



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

Технології розроблення програмного забезпечення
Лабораторна робота №5
ШАБЛОНИ «ADAPTER», «BUILDER», «COMMAND»,
«CHAIN OF RESPONSIBILITY», «PROTOTYPE»

Виконала:
Студентка групи ІА-22
Микитенко Ірина

Перевірив:
Мягкий Михайло Юрійович

Зміст

1. Короткі теоритичні відомості	3
2. Реалізація не менше 3х класів згідно з вибраною темою.	8
3. Реалізація шаблону Command	9
4. Діаграма класів для паттерну Command	13
5. Проблема, яку допоміг вирішити шаблон Command	13
6. Переваги застосування патерну Command	14

Тема: Шаблони «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype»

Мета: Ознайомитися з шаблонами проектування «Adapter», «Builder»,

«

С

о

т

т

а

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 аудіо-форматів, еквалайзер.

а

і

1. Короткі теоритичні відомості

Анти-патерни (anti-patterns) — це погані рішення проблем, що часто використовуються в проектуванні. Вони є протилежністю «шаблонам проектування», які описують хороші практики. Анти-патерни в управлінні та розробці ПЗ включають:

Управління розробкою ПЗ:

- Дим і дзеркала: Демонстрація неповних функцій.
- Роздування ПЗ: Збільшення вимог до ресурсів у наступних версіях.
- Функції для галочки: Наявність непов'язаних функцій для реклами.

Розробка ПЗ:

- Інверсія абстракції: Приховування частини функціоналу.
- Невизначена точка зору: Моделі без чіткої специфікації.
- Великий клубок бруду: Система без чіткої структури.
- Бензинова фабрика: Необов'язкова складність дизайну.
- Затичка на введення даних: Недостатня обробка неправильного введення.

ООП:

- Базовий клас-утиліта: Спадкування замість делегування.

«

Р

г

- Божественний об'єкт: Надмірна концентрація функцій в одному класі.
- Самотність: Надмірне використання патерну Singleton.

Анти-патерни в програмуванні:

- Непотрібна складність: Ускладнення рішень без необхідності.
- Дія на відстані: Несподівані взаємодії між різними частинами системи.
- Накопичити і запустити: Використання глобальних змінних для параметрів.
- Сліпа віра: Недостатня перевірка результатів або виправлень.
- Активне очікування: Використання ресурсів під час очікування події замість асинхронного підходу.
- Кешування помилки: Забуте скидання прапора після обробки помилки.
- Перевірка типу замість інтерфейсу: Перевірка типу замість використання інтерфейсу.
- Кодування шляхом виключення: Обробка випадків через винятки.
- Потік лави: Залишення поганого коду через високу вартість його видалення.
- Спагеті-код: Заплутаний код з некерованим виконанням.

Методологічні анти-патерни:

- Програмування методом копіювання-вставки: Копіювання коду замість створення спільних рішень.
- Дефакторінг: Заміна функціональності документацією.
- Золотий молоток: Впевненість у застосуванні одного рішення для всіх проблем.
- Передчасна оптимізація: Оптимізація без достатньої інформації.
- Винахід колеса: Створення нових рішень замість використання існуючих.
- Винахід квадратного колеса: Погане рішення, коли є хороше.
- Самознищення: Помилка, що призводить до фатальної поведінки програми.

Анти-патерни управління конфігурацією:

- DLL-пекло: Проблеми з версіями і доступністю DLL.
- Пекло залежностей: Проблеми з версіями залежних продуктів.

Організаційні анти-патерни:

- Аналітичний параліч: Надмірний аналіз без реалізації.
- Дійна корова: Неінвестування в розвиток продукту.
- Тривале старіння: Перенесення зусиль на портинг замість нових рішень.
- Скидування витрат: Перенесення витрат на інші відділи.
- Повзуче покращення: Додавання змін, що знижують загальну якість.
- Розробка комітетом: Розробка без централізованого керівництва.
- Ескалація зобов'язань: Продовження хибного рішення.
- Розповзання рамок: Втрата контролю над проектом.
- Замикання на продавці: Залежність від одного постачальника.

Шаблон "Adapter" (Адаптер)

Призначення:

Адаптер використовується для приведення інтерфейсу одного об'єкта до іншого, забезпечуючи сумісність між різними системами.

Проблема:

Існує потреба адаптувати різні формати даних (наприклад, XML до JSON) без зміни вихідного коду.

Рішення:

Створення адаптера, який «перекладає» інтерфейси одного об'єкта для іншого (наприклад, XML_To_JSON_Adapter).

Приклад:

Адаптер для зарядки ноутбука в різних країнах, що дозволяє підключити пристрій до різних типів розеток.

Переваги:

Приховує складність взаємодії різних інтерфейсів від користувача.

Недоліки:

Ускладнює код через додаткові класи.

Шаблон "Builder" (Будівельник)

Призначення:

Шаблон відділяє процес створення об'єкта від його представлення, що зручно для складних або багатформатних об'єктів.

Проблема:

Ускладнене створення об'єкта, наприклад, формування відповіді web-сервера з кількох частин (заголовки, статуси, вміст).

Рішення:

Кожен етап створення об'єкта абстрагується в окремий метод будівельника, що дозволяє контролювати процес побудови.

Приклад:

Будівництво будинку за етапами, де кожен етап виконується окремою командою будівельників.

Переваги:

Забезпечує гнучкість та незалежність від внутрішніх змін у процесі створення.

Недоліки:

Клієнт залежить від конкретних класів будівельників, що може обмежити можливості.

Шаблон "Command" (Команда)

Призначення:

Перетворює виклик методу в об'єкт-команду, що дозволяє гнучко керувати діями (додавати, скасовувати, комбінувати команди).

Проблема:

Як організувати обробку кліків у текстовому редакторі без дублювання коду для різних кнопок.

Рішення:

Створити окремий клас-команду для кожної дії, який буде викликати методи бізнес-логіки через об'єкт інтерфейсу.

Приклад:

Офіціант у ресторані приймає замовлення (команда) і передає кухарю для виконання (отримувач), без прямого контакту між клієнтом і кухарем.

Переваги:

- Знімає залежність між об'єктами, що викликають і виконують операції
- Дозволяє скасовувати, відкладати та комбінувати команди
- Підтримує принцип відкритості/закритості

Недоліки:

- Ускладнює код додатковими класами

Шаблон "Chain of Responsibility"

Призначення:

Ланцюг відповідальності дозволяє організувати обробку запиту, передаючи його послідовно через ланцюг об'єктів, поки не буде знайдено обробник, здатний виконати запит. Це зручно для побудови гнучкої системи обробників, коли важливо зменшити залежність клієнта від обробників і структурувати обробку.

Проблема:

Припустимо, потрібно реалізувати систему обробки онлайн-замовлень, де доступ мають тільки авторизовані користувачі, а адміністратори мають додаткові права. Ці перевірки слід виконувати послідовно. Однак при додаванні нових перевірок код стає перевантаженим умовними операторами, що ускладнює підтримку.

Рішення:

Створити для кожної перевірки окремий клас з методом, який виконує потрібні дії. Далі всі об'єкти-обробники об'єднуються в ланцюг, де кожен обробник має посилання на наступного. Запит передається першому обробнику ланцюга, який або самостійно обробляє його, або передає далі. Якщо обробник не може виконати дію, запит продовжує передаватися далі по ланцюгу, аж поки не знайдеться відповідний обробник або ланцюг закінчиться.

Приклад з життя:

У JavaScript події в браузері можна обробляти на різних рівнях DOM-дерева. Наприклад, для таблиці з рядками, кожен з яких має кнопку для видалення, можна поставити обробник не на кожну кнопку, а на `tbody`, що містить усі рядки. Якщо подія виникає на кнопці, але вона не має обробника, подія "піднімається" по ланцюгу: від кнопки до рядка, потім до `tbody`, що дозволяє зручно обробляти події динамічно.

Переваги:

- Зменшує залежність між клієнтом і обробниками.
- Відповідає принципу єдиного обов'язку.
- Підтримує принцип відкритості/закритості.

Недоліки:

- Запит може залишитися без обробки, якщо жоден обробник його не обробить.

Шаблон "Prototype"**Призначення:**

Шаблон "Prototype" використовується для створення об'єктів шляхом копіювання існуючого шаблонного об'єкта. Це дозволяє спростувати процес

створення об'єктів, коли їх структура заздалегідь відома, та уникати прямого створення нових екземплярів.

Проблема:

Якщо потрібно скопіювати об'єкт, звичайний спосіб — створити новий об'єкт і вручну копіювати його поля. Але деякі частини об'єкта можуть бути приватними, що ускладнює доступ до них і порушує інкапсуляцію.

Рішення:

Шаблон "Prototype" доручає самим об'єктам реалізовувати метод `clone()`, який повертає їх копію. Це дозволяє створювати нові об'єкти без прив'язки до їхніх конкретних класів, залишаючи логіку копіювання всередині класу.

Приклад з життя:

У виробництві перед масовим випуском виготовляються прототипи, що дозволяють протестувати виріб. Ці прототипи служать шаблонами і не беруть участі в подальшому виробництві.

Переваги:

- Дозволяє клонувати об'єкти без прив'язки до конкретного класу.
- Зменшує дублювання коду ініціалізації.
- Прискорює процес створення об'єктів.
- Є альтернативою підкласам при створенні складних об'єктів.

Недоліки:

- Складність клонування об'єктів, що містять посилання на інші об'єкти.

2. Реалізація не менше 3х класів згідно з вибраною темою.

Структура проекту з реалізованими класами зображена на рисунку 1.

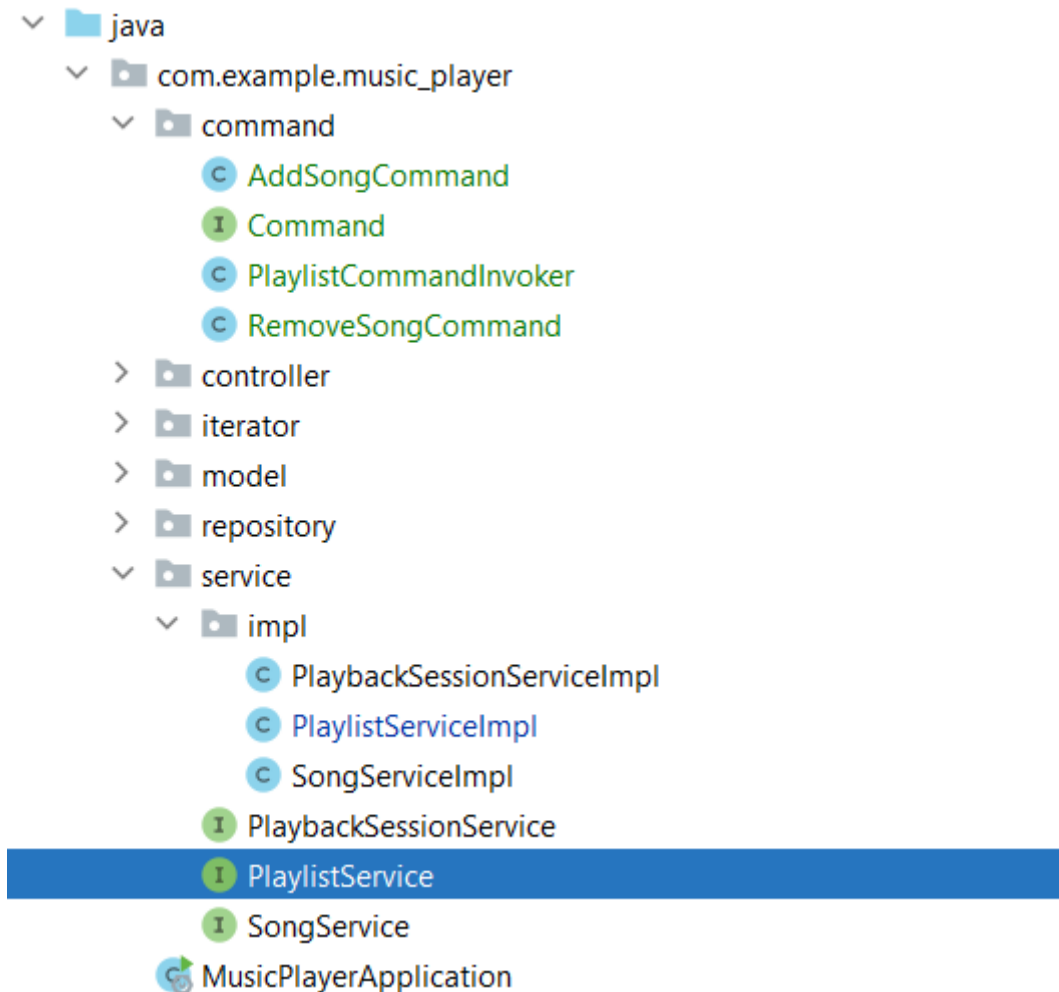


Рисунок 1 – структура проекту

У ході лабораторної роботи була реалізована система музичного плеєра, де основним завданням було використання патерну Command.

3. Реалізація шаблону Command .

Паттерн "Команда" (Command) дозволяє інкапсулювати дію як об'єкт, що дозволяє легко виконувати, скасувати або чергувати команди. У вашому прикладі цей патерн застосовується для управління операціями над плейлистами, такими як додавання і видалення пісень. Патерн **Command** допомагає зробити код гнучкішим і розширюваним, оскільки всі дії організовані як команди, що можна виконати чи скасувати.

1. Command (Інтерфейс Команди)

- Призначення: Інтерфейс Command визначає метод execute, який виконують усі команди. Це дозволяє визначити загальний контракт для всіх операцій, що реалізуються в системі.

- Навіщо потрібен: Він забезпечує узгодженість в системі, щоб будь-яка команда, яка виконує дію, мала однаковий метод execute, що робить код чистішим і гнучкішим. Даний інтерфейс зображений на рисунку 2.



```

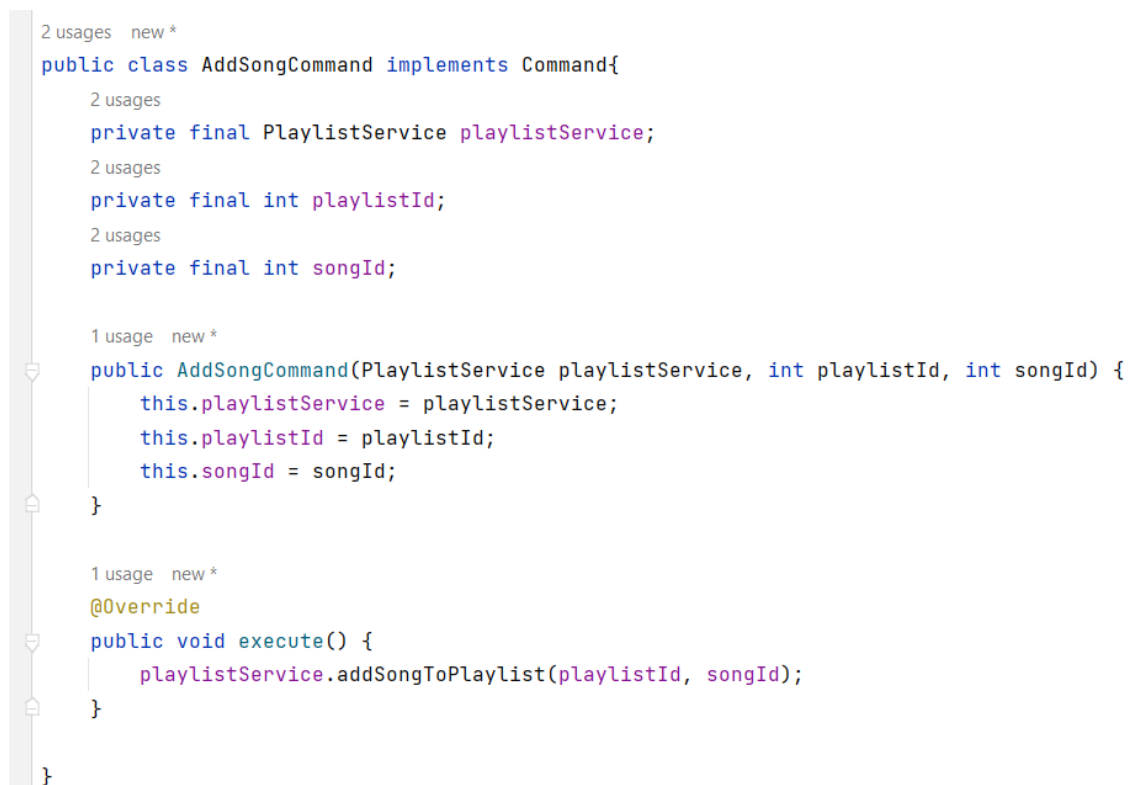
7 usages 2 implementations new *
public interface Command {
    1 usage 2 implementations new *
    void execute();
}

```

Рисунок 2 – інтерфейс Command

2. AddSongCommand (Додати Пісню)

- Призначення: Цей клас реалізує команду для додавання пісні в плейлист.
- Що виконує: Він приймає PlaylistService, playlistId, і songId, щоб мати змогу виконати додавання пісні. Даний клас зображений на рисунку 3.



```

2 usages new *
public class AddSongCommand implements Command{
    2 usages
    private final PlaylistService playlistService;
    2 usages
    private final int playlistId;
    2 usages
    private final int songId;

    1 usage new *
    public AddSongCommand(PlaylistService playlistService, int playlistId, int songId) {
        this.playlistService = playlistService;
        this.playlistId = playlistId;
        this.songId = songId;
    }

    1 usage new *
    @Override
    public void execute() {
        playlistService.addSongToPlaylist(playlistId, songId);
    }
}

```

Рисунок 3 – AddSongCommand

3. RemoveSongCommand (Видалити Пісню)

- Призначення: Цей клас реалізує команду для видалення пісні з плейлиста.
- Що виконує: Приймає PlaylistService, playlistId, і songId для видалення конкретної пісні з певного плейлиста. Даний клас зображений на рисунку 4.

```

2 usages new *
public class RemoveSongCommand implements Command {

    2 usages
    private final PlaylistService playlistService;
    2 usages
    private final int playlistId;
    2 usages
    private final int songId;

    1 usage new *
    public RemoveSongCommand(PlaylistService playlistService, int playlistId, int songId) {
        this.playlistService = playlistService;
        this.playlistId = playlistId;
        this.songId = songId;
    }

    1 usage new *
    @Override
    public void execute() {
        playlistService.removeSongFromPlaylist(playlistId, songId);
    }
}

```

Рисунок 4 – RemoveSongCommand

4. PlaylistCommandInvoker (Invoker)

- Призначення: Цей клас зберігає та виконує команди.
- Що виконує: Містить метод setCommand для зберігання команди, яку потрібно виконати, і метод executeCommand для її виконання.
- Навіщо потрібен: Він виступає проміжною ланкою між викликом команди та її виконанням. Це дозволяє змінювати команду або зберігати її для подальшого виконання, а також реалізувати шаблон виклику дій. Даний клас зображений на рисунку 5.

```

3 usages new *
public class PlaylistCommandInvoker {

    2 usages
    private Command command;

    2 usages new *
    public void setCommand(Command command) {
        this.command = command;
    }

    2 usages new *
    public void executeCommand() {
        command.execute();
    }
}

```

Рисунок 5 – PlaylistCommandInvoker

5. PlaylistService (Інтерфейс Сервісу Плейлиста)

- Призначення: Інтерфейс визначає основні операції над плейлистами, такі як додавання пісні, видалення, отримання та оновлення плейлистів.
- Навіщо потрібен: Цей інтерфейс є своєрідним Receiver, тобто отримувачем команд. Команди викликають методи цього інтерфейсу, щоб виконати відповідні дії. Даний інтерфейс зображений на рисунку 6.

```
public interface PlaylistService {
    1 usage 1 implementation imykytenko
    Playlist createPlaylist(Playlist playlist);
    2 usages 1 implementation imykytenko
    Playlist getPlaylistById(int id);
    1 usage 1 implementation imykytenko
    List<Playlist> getAllPlaylists();
    1 usage 1 implementation imykytenko
    Playlist updatePlaylist(int id, Playlist playlist);
    1 usage 1 implementation imykytenko
    void deletePlaylist(int id);
    2 usages 1 implementation imykytenko
    void addSongToPlaylist(int playlistId, int songId);
    2 usages 1 implementation imykytenko
    void removeSongFromPlaylist(int playlistId, int songId);
    1 usage 1 implementation new *
    List<Song> getSongsInPlaylist(int playlistId);
}
```

Рисунок 6 – PlaylistService

На рисунку 7 ми можемо бачити безпосередню реалізацію в PlaylistServiceImpl

```

@Override
public void addSongToPlaylist(int playlistId, int songId) {
    Playlist playlist = playlistRepository.findById(playlistId).orElse( other: null);
    Song song = songRepository.findById(songId).orElse( other: null);
    if (playlist != null && song != null) {
        Command addSongCommand = new AddSongCommand( playlistService: this, playlistId, songId);
        commandInvoker.setCommand(addSongCommand);
        commandInvoker.executeCommand();
    }
}

2 usages  imykytenko *
@Override
public void removeSongFromPlaylist(int playlistId, int songId) {
    Playlist playlist = playlistRepository.findById(playlistId).orElse( other: null);
    Song song = songRepository.findById(songId).orElse( other: null);
    if (playlist != null && song != null) {
        Command removeSongCommand = new RemoveSongCommand( playlistService: this, playlistId, songId);
        commandInvoker.setCommand(removeSongCommand);
        commandInvoker.executeCommand();
    }
}

```

Рисунок 7 – PlaylistServiceImpl

4. Діаграма класів для паттерну Command

Діаграма класів, які реалізують паттерн Command зображена на рисунку 8.

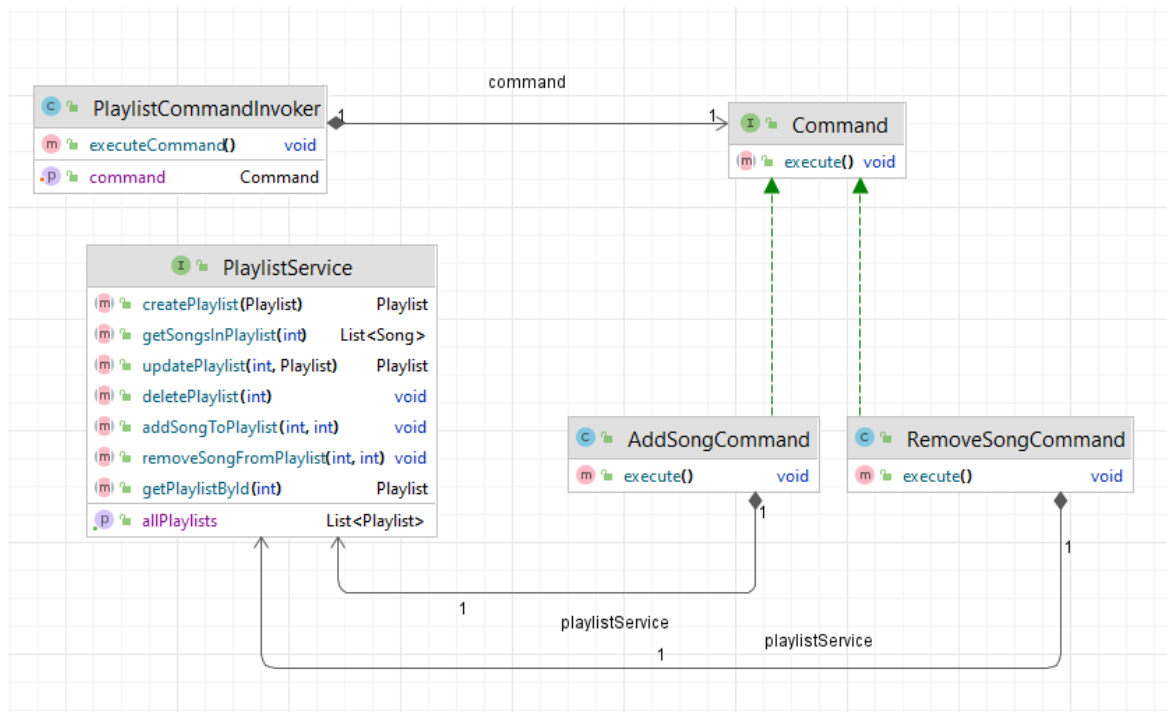


Рисунок 8 - Діаграма класів

5. Проблема, яку допоміг вирішити шаблон Command

Шаблон проектування Command вирішив проблему щільного зв'язку між методами додавання та видалення пісень і основною логікою роботи сервісу.

У початковій версії `addSongToPlaylist` та `removeSongFromPlaylist` містили безпосередній доступ до репозиторіїв та маніпулювали об'єктами напряму. Це ускладнювало підтримку коду, знижувало його гнучкість та унеможливлювало динамічне керування операціями над плейлистами.

Шаблон `Command` дозволив інкапсулювати операції як окремі об'єкти-команди, що надало наступні переваги:

- **Зменшення зв'язності:** Код `PlaylistServiceImpl` більше не потребує знання, як виконуються конкретні операції, адже вони інкапсульовані в командних класах.
- **Легкість у розширенні:** Додавання нових дій, таких як "відновити видалення пісні" або "змінити порядок пісень", потребує лише створення нових команд без змін у вже існуючих методах.
- **Можливість динамічного виконання:** Команди можна додавати в чергу, відкладати їх виконання або навіть скасовувати, якщо буде реалізована функція "undo".

6. Переваги застосування патерну `Command`

1. **Інкапсуляція запиту як об'єкта:** `Command` дозволяє перетворити операцію на окремий об'єкт, що робить запити гнучкими та легкими для повторного використання, чергування, скасування або відкладання.
2. **Спрощення змін у коді:** Додавання нових дій більше не потребує модифікації основного сервісу — достатньо додати новий клас-команду.
3. **Гнучке керування операціями:** `Command` дозволяє легко впроваджувати механізми "undo"/"redo" і зберігати історію виконаних операцій, що корисно у випадку з музичним плеєром для динамічного керування плейлистом.
4. **Можливість поєднання з іншими патернами:** `Command` добре працює в поєднанні з такими патернами, як `Invoker` (отримувач команд), що дозволяє більш чітко структуровано виконувати операції у певному порядку, і `Composite`, що допомагає виконувати комплексні команди з декількох простих.

Висновок: У даній лабораторній роботі я реалізувала патерн проєктування `Command`, щоб оптимізувати структуру роботи з плейлистом у музичному плеєрі. Патерн дозволив інкапсулювати операції додавання та видалення пісень як окремі команди, що зробило систему більш гнучкою, легко масштабованою та модульною.