



Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Лабораторна робота №5
Технологія розробки програмного забезпечення
«Патерни проектування»

Виконала
студентка групи ІА–32:
Ткачук М. С.

Перевірив:
Мягкий М. Ю.

Київ 2025

Зміст

Теоретичні відомості	4
Діаграма класів	8
Патерн Adapter.....	10
Фрагменти коду	12
Вихідний код	21
Висновок	21
Контрольні запитання.....	22

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

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

Шаблон «Adapter»

Призначення:

Патерн Adapter використовується для узгодження інтерфейсів двох класів, які несумісні між собою. Він дозволяє «обгорнути» сторонній або застарілий компонент у новий інтерфейс, щоб клієнтський код працював з єдиною абстракцією.

Проблема:

Коли система працює з різними модулями, що мають різні інтерфейси (наприклад, різні аудіо-кодеки), код ускладнюється великою кількістю умов та перевірок.

Суть рішення:

Створюється єдиний інтерфейс (наприклад, IPlayer), і для кожного компоненту свій адаптер, який переводить виклики до відповідного формату. Клієнт взаємодіє лише з адаптером, а не з конкретною реалізацією.

Переваги:

- Просте додавання нових форматів без зміни існуючого коду.
- Зменшення залежностей між компонентами.

Недоліки:

Збільшується кількість класів.

Шаблон «Builder»

Призначення:

Патерн Builder відокремлює процес створення складного об'єкта від його структури. Один і той же процес побудови може створювати різні варіанти об'єкта.

Проблема:

Коли створення об'єкта складається з багатьох кроків (наприклад, формування HTTP-відповіді), код конструктора стає надто громіздким і негнучким.

Суть рішення:

Створюється абстрактний будівельник з методами «побудувати частину», а директор керує послідовністю викликів. Конкретні реалізації будівельника створюють різні версії продуктів.

Переваги:

- Контрольований процес створення.
- Можливість створювати різні варіанти об'єктів одним алгоритмом.

Недоліки:

Клієнт залежить від конкретного будівельника, якщо він самостійно отримує результат.

Шаблон «Command»**Призначення:**

Патерн Command перетворює операцію на окремий об'єкт. Команда зберігає дію, її параметри та може підтримувати скасування, черги, логування.

Проблема:

У UI-застосунку одна дія може викликатися з кнопки, меню чи контекстного меню. Без патерну логіка дублюється в кількох обробниках.

Суть рішення:

Створюється інтерфейс ICommand з методом Execute().

Кнопки та пункти меню не містять логіки вони викликають відповідну команду. Це відокремлює UI від бізнес-логіки.

Переваги:

- Підтримка undo/redo.

- Легке додавання нових команд (ОСР).
- Відсутність дублювання коду.

Недоліки:

Збільшується кількість класів.

Шаблон «Chain of Responsibility»**Призначення:**

Патерн Chain of Responsibility дозволяє передавати запит послідовністю обробників, де кожен може обробити запит або передати далі.

Проблема:

Складні UI-форми, де кожен елемент може додати власні пункти в контекстне меню. Логіка швидко перетворюється на хаос із перевітками.

Суть рішення:

Кожен компонент має метод UpdateContextMenu() та посилання на «батьківський» елемент. Компонент обробляє меню та передає виклик вгору по ланцюгу.

Переваги:

- Менше залежностей між елементами.
- Гнучке додавання та видалення обробників.

Недоліки:

Запит може ніким не бути оброблений.

Шаблон «Prototype»**Призначення:**

Патерн Prototype створює нові об'єкти шляхом клонування наявного «шаблонного» екземпляра. Це корисно, коли об'єкти складні або їх створення ресурсозатратне.

Проблема:

У редакторі рівнів потрібно створювати багато типів об'єктів (стіни, платформи, декоративні елементи). Без прототипів доведеться створювати паралельну ієрархію кнопок або фабрик.

Суть рішення:

Базовий клас `GameObject` має метод `Clone()`. Кнопки зберігають посилання на прототип і створюють нові об'єкти через клонування.

Переваги:

- Швидке створення складних об'єктів.
- Зменшує ієрархію класів.
- Гнучкість - клоновані об'єкти легко модифікувати незалежно.

Недоліки:

Реалізація глибокого копіювання може бути складною.

Велика кількість прототипів може ускладнювати проект.

Хід Роботи

Діаграма класів

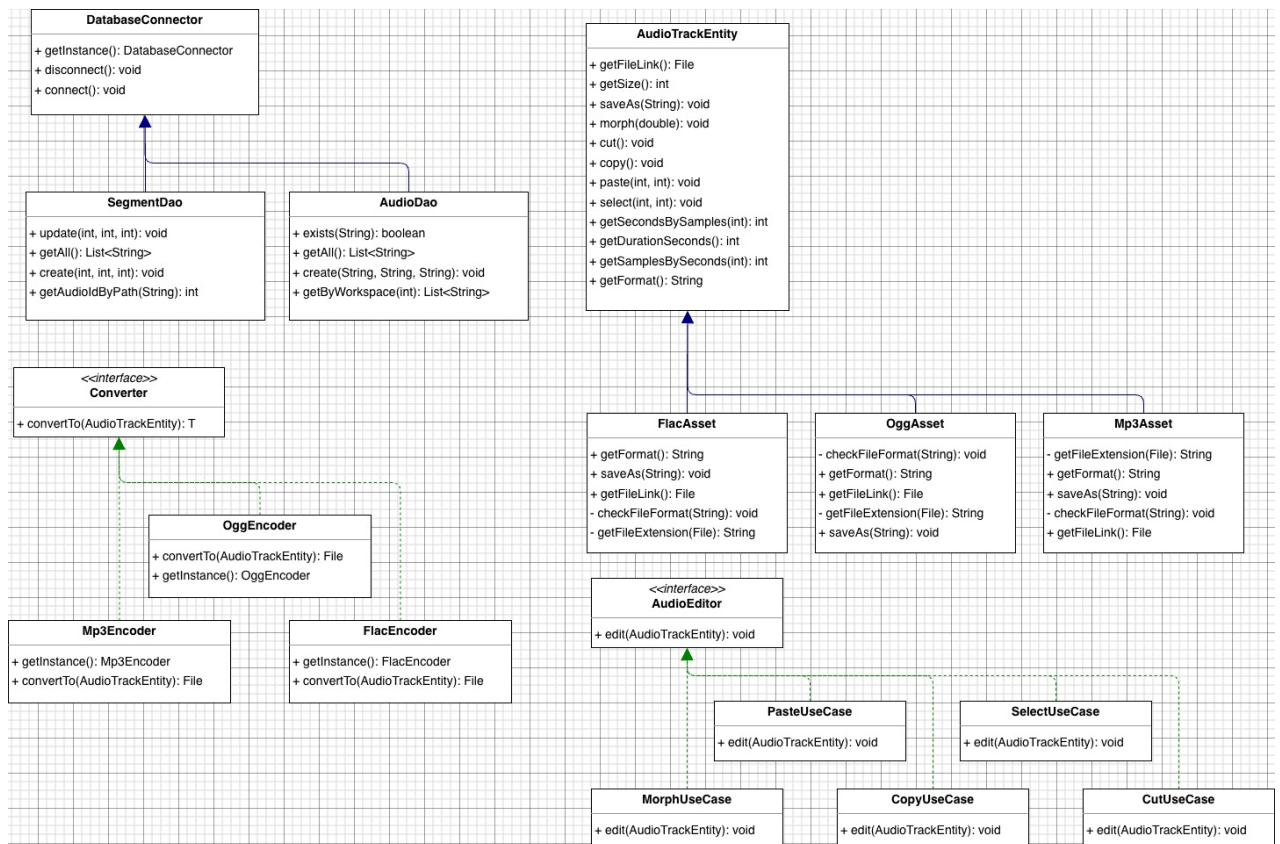


Рисунок 1 – Діаграма класів

На рисунку 1 наведено діаграму класів аудіоредактора, що відображає логічну структуру програмного забезпечення, основні класи, їхні атрибути, методи та взаємозв'язки між ними.

Центральним елементом системи є клас **AudioTrackEntity**, який описує сутність аудіотреку та містить методи для виконання основних операцій редагування **cut()**, **copy()**, **paste()**, **morph()**, а також методи для роботи з форматом файлу, отримання семплів і тривалості (**getDurationSeconds()**, **getFormat()**, **getSampleBySeconds()**).

Цей клас є базовим для роботи з різними типами звукових файлів і виступає ядром усієї системи.

Від `AudioTrackEntity` наслідуються класи `Mp3Asset`, `OggAsset` та `FlacAsset`, що реалізують підтримку різних аудіоформатів. Кожен із цих класів має власні методи для збереження (`saveAs()`), перевірки формату (`checkFileFormat()`), а також отримання розширення файлу (`getFileExtension()`), що дозволяє системі працювати з популярними форматами MP3, OGG та FLAC.

Окремий клас `AudioEditor` реалізує високорівневу логіку редагування файлів. Він містить метод `edit()`, який викликає відповідні операції залежно від сценарію редагування: копіювання, вставлення, вирізання або деформації звуку. Для реалізації цих сценаріїв використовуються окремі класи `CutUseCase`, `CopyUseCase`, `SelectUseCase` і `MorphUseCase`, кожен із яких виконує власну операцію редагування через метод `edit(AudioTrackEntity)`.

Для кодування звукових доріжок у різні формати використовується абстрактний клас `Converter<T>`, який містить універсальний метод `convertTo(AudioTrackEntity)`. Від нього наслідуються конкретні реалізації `Mp3Encoder`, `OggEncoder` та `FlacEncoder`, які забезпечують перетворення аудіо у відповідний формат.

Компонент `DatabaseConnector` відповідає за взаємодію з локальною базою даних SQLite. Він реалізує підключення, відключення та отримання екземпляра підключення (`getInstance()`, `connect()`, `disconnect()`). З ним працюють два DAO-класи `AudioDao` і `SegmentDao`, які реалізують доступ до даних аудіофайлів і сегментів. `AudioDao` забезпечує методи `create()`, `getAll()`, `exists()` і `getByWorkspace()`, а `SegmentDao`: операції `create()`, `update()` та `getAudioIdByPath()`. Це дає змогу системі зберігати інформацію про проекти, сегменти та аудіофайли у структурованому вигляді.

Діаграма класів демонструє, як система аудіоредактора розділена на логічні шари:

- рівень даних (DAO-класи, DatabaseConnector),
- рівень бізнес-логіки (AudioEditor, UseCase-класи),
- рівень моделей (AudioTrackEntity і похідні класи для форматів),
- рівень перетворень (Converter і Encoder-класи).

Патерн Adapter

Патерн Adapter це шаблон проєктування, який дозволяє узгодити інтерфейси різних компонентів системи, роблячи їх сумісними без необхідності змінювати їх внутрішню реалізацію. Його основна ідея полягає в тому, щоб створити проміжний об'єкт адаптер, який перетворює один інтерфейс у інший. Завдяки цьому компоненти з різною структурою даних або різними методами можуть коректно взаємодіяти між собою. Використання Adapter особливо доречне в системах, де потрібно інтегрувати нові модулі без переписування існуючої архітектури

У межах розроблюваного аудіоредактора патерн Adapter застосовано для уніфікації роботи з різними форматами аудіо. Оскільки кожен формат файлу має власні правила структурування даних, зберігання метаданих та доступу до інформації, пряме використання цих класів у високорівневих алгоритмах редагування ускладнювало б систему. Саме тому було впроваджено клас SegmentEncodingAdapter, який адаптує об'єкти типу SegmentEntity до форматів FlacAsset, OggAsset і Mp3Asset.

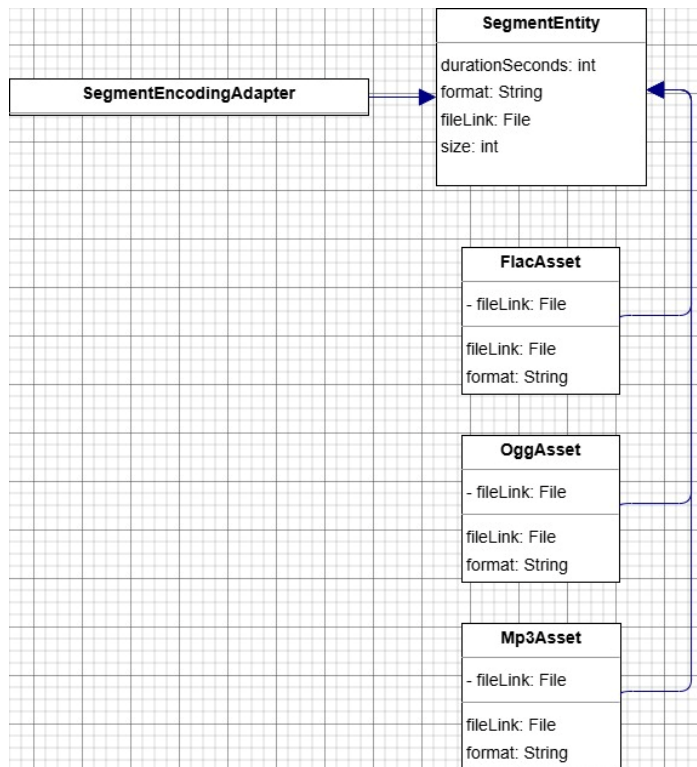


Рисунок 2 – Схема патерну Adapter

Фрагменты коду:

App.java

```
package app.soundlab.audiotrack;
```

```
public class App {
```

```
    public static void main(String[] args) {
```

```
        SegmentEntity legacySegment = SegmentEntity.microphoneCapture(
```

```
            "Lab 5 Groove",
```

```
            "Engineer Ola",
```

```
            5
```

```
        );
```

```
        EncodingRequest adapter = new SegmentEncodingAdapter(legacySegment,  
"mp3");
```

```
        ConsoleAudioEncoder encoder = new ConsoleAudioEncoder();
```

```
        encoder.export(adapter);
```

```
    }
```

```
}
```

ConsoleAudioEncoder.java

```
package app.soundlab.audiotrack;
```

```
import java.io.IOException;

import java.io.InputStream;

public class ConsoleAudioEncoder {

    public void export(EncodingRequest request) {

        System.out.println("Encoding job started:");

        System.out.println("Summary: " + request.summary());

        System.out.println("Target codec: " + request.targetCodec());

        System.out.println("Duration (s): " + request.durationSeconds());

        int bytes = consume(request.dataStream());

        System.out.println("Streamed bytes: " + bytes);

        System.out.println("=== Encoding job finished ===");

    }

    private int consume(InputStream inputStream) {

        byte[] buffer = new byte[1024];

        int total = 0;

        int read;

        try (InputStream stream = inputStream) {

            while ((read = stream.read(buffer)) != -1) {

                total += read;

            }

        }

    }

}
```

```
    }  
  
    } catch (IOException e) {  
  
        throw new IllegalStateException("Failed to consume stream: " +  
e.getMessage(), e);  
  
    }  
  
    return total;  
  
    }  
}
```

EncoderRequest.java

```
package app.soundlab.audiotrack;
```

```
import java.io.InputStream;
```

```
public interface EncodingRequest {
```

```
    String targetCodec();
```

```
    InputStream dataStream();
```

```
    int durationSeconds();
```

```
    String summary();
```

```
}
```

SegmentEncodingAdapter.java

```
package app.soundlab.audiotrack;
```

```
import java.io.ByteArrayInputStream;
```

```
import java.io.ByteArrayOutputStream;
```

```
import java.io.InputStream;
```

```
public class SegmentEncodingAdapter implements EncodingRequest {
```

```
    private final SegmentEntity legacySegment;
```

```
    private final String desiredCodec;
```

```
    public SegmentEncodingAdapter(SegmentEntity legacySegment, String  
desiredCodec) {
```

```
        this.legacySegment = legacySegment;
```

```
        this.desiredCodec = desiredCodec;
```

```
    }
```

```
@Override
```

```
public String targetCodec() {
```

```
    return desiredCodec;
```

```
}
```

```
@Override
```

```
public InputStream dataStream() {  
  
    byte[] legacySamples = legacySegment.dumpSamples();  
  
    ByteArrayOutputStream normalized = new  
ByteArrayOutputStream(legacySamples.length * 2);  
  
    for (byte sample : legacySamples) {  
  
        normalized.write(sample);  
  
        normalized.write(0);  
  
    }  
  
    return new ByteArrayInputStream(normalized.toByteArray());  
  
}
```

```
@Override
```

```
public int durationSeconds() {  
  
    int seconds = legacySegment.rawSampleCount() /  
legacySegment.legacySampleRate();  
  
    return Math.max(1, seconds);  
  
}
```

```
@Override
```

```
public String summary() {
```

```
        return legacySegment.displayName() +  
            " captured as " + legacySegment.getRecorderFormat() +  
            ", adapted for " + desiredCodec +  
            " by " + legacySegment.supervisedBy();  
    }  
}
```

SegmentEntity.java

```
package app.soundlab.audiotrack;  
  
import java.time.LocalDateTime;  
import java.util.Arrays;  
import java.util.Random;  
  
public final class SegmentEntity {  
  
    private final String title;  
  
    private final String engineer;  
  
    private final LocalDateTime capturedAt;  
  
    private final String recorderFormat;  
  
    private final int sampleRate;  
  
    private final byte[] rawSamples;
```

```

private SegmentEntity(
    String title,
    String engineer,
    String recorderFormat,
    int sampleRate,
    byte[] rawSamples
) {
    this.title = title;
    this.engineer = engineer;
    this.recorderFormat = recorderFormat;
    this.sampleRate = sampleRate;
    this.rawSamples = rawSamples;
    this.capturedAt = LocalDateTime.now();
}

public static SegmentEntity microphoneCapture(String title, String engineer, int
seconds) {
    int sampleRate = 44100;
    int length = Math.max(1, seconds) * sampleRate;
    byte[] samples = new byte[length];
    Random random = new Random(title.hashCode() ^ engineer.hashCode());
    for (int i = 0; i < length; i++) {

```

```
        double angle = (i % sampleRate) / (double) sampleRate * Math.PI * 2;

        double wave = Math.sin(angle * 3);

        samples[i] = (byte) (wave * 90 + random.nextInt(20) - 10);
    }

    return new SegmentEntity(title, engineer, "PCM_S16LE", sampleRate,
samples);
}
```

```
public String getRecorderFormat() {
    return recorderFormat;
}
```

```
public byte[] dumpSamples() {
    return Arrays.copyOf(rawSamples, rawSamples.length);
}
```

```
public int legacySampleRate() {
    return sampleRate;
}
```

```
public int rawSampleCount() {
    return rawSamples.length;
}
```

```

public String displayName() {

    return title;

}


public String supervisedBy() {

    return engineer;

}

public LocalDateTime capturedAt() {

    return capturedAt;

}

@Override

public String toString() {

    return "SegmentEntity{" +

        "title=" + title + "\" +

        ", engineer=" + engineer + "\" +

        ", capturedAt=" + capturedAt +

        ", recorderFormat=" + recorderFormat + "\" +

        ", sampleRate=" + sampleRate +

        ", samples=" + rawSamples.length +

        "'}';

}

}

```

Вихідний код:

<https://github.com/mandarinchik21/AudioEditor/tree/main/lab5>

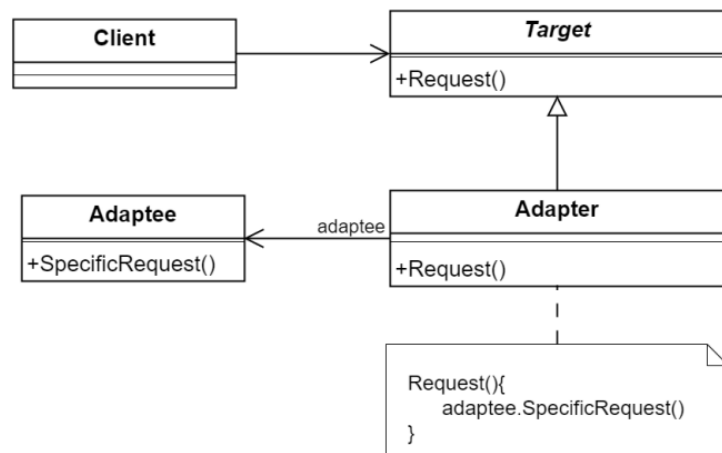
Висновок: У цій лабораторній роботі було досліджено призначення патернів «Adapter», «Builder», «Command», «Chain of Responsibility» та «Prototype», а також реалізовано частину функціоналу аудіоредактора із використанням Adapter. На прикладі редагування аудіофайлів та роботи з різними форматами продемонстровано, як патерни допомагають спростити архітектуру, зменшити залежності між компонентами, підвищити гнучкість системи та забезпечити можливість легкого розширення функціональності. Застосування шаблонів у структурі редактора підтвердило їхню ефективність у створенні модульного, зрозумілого та масштабованого програмного забезпечення.

Контрольні запитання

1. Яке призначення шаблону «Адаптер»?

Адаптер призначений для узгодження двох несумісних інтерфейсів. Він обгортає існуючий клас і надає йому інший інтерфейс, зручний для клієнтського коду

2. Нарисуйте структуру шаблону «Адаптер».



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

- **Client** - використовує інтерфейс **Target**
- **Target** - загальний інтерфейс для клієнта
- **Adapter** - реалізує **Target**, містить посилання на **Adaptee**
- **Adaptee** - існуючий клас, який потрібно адаптувати

Взаємодія: **Client** викликає методи **Target**. Насправді працює з об'єктом **Adapter**, а той усередині транслює виклики до **Adaptee**.

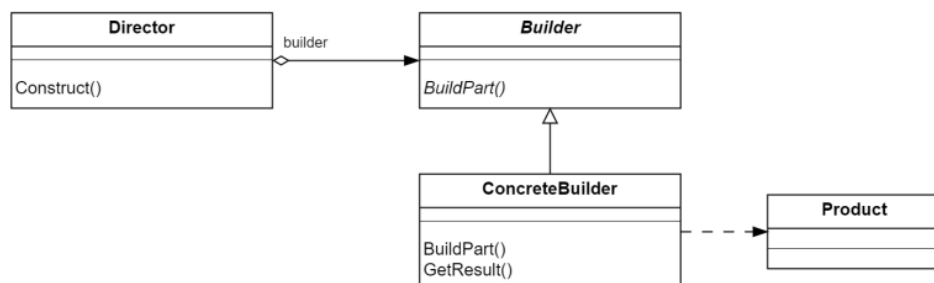
4. Яка різниця між реалізацією «Адаптера» на рівні об'єктів та на рівні класів?

- Об'єктний адаптер
 - Використовує композицію: Adapter містить об'єкт Adaptee.
 - Гнучкіший, можна адаптувати підкласи, міняти екземпляри під час роботи.
- Класовий адаптер
 - Використовує наслідування: Adapter наслідує і Target, і Adaptee.
 - Жорсткіший, прив'язаний до конкретного класу, але дешевший за кількістю об'єктів.

5. Яке призначення шаблону «Будівельник»?

Будівельник відокремлює процес побудови складного об'єкта від його внутрішньої структури. Один і той самий алгоритм побудови може створювати різні варіанти продукту.

6. Нарисуйте структуру шаблону «Будівельник».



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

- **Director** - знає порядок кроків, викликає методи будівельника

- Builder - оголошує методи побудови частин продукту
- ConcreteBuilder - реалізує побудову конкретного продукту, зберігає проміжний стан
- Product - результат роботи конкретного будівельника

Взаємодія: клієнт створює ConcreteBuilder, передає його у Director. Director викликає кроки будівництва, а потім клієнт бере готовий Product з будівельника.

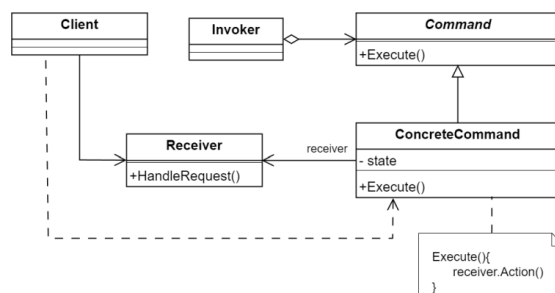
8. У яких випадках варто застосовувати шаблон «Будівельник»?

- Коли об'єкт складається з багатьох частин та етапів створення.
- Коли потрібно створювати різні представлення одного продукту (різні конфігурації).
- Коли потрібно приховати складний конструктор і дати зручний покроковий інтерфейс.

9. Яке призначення шаблону «Команда»?

Команда інкапсулює запит в окремий об'єкт. Це дозволяє передавати дії як параметри, ставити їх у чергу, логувати, виконувати відкладено та реалізовувати скасування.

10. Нарисуйте структуру шаблону «Команда».



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

- Command - оголошує метод Execute()
- ConcreteCommand - зберігає посилання на Receiver, реалізує Execute()
- Receiver - містить фактичну бізнес-логіку
- Invoker - зберігає посилання на команду та викликає її
- Client - створює ConcreteCommand і прив'язує її до Invoker

Взаємодія: Client налаштовує систему, Invoker викликає Execute(), команда делегує роботу Receiver.

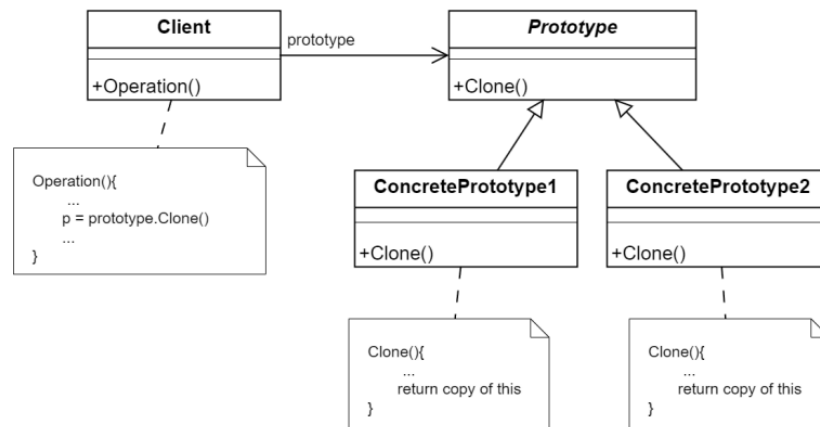
12. Розкажіть як працює шаблон «Команда».

1. Клієнт створює Receiver і ConcreteCommand, передає в команду посилання на Receiver.
2. Команда передається Invoker (кнопка, меню, гаряча клавіша).
3. Коли користувач виконує дію, Invoker викликає command.Execute().
4. Всередині Execute() команда викликає потрібні методи Receiver.
5. Для undo команда може зберігати стан і мати метод Unexecute().

13. Яке призначення шаблону «Прототип»?

Прототип дозволяє створювати нові об'єкти шляхом клонування існуючих екземплярів. Це корисно, коли створення об'єкта дороге або коли треба швидко отримувати багато схожих об'єктів.

14. Нарисуйте структуру шаблону «Прототип»



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

- **Prototype** - оголошує метод `Clone()`
- **ConcretePrototype** - реалізує клонування себе (поверхневе або глибоке)
- **Client** - не створює об'єкти через конструктор, а працює з прототипами, викликаючи `Clone()`

Взаємодія: клієнт бере потрібний прототип і викликає `Clone()`, отримуючи новий об'єкт з таким самим станом.

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

Кілька типових прикладів:

- Обробка подій у графічному інтерфейсі
 - Клік миші передається від дочірнього елемента до батьківських, поки хтось не обробить подію.
- Система логування

- Повідомлення проходить через ланцюжок логерів: консоль, файл, віддалений сервер. Кожен може обробити або передати далі.
- Обробка HTTP запиту в веб-фреймворках
 - Ланцюжок middleware: автентифікація, авторизація, логування, кешування, тощо.
- Перевірка прав доступу або валідація
 - Запит проходить послідовність перевірок, і кожен обробник вирішує, обробляти чи передавати далі.