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

ЛАБОРАТОРНА РОБОТА №5

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування»

Виконав:

студент групи ІА-33

Грицай Андрій

Перевірив:

асистент кафедри ІСТ

Мягкий Михайло Юрійович

Київ - 2025

Тема: Патерни проектування
Мета: Вивчити структуру шаблонів «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype» 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

Адаптер (Adapter)

Шаблон проєктування "Адаптер" дозволяє об'єднати інтерфейси двох несумісних класів. Це корисно, коли потрібно використовувати наявний клас, але його інтерфейс не відповідає потребам клієнтського коду. Адаптер огортає клас-джерело і надає інтерфейс, що сумісний з інтерфейсом, який очікує клієнт. Таким чином, клієнтський код може працювати з адаптером, не змінюючи свій інтерфейс. У Java цей шаблон часто реалізується шляхом створення нового класу, який наслідує або реалізує необхідний інтерфейс і всередині використовує методи класу-джерела. Шаблон дозволяє покращити модульність і повторне використання коду, зберігаючи при цьому стабільний інтерфейс для клієнта.

Будівельник (Builder)

Шаблон "Будівник" забезпечує гнучкий спосіб створення складних об'єктів, дозволяючи будувати їх покроково. Це особливо корисно, коли об'єкт має багато опціональних параметрів або складну структуру. Замість того щоб створювати об'єкт через конструктор з численними параметрами, будівник дає можливість додавати значення по черзі і в кінці отримати готовий об'єкт. У Java це часто реалізується за допомогою вкладеного статичного класу "Builder" всередині основного класу, де кожен метод будівника повертає об'єкт будівника, дозволяючи ланцюжковий виклик методів. Така реалізація робить код читабельнішим і гнучкішим для створення об'єктів з різними конфігураціями.

Команда (Command)

Шаблон "Команда" інкапсулює дію або запит як об'єкт, дозволяючи відкладене виконання операції, її скасування або збереження для подальшого використання. Кожна команда реалізує інтерфейс із методами для виконання дії (наприклад, execute). Це дозволяє відокремити клієнтський код від отримувача дії. Команди можна комбінувати, записувати в лог або ставити в чергу. У Java цей шаблон можна реалізувати за допомогою інтерфейсу Command і його

конкретних реалізацій для різних операцій. Це полегшує обробку запитів в інтерфейсі, де користувач може обирати команди, а також зручно для реалізації таких функцій, як "Скасувати" або "Повторити" в додатках

Ланцюг відповідальностей (Chain of Responsibility)

Шаблон "Ланцюг відповідальностей" дозволяє передавати запит вздовж ланцюга обробників, де кожен обробник має шанс обробити запит або передати його далі. Це дозволяє клієнтському коду не знати, хто саме оброблятиме запит, і забезпечує гнучкість в додаванні чи видаленні обробників. Кожен обробник реалізує інтерфейс із методом для обробки запиту і зберігає посилання на наступного обробника в ланцюзі. У Java цей шаблон можна реалізувати шляхом створення абстрактного класу для обробників, де кожен конкретний обробник перевіряє, чи здатен він обробити запит, або передає його далі. Це зручно для систем обробки помилок або запитів з різними рівнями доступу.

Прототип (Prototype)

Шаблон "Прототип" дозволяє створювати нові об'єкти шляхом копіювання вже наявного об'єкта (прототипу) замість створення об’єктів "з нуля". Це ефективно, коли створення нового об'єкта є складним або ресурсозатратним, а також корисно для створення об'єктів зі схожими початковими станами. У Java шаблон може бути реалізований через інтерфейс Cloneable, що вимагає реалізації методу clone, який повертає копію об’єкта. Завдяки цьому шаблону можна створювати об'єкти зі схожою конфігурацією без необхідності повторювати весь процес ініціалізації, зберігаючи при цьому незалежність їхніх станів.

Хід роботи

Музичний програвач має такі функціональні можливості, як керування відтворенням (грати, пауза, стоп) та навігація по списку треків (наступний, попередній). У цій роботі використано патерн "Команда" (Command). Цей патерн обрано як архітектурне рішення для інкапсуляції дій користувача над медіаплеєром у вигляді окремих об'єктів. У процесі розробки стало зрозуміло, що для гнучкого керування плеєром необхідно відокремити бізнес-логіку відтворення від графічного інтерфейсу (кнопок), який ініціює ці дії. Без використання патерну Command реалізація обробки подій призвела б до жорсткої зв'язаності між контролером та логікою плеєра.

Патерн Command вирішує цю проблему, дозволяючи:

- Інкапсулювати кожну дію (наприклад, PlayCommand або NextCommand) як окремий об'єкт.
- Забезпечити для кожного об'єкта уніфікований метод execute() для виконання дії.
- Легко додавати нові команди керування без зміни існуючого коду контролера або інтерфейсу.

У проєкті цей підхід реалізований наступним чином:

- **Command (Інтерфейс):** Command.
- **Concrete Commands (Конкретні Команди):** Класи PlayCommand, PauseCommand, StopCommand, NextCommand, PreviousCommand.
- **Receiver (Отримувач):** Клас MusicPlayer, який містить логіку відтворення та методи для зміни стану (play, pause, stop, next).
- **Invoker (Ініціатор):** Клас MusicPlayerController (та елементи GUI), який створює та викликає конкретні команди у відповідь на дії користувача.

Реалізація класів

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

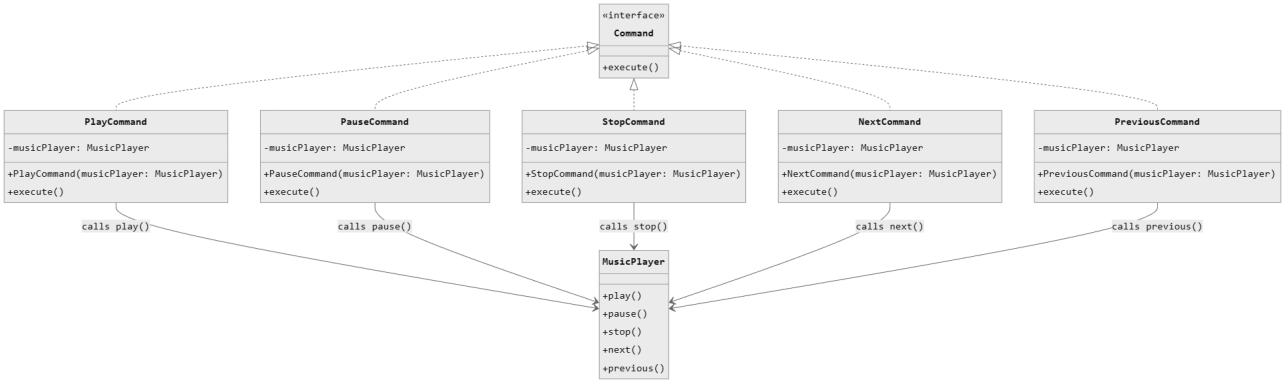


Рисунок 1. - Діаграма класів, яка представляє використання шаблону в реалізації системи

Опис створених об’єктів:

Command (інтерфейс). Command є абстрактною основою патерну. Він визначає загальний інтерфейс (контракт) для всіх команд, що керують музичним плеєром. Інтерфейс декларує єдиний метод execute(), який викликає відповідну дію на отримувачі.

```
package com.example.labs.common.patterns.command;
public interface Command {
    void execute();
}
```

Concrete Commands (конкретні команди):

PlayCommand (конкретна команда). Цей клас реалізує інтерфейс Command і відповідає за початок або відновлення відтворення. Він зберігає посилання на MusicPlayer і в методі execute() викликає метод play() об'єкта-отримувача.

```
public class PlayCommand implements Command {
    private MusicPlayer musicPlayer;
    public PlayCommand(MusicPlayer musicPlayer) {
        this.musicPlayer = musicPlayer;
    }
    @Override
    public void execute() {
```

```
        musicPlayer.play();
    }
}
```

PauseCommand (конкретна команда). Цей клас реалізує інтерфейс Command і відповідає за призупинення відтворення. Він зберігає посилання на MusicPlayer і в методі execute() викликає метод pause() об'єкта-отримувача.

```
public class PauseCommand implements Command {
    private MusicPlayer musicPlayer;
    public PauseCommand(MusicPlayer musicPlayer) {
        this.musicPlayer = musicPlayer;
    }
    @Override
    public void execute() {
        musicPlayer.pause();
    }
}
```

StopCommand (конкретна команда). Клас реалізує інтерфейс Command для повної зупинки відтворення. Метод execute() делегує виконання методу stop() класу MusicPlayer, що скидає поточний стан відтворення.

```
public class StopCommand implements Command {
    private MusicPlayer musicPlayer;
    public StopCommand(MusicPlayer musicPlayer) {
        this.musicPlayer = musicPlayer;
    }
    @Override
    public void execute() {
        musicPlayer.stop();
    }
}
```

NextCommand (конкретна команда). Цей клас інкапсулює дію переходу до наступного треку. Реалізуючи інтерфейс Command, він у методі execute() звертається до методу next() об'єкта MusicPlayer.

```
public class NextCommand implements Command {
    private MusicPlayer musicPlayer;
    public NextCommand(MusicPlayer musicPlayer) {
        this.musicPlayer = musicPlayer;
    }
    @Override
    public void execute() {
        musicPlayer.next();
    }
}
```

PreviousCommand (конкретна команда). Клас відповідає за повернення до попереднього треку. Він реалізує інтерфейс Command і викликає метод previous() на пов'язаному екземплярі MusicPlayer при виконанні команди.

```
public class PreviousCommand implements Command {
```

```
    private MusicPlayer musicPlayer;
    public PreviousCommand(MusicPlayer musicPlayer) {
        this.musicPlayer = musicPlayer;
    }
    @Override
    public void execute() {
        musicPlayer.previous();
    }
}
```

MusicPlayer (Отримувач / Receiver). Цей клас виступає "Отримувачем" у патерні Command. Він містить основну бізнес-логіку відтворення музики, керування станом (відтворення, пауза, зупинка) та перемикання треків. Самі об'єкти команд не виконують обчислень, а лише викликають методи цього класу (play(), pause(), stop(), next()), який знає, як саме виконати запит.

```
public class MusicPlayer {
    // ...
    public void play() { }
    public void pause() { }
    public void stop() { }
    public void next() { */ }
    // ...
}
```

Висновок: Виконуючи цю лабораторну роботу, я ознайомився з такими патернами, як Adapter, Builder, Command, Chain of Responsibility, Prototype. Особливу увагу я приділив реалізації шаблону Command, детально описавши його основну логіку та функціональність у контексті мого проєкту музичного програвача. Цей патерн було обрано як архітектурне рішення для спрощення реалізації функціоналу керування відтворенням (Play, Pause, Stop, Next, Previous). Я зрозумів, наскільки цей патерн спрощує процес управління діями, дозволяючи відокремити ініціатора (контролер/інтерфейс) від виконавця (MusicPlayer).

Використання шаблону Command забезпечує:

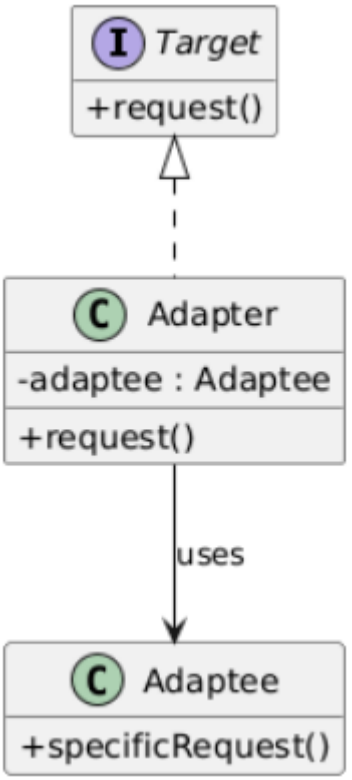
- Гнучкість та розширюваність архітектури.
- Можливість легко додавати нові команди без зміни коду ініціатора.
- Чітку та структуровану архітектуру програми, особливо для відокремлення бізнес-логіки плеєра від користувацького інтерфейсу.

Контрольні запитання

1. Яке призначення шаблону «Адаптер»?

Призначення шаблону «Адаптер» - забезпечити спільну роботу класів з несумісними інтерфейсами. Він діє як "перехідник" або "обгортка" навколо існуючого класу (Adaptee), перетворюючи його інтерфейс на інший, очікуваний клієнтським кодом (Client)

2. Нарисуйте структуру шаблону «Адаптер».



3. Які класи входять в шаблон «Адаптер», та яка між ними взаємодія?

Client (Клієнт): Клас, який хоче використовувати функціонал, але очікує певний інтерфейс (Target).

Target (Цільовий інтерфейс): Інтерфейс, який використовує Client.

Adapter (Адаптер): Клас, який реалізує інтерфейс Target і містить посилання на об'єкт Adaptee. Він перенаправляє виклики від Client до Adaptee, виконуючи необхідні перетворення.

Adaptee (Об'єкт, що адаптується): Існуючий клас з несумісним інтерфейсом.

4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?

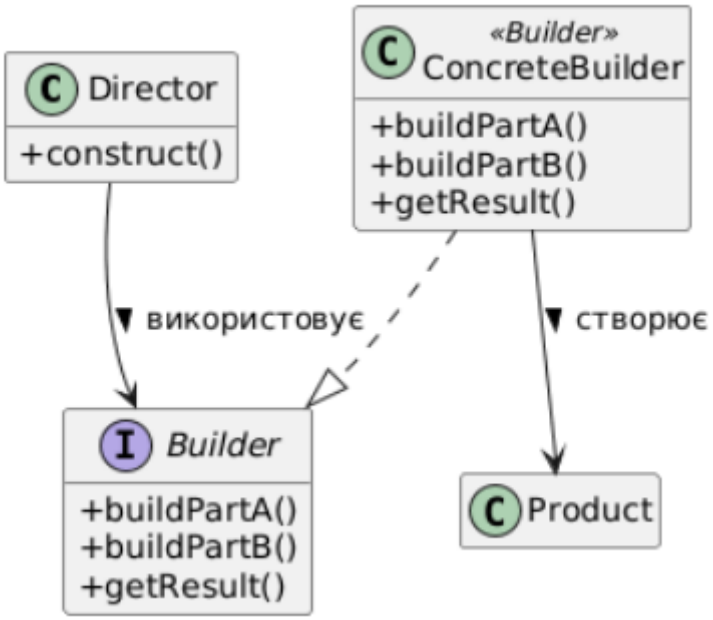
Адаптер об'єктів: Використовує композицію. Клас Adapter містить екземпляр класу Adaptee. Цей підхід є більш гнучким, оскільки дозволяє адаптувати будь-який підклас Adaptee.

Адаптер класів: Використовує множинне успадкування (в Java реалізується через успадкування класу та реалізацію інтерфейсу). Клас Adapter одночасно успадковує Adaptee та реалізує інтерфейс Target. Цей підхід менш гнучкий.

5. Яке призначення шаблону «Будівельник»?

Шаблон «Будівельник» (Builder) використовується для покрокового створення складних об'єктів. Він дозволяє відокремити процес конструювання об'єкта від його представлення, завдяки чому один і той самий процес конструювання може створювати різні представлення

6. Нарисуйте структуру шаблону «Будівельник».



7. Які класи входять в шаблон «Будівельник», та яка між ними взаємодія?

Product (Продукт): Складний об'єкт, який створюється.

Builder (Будівельник): Інтерфейс для створення частин об'єкта Product.

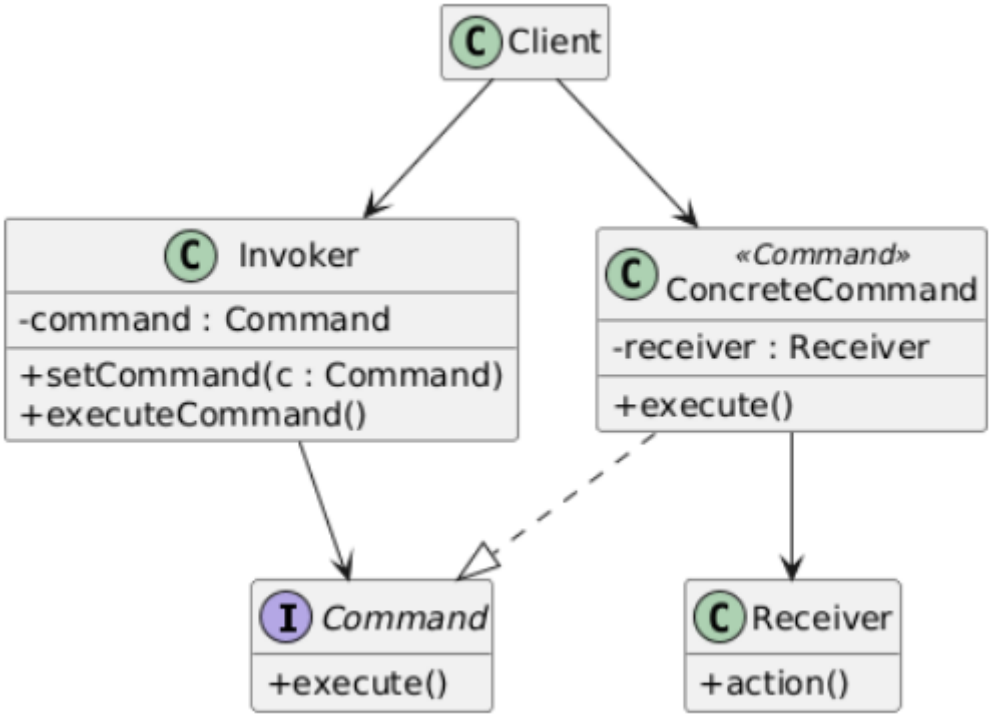
ConcreteBuilder (Конкретний будівельник): Реалізує інтерфейс Builder і конструює конкретне представлення продукту.

Director (Директор): Клас, який керує процесом побудови, використовуючи об'єкт Builder.

8. У яких випадках варто застосовувати шаблон «Будівельник»"? Коли процес створення об'єкта є складним, багатоетапним, або коли конструктор має занадто багато параметрів (особливо опціональних). Також, коли потрібно створювати різні представлення одного й того ж об'єкта.

9. Яке призначення шаблону «Команда»? Шаблон «Команда» (Command) інкапсулює запит на виконання дії як об'єкт. Це дозволяє параметризувати клієнтські об'єкти різними запитами, ставити запити в чергу, логувати їх, а також підтримувати операції скасування (undo).

10. Нарисуйте структуру шаблону «Команда».



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

Command: Інтерфейс, що оголошує метод для виконання операції (execute).

ConcreteCommand: Реалізує інтерфейс Command, містить посилання на Receiver і викликає його методи.

Client: Створює об'єкт ConcreteCommand і встановлює його одержувача. Invoker: Просить команду виконати запит.

Receiver: "Одержувач", який знає, як виконати операцію.

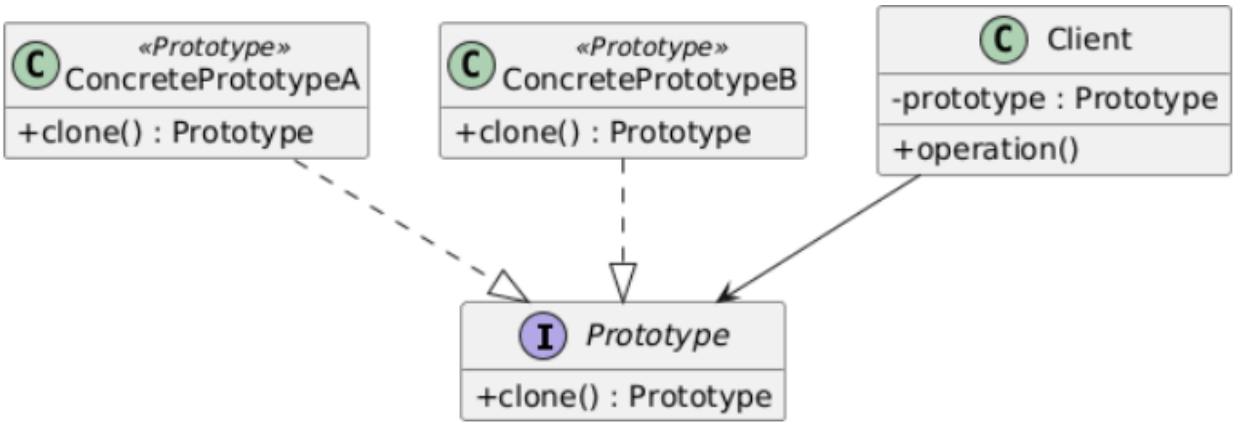
12. Розкажіть як працює шаблон «Команда».

Клієнт створює об'єкт-команду, пов'язуючи її з конкретним одержувачем. Потім цей об'єкт-команда передається ініціатору (Invoker), наприклад, кнопці в меню. Коли користувач натискає кнопку, ініціатор викликає метод execute() у команди, а команда, в свою чергу, викликає потрібний метод у свого одержувача.

13. Яке призначення шаблону «Прототип»?

Шаблон «Прототип» (Prototype) дозволяє створювати нові об'єкти шляхом копіювання існуючого об'єкта (прототипу). Це дозволяє уникнути прив'язки до класів об'єктів, що створюються.

14. Нарисуйте структуру шаблону «Прототип».



15. Які класи входять в шаблон «Прототип», та яка між ними взаємодія?

Prototype: Інтерфейс, що оголошує метод клонування (clone).

ConcretePrototype: Реалізує інтерфейс Prototype та метод clone.

Client: Створює новий об'єкт, викликаючи метод clone у прототипу.

16. Які можна привести приклади використання шаблону «Ланцюжок відповідальності»?

Обробка подій в GUI: Коли користувач клікає на кнопку, подія спочатку обробляється кнопкою, потім може бути передана батьківській панелі, потім вікну, і так далі, доки не буде оброблена. Системи логування: Повідомлення може проходити через ланцюжок обробників: один записує в консоль, інший - у файл, третій - відправляє по email, залежно від рівня важливості. Системи авторизації та валідації: Запит користувача може проходити через ланцюжок перевірок: перевірка автентифікації, перевірка прав доступу, перевірка валідності даних.