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

із дисципліни «*Технології розроблення програмного забезпечення*»

Патерни проектування

Аудіоредактор

Виконала
студентка групи ІА–34
Кузьменко В. С.

Перевірив
викладач
Мягкий М. Ю.

Київ 2025

ЗМІСТ

Тема: Патерни проектування.

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

Завдання:

• Ознайомитись з короткими теоретичними відомостями.

• Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

• Реалізувати один з розглянутих шаблонів за обраною темою.

• Реалізувати не менше 3-х класів відповідно до обраної теми.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

Тема роботи:

5 Аудіо редактор (singleton, adapter, observer, mediator, composite, client-server)
Аудіо редактор повинен володіти наступним функціоналом: представлення аудіо даних будь-якого формату в WAVE-формі, вибір і подальші операції копіювання / вставки / вирізання / деформації по сегменту аудіозапису, можливість роботи з декількома звуковими доріжками, кодування в найбільш поширених форматах (ogg, flac, mp3).

Теоретичні відомості

Анти-шаблони проєктування (anti-patterns) — це повторювані рішення проблем у розробці програмного забезпечення, які виявляються неефективними або шкідливими в довгостроковій перспективі. Вони виникають через недоліки у розумінні принципів проектування, тиск дедлайнів або недостатній досвід розробників. Анти-шаблони можуть призводити до зниження продуктивності, складності коду, проблем із масштабованістю та підтримкою.

Приклади анти-шаблонів:

1. Spaghetti Code — код без структури, важкий для розуміння.
2. God Object — об’єкт, який має занадто багато відповідальностей.
3. Golden Hammer — використання одного підходу для вирішення всіх проблем.
4. Magic Numbers — використання числових значень у коді без пояснення їхнього значення.

Анти-патерни в управлінні розробкою ПЗ:

- Дим і дзеркала: демонстрація незавершених функцій.
- Роздування ПЗ: збільшення вимог до ресурсів без виправданих змін.
- Функції для галочки: додавання непотрібних функцій для вигляду.

Анти-патерни в розробці ПЗ:

- Великий клубок бруду: складні і непідтримувані системи.
- Інверсія абстракції: приховування функціоналу без необхідності.
- Роздування інтерфейсу: занадто великі інтерфейси.

Анти-патерни в ООП:

- Божественний об'єкт: концентрація надмірної логіки в одному класі.

- Самотність (Singletonitis): надмірне використання патерну "Одинак".

Анти-патерни в програмуванні:

- Непотрібна складність: введення зайвої складності.
- Жорстке кодування: використання фіксованих значень в коді замість гнучких рішень.
- Спагеті-код: заплутаний і важкий для підтримки код.

Методологічні анти-патерни:

- Копіювання-вставка: копіювання коду замість створення спільних рішень.
- Передчасна оптимізація: оптимізація без достатньої інформації.

Adapter (Адаптер)

Призначення: дозволяє об'єктам із несумісними інтерфейсами працювати разом, забезпечуючи проміжний інтерфейс.

Коли використовувати:

- Коли потрібно узгодити інтерфейс існуючого класу з інтерфейсом, який очікує клієнт.
- Для роботи із застарілими класами без їх зміни.

Реалізація: створюється клас-адаптер, який трансформує інтерфейс одного класу в інтерфейс іншого.

Переваги: зменшує залежність від зовнішніх бібліотек, сприяє повторному використанню коду.

Недоліки: ускладнює структуру, якщо занадто багато адаптерів.

Builder (Будівельник)

Призначення: розділяє створення складного об’єкта на етапи, дозволяючи поетапно будувати різні його представлення.

Коли використовувати:

- Коли потрібно створювати складні об'єкти з великою кількістю параметрів.
- Для спрощення читабельності коду при конфігуруванні об’єктів.

Реалізація: використовується клас-будівельник, який крок за кроком створює об'єкт. Завершений об'єкт повертається через метод build().

Переваги: забезпечує гнучкість у створенні об'єктів, робить код зрозумілішим.

Недоліки: збільшує кількість класів.

Command (Команда)

Призначення: інкапсулює запит у вигляді об'єкта, дозволяючи відкладати виконання запиту, реєструвати його або підтримувати скасування операцій.

Коли використовувати:

- Для реалізації системи скасування та повтору дій.
- Для створення черги команд або логування.

Реалізація: створюється інтерфейс Command із методом execute(). Конкретні команди реалізують цей інтерфейс і виконують певну дію.

Переваги: знижує залежність між відправником і отримувачем запиту, дозволяє створювати складні макрокоманди.

Недоліки: ускладнює архітектуру через додаткові класи.

Chain of Responsibility (Ланцюг відповідальності)

Призначення: дозволяє передавати запити вздовж ланцюга обробників, поки один із них не виконає запит.

Коли використовувати:

- Коли потрібно уникнути тісного зв’язку між відправником і отримувачем запиту.

- Для реалізації динамічного оброблення запитів.

Реалізація: кожен обробник містить посилання на наступний обробник у ланцюгу. Запит передається від одного обробника до іншого.

Переваги: зменшує кількість залежностей між об’єктами, полегшує додавання нових обробників.

Недоліки: може бути важко відстежити, який обробник виконав запит.

Prototype (Прототип)

Призначення: дозволяє створювати нові об'єкти, копіюючи вже існуючі, замість створення об'єктів із нуля.

Коли використовувати:

- Коли створення об'єкта є дорогим або складним.
- Коли потрібно уникнути створення об’єктів через конструктор.

Реалізація: клас має метод clone(), який повертає копію об'єкта.

Переваги: швидке створення об'єктів, дозволяє уникати складної ініціалізації.

Недоліки: складність у копіюванні об'єктів із вкладеними залежностями.

Хід роботи

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціоналу робочої програми у вигляді 3 класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
3. Застосування одного з розглянутих шаблонів при реалізації програми.

Шаблон Adapter

Структура та обґрунтування вибору

Шаблон «Адаптер» використовується для поєднання двох систем із несумісними інтерфейсами шляхом створення проміжного класу-«перехідника». Це дозволяє забезпечити взаємодію між різними компонентами без потреби змінювати їх вихідний код.

У цьому проєкті шаблон Adapter застосовано для узгодження роботи між класами ієрархії Audiotrack та іншими частинами програми або зовнішніми бібліотеками. Основне його призначення - уніфікувати доступ до аудіоданих і приховати деталі реалізації різних форматів (Mp3, Flac, Ogg).

Клас AudioAdapter виконує роль «перекладача», що надає єдиний інтерфейс для роботи з об’єктами Audiotrack, незалежно від їхнього типу. Наприклад, метод adaptAttributes() форматує аудіоатрибути у вигляді, сумісному з вимогами зовнішньої бібліотеки, а adaptFile() повертає об’єкт файлу у стандартизованому форматі.

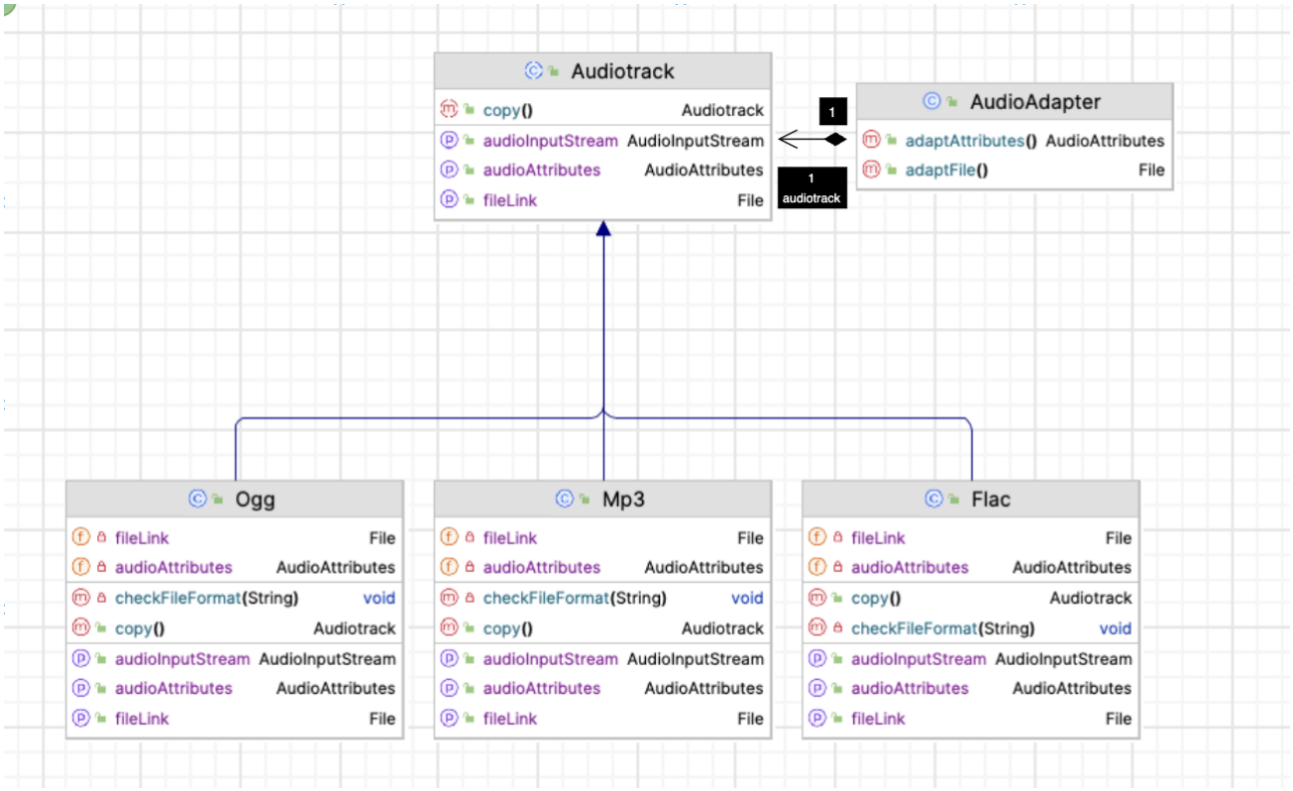


Рисунок 1. Структура класів реалізованого шаблону Adapter

Audiotrack - це абстрактний клас, який задає спільний інтерфейс для всіх аудіоформатів. Він містить методи для отримання характеристик аудіо, доступу до файлу та створення копії треку. Його нащадки - Mp3, Ogg і Flac - реалізують ці методи відповідно до особливостей кожного формату.

AudioAdapter виступає проміжною ланкою між різними реалізаціями Audiotrack і рештою програми. Він містить екземпляр класу Audiotrack і надає уніфікований інтерфейс для доступу до даних незалежно від формату файлу. Таким чином, адаптер узгоджує роботу з різними типами аудіо, забезпечуючи їх використання в єдиному стандартизованому вигляді.

Код системи

Лістинг 1- abstract class Audiotrack

```
package org.example.audiotrack;

import it.sauronsoftware.jave.AudioAttributes;

import javax.sound.sampled.AudioInputStream;
import java.io.File;

public abstract class Audiotrack {
    public abstract AudioAttributes getAudioAttributes();
    public abstract File getFileLink();
    public abstract AudioInputStream getAudioInputStream();
    public abstract Audiotrack copy();
}
```

Лістинг 2 - class AudioAdapter

```
package org.example.audiotrack;

import it.sauronsoftware.jave.AudioAttributes;

import java.io.File;

public class AudioAdapter{
    private final Audiotrack audiotrack;

    public AudioAdapter(Audiotrack audiotrack) {
        this.audiotrack = audiotrack;
    }

    public AudioAttributes adaptAttributes() {
        return audiotrack.getAudioAttributes();
```

```
    }

    public File adaptFile() {
        return audiotrack.getFileLink();
    }
}
```

Лістинг 3 - class Flac

```
package org.example.audiotrack;

import it.sauronsoftware.jave.AudioAttributes;

import javax.sound.sampled.AudioInputStream;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;


public class Flac extends Audiotrack {
    private AudioAttributes audioAttributes;
    private File fileLink;

    public Flac(String filePath) {
        checkFileFormat(filePath);
        File audioFile = new File(filePath);

        fileLink = audioFile;
        audioAttributes = new AudioAttributes();
        audioAttributes.setCodec("flac");
        audioAttributes.setBitRate(128000);
        audioAttributes.setChannels(2);
        audioAttributes.setSamplingRate(44100);
    }


    private void checkFileFormat(String path) {
        try {
            if (!Files.probeContentType(Path.of(path)).equals("audio/x-flac")) {
                throw new IllegalArgumentException("Wrong audio format");
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    public AudioAttributes getAudioAttributes() {
        return audioAttributes;
    }

    public File getFileLink() {
        return fileLink;
    }

    @Override
    public AudioInputStream getAudioInputStream() {
        return null;
    }

    @Override
    public Audiotrack copy() {
        return null;
    }
}
```

Лістинг 3 - class Ogg

```
package org.example.audiotrack;

import it.sauronsoftware.jave.AudioAttributes;

import javax.sound.sampled.AudioInputStream;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class Ogg extends Audiotrack {
    private AudioAttributes audioAttributes;
    private File fileLink;

    public Ogg(String filePath) {
        checkFileFormat(filePath);
        File audioFile = new File(filePath);
        fileLink = audioFile;
        audioAttributes = new AudioAttributes();
        audioAttributes.setCodec("vorbis");
        audioAttributes.setBitRate(128000);
        audioAttributes.setChannels(2);
        audioAttributes.setSamplingRate(44100);
    }


    private void checkFileFormat(String path) {
        try {
            if (!Files.probeContentType(Path.of(path)).equals("audio/ogg")) {
                throw new IllegalArgumentException("Wrong audio format");
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }


    public AudioAttributes getAudioAttributes() {
        return audioAttributes;
    }


    public File getFileLink() {
        return fileLink;
    }

    @Override
    public AudioInputStream getAudioInputStream() {
        return null;
    }

    @Override
    public Audiotrack copy() {
        return null;
    }
}
```

Лістинг 4 – class Mp3

```
package org.example.audiotrack;

import it.sauronsoftware.jave.AudioAttributes;
```

```
import javax.sound.sampled.AudioInputStream;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class Mp3 extends Audiotrack {
    private AudioAttributes audioAttributes;
    private File fileLink;

    public Mp3(String filePath) {
        checkFileFormat(filePath);
        File audioFile = new File(filePath);
        fileLink = audioFile;
        audioAttributes = new AudioAttributes();
        audioAttributes.setCodec("libmp3lame");
        audioAttributes.setBitRate(128000);
        audioAttributes.setChannels(2);
        audioAttributes.setSamplingRate(44100);
    }


    private void checkFileFormat(String path) {
        try {
            if (!Files.probeContentType(Path.of(path)).equals("audio/mpeg")) {
                throw new IllegalArgumentException("Wrong audio format");
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    public AudioAttributes getAudioAttributes() {
        return audioAttributes;
    }

    public File getFileLink() {
        return fileLink;
    }

    @Override
    public AudioInputStream getAudioInputStream() {
        return null;
    }

    @Override
    public Audiotrack copy() {
        return null;
    }
}
```

Клас **Audiotrack** є базовим (абстрактним) класом, який визначає спільний інтерфейс для всіх типів аудіоформатів. Він містить абстрактні методи, які мають бути реалізовані в похідних класах:

- getAudioAttributes() - повертає атрибути аудіо (бітрейт, частота дискретизації, кількість каналів тощо);

- getFileLink() - повертає посилання на аудіофайл;
- getAudioInputStream() - повертає потік даних аудіо;
- copy() - створює копію об’єкта треку.

Клас **AudioAdapter** реалізує патерн «Адаптер», який дозволяє працювати з різними типами аудіоформатів через уніфікований інтерфейс. Він містить поле audiotrack, що є посиланням на будь-який об’єкт-нащадок класу Audiotrack (наприклад, Mp3, Ogg, або Flac).

Основні методи:

- adaptAttributes() - повертає атрибути аудіо через виклик getAudioAttributes() у відповідному форматі;
- adaptFile() - повертає посилання на файл через метод getFileLink().

Завдяки цьому адаптер усуває відмінності між форматами, дозволяючи використовувати їх однаково.

Клас **Flac** є реалізацією абстрактного класу Audiotrack для формату FLAC. У конструкторі відбувається перевірка правильності формату файлу (через Files.probeContentType()), після чого задаються атрибути звуку:

- кодек - "flac"
- бітрейт -128000
- кількість каналів - 2
- частота дискретизації - 44100 Гц.

Методи getAudioAttributes() та getFileLink() повертають відповідні значення, а getAudioInputStream() і copy() залишені для подальшої реалізації.

Клас **Ogg** аналогічний до Flac, але призначений для формату OGG. Перевірка формату здійснюється на "audio/ogg", а кодек визначено як "vorbis". Встановлюються ті самі основні параметри звуку: бітрейт, канали, частота дискретизації. Таким чином, клас описує специфіку аудіо OGG, дотримуючись єдиної архітектури.

Клас **Mp3** реалізує підтримку формату MP3. При створенні об’єкта виконується перевірка MIME-типу "audio/mpeg". Кодек задається як "libmp3lame", решта параметрів (бітрейт, канали, частота дискретизації) збігаються з іншими форматами. Методи реалізовані аналогічно до інших форматів, що забезпечує сумісність і уніфіковану роботу з аудіо.

Висновок

Під час виконання лабораторної роботи було детально проаналізовано п’ять ключових шаблонів проєктування: Adapter**,** Builder**,** Command**,** Chain of Responsibility та Prototype. Кожен із них виконує власну функцію, допомагаючи створювати більш гнучку, зрозумілу та підтримувану архітектуру програмного забезпечення.

Зокрема:

- Adapter забезпечує взаємодію між класами з несумісними інтерфейсами.
- Builder спрощує процес створення складних об’єктів, розділяючи етапи побудови й кінцевий результат.
- Command інкапсулює запити у вигляді об’єктів, що дає змогу гнучко керувати виконанням операцій.
- Chain of Responsibility передає запит уздовж ланцюжка обробників, що дозволяє легко змінювати або розширювати логіку обробки.
- Prototype дає можливість створювати нові об’єкти шляхом копіювання вже існуючих, зберігаючи їхній стан.

Особливий акцент у роботі зроблено на шаблоні Adapter, який був реалізований у межах проєкту «Аудіоредактор». Його використання дозволило об’єднати власний інтерфейс програми з зовнішньою бібліотекою для обробки аудіофайлів. Завдяки цьому вдалося уникнути змін у коді бібліотеки, зменшити залежність від її внутрішньої реалізації та забезпечити універсальність у роботі з різними аудіоформатами.

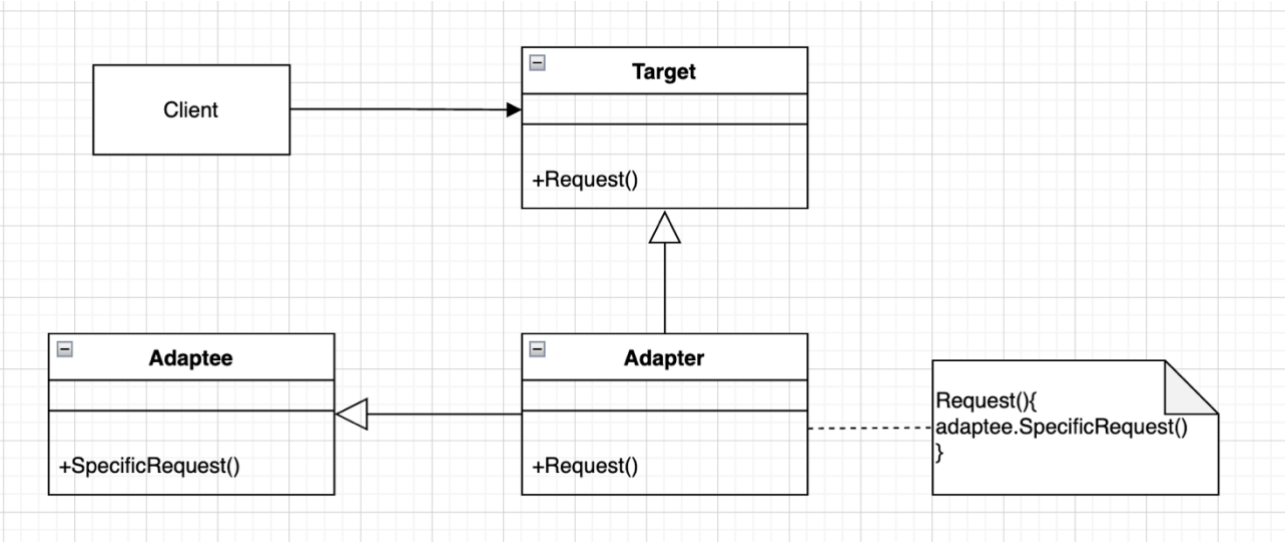
Застосування шаблонів проєктування продемонструвало їхню ефективність у розв’язанні типових інженерних задач і побудові масштабованої архітектури. Практична реалізація шаблону Adapter у проєкті підтвердила його значення для інтеграції сторонніх компонентів, підвищення гнучкості та модульності системи. Такий підхід спрощує подальший розвиток і підтримку програмного продукту, роблячи його більш стійким до змін і розширень у майбутньому.

Відповіді на теоретичні питання

1. Призначення шаблону «Адаптер»

Шаблон Adapter (Адаптер) призначений для узгодження роботи класів із несумісними інтерфейсами. Він створює “перехідник”, який дозволяє об’єктам взаємодіяти, не змінюючи їхній вихідний код.

2. Структура шаблону «Адаптер»



3. Класи шаблону «Адаптер» і взаємодія

- Target - інтерфейс, який очікує клієнт.
- Client - клас, що використовує Target.
- Adaptee - клас із несумісним інтерфейсом.
- Adapter - проміжний клас, який реалізує Target і викликає методи Adaptee.

Взаємодія:
Клієнт звертається до адаптера через інтерфейс Target. Адаптер, у свою чергу, викликає відповідні методи класу Adaptee.

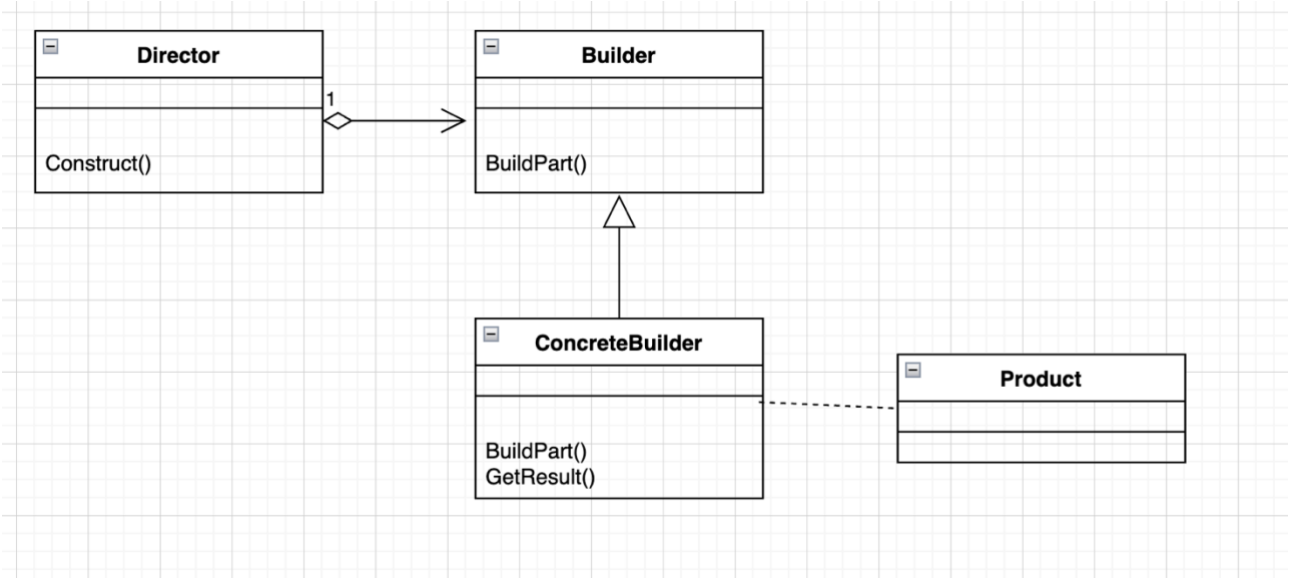
4. Різниця між реалізацією адаптера на рівні об’єктів і класів

- Об’єктний адаптер - використовує композицію (адаптер має посилання на об’єкт Adaptee). Гнучкіший варіант.
- Класовий адаптер - використовує наслідування від Adaptee. Може перевизначати методи, але обмежений мовами, які не підтримують множинне наслідування (наприклад, Java).

5. Призначення шаблону «Будівельник»

Шаблон Builder (Будівельник) використовується для поетапного створення складних об’єктів. Він відокремлює процес побудови об’єкта від його представлення, щоб одним і тим самим процесом можна було створювати різні варіанти об’єктів.

6. Структура шаблону «Будівельник»



7. Класи шаблону «Будівельник» і взаємодія

- Builder - абстрактний інтерфейс для створення частин продукту.
- ConcreteBuilder - конкретна реалізація, що створює об’єкт.
- Director - керує процесом побудови, використовуючи Builder.
- Product - кінцевий об’єкт, який створюється.

Взаємодія:
Director викликає методи Builder у певній послідовності: ConcreteBuilder формує Product - результат повертається клієнту.

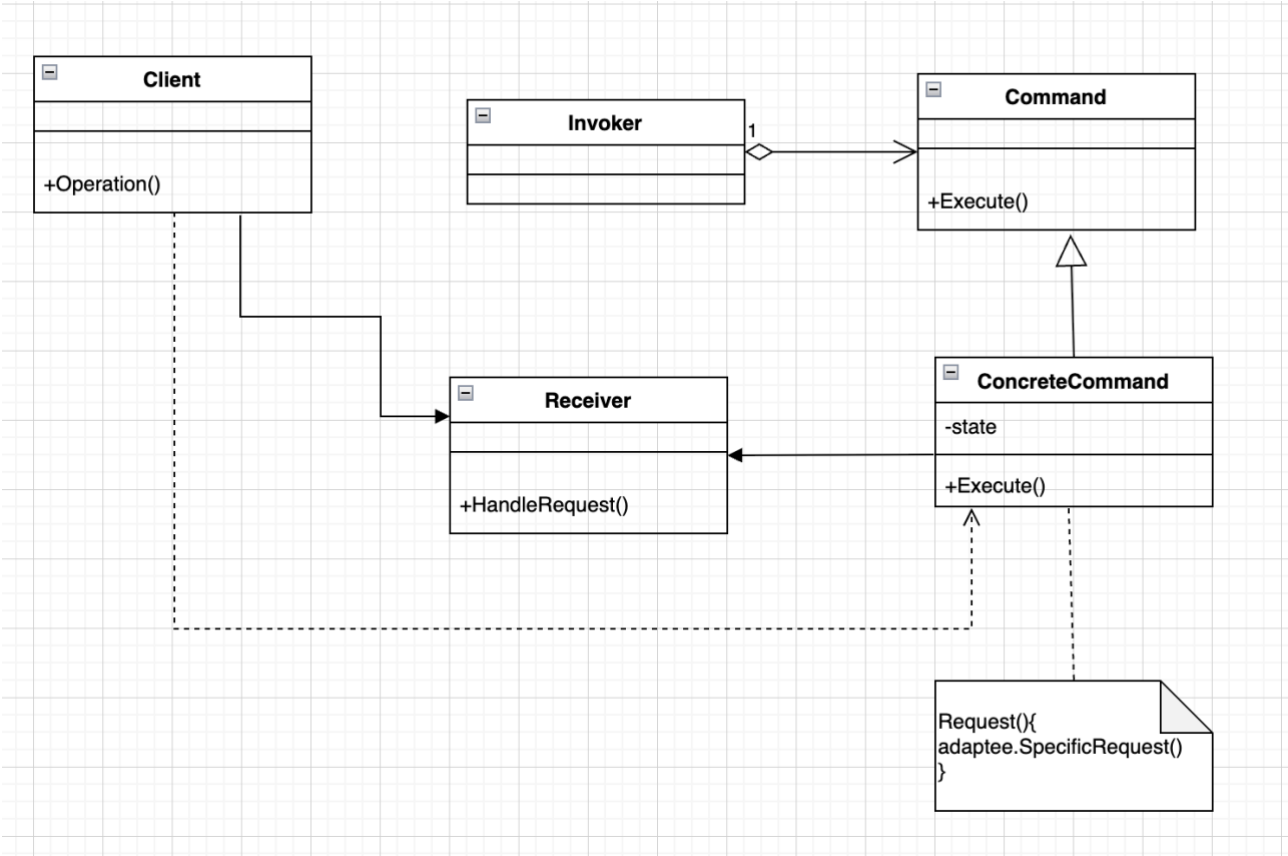
8. Коли застосовувати шаблон «Будівельник»

- Коли об’єкт має багато параметрів, і потрібно спростити його створення.
- Коли потрібно створювати різні представлення одного об’єкта.
- Коли важливо ізолювати процес побудови від структури об’єкта.

9. Призначення шаблону «Команда»

Шаблон Command (Команда) перетворює запит у незалежний об’єкт, який містить усю інформацію для виконання дії. Це дозволяє зберігати, чергувати, скасовувати або повторно виконувати команди.

10. Структура шаблону «Команда»



11. Класи шаблону «Команда» і взаємодія

- Command - інтерфейс із методом execute().
- ConcreteCommand - конкретна команда, що виконує дію через Receiver.
- Receiver - виконує реальну роботу.
- Invoker - ініціює виконання команди.
- Client - створює команди та передає їх виконавцю.

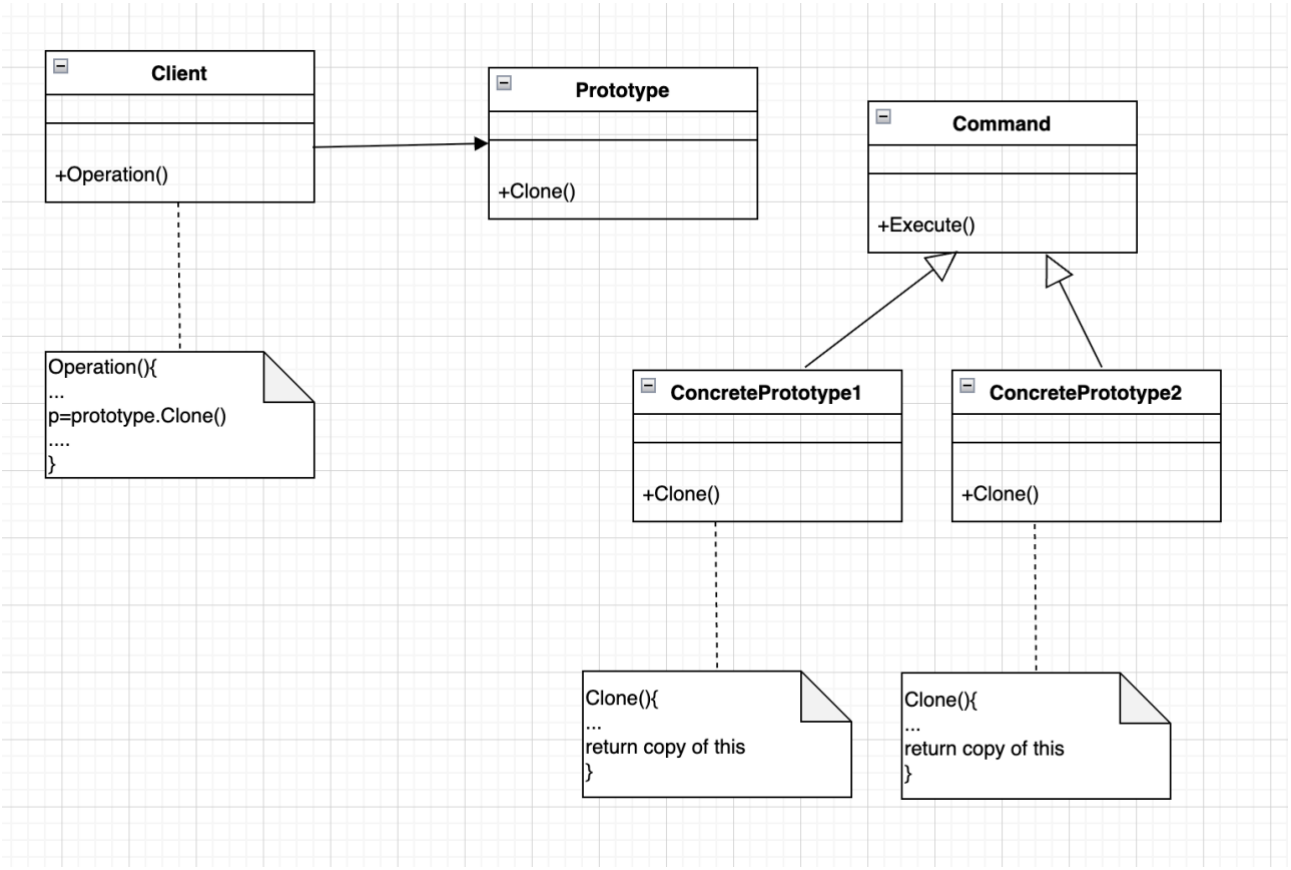
12. Як працює шаблон «Команда»

Клієнт створює об’єкт команди і передає його виконавцю (Invoker). Коли потрібно виконати дію, виконавець викликає метод execute() команди, а та звертається до Receiver, який здійснює конкретну операцію.

13. Призначення шаблону «Прототип»

Шаблон Prototype (Прототип) дозволяє створювати нові об’єкти шляхом копіювання вже існуючих. Це спрощує створення складних об’єктів, коли їхнє створення “з нуля” є ресурсомістким.

14. Структура шаблону «Прототип»



15. Класи шаблону «Прототип» і взаємодія

- Prototype - оголошує метод clone().
- ConcretePrototype - реалізує клонування об’єкта.
- Client - створює нові об’єкти через копіювання прототипів.

Взаємодія:

Client викликає метод clone() у Prototype, отримуючи новий екземпляр об’єкта з таким самим станом.

16. Приклади використання шаблону «Ланцюжок відповідальності»

- Обробка подій у графічному інтерфейсі (GUI), коли подія передається від одного елемента до іншого.
- Обробка запитів у вебсервері (фільтри, middleware).
- Логування з різними рівнями (info, warning, error).
- Системи технічної підтримки, де запит передається від нижчого до вищого рівня.
- Модуль перевірки даних, де кожен обробник перевіряє свій аспект.