводить пісні по черзі.
- Потім отримує ітератор перемішування через createShuffleIterator() і демонструє випадковий порядок.
- Клієнтський код (цикли for) виглядає ідентично для обох випадків, оскільки працює через абстракцію MusicIterator.

Код класу MusicPlayerClient

```
public class MusicPlayerClient {
    public static void main(String[] args) {
        Playlist myPlaylist = new Playlist();
        myPlaylist.addSong(new Song("Billie Jean", "Michael Jackson"));
        myPlaylist.addSong(new Song("Imagine", "John Lennon"));
        myPlaylist.addSong(new Song("Smells Like Teen Spirit", "Nirvana"));
        myPlaylist.addSong(new Song("Shape of My Heart", "Sting"));

        System.out.println("--- Normal Playback ---");
        MusicIterator normalIterator = myPlaylist.createIterator();

        for (normalIterator.first(); !normalIterator.isDone();
normalIterator.next()) {
```

```
            System.out.println("Playing: " + normalIterator.currentItem());
        }

        System.out.println("\n--- Shuffle Playback ---");
        MusicIterator shuffleIterator = myPlaylist.createShuffleIterator();

        for (shuffleIterator.first(); !shuffleIterator.isDone();
shuffleIterator.next()) {
            System.out.println("Playing: " + shuffleIterator.currentItem());
        }

        System.out.println("\n--- Checking original order ---");
        normalIterator.first();
        System.out.println("First song is still: " +
normalIterator.currentItem().getTitle());
    }
}
```

3. Опис результатів коду

Для демонстрації роботи шаблону «Iterator», запустимо наш клас MusicPlayerClient. Програма створить плейлист із 4 пісень та виконає два проходи по ньому.

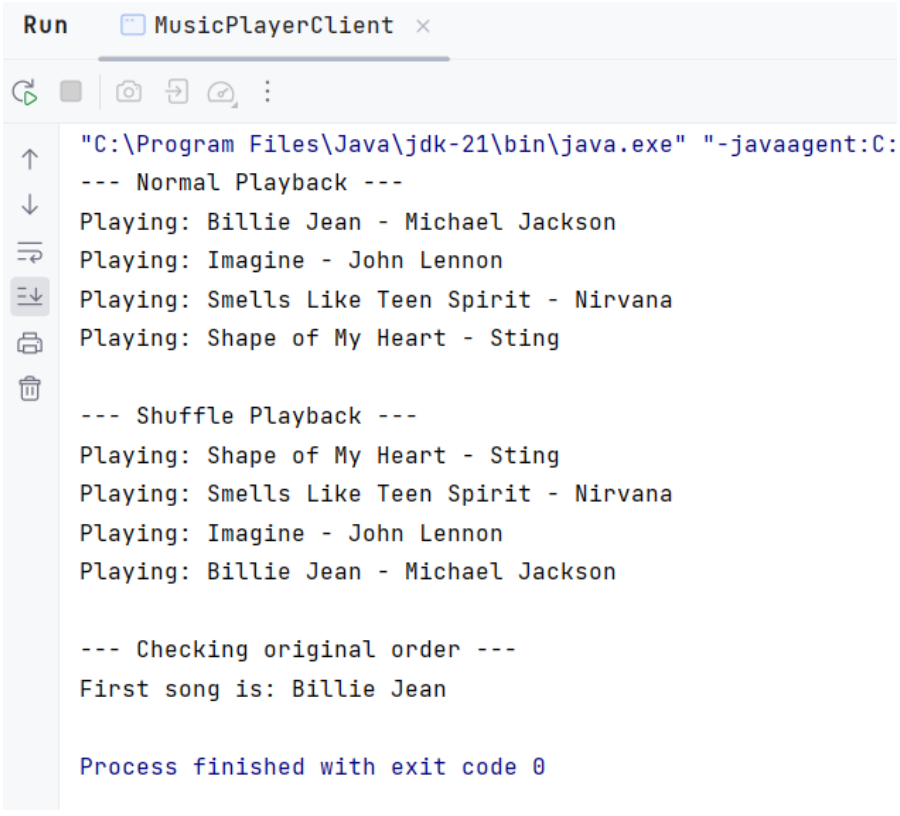


Рисунок 1 – консольний вивід роботи програми

Аналізуючи вивід, можна дійти висновку, що:

- У першому блоці (Normal Playback) пісні відтворюються точно в тому порядку, в якому були додані (індекси 0, 1, 2, 3). Це результат роботи SequentialIterator.
- У другому блоці (Shuffle Playback) ті самі пісні відтворюються у випадковому порядку. Це демонструє роботу ShuffleIterator, який згенерував власну послідовність індексів.
- Остання перевірка підтверджує, що внутрішній стан об'єкта Playlist не змінився - першою піснею все ще залишається "Billie Jean". Це підтверджує безпеку використання ітераторів: кожен ітератор має власний стан обходу і не впливає на саму колекцію.

4. Діаграми класів

Для візуалізації архітектури розробленої системи та взаємозв'язків між її компонентами побудуємо UML-діаграми класів.

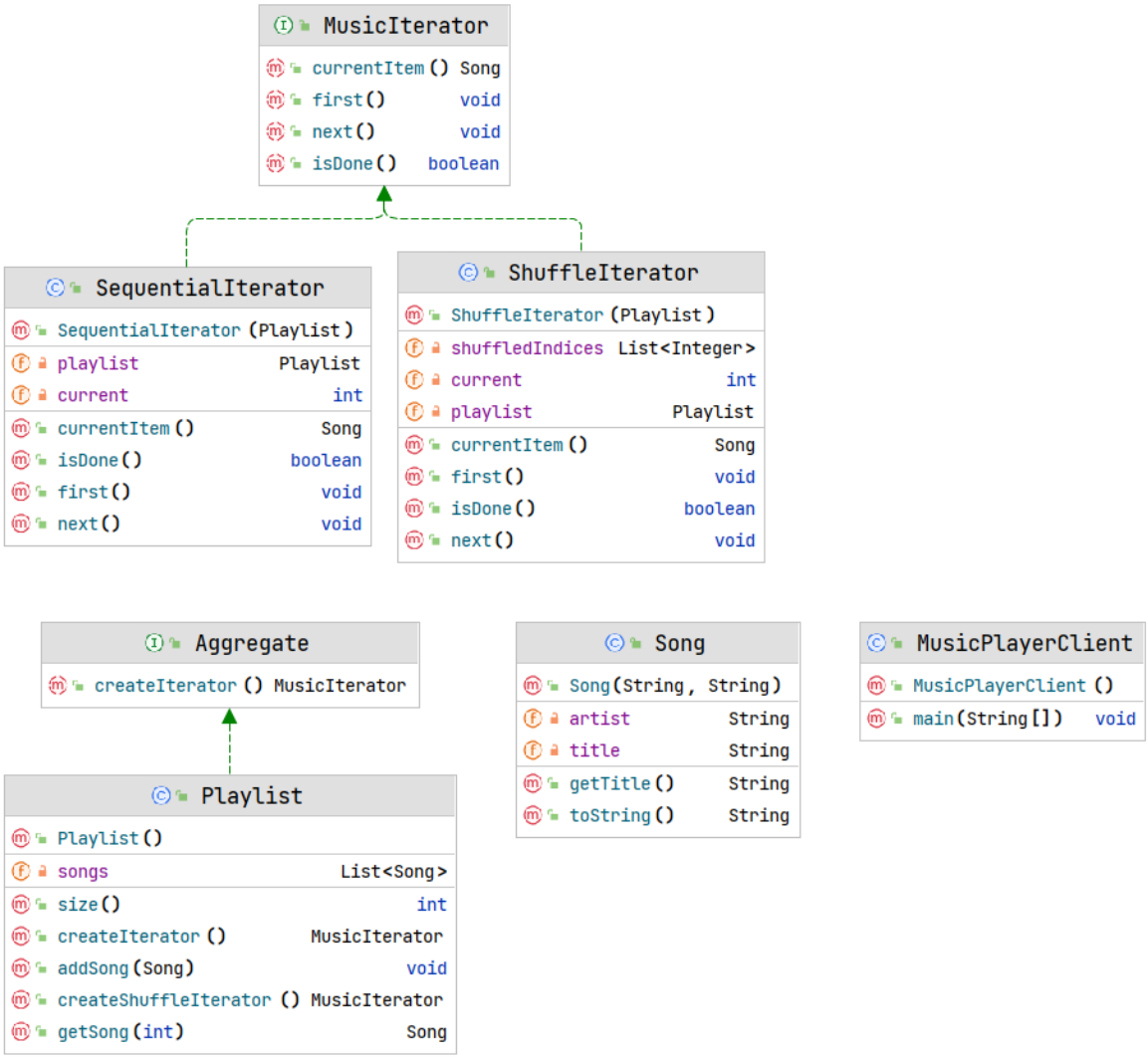


Рисунок 2 – Діаграма класів (спрощена)

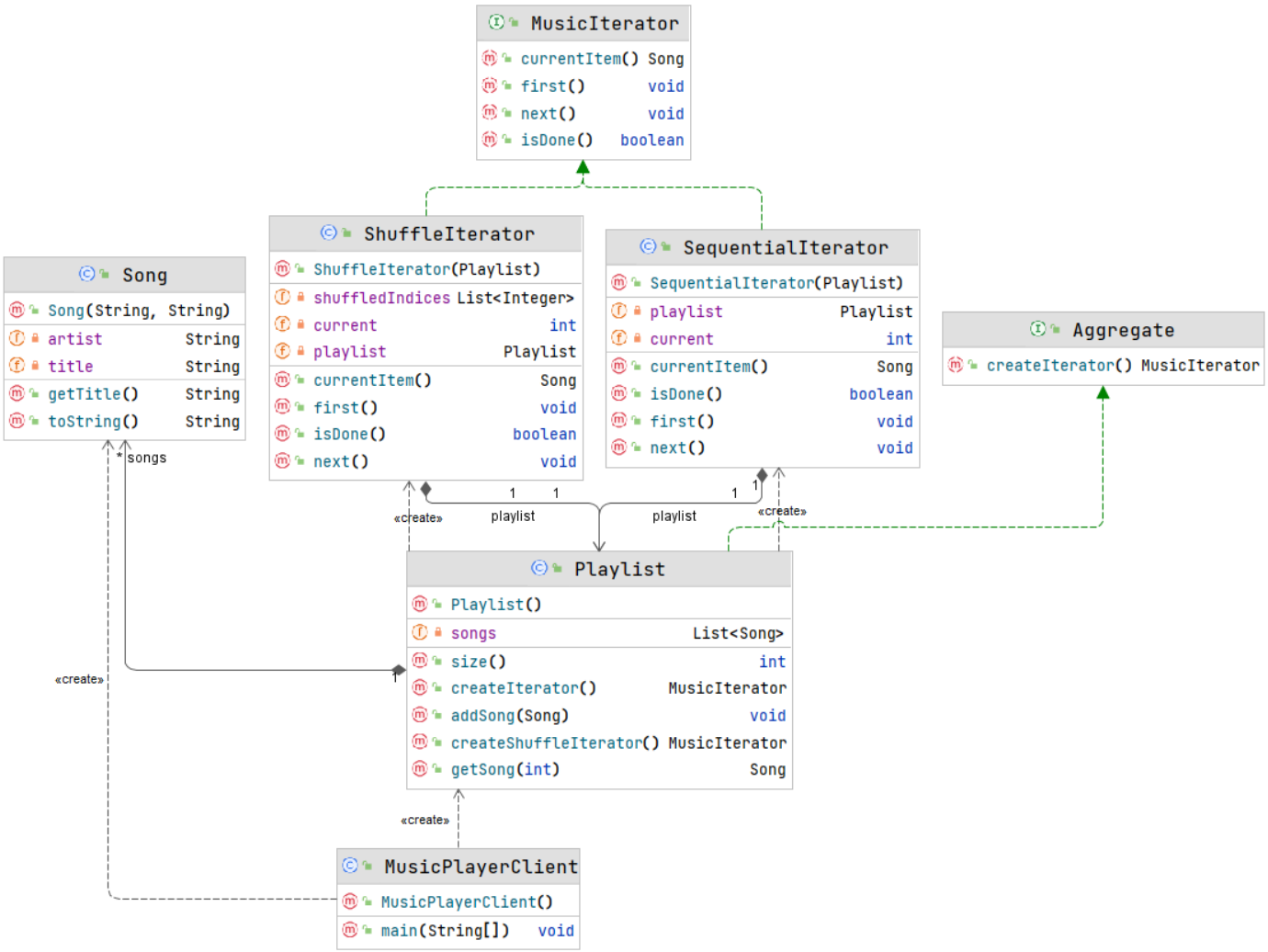


Рисунок 3 – Діаграма класів (повна)

Висновки

В ході виконання даної лабораторної роботи було успішно вивчено та реалізовано патерн проектування «Ітератор» для вирішення задачі керування відтворенням музичних треків у програвачі. Була створена чітка структура взаємодії об’єктів: клас-агрегат Playlist, що відповідає за зберігання даних, та ієрархія ітераторів (SequentialIterator, ShuffleIterator), що відповідають за логіку обходу цих даних. Впровадження інтерфейсу MusicIterator дозволило уніфікувати спосіб доступу до елементів колекції незалежно від обраного режиму відтворення (послідовний чи випадковий). Практична реалізація продемонструвала ключові переваги патерну: можливість мати кілька активних переборів однієї колекції одночасно, спрощення інтерфейсу агрегату та легку розширюваність системи.

Контрольні запитання

1. Що таке шаблон проєктування?

Шаблон проєктування - це перевірене, багаторазово застосовне архітектурне рішення для типової проблеми, що виникає при проєктуванні програмного забезпечення. Це не готовий код, який можна скопіювати, а опис схеми взаємодії класів та об'єктів для вирішення задачі.

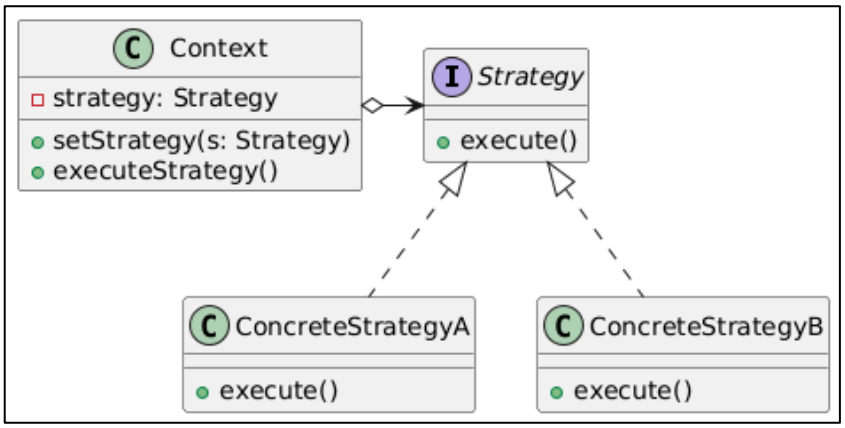
2. Навіщо використовувати шаблони проєктування?

- Перевірені рішення: Вони базуються на досвіді багатьох розробників, що дозволяє уникати типових помилок.
- Спільна мова: Назви патернів дозволяють розробникам швидше розуміти один одного без пояснення деталей коду.
- Гнучкість та розширюваність: Шаблони сприяють написанню коду, який легше підтримувати та модифікувати.
- Уникнення "винахідництва велосипеда": Економія часу на пошук архітектурних рішень.

3. Яке призначення шаблону «Стратегія»?

Шаблон «Стратегія» (Strategy) визначає сімейство алгоритмів, інкапсулює кожен з них в окремий клас і робить їх взаємозамінними. Патерн дозволяє підміняти алгоритми (стратегії) під час виконання програми незалежно від клієнта, який їх використовує.

4. Нарисуйте структуру шаблону «Стратегія».



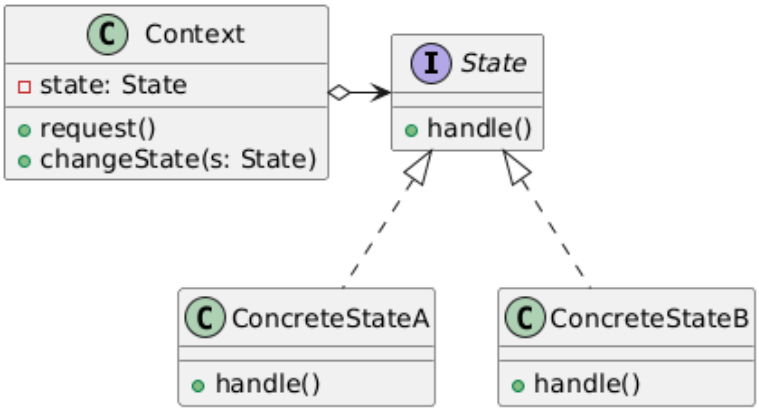
5. Які класи входять в шаблон «Стратегія», та яка між ними взаємодія?

- Context (Контекст): Зберігає посилання на об'єкт Strategy. Спілкується зі стратегією через інтерфейс, не знаючи конкретної реалізації.
- Strategy (Стратегія): Спільний інтерфейс для всіх підтримуваних алгоритмів.
- ConcreteStrategy (Конкретна стратегія): Класи, що реалізують конкретний алгоритм.
- Взаємодія: Клієнт створює об'єкт конкретної стратегії та передає його в Контекст. Контекст делегує виконання роботи об'єкту стратегії.

6. Яке призначення шаблону «Стан»?

Шаблон «Стан» (State) дозволяє об'єкту змінювати свою поведінку при зміні його внутрішнього стану. Ззовні це виглядає так, ніби об'єкт змінив свій клас. Патерн допомагає уникнути громіздких умовних операторів (if, switch) при управлінні станами.

7. Нарисуйте структуру шаблону «Стан».



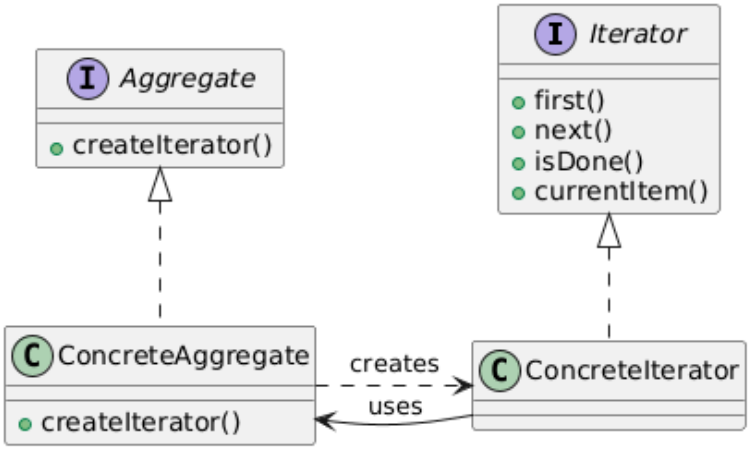
8. Які класи входять в шаблон «Стан», та яка між ними взаємодія?

- Context (Контекст): Об'єкт, поведінка якого залежить від стану. Зберігає посилання на поточний об'єкт State.
- State (Стан): Інтерфейс, що описує методи, які повинні реалізувати всі конкретні стани.
- ConcreteState (Конкретний стан): Реалізує поведінку, специфічну для певного стану контексту.
- Взаємодія: Контекст делегує виклики методів поточному об'єкту стану. Самі стани можуть ініціювати зміну стану в Контексті.

9. Яке призначення шаблону «Ітератор»?

Шаблон «Ітератор» (Iterator) надає спосіб послідовного доступу до елементів агрегатного об'єкта (колекції) без розкриття його внутрішнього представлення (масив, список, дерево тощо).

10. Нарисуйте структуру шаблону «Ітератор».



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

- Iterator (Ітератор): Інтерфейс для доступу та обходу елементів.
- ConcreteIterator: Реалізує інтерфейс, відслідковує поточну позицію обходу.
- Aggregate (Агрегат): Інтерфейс колекції, що має метод для створення ітератора.
- ConcreteAggregate: Конкретна колекція, яка повертає екземпляр сумісного ітератора.
- Взаємодія: Клієнт отримує ітератор від агрегату і використовує його методи (next, currentItem) для перебору, не знаючи про внутрішню структуру колекції.

12. В чому полягає ідея шаблона «Одинак»?

Ідея шаблона «Одинак» (Singleton) полягає в тому, щоб гарантувати, що клас має лише один екземпляр, і надати глобальну точку доступу до цього екземпляра.

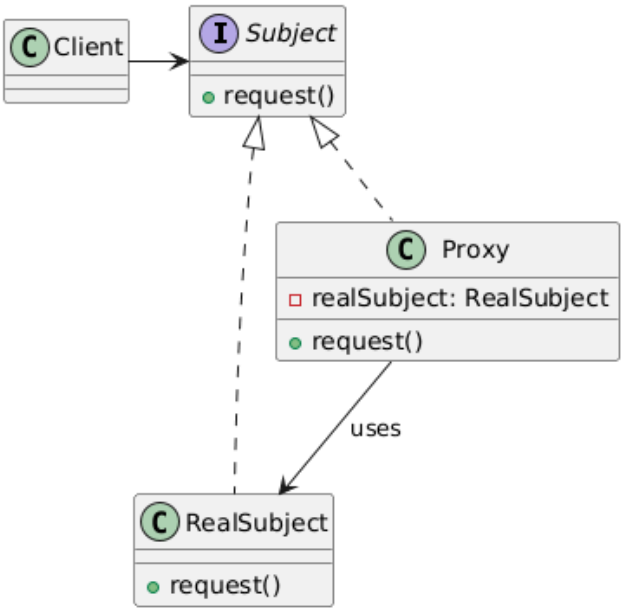
13. Чому шаблон «Одинак» вважають «анти-шаблоном»?

- Глобальний стан: Створює приховані залежності між класами, що ускладнює розуміння коду.
- Проблеми з тестуванням: Важко писати юніт-тести, оскільки Одинак важко замінити на макет (mock) об'єкт, і стан може зберігатися між тестами.
- Порушення SRP: Клас контролює і свою логіку, і своє створення (порушення принципу єдиної відповідальності).
- Проблеми багатопоточності: Потребує складного коду для безпечної ініціалізації в багатопотоковому середовищі.

14. Яке призначення шаблону «Проксі»?

Шаблон «Проксі» (Proxy, Замісник) надає об'єкт-замінник замість реального об'єкта, щоб контролювати доступ до нього. Це дозволяє виконувати певні дії до або після звернення до реального об'єкта (лінива ініціалізація, кешування, перевірка прав доступу, логування).

15. Нарисуйте структуру шаблону «Проксі».



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

- Subject (Суб'єкт): Спільний інтерфейс для RealSubject та Proxy. Дозволяє використовувати Проксі всюди, де очікується Реальний суб'єкт.
- RealSubject (Реальний суб'єкт): Клас, що виконує основну бізнес-логіку.
- Proxy (Замісник): Зберігає посилання на реальний об'єкт. Реалізує той самий інтерфейс.
- Взаємодія: Клієнт звертається до Proxy. Proxy може виконати проміжну роботу (наприклад, перевірити права), створити RealSubject (якщо його ще немає) і передати йому виклик.