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 №5

«Патерни проектування»

Виконав:
студент групи ІА–32
Лось Ярослав

Київ 2025

Мета: Вивчити структуру шаблонів «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype» та навчитися застосовувати їх в реалізації програмної системи.

Зміст

Теоретичні відомості

Шаблон «Command»

Призначення патерну: Шаблон "command" (команда) перетворить звичайний виклик методу в клас. Таким чином дії в системі стають повноправними об'єктами. Це зручно в наступних випадках:

- Коли потрібна розвинена система команд – відомо, що команди будуть добавлятися;
- Коли потрібна гнучка система команд – коли з'являється необхідність додавати командам можливість відміни, логування і інш.;
- Коли потрібна можливість складання ланцюжків команд або виклику команд в певний час.

Об'єкт команда сама по собі не виконує ніяких фактичних дій окрім перенаправлення запиту одержувачеві (тобто команди все ж виконуються одержувачем), однак ці об'єкти можуть зберігати дані для підтримки додаткових функцій відміни, логування і інш. Наприклад, команда вставки символу може запам'ятовувати символ, і при виклику відміни викликати відповідну функцію витирання символу. Можна також визначити параметр «застосовності» команди (наприклад, на картинці писати не можна) – і використати цей аттрибут для засвічування піктограми в меню.

Такий підхід до команд дозволяє побудувати дуже гнучку систему команд, що настроюється. У більшості додатків це буде зайвим (використовується спрощений варіант), проте життєво важливий в додатках з великою кількістю команд (редактори).

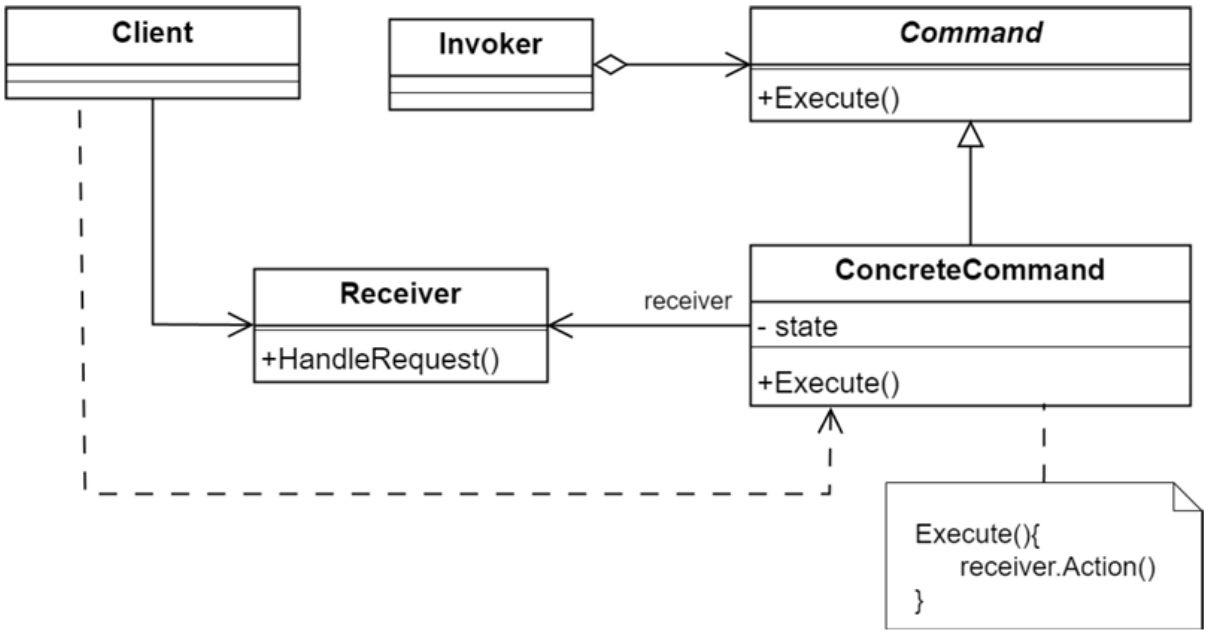


Рисунок 5.3. Структура патерну Команда

Проблема: Ви реалізуєте товстий клієнт який має багатий візуальний інтерфейс: має меню, кнопки і контекстне меню. Кожна дія, яку можна виконати, має три варіанти виконання – через меню, натисканням кнопки та через контекстне меню. Реалізацію кожної дії можна розмістити в обробнику візуального елементу, але тоді потрібно буде продублювати цей функціонал для меню, кнопки та контекстного меню. Таким чином ми будемо мати дублювання коду і при зміні функціоналу змінювати його в трьох місцях.

Додатковим викликом буде реалізувати автоматизоване тестування такої системи, тому що нам необхідно емулювати натискання кнопок користувачем.

Рішення: Виділення функціоналу який виконується по натисканню на кнопку в окремий клас дозволяє відвязати візуальну частину від логіки обробки. Таким чином ми будемо мати шар з UI елементами і шар з логікою обробки дій виконаних на UI. Це дозволяє один і той самий об'єкт з шару логіки обробки дій використати для реакцію на натискання кнопки і пункту меню і контекстного меню. Кнопки тепер не знають про конкретні класи реалізації команд, а взаємодіють з об'єктами команд через загальний інтерфейс. Ще одна перевага, що тепер ми можемо достатньо просто реалізувати механізм enable-disable для кнопок та

контекстного меню через виклик у об'єкта команди метода IsEnabled() або через прив'язку (binding) на поле IsEnabled у об'єкта команди.

При такій реалізації також набагато простіше організувати тестування застосунку, тому що ми тепер не потрібно емулювати дії користувача, а достатньо протестувати шар логіки обробки використовуючи модульні тести (unit tests).

Переваги та недоліки:

+ Ініціатор виконання команди не знає деталей реалізації виконавця команди.

+ Підтримує операції скасування та повторення команд.

+ Послідовність команд можна логувати і при необхідності виконати цю послідовність ще раз.

+ Простота розширення за рахунок додавання нових команд без необхідності внесення змін в уже існуючий код (принцип відкритості закритості).

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.

1. Загальний опис роботи

Основна мета роботи полягає в тому, щоб роз'єднати об'єкт, який ініціює дію (кнопка на інтерфейсі), від об'єкта, який цю дію виконує (аудіо-двигун програвача). Для цього використаємо патерн **Command** (Команда).

В контексті нашої теми, ми реалізуємо систему керування, де натискання кнопок ("Play", "Stop", "Rewind") не викликає методи програвача напряму. Замість цього створюються об'єкти-команди, які інкапсулюють запит. Це дозволить гнучко налаштовувати панель керування (Invoker), змінювати призначення кнопок "на льоту" та потенційно реалізувати чергу команд або їх скасування.

Програмна система складається з наступних компонентів:

- **Receiver**: Клас AudioSystem, який знає, як реально виконувати операції (грати музику, зупиняти, змінювати гучність).
- **Command (Інтерфейс)**: Визначає метод execute().
- **Concrete Commands**: Класи PlayCommand, StopCommand, RewindCommand, які звязують дію з користувачем.
- **Invoker**: Клас PlayerControlPanel (пульт), який зберігає посилання на команди і викликає їх виконання при натисканні кнопок.
- **Client**: Демонстраційний клас, що налаштовує пульт.

2. Опис класів програмної системи

Клас AudioSystem (Receiver)

Опис:

Клас AudioSystem виступає в ролі Одержувача (Receiver) у термінології патерну Команда. Він представляє собою бізнес-логіку програми — аудіо-двигун, який вміє виконувати фактичні операції з медіа-контентом. Цей клас є повністю незалежним від інтерфейсу користувача та команд; він нічого не знає про те, хто і коли його викликає.

Ключові характеристики:

- Містить методи, що реалізують конкретну роботу: startPlaying() (початок відтворення), stopPlaying() (зупинка) та rewind() (перемотка).
- У реальній системі цей клас взаємодіяв би з драйверами звукової карти або аудіо-бібліотеками.

- Є об'єктом, над яким виконуються всі команди. Саме посилання на екземпляр цього класу передається в конструктори конкретних команд.

```
public class AudioSystem {
    1 usage
    public void startPlaying() {
        System.out.println("[AudioSystem] Музика грає");
    }

    1 usage
    public void stopPlaying() {
        System.out.println("[AudioSystem] Музику зупинено");
    }

    1 usage
    public void rewind() {
        System.out.println("[AudioSystem] Перемотка на початок треку");
    }
}
```

Рисунок 1 – Код класу AudioSystem

Інтерфейс Command

Опис:

Інтерфейс Command є абстракцією, яка уніфікує всі можливі дії в системі. Він дозволяє Ініціатору (Invoker) працювати з будь-якими командами однаково, не вдаючись у подробиці їхньої реалізації. Це ключовий елемент для забезпечення принципу слабкої зв'язності.

Характеристики:

- Оголошує єдиний метод execute(), який не приймає параметрів. Вся необхідна інформація (посилання на отримувача, налаштування) закладається в команду при її створенні через конструктор.
- Забезпечує поліморфізм: пульт керування може викликати execute() для будь-якого об'єкта, що імплементує цей інтерфейс.

```
public interface Command {
    3 usages  3 implementations
    void execute();
}
```

Рисунок 2 – Код інтерфейсу Command

Класи PlayCommand, StopCommand, RewindCommand

Опис:

Це набір класів Конкретних Команд, кожен з яких імплементує інтерфейс Command. Їхнє головне завдання — зв'язати дію (наприклад, "натискання кнопки Play") з конкретним виконавцем цієї дії (AudioSystem). Вони інкапсулюють запит у вигляді об'єкта.

Характеристики:

- Конструктор: Кожна команда приймає в конструкторі посилання на об'єкт AudioSystem і зберігає його у приватному полі. Це ініціалізація зв'язку "Command - Receiver".
- Метод execute(): Реалізує метод інтерфейсу, викликаючи відповідний метод у збереженого об'єкта AudioSystem.
    - PlayCommand.execute() викликає audioSystem.startPlaying().
    - StopCommand.execute() викликає audioSystem.stopPlaying().
    - RewindCommand.execute() викликає audioSystem.rewind().
- Ці класи є обгортками, які лише делегують виконання.

```java
public class PlayCommand implements Command {
    2 usages
    private AudioSystem audioSystem;

    1 usage
    public PlayCommand(AudioSystem audioSystem) {
        this.audioSystem = audioSystem;
    }

    3 usages
    @Override
    public void execute() {
        audioSystem.startPlaying();
    }
}
```

Рисунок 3 – Код класу PlayCommand

```java
public class StopCommand implements Command {
    2 usages
    private AudioSystem audioSystem;

    1 usage
    public StopCommand(AudioSystem audioSystem) {
        this.audioSystem = audioSystem;
    }

    3 usages
    @Override
    public void execute() {
        audioSystem.stopPlaying();
    }
}
```

Рисунок 4 – Код класу StopCommand

```java
public class RewindCommand implements Command {
    2 usages
    private AudioSystem audioSystem;

    1 usage
    public RewindCommand(AudioSystem audioSystem) {
        this.audioSystem = audioSystem;
    }

    3 usages
    @Override
    public void execute() {
        audioSystem.rewind();
    }
}
```

Рисунок 5 – Код класу RewindCommand

Клас PlayerControlPanel (Invoker)

Опис:

Клас PlayerControlPanel виконує роль Ініціатора (Invoker). Це абстракція панелі керування музичним центром або графічного інтерфейсу плеєра. Він не містить бізнес-логіки, але знає коли треба виконати команду.

Ключові характеристики:

- Агрегація команд: Містить поля типу Command (playButton, stopButton, rewindButton), а не посилання на AudioSystem. Тобто пульт відв'язаний від плеєра.
- Конфігурація: Надає методи setPlayCommand(), setStopCommand(), що дозволяє динамічно змінювати поведінку кнопок під час виконання програми.
- Ініціація дій: Методи pressPlay(), pressStop() імітують фізичне натискання кнопок. Вони викликають метод execute() у відповідного об'єкта команди.

Код класу PlayerControlPanel

```
public class PlayerControlPanel {
    private Command playButton;
    private Command stopButton;
    private Command rewindButton;

    public void setPlayCommand(Command playButton) {
        this.playButton = playButton;
    }

    public void setStopCommand(Command stopButton) {
        this.stopButton = stopButton;
    }

    public void setRewindCommand(Command rewindButton) {
        this.rewindButton = rewindButton;
    }

    public void pressPlay() {
        System.out.print("Натиснуто Play: ");
        if (playButton != null) playButton.execute();
    }

    public void pressStop() {
        System.out.print("Натиснуто Stop: ");
        if (stopButton != null) stopButton.execute();
    }

    public void pressRewind() {
        System.out.print("Натиснуто Rewind: ");
        if (rewindButton != null) rewindButton.execute();
    }
}
```

Клас CommandClient

Опис:

Клас DemoCommandClient виступає в ролі клієнта, який налаштовує систему перед використанням. Він відповідає за створення всіх об'єктів та встановлення зв'язків між ними.

Етапи роботи клієнта:

1. Створення Receiver: Створюється екземпляр AudioSystem.
2. Створення Commands: Створюються об'єкти команд, у конструктори яких передається створений раніше AudioSystem.
3. Створення Invoker: Створюється пульт PlayerControlPanel.
4. Конфігурація: Створені команди передаються у пульт через сетери.
5. Демонстрація: Викликаються методи натискання кнопок на пульті, демонструючи, що запит проходить через команду до одержувача.

```
public class CommandClient {
    public static void main(String[] args) {
        AudioSystem myAudio = new AudioSystem();

        Command play = new PlayCommand(myAudio);
        Command stop = new StopCommand(myAudio);
        Command rewind = new RewindCommand(myAudio);

        PlayerControlPanel remote = new PlayerControlPanel();

        remote.setPlayCommand(play);
        remote.setStopCommand(stop);
        remote.setRewindCommand(rewind);

        System.out.println("- Тестування пульта -");
        remote.pressPlay();
        remote.pressRewind();
        remote.pressStop();
    }
}
```

Рисунок 6 – Код класу CommandClient

3. Опис результатів коду

Для перевірки працездатності розробленої архітектури, запустимо головний метод класу CommandClient. Програма відтворить дії користувача, який натискає кнопки на пульті керування.

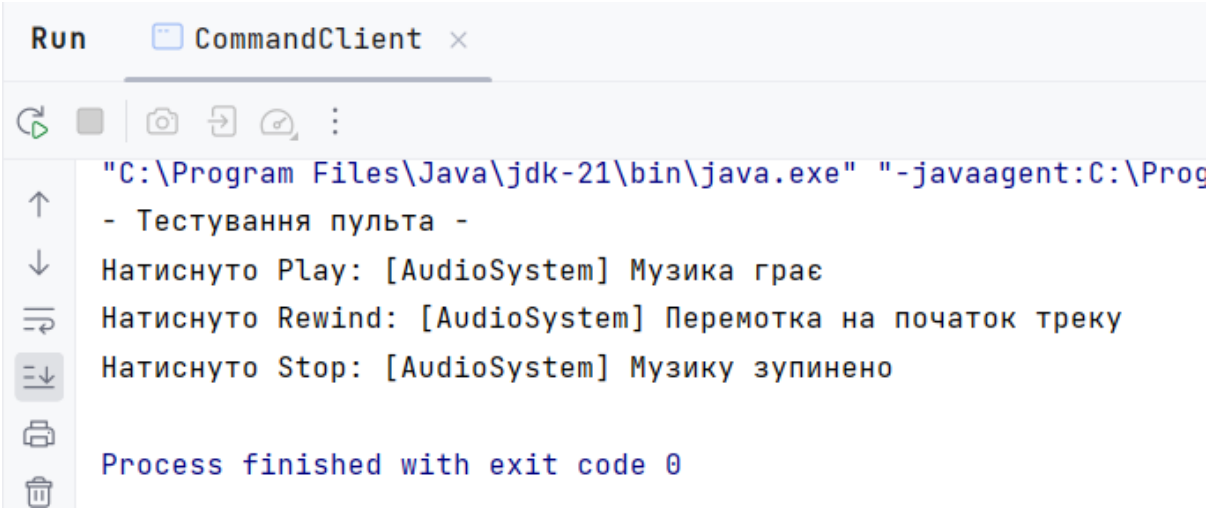


Рисунок 7 – Результат виконання програми

Аналіз результатів:

Вивід у консолі підтверджує правильність побудованого ланцюжка:

1. Клієнт викликав метод remote.pressPlay().
2. Об'єкт remote (Invoker) викликав playButton.execute().
3. Об'єкт PlayCommand делегував виклик до audioSystem.startPlaying().
4. Об'єкт audioSystem (Receiver) вивів повідомлення "[AudioSystem] Музика грає".

Така сама послідовність спрацювала для команд перемотки та зупинки. Отже, наша архітектура патерну Command успішно ізолювала об'єкт ініціювання запиту від об'єкта його виконання.

4. Діаграми класів

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

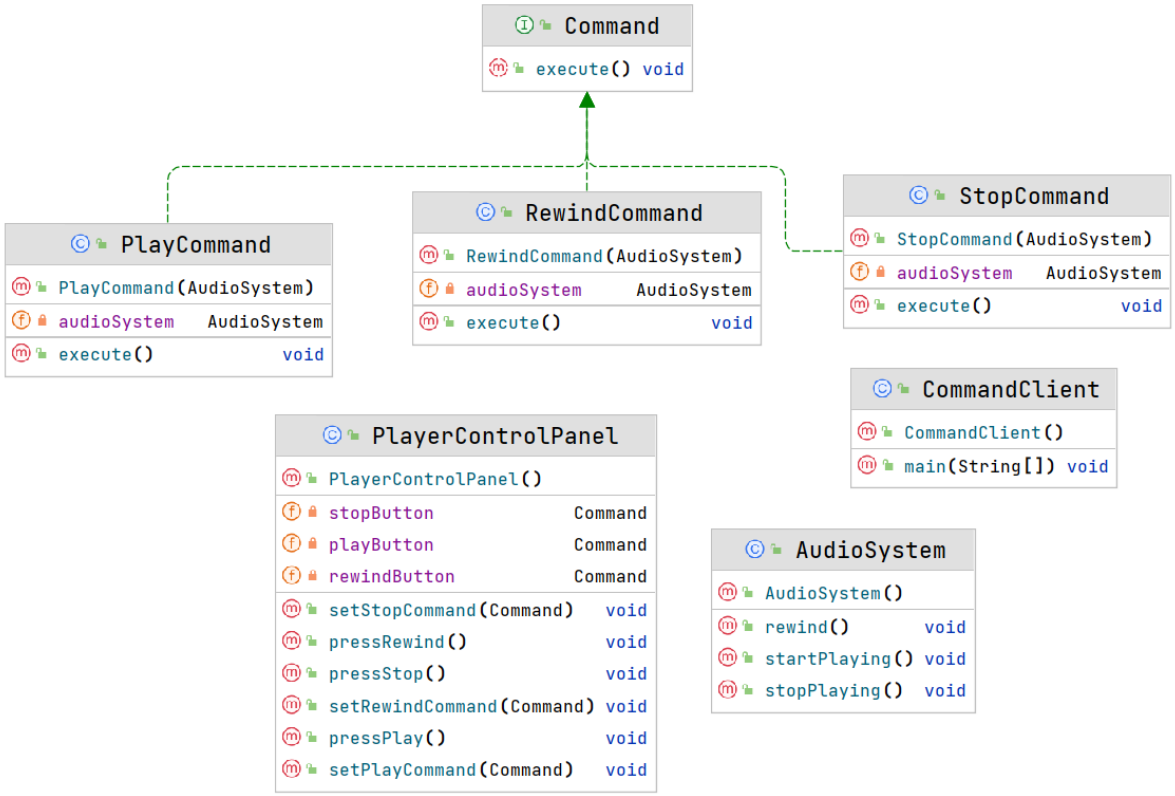


Рисунок 8 – Діаграма класів (спрощена)

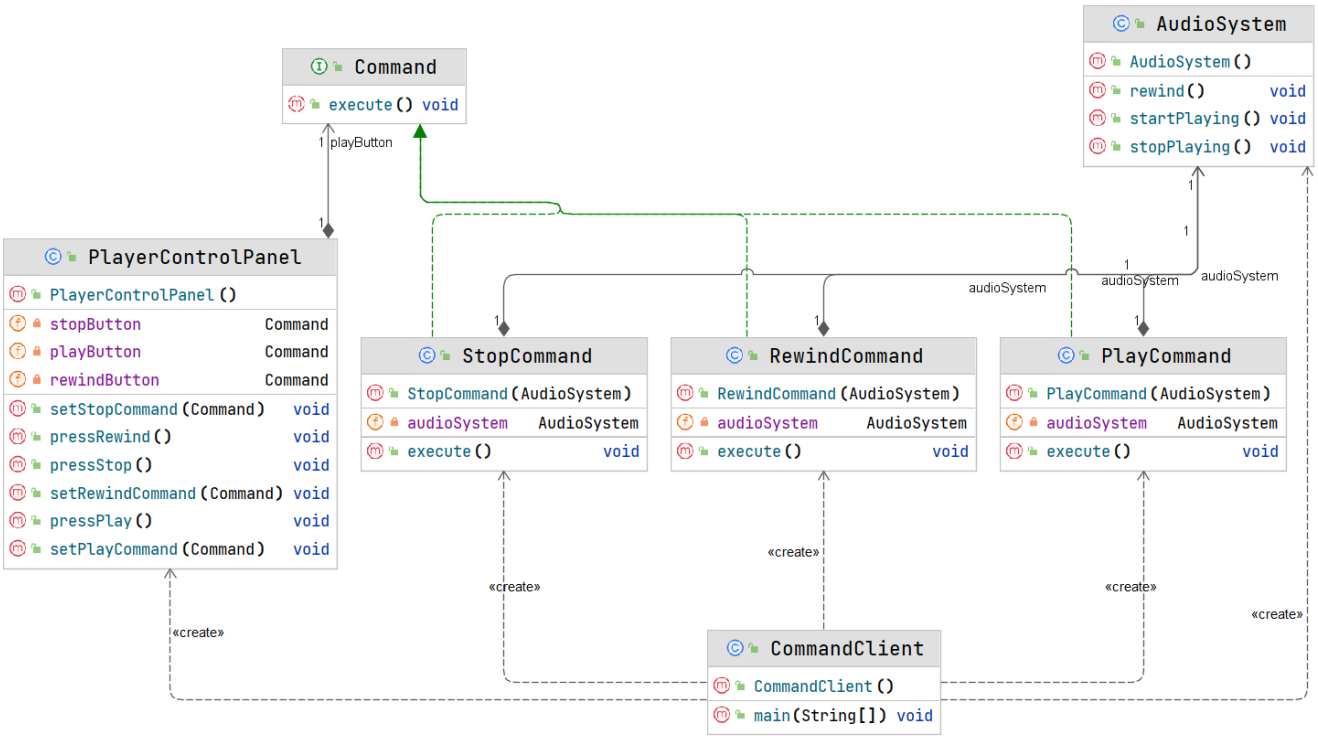


Рисунок 9 – Діаграма класів (повна)

Висновки

В ході виконання роботи було реалізовано патерн «Command» для системи управління музичним програвачем. Це дозволило інкапсулювати запити на відтворення, зупинку та перемотку у вигляді окремих об'єктів.

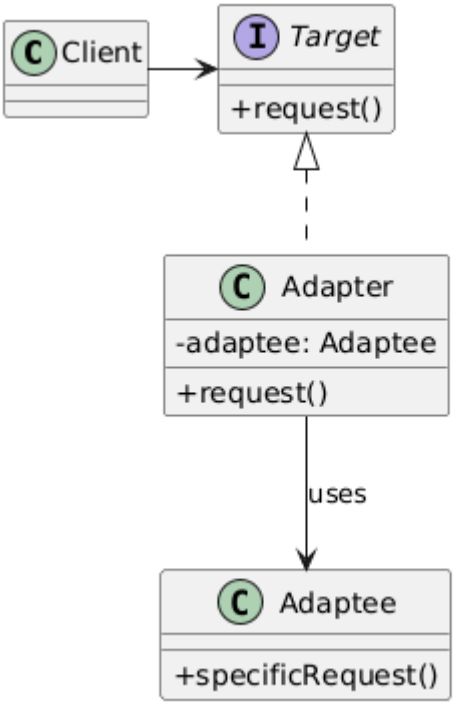
Головною перевагою такого підходу стало повне відокремлення класу, що викликає операцію (Invoker, пульт керування), від класу, що її виконує (Receiver, аудіосистема). Це означає, що ми можемо додавати нові команди (наприклад, "Mute", "Next Track"), створюючи нові класи команд, не змінюючи код існуючих кнопок пульта. Також ця архітектура дозволяє в майбутньому легко додати функціонал черги відтворення (MacroCommand) або скасування останньої дії (Undo), оскільки кожна дія є об'єктом.

Контрольні запитання

1. Яке призначення шаблону «Адаптер»?

Шаблон «Адаптер» (Adapter) призначений для приведення інтерфейсу одного класу до інтерфейсу іншого класу, на який очікує клієнт. Він дозволяє об'єктам з несумісними інтерфейсами працювати разом. Це свого роду "перехідник" між двома різними системами.

2. Нарисуйте структуру шаблону «Адаптер».



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

- Client: Клас, який використовує об'єкти через інтерфейс Target.
- Target: Інтерфейс, який очікує Клієнт.
- Adaptee: Існуючий клас з корисним функціоналом, але несумісним інтерфейсом.
- Adapter: Реалізує інтерфейс Target і зберігає посилання на Adaptee.
- Взаємодія: Клієнт викликає метод request() у Адаптера. Адаптер всередині цього методу конвертує виклик у specificRequest() об'єкта Adaptee.

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

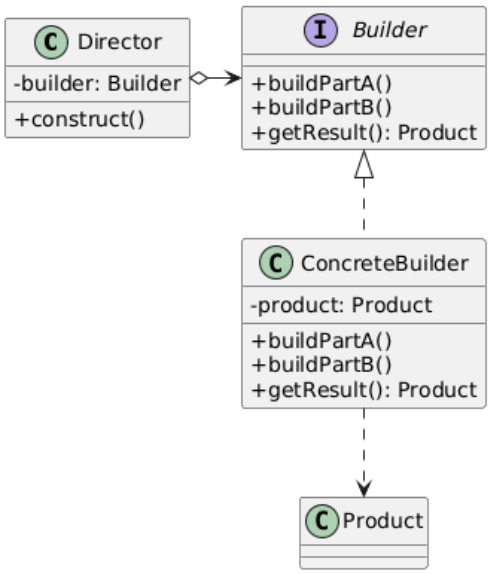
Адаптер об'єкта (Object Adapter): Використовує композицію (зберігає посилання на об'єкт Adaptee). Він реалізує інтерфейс Target і делегує виклики об'єкту Adaptee. Це гнучкіший підхід, що дозволяє адаптувати не тільки сам клас Adaptee, а і його підкласи.

Адаптер класу (Class Adapter): Використовує множинне успадкування (успадковує і від Target, і від Adaptee). Цей підхід можливий лише в мовах, що підтримують множинне успадкування (наприклад, C++). Він не може адаптувати підкласи Adaptee, але дозволяє перевизначати методи Adaptee.

5. Яке призначення шаблону «Будівельник»?

Шаблон «Будівельник» (Builder) відокремлює процес конструювання складного об'єкта від його представлення. Це дозволяє використовувати один і той самий процес конструювання для створення різних представлень об'єкта. Також він допомагає уникнути "телескопічних конструкторів" з великою кількістю параметрів.

6. Нарисуйте структуру шаблону «Будівельник».



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

- Builder: Інтерфейс для створення частин об'єкта Product.
- ConcreteBuilder: Реалізує інтерфейс Builder, конструює та збирає частини продукту. Зберігає створений продукт.
- Director: Керує порядком виконання будівельних кроків. Він знає *як* будувати, але не знає, що саме будується.
- Product: Складний об'єкт, який створюється.

Взаємодія: Клієнт створює ConcreteBuilder і передає його Director. Director викликає методи будівельника по черзі. В кінці Клієнт забирає готовий Product у ConcreteBuilder.

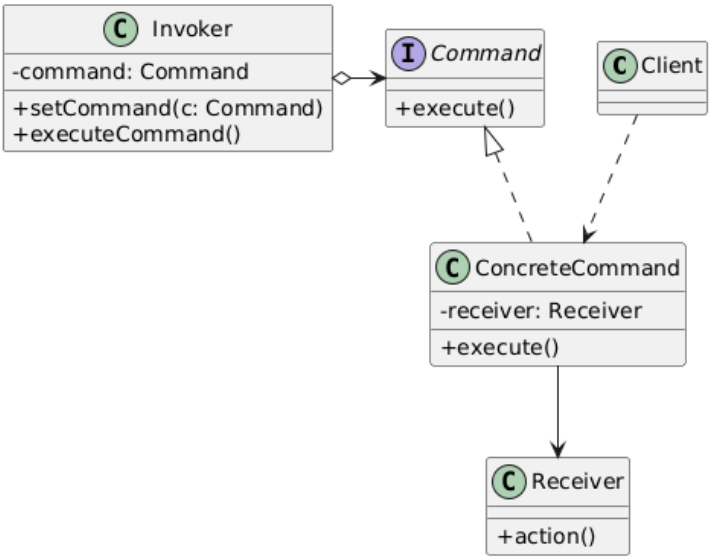
8. У яких випадках варто застосовувати шаблон «Будівельник»"?

- Коли процес створення об'єкта складний і складається з багатьох кроків.
- Коли конструктор класу вимагає занадто багато параметрів, багато з яких є необов'язковими.
- Коли потрібно створювати різні представлення одного й того ж об'єкта (наприклад, звіт у HTML та звіт у PDF), використовуючи однаковий процес побудови.

9. Яке призначення шаблону «Команда»?

Шаблон «Команда» (Command) інкапсулює запит як об'єкт, дозволяючи цим параметризувати клієнтів параметрами запитів, ставити запити в чергу, логувати їх, а також підтримувати операції скасування (Undo) дій.

10. Нарисуйте структуру шаблону «Команда».



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

- Command: Інтерфейс із методом execute().
- ConcreteCommand: Реалізує зв'язок між Receiver та дією. Викликає метод Receiver у своєму execute().
- Receiver (Одержувач): Знає, як виконувати операції бізнес-логіки.
- Invoker (Ініціатор): Зберігає команду і викликає її метод execute(). Не знає нічого про Receiver.
- Client: Створює ConcreteCommand, налаштовує його на певного Receiver і передає команду до Invoker.

12. Розкажіть як працює шаблон «Команда».

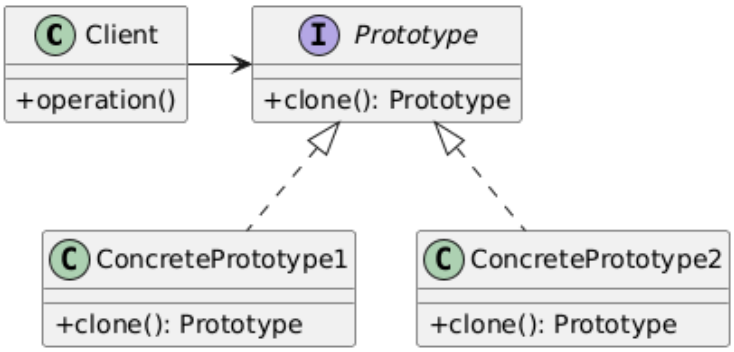
Клієнт створює екземпляр команди та зв'язує його з конкретним одержувачем. Потім ця команда передається ініціатору (наприклад, кнопці). Коли користувач взаємодіє з ініціатором, той викликає метод execute() у команди. Команда, в свою чергу, делегує виконання дії одержувачу. Це розриває пряму залежність між тим, хто викликає дію, і тим, хто її виконує.

13. Яке призначення шаблону «Прототип»?

Шаблон «Прототип» (Prototype) дозволяє копіювати об'єкти, не вдаючись у подробиці їхньої реалізації та не залежачи від їхніх класів. Використовується, коли

створення об'єкта "з нуля" є дорогим (забирає багато ресурсів) або складним, а простіше скопіювати існуючий екземпляр.

14. Нарисуйте структуру шаблону «Прототип».



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

- Prototype: Інтерфейс, який оголошує метод clone().
- ConcretePrototype: Реалізує метод clone() для копіювання самого себе.
- Client: Створює нові об'єкти шляхом звернення до прототипу з вимогою клонувати себе.

Взаємодія: Клієнт замість використання оператора new викликає метод clone() у вже існуючого об'єкта-прототипу.

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

Шаблон «Ланцюжок відповідальності» дозволяє передавати запит послідовно через ланцюжок потенційних обробників, поки один з них не обробить запит.

Приклади:

1. Обробка подій у графічних інтерфейсах (GUI): Наприклад, клік по кнопці. Якщо кнопка не обробляє клік, подія передається батьківському контейнеру (панелі), потім вікну і так далі.
2. Система логування: Повідомлення може бути відправлене в консоль, у файл, або на пошту адміністратору залежно від рівня критичності (Info, Warning, Error), проходячи через ланцюжок логерів.

3. Банкомат: Видача грошей. Запит на суму передається від касети з найбільшими купюрами до менших (видати 1000, решту передати далі -> видати 500 -> видати 100).