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

Лабораторна робота № 8

з дисципліни «Технології розроблення програмного забезпечення»

Тема лабораторної роботи: «Патерни проектування»

Тема проєкта: «Аудіо редактор»

Виконала:
студентка групи ІА-33
Котик Іванна

Перевірив:
асистент кафедри ІСТ
Мягкий Михайло Юрійович

Київ 2025

Зміст

Тема: Патерни проектування.

Мета: Вивчити структуру шаблонів «Composite», «Flyweight» (Пристосуванець), «Interpreter», «Visitor» та навчитися застосовувати їх в реалізації програмної системи.

Тема проєкта: 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).

Короткі теоретичні відомості:

Шаблон «Composite» (Компонувальник)

Призначення: Шаблон використовується для складання об'єктів у деревоподібну структуру для подання ієрархій типу «частина-цілого». Даний шаблон дозволяє уніфіковано обробляти як поодинокі об'єкти (листи), так і об'єкти з вкладеністю (композиції).

Ідея: Клієнт взаємодіє з усіма елементами через єдиний інтерфейс Component. Якщо операція викликається у Leaf (листовий елемент), вона виконується. Якщо операція викликається у Composite (складений об'єкт), він рекурсивно застосовує цю операцію до всіх своїх дочірніх елементів . Це дозволяє клієнту не замислюватися, чи працює він з одним об'єктом чи з цілою групою.

Основні елементи:

- Component — оголошує загальний інтерфейс як для Leaf (листів), так і для Composite (контейнерів), включаючи операції управління дочірніми елементами.
- Leaf — представляє кінцевий об'єкт в ієрархії (лист), який не має дочірніх елементів .
- Composite — представляє вузол, який може мати дочірні елементи (children) та реалізує операції Add, Remove, GetChild .
- Client — працює з об'єктами через інтерфейс Component.

Приклад: Простим прикладом є представлення графічного інтерфейсу, де Composite — це Форма або Панель, а Leaf — це Кнопка або Напис. Виклик операції Operation() (наприклад, "розтягування") у Форми призведе до рекурсивного виклику цієї ж операції для всіх дочірніх елементів.

Переваги:

- Спрощує представлення та роботу з деревоподібною структурою.
- Додає гнучкості в роботі зі складними об'єктами та рекурсивними операціями.
- Дозволяє легко додавати та видаляти об'єкти в ієрархії.

Недоліки:

- Вимагає додаткових зусиль для початкового впровадження.
- Вимагає добре спроєктованого загального інтерфейсу.

Шаблон «Flyweight» (Пристосуванець)

Призначення: Шаблон використовується для зменшення кількості об'єктів у додатку шляхом поділу (спільного використання) цих об'єктів між різними ділянками програми. Це дозволяє значно заощадити оперативну пам'ять.

Ідея: Патерн розділяє стан об'єкта на два типи:

1. Внутрішній стан (Intrinsic): Це дані, які є спільними та незмінними (наприклад, код літери). Вони зберігаються всередині самого об'єкта-легковаговика.
2. Зовнішній стан (Extrinsic): Це контекстні дані, унікальні для кожного використання (наприклад, позиція літери на екрані — рядок і стовпчик). Вони зберігаються клієнтом і передаються легковаговику як параметри методів .

Основні елементи:

1. Flyweight — оголошує інтерфейс для операцій, що приймають зовнішній стан (Operation(in extrinsicState)).
2. ConcreteFlyweight — реалізує інтерфейс і зберігає внутрішній стан.
3. FlyweightFactory — керує пулом легковаговиків. При запиті (GetFlyweight) вона або повертає існуючий об'єкт з пулу, або створює новий.
4. Client — зберігає зовнішній стан і передає його Flyweight.

Приклад: У текстовому редакторі не створюється окремий об'єкт для кожної літери на екрані. Натомість, створюється 26 об'єктів-легковаговиків (для кожної літери алфавіту). Клієнтський код (редактор) просто зберігає зовнішній стан (координати) і передає їх потрібному об'єкту-літері для відображення.

Переваги:

- Заощаджує оперативну пам'ять.

Недоліки:

- Витрачає процесорний час на пошук/обчислення зовнішнього стану (контексту).
- Ускладнює код програми внаслідок введення додаткових класів[30].

Шаблон «Interpreter» (Інтерпретатор)

Призначення: Даний шаблон використовується для подання граматики і інтерпретатора для обраної мови.

Ідея: Граматика мови представляється у вигляді ієрархії класів. Клієнт будує речення у вигляді абстрактного синтаксичного дерева (AST). Існує два типи вузлів дерева:

1. TerminalExpression: Представляє кінцеві символи граматики (база рекурсії).
2. NonterminalExpression: Представляє складені правила. Він рекурсивно викликає Interpret() для своїх дочірніх виразів, перш ніж обчислити власний результат.
3. Context містить глобальну інформацію, що використовується інтерпретатором

Основні елементи:

- AbstractExpression — оголошує абстрактний метод Interpret(in Context).
- TerminalExpression — реалізує Interpret() для термінальних символів .

- NonterminalExpression — реалізує Interpret() для нетермінальних символів, зазвичай викликаючи Interpret() для своїх дочірніх об'єктів.
- Context — містить глобальну інформацію для інтерпретатора.
- Client — будує AST з об'єктів виразів.

Приклад: Пошук рядків за зразком (регулярні вирази), де кожен елемент зразка (літерал, зірочка, група) є вузлом в AST.

Переваги:

- Граматику стає легко розширювати та змінювати.
- Можна легко змінювати спосіб обчислення виразів.

Недоліки:

- Супроводження граматики з великою кількістю правил є проблематичним.

Шаблон «Visitor» (Відвідувач)

Призначення: Шаблон «Відвідувач» дозволяє додавати нові операції до елементів, не змінюючи класи цих елементів. Він дозволяє групувати однотипні операції (наприклад, «розрахунок вартості»), що застосовуються до різнотипних об'єктів, в одному класі.

Ідея: Патерн розділяє логіку (Відвідувач) і дані (Елемент).

1. Елемент (ConcreteElement) не містить логіки. Він має лише метод Accept(in visitor: Visitor).

2. Відвідувач (ConcreteVisitor) містить всю логіку. Він має окремий метод для кожного типу елемента (VisitConcreteElementA, VisitConcreteElementB).
3. При виклику elementA.Accept(visitor), elementA негайно викликає visitor.VisitConcreteElementA(this). Цей механізм називається «подвійною диспетчеризацією».

Основні елементи:

- Visitor — інтерфейс, що оголошує методи Visit...() для кожного конкретного елемента .
- ConcreteVisitor — реалізує інтерфейс Visitor та містить логіку операції .
- Element — інтерфейс, що оголошує метод Accept(in visitor: Visitor).
- ConcreteElementA / B — реалізують Accept(), викликаючи відповідний метод Visit...() у відвідувача .
- ObjectStructure — структура (наприклад, список), що містить елементи.

Приклад: Система компілятора. Замість того, щоб класи AST (Виклик Методу, Умовний Вираз) містили логіку, створюються окремі відвідувачі: TypeCheckVisitor (для перевірки типів) та CodeGenerationVisitor (для генерації коду).

Хід роботи:

1. Діаграма класів.

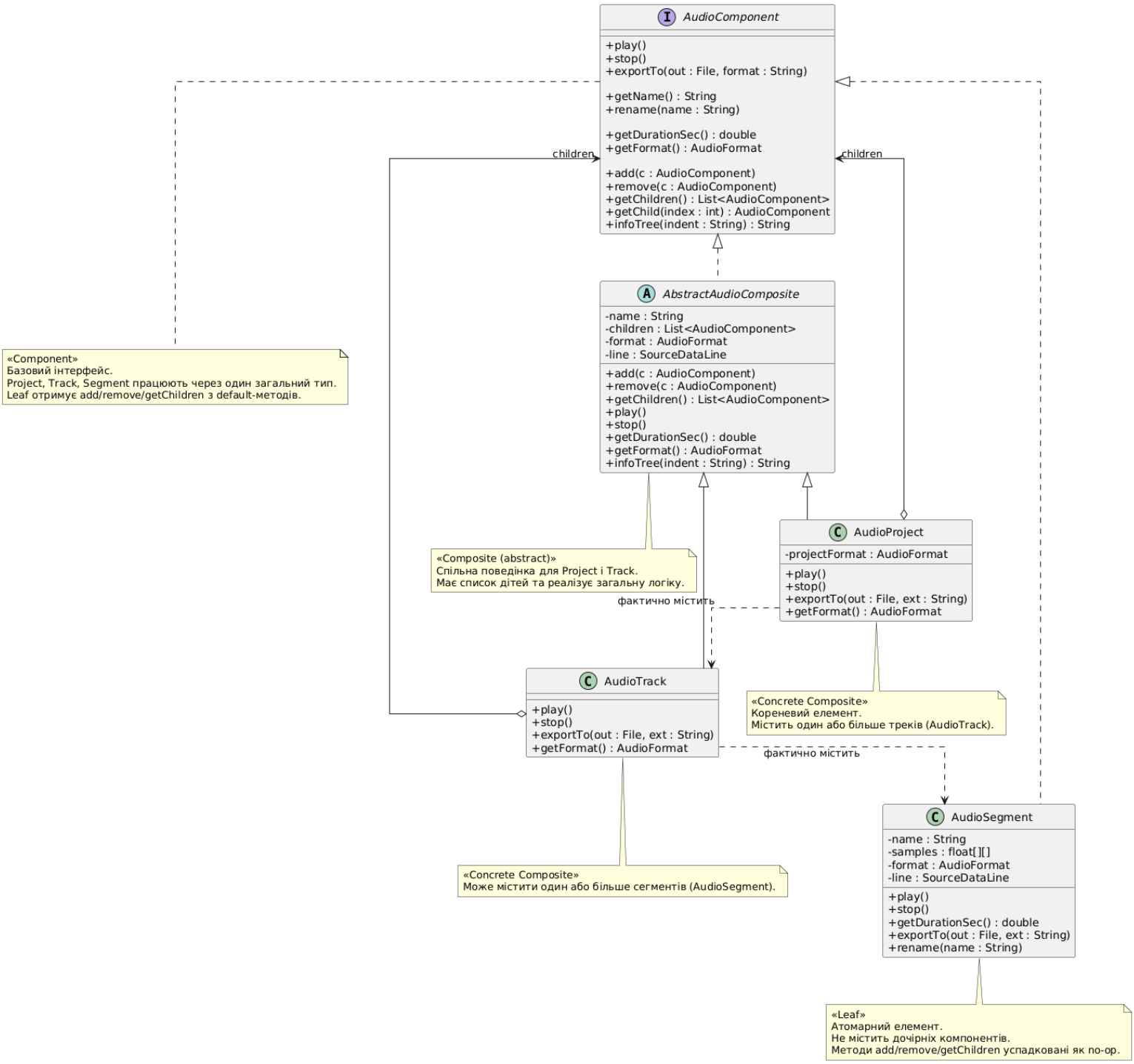


Рисунок 1.1 – Представлення патенту «Компонувальник» на діаграмі класів

2.1. Реалізація шаблону «Composite» у системі для Audio Editor.

Призначення:

Шаблон Компонувальник (Composite) застосовується для подання ієрархій типу «частина—ціле» (Part—Whole Hierarchy) та забезпечення уніфікованої обробки як поодиноких об'єктів, так і їхніх груп.

Метою впровадження даного шаблону в системі Audio Editor було:

- Спрощення логіки клієнта: Дозволити модулям (таким як ExportModule, PlaybackModule) звертатися до окремого аудіо-сегмента (AudioSegment) та до всієї звукової доріжки (AudioTrack) чи проєкту (AudioProject) через один і той самий інтерфейс (AudioComponent).
- Деревоподібне подання даних: Структуру проєкту (Проєкт - Треки - Сегменти) подано у вигляді дерева, що є природним для аудіоредакторів.
- Обробка рекурсивних операцій: Забезпечити рекурсивний прохід операцій, таких як play() або exportTo(), по всій ієрархії об'єктів.

Проблема:

У процесі розроблення програмного забезпечення Audio Editor виникла потреба у створенні гнучкої та масштабованої структури для роботи з аудіоданими.

У типовому аудіоредакторі існує природна ієрархія:

- Проєкт складається з кількох треків;
- Трек складається з кількох сегментів аудіо;
- Кожен сегмент є атомарним елементом, що містить власні PCM-дані.

Без застосування шаблону Composite доведеться:

- писати окрему логіку обробки для проєкту, треку та сегмента;
- дублювати методи відтворення, експорту та обчислення тривалості;
- вручну перевіряти тип об’єкта через instanceof;
- постійно змінювати клієнтський код при додаванні нових типів вузлів.

Це суперечить принципам SOLID, ускладнює підтримку та розширення редактора, а також робить код важким для читання.

Рішення:

Для усунення цих проблем у системі було застосовано шаблон «Composite», який дозволив представити аудіопроєкт у вигляді дерева, де кожен елемент реалізує спільний інтерфейс AudioComponent.

Усі об’єкти – проєкт, треки, сегменти – працюють через один загальний тип, що дозволяє виконувати над ними однакові операції:

- play()
- stop()
- exportTo()
- getDurationSec()
- infoTree() (побудова дерева у текстовому вигляді)

Таким чином, клієнтський код (UI, модулі Mediator, редакторські операції) працює не з конкретними класами, а з єдиним абстрактним компонентом.

Архітектура рішення:

У системі Audio Editor шаблон «Composite» застосовано для побудови деревоподібної структури аудіооб’єктів, де проєкт, трек і сегмент обробляються однаково завдяки спільному інтерфейсу.

Інтерфейс AudioComponent: Визначає базові операції для всіх елементів аудіомоделі: відтворення, зупинку, експорт, отримання формату та тривалості, а також роботу з дочірніми елементами. Клієнтський код працює з усією структурою через цей інтерфейс, не розрізняючи типи об’єктів.

Абстрактний клас AbstractAudioComposite: Містить реалізацію спільної логіки для складених елементів — списку дітей, рекурсивного підрахунку тривалості, базового відтворення, операцій add/remove/getChildren. Від нього успадковуються всі композити — AudioProject і AudioTrack.

AudioProject — кореневий Composite: Представляє увесь проєкт і містить список треків. Реалізує мішкування аудіо, відтворення всього проєкту та експорт у вибраний формат. Виступає верхнім рівнем ієрархії.

AudioTrack — композит середнього рівня: Містить сегменти, відповідає за побудову треку з кількох аудіочастин, їх відтворення та експорт. Є вкладеним Composite-елементом.

AudioSegment — Leaf: Атомарний елемент структури, що містить аудіосемпли й не має дочірніх компонентів. Реалізує реальне відтворення через SourceDataLine. Операції add/remove успадковані, але не використовуються.

Проєкт – містить треки – містять сегменти.

Методи play(), exportTo(), getDurationSec() виконуються рекурсивно, тому клієнту не потрібно знати, чи має елемент дочірні об’єкти.

UI, імпорт, експорт і модулі роботи з хвильовою формою працюють зі структурою через один тип — AudioComponent.

Переваги:

- спрощене управління деревом аудіооб’єктів;
- можливість легко додавати нові рівні ієрархії (групи треків, ефекти);
- рекурсивні операції реалізуються автоматично;
- зменшення зв'язності та дублювання коду.

2.2. Код реалізації шаблона «Composite».

У системі Audio Editor шаблон «Composite» реалізовано на рівні побудови ієрархічної структури аудіооб’єктів — проєктів, треків і сегментів. Ця структура дозволяє обробляти як складені аудіоелементи (проєкт або трек), так і атомарні (сегмент) однаковим чином через спільний інтерфейс AudioComponent.

Нижче наведено повну реалізацію шаблона з виділенням компонентів: Component, Composite та Leaf.

2.2.1. Інтерфейс AudioComponent (Component).

```
public interface AudioComponent {
    void play();
    void stop();
    void exportTo(File out, String format) throws Exception;

    String getName();
    void rename(String name);

    double getDurationSec();
    AudioFormat getFormat();
```

```
    default void add(AudioComponent c) { throw new UnsupportedOperationException(); }
    default void remove(AudioComponent c) { throw new UnsupportedOperationException(); }
    default List<AudioComponent> getChildren() { return List.of(); }

    default AudioComponent getChild(int index) {
        return getChildren().get(index);
    }

    default String infoTree(String indent) {
        return indent + "• " + getName() + "\n";
    }
}
```

2.2.2. Абстрактний клас AbstractAudioComposite.

```
public abstract class AbstractAudioComposite implements AudioComponent {

    protected String name;
    protected final List<AudioComponent> children = new ArrayList<>();

    protected AudioFormat format = new AudioFormat( 44100, 16, 1, true, false );
    protected volatile SourceDataLine line;

    protected AbstractAudioComposite(String name) {
        this.name = name;
    }

    @Override public String getName() { return name; }
    @Override public void rename(String name) { this.name = name; }
    @Override public void add(AudioComponent c) {
        children.add(c);
        if (c.getFormat() != null) {
            this.format = c.getFormat();
        }
    }
    @Override public void remove(AudioComponent c) { children.remove(c); }
    @Override public List<AudioComponent> getChildren() {
        return Collections.unmodifiableList(children);
    }
    @Override
    public double getDurationSec() {
        return children.stream()
```

```
            .mapToDouble(AudioComponent::getDurationSec)
            .max().orElse(0.0);
    }
    @Override
    public void play() {
        for (AudioComponent c : children)
            c.play();
    }
    @Override
    public void stop() {
        SourceDataLine lineToStop = this.line;
        if (lineToStop != null) {
            try {
                lineToStop.stop();
                lineToStop.close();
            } catch (Exception ignore) {
            } finally {
                if (this.line == lineToStop) {
                    this.line = null;
                }
            }
        }
    }
    @Override
    public String infoTree(String indent) {
        var sb = new StringBuilder(indent)
                .append("• ")
                .append(getName())
                .append("\n");

        for (AudioComponent c : children)
            sb.append(c.infoTree(indent + "  "));
        return sb.toString();
    }
    @Override
    public void exportTo(File out, String format) throws Exception {
        throw new UnsupportedOperationException("Export not implemented in base class");
    }
    @Override
    public AudioFormat getFormat() {
        for (AudioComponent c : children) {
            AudioFormat f = c.getFormat();
            if (f != null) return f;
        }
        return PcmUtils.getStandardFormat();
    }
}
```

2.2.3. Клас AudioProject (Concrete Composite)

```
public class AudioProject extends AbstractAudioComposite {
    private final AudioFormat projectFormat;
    public AudioProject(String name) {
        super(name);
        this.projectFormat = PcmUtils.getStandardFormat();
    }

    @Override
    public AudioFormat getFormat() {
        return this.projectFormat;
    }

    @Override
    public double getDurationSec() {
        return children.stream()
                .mapToDouble(AudioComponent::getDurationSec)
                .max().orElse(0.0);
    }

    @Override
    public void play() {
        stop();
        try {
            float[][] mix = PcmUtils.mixProject(this);
            byte[] pcm = PcmUtils.toPCM16(mix);

            DataLine.Info info = new DataLine.Info(SourceDataLine.class, projectFormat);

            SourceDataLine localLine = (SourceDataLine) AudioSystem.getLine(info);
            this.line = localLine;

            localLine.open(projectFormat);
            localLine.start();
            localLine.write(pcm, 0, pcm.length);
            localLine.drain();

        } catch (Exception ex) {
            System.out.println("Playback interrupted (or error): " + ex.getMessage());

        } finally {
            if (this.line != null) {
                try {
                    this.line.close();
                } catch (Exception ignore) {}
                this.line = null;
            }
```

```
        }
    }

    @Override
    public void stop() {
        super.stop();

        for (AudioComponent c : children)
            c.stop();
    }

    @Override
    public void exportTo(File out, String ext) throws Exception {
        float[][] mix = PcmUtils.mixProject(this);
        PcmUtils.writeWav(mix, this.projectFormat, out);
    }
}
```

2.2.4. Клас AudioTrack (Concrete Composite).

```
public class AudioTrack extends AbstractAudioComposite {

    public AudioTrack(String name) {
        super(name);
    }

    @Override
    public void play() {
        for (AudioComponent c : children)
            c.play();
    }

    @Override
    public void stop() {
        for (AudioComponent c : children)
            c.stop();
    }

    @Override
    public void exportTo(File out, String formatExt) throws Exception {
        PcmUtils.concatTrackToFile(this, out, formatExt);
    }

    @Override
    public AudioFormat getFormat() {
        for (AudioComponent c : children) {
            if (c instanceof AudioSegment s) {
                return s.getFormat();
            }
        }
        return PcmUtils.getStandardFormat();
    }
}
```

2.2.5. Клас AudioSegment (Leaf).

```
public class AudioSegment implements AudioComponent {

    private String name;
    private float[][] samples;
    private final AudioFormat format;

    private volatile SourceDataLine line;

    public AudioSegment(String name, float[][] samples, AudioFormat fmt) {
        this.name = name;
        this.samples = samples;
        this.format = fmt;
    }

    public void setSamples(float[][] newSamples) {
        stop();
        this.samples = newSamples;
    }

    @Override
    public void play() {
        stop();
        try {
            byte[] buf = PcmUtils.toPCM16(samples);
            DataLine.Info info = new DataLine.Info(SourceDataLine.class, format);
            SourceDataLine localLine = (SourceDataLine) AudioSystem.getLine(info);
            this.line = localLine;
            localLine.open(format);
            localLine.start();
            localLine.write(buf, 0, buf.length);
            localLine.drain();
        } catch (Exception ex) {
            System.out.println("Segment playback interrupted: " + ex.getMessage());
        } finally {
            if (this.line != null) {
                try { this.line.close(); } catch (Exception ignore) {}
                this.line = null;
            }
        }
    }

    @Override
    public void stop() {
        SourceDataLine lineToStop = this.line;
        if (lineToStop != null) {
            try {
                lineToStop.stop();
                lineToStop.close();
```

```
            } catch (Exception ignore) {
            } finally {
                if (this.line == lineToStop) {
                    this.line = null;
                }
            }
        }
    }

    @Override
    public double getDurationSec() {
        if (samples == null || samples.length == 0) return 0.0;
        return samples[0].length / format.getSampleRate();
    }

    @Override
    public void exportTo(File out, String formatExt) throws Exception {
        PcmUtils.writeWav(this.samples, this.format, out);
    }

    @Override public String getName() { return name; }
    @Override public void rename(String newName) { this.name = newName; }

    public float[][] getSamples() { return samples; }

    @Override
    public AudioFormat getFormat() { return format; }
}
```

2.3. Пояснення коду реалізації шаблону «Composite».

Фундаментом структури є інтерфейс **AudioComponent**, який виступає у ролі Компонента — загального типу для всіх вузлів дерева. Він чітко визначає набір операцій, які мають сенс для будь-якої аудіо-сутності: play(), stop(), exportTo(), getDurationSec() та маніпуляції з назвою (getName(), rename()). Таким чином, будь-який модуль редактора (UI, модуль експорту) може викликати, наприклад, exportTo() для всього проєкту, окремого треку чи навіть одного сегмента, не розрізняючи їхні конкретні класи.

Інтерфейс також включає методи для управління дочірніми елементами (add(), remove(), getChildren()). У цій реалізації вони надані як методи за замовчуванням і викидають UnsupportedOperationException. Цей підхід є частиною «безпечного» (варіанту шаблону Composite, коли атомарні елементи (AudioSegment) не

зобов'язані реалізовувати логіку управління дітьми, але зберігають єдину сигнатуру інтерфейсу.

AbstractAudioComposite є абстрактним базовим класом, що інкапсулює спільну логіку для всіх Складених об'єктів, а саме AudioProject та AudioTrack. Головним завданням цього класу є управління колекцією дочірніх AudioComponent та реалізація рекурсивного проходження.

Методи add(), remove() та getChildren() тут реалізовані повністю, працюючи з внутрішнім списком children. Ключові операції, такі як play(), імплементовані через просту ітерацію, яка делегує виклик кожному дочірньому елементу. Обчислення тривалості (getDurationSec()) також є рекурсивним, але з логічним винятком для аудіо: воно повертає максимальну тривалість серед дітей, що коректно відображає загальну довжину мультитрекового проєкту.

Конкретні класи **AudioProject** (корінь) та **AudioTrack** (внутрішній вузол) успадковують логіку Composite, але переозначають операції, які вимагають специфічної обробки.

- **AudioProject** відповідає за кінцевий аудіовивід. Його метод play() не просто делегує, а виконує складну операцію мікшування аудіоданих з усіх дочірніх треків в єдиний PCM-потік, який потім відтворюється через SourceDataLine. Аналогічно, exportTo() реалізує мікшування та запис фінального WAV-файлу.
- **AudioTrack** є проміжним композитом. Його основна функція — групування сегментів. Його метод exportTo() виконує конкатенацію аудіоданих сегментів у єдиний трековий файл.

Клас **AudioSegment** є Листком ієрархії. Це атомарний елемент, який містить безпосередньо аудіодані і не має дочірніх вузлів.

Усі уніфіковані операції тут реалізуються на базовому рівні:

- play() відтворює власні семпли через SourceDataLine.
- exportTo() записує лише власні семпли у файл.
- getDurationSec() обчислює тривалість на основі внутрішніх даних та аудіоформату.

Оскільки AudioSegment є кінцевим вузлом, він ігнорує логіку управління дітьми.

Завдяки цій структурі, клієнтський код досягає високого рівня абстракції. Будь-яка операція, викликана на кореневому AudioProject (наприклад, audioProject.play()), автоматично поширюється вниз ієрархією: проєкт мікшує, треки групують, а сегменти надають реальні дані для відтворення. Це гарантує, що при додаванні нових рівнів ієрархії клієнтський код залишиться незмінним, дотримуючись принципів SOLID.

2.4. Використання шаблону «Composite» у системі Audio Editor.

Після впровадження шаблону «Composite» у системі Audio Editor робота з аудіоієрархією відбувається через універсальний інтерфейс AudioComponent. Це означає, що різні частини системи можуть виконувати операції над будь-яким елементом ієрархії без знання його конкретного типу.

Нижче наведено список класів і модулів, у яких фактично використовується Composite — тобто здійснюється виклик методів play(), stop(), exportTo(), getChildren() або маніпуляція об’єктами AudioProject, AudioTrack, AudioSegment.

Клас	Метод	Використання Composite	Пояснення
ImportModule	receive()	track.add(segment)	Створюється новий Leaf-елемент (AudioSegment) і додається до Composite-елемента (AudioTrack), використовуючи його успадкований метод add().
EditorView	playComposite(), stopComposite(), exportComposite()	audioProject.play(), audioProject.stop(), audioProject.exportTo()	Виклик уніфікованих методів на кореневому компоненті (AudioProject), що автоматично запускає рекурсивне виконання операції по всіх дочірніх треках і сегментах.
ExportModule	onExport()	project.getChildren(), project.exportTo(), selectedTrack.exportTo()	Використовується getChildren() для отримання списку треків (елементи Composite). Викликається exportTo() або для AudioProject (для міксу), або для конкретного AudioTrack (делегування операції).
TrackModule	onAddTrack(), getTrack(), getMainSegment()	project.add(new AudioTrack(name)), project.getChildren().stream().filter(...)	Додавання нового Composite-елемента (AudioTrack) до кореневого композита (AudioProject). Навігація: Використання getChildren() для пошуку конкретного треку (AudioTrack), а потім першого сегмента (AudioSegment) у ньому (Leaf).
ClipboardModule	onCopy(), onCut(), onPaste()	mainSegment.getSamples(), mainSegment.setSamples(newSamples)	Пряма модифікація Leaf-елемента (AudioSegment). Операції (як-от вирізання) змінюють внутрішні семпли, використовуючи його публічний API (setSamples).
WaveformModule	onWaveformFx()	mainSegment.getSamples(), mainSegment.setSamples(newSamples)	Застосування ефектів (наприклад, reverse) здійснюється через читання та перезапис даних Leaf-елемента (AudioSegment).
PlaybackModule	onStart()	track.getFormat(), PcmUtils.concatTrack(track)	Виклик getFormat() на AudioTrack (який делегує його сегменту) та використання утиліти для конкатенації всіх сегментів, що входять до треку, в

			єдиний потік PCM для відтворення.

Шаблон Composite активно використовується в багатьох компонентах Audio Editor.

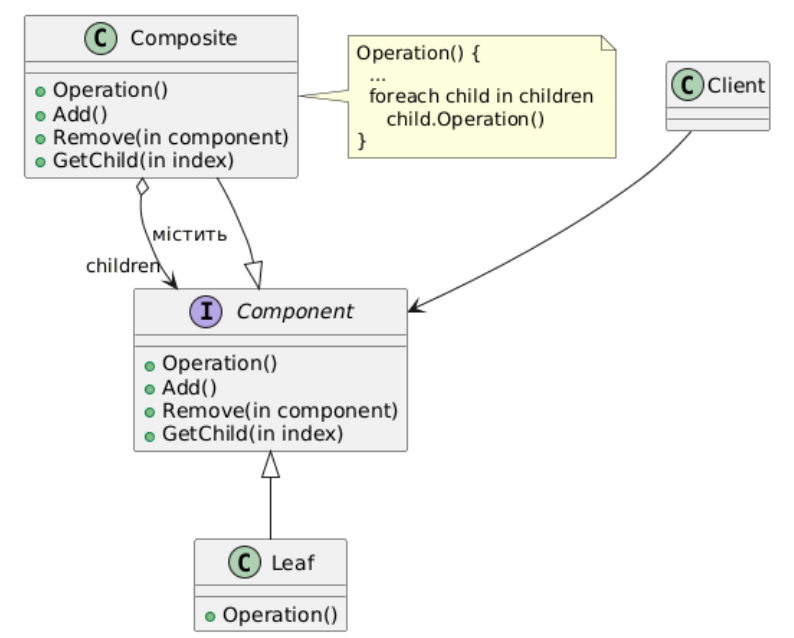
Усі модулі – ImportModule, TrackModule, ExportModule, PlaybackModule, WaveformModule, ClipboardModule – працюють із аудіоданими через спільний інтерфейс AudioComponent.

Відповіді на контрольні запитання:

1. Яке призначення шаблону «Композит»?

Шаблон використовується для складання об'єктів в деревоподібну структуру для подання ієрархій типу «частина цілого». Даний шаблон дозволяє уніфіковано обробляти як поодинокі об'єкти, так і об'єкти з вкладеністю.

2. Нарисуйте структуру шаблону «Композит».



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

Класи, що входять в шаблон «Композит»:

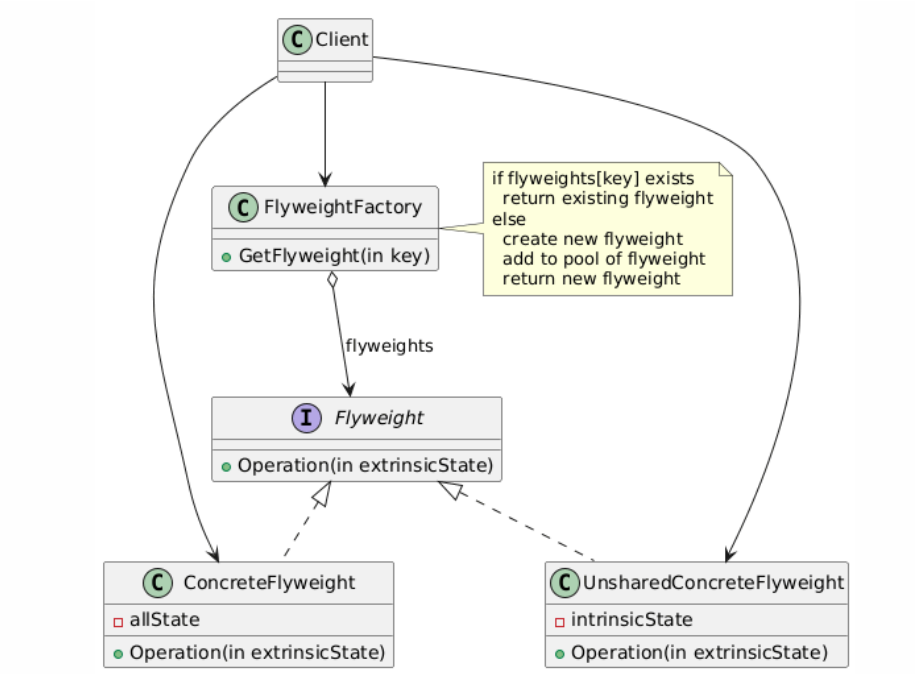
- *Client – Маніпулює об'єктами в композиції через загальний інтерфейс Component.*
- *Component – оголошує інтерфейс для компонованих об'єктів; Надає відповідну реалізацію операцій за замовчуванням, загальну для всіх класів; Оголошує єдиний інтерфейс для доступу до нащадків та управління ними; Визначає інтерфейс для доступу до батька компонента в рекурсивної структурі і при необхідності реалізує його (можливість необов'язкова);*
- *Leaf (Leaf_1, Leaf_2) – об'єкт того ж типу що і Composite, але без реалізації контейнерних функцій; Представляє листові вузли композиції і не має нащадків; Визначає поведінку примітивних об'єктів в композиції; Входить до складу контейнерних об'єктів;*
- *Composite – визначає поведінку контейнерних об'єктів, у яких є нащадки; Зберігає ієрархію компонентів-нащадків; Реалізує пов'язані з управління нащадками (контейнерні) операції в інтерфейсі класу Component;*

Взаємодія: Клієнт використовує інтерфейс Component для роботи з будь-яким елементом структури. Коли клієнт викликає операцію (Operation()) у Leaf, вона просто виконується. Коли операція викликається у Composite, він рекурсивно передає цей виклик всім своїм дочірнім елементам .

4. Яке призначення шаблону «Легковаговик»?

Шаблон використовується для зменшення кількості об'єктів в додатку шляхом поділу цих об'єктів між ділянками додатку. Flyweight являє собою поділюваний об'єкт.

5. Нарисуйте структуру шаблону «Легковаговик».



6. Які класи входять в шаблон «Легковаговик», та яка між ними взаємодія?

Класи, що входять в шаблон «Легковаговик»:

- *Flyweight – Оголошує інтерфейс, через який легковаговики можуть отримувати зовнішній стан (Operation(in extrinsicState)) .*
- *ConcreteFlyweight – Реалізує інтерфейс Flyweight і зберігає внутрішній стан (спільний, незмінний) .*
- *UnsharedConcreteFlyweight – Клас для об'єктів, які не поділяються, але мають спільний інтерфейс Flyweight.*
- *FlyweightFactory – Створює та керує пулом об'єктів Flyweight.*

- *Client – Зберігає або обчислює зовнішній стан (контекст) і передає його легковаговикам при виклику їх методів.*

Взаємодія: Client запитує об'єкт у FlyweightFactory за допомогою ключа (GetFlyweight). Фабрика перевіряє, чи існує такий об'єкт у пулі: якщо так, вона повертає існуючий екземпляр; якщо ні — створює новий, додає його в пул і повертає . Потім Client викликає метод Operation() у легковаговика, передаючи йому зовнішній стан.

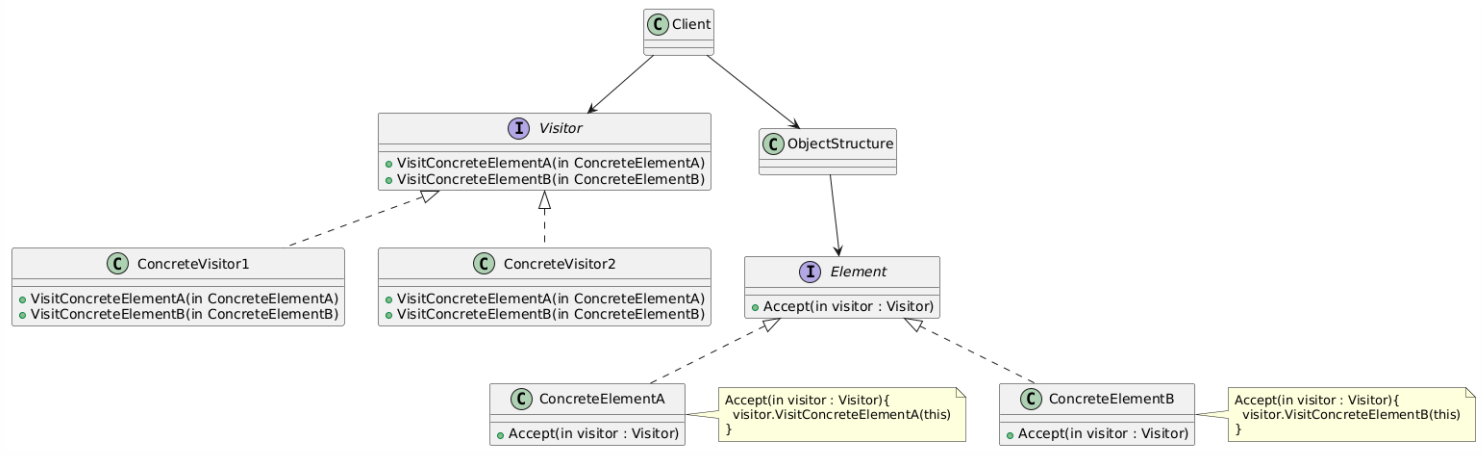
7. Яке призначення шаблону «Інтерпретатор»?

Призначення шаблону «Інтерпретатор» – представити граматику для певної мови (наприклад, скриптової) та надати інтерпретатор, який може обробляти вирази (речення) цією мовою.

8. Яке призначення шаблону «Відвідувач»?

Шаблон відвідувач дозволяє вказувати операції над елементами без зміни структури конкретних елементів. Таким чином вкрай зручно додавати нові операції, проте дуже важко додавати нові елементи в ієрархію (необхідно додавати відповідні методи для обробки їх відвідувань в кожного відвідувача). Даний шаблон дозволяє групувати однотипні операції, що застосовуються над різнотипними об'єктами.

9. Нарисуйте структуру шаблону «Відвідувач».



10. Які класи входять в шаблон «Відвідувач», та яка між ними взаємодія?

До шаблону «Відвідувач» входять такі класи:

- *Visitor – Інтерфейс, що оголошує окремий метод Visit...() для кожного класу ConcreteElement в ієрархії (наприклад, VisitConcreteElementA, VisitConcreteElementB) .*
- *ConcreteVisitor – Клас, що реалізує інтерфейс Visitor та містить фактичну логіку операції для кожного типу елемента .*
- *Element – Інтерфейс, що оголошує метод Accept(in visitor: Visitor), який приймає відвідувача .*
- *ConcreteElementA / B – Класи, що реалізують інтерфейс Element. Їхній метод Accept() викликає відповідний метод у відвідувача, передаючи себе.*
- *ObjectStructure – Структура (наприклад, колекція), яка зберігає об'єкти Element і дозволяє клієнту обходити їх за допомогою відвідувача.*
- *Client – Ініціює операцію, створюючи ConcreteVisitor і передаючи його в метод Accept() елементів (які містяться в ObjectStructure).*

Взаємодія: Клієнт створює об'єкт ConcreteVisitor і викликає метод Accept() у об'єкта Element. Елемент негайно викликає відповідний метод у відвідувача,

передаючи себе як аргумент. Це дозволяє відвідувачу виконати операцію над цим елементом, не змінюючи клас самого елемента .

Висновки:

У ході лабораторної роботи було вивчено призначення та структуру шаблонів проектування «Composite», «Flyweight», «Interpreter» і «Visitor», а також розглянуто особливості їх застосування у програмних системах. На практичній частині було реалізовано шаблон «Composite» у проєкті аудіоредактора, що дозволило подати аудіодані у вигляді ієрархії «проєкт – треки – сегменти» та організувати роботу з ними через єдиний універсальний інтерфейс.

Завдяки впровадженню цього шаблону клієнтський код став простішим і більш гнучким, а операції відтворення, експорту та обробки аудіо отримали рекурсивну реалізацію, яка не залежить від типу об’єкта. Це забезпечило краще розділення відповідальностей, зменшило дублювання логіки та полегшило розширення системи в майбутньому.