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

Лабораторна робота № 7

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 «Mediator», «Facade», «Bridge», «Template method» 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 «Mediator»

Призначення:

Шаблон "Mediator" (Посередник) використовується для визначення взаємодії об'єктів через єдиний об'єкт-посередник, замість того, щоб об'єкти посилалися безпосередньо один на одного. Це дозволяє інкапсулювати складну логіку взаємодії в одному місці.

Проблема:

Виникає при розробці складних візуальних форм, де багато компонентів тісно пов'язані. Дії в одному компоненті (наприклад, вибір прапорця) можуть вимагати змін у багатьох інших (відключення полів, приховування блоків). Це створює тисячі зв'язків, що робить код складним для розуміння та виправлення помилок.

Рішення:

Впроваджується клас-посередник. Усі компоненти взаємодіють лише з ним, а не один з одним. Компоненти повідомляють посередника про свої дії, а посередник вже вирішує, які інші компоненти мають на це відреагувати.

Переваги:

- Спрощує розуміння та супровід коду, оскільки вся логіка взаємодії централізована.
- Зменшує залежності між компонентами, підвищуючи гнучкість системи.
- Дозволяє додавати нових посередників (і нову логіку) без зміни самих компонентів.

Недоліки:

- З часом посередник може перетворитися на надто складний об'єкт («God Object»), який важко підтримувати.

Шаблон «Facade»

Призначення:

Шаблон "Facade" (Фасад) надає єдиний, уніфікований інтерфейс до складної підсистеми, приховуючи її внутрішні деталі. Це дозволяє уникнути "спагеті-коду", де все тісно пов'язано.

Проблема:

Виникає при створенні компонента зі складною внутрішньою структурою класів (наприклад, клієнта для роботи з різними протоколами HTTP та TCP/IP).

Інструкції з використання такого компонента виходять заплутаними. Крім того, будь-яка зміна внутрішньої структури змусить усіх клієнтів (користувачів компонента) переписувати свій код.

Рішення:

Створюється один фасадний клас (наприклад, InternetClient) з простим набором публічних методів для налаштування та використання. Вся складна логіка та внутрішні класи приховані (наприклад, позначені як internal). Клієнти працюють лише з цим простим фасадом. Внутрішню структуру можна змінювати, не впливаючи на клієнтський код.

Переваги:

- Інкапсулює (приховує) внутрішню структуру модуля від клієнтського коду.
- Значно спрощує інтерфейс для роботи з підсистемою.

Недоліки:

- Знижується гнучкість у налаштуванні підсистеми, оскільки клієнт обмежений лише тими методами, які надає фасад.

Шаблон «Bridge»

Призначення:

Шаблон "Bridge" (Міст) використовується для розділення абстракції та її реалізації, дозволяючи їм змінюватися незалежно одна від одної.

Проблема:

Виникає, коли потрібно реалізувати кілька варіантів абстракції (наприклад, фігури: Лінія, Коло) і водночас кілька варіантів реалізації (наприклад, способи відображення: На екрані, На принтері). Звичайне наслідування призводить до "вибуху" кількості класів (LinePrint, LineDraw, CirclePrint, CircleDraw). Додавання нової фігури або нового способу відображення стає дуже складним.

Рішення: Створюються дві незалежні ієрархії класів.

1. Абстракція (наприклад, базовий клас Shape та його нащадки Line, Circle).
2. Реалізація (наприклад, інтерфейс DrawApi та його реалізації WindowDrawApi, PrinterDrawApi). Об'єкт абстракції (Shape) містить посилання на об'єкт реалізації (DrawApi) і делегує йому виконання роботи (рисування).

Переваги:

- Дозволяє змінювати абстракції (фігури) та реалізації (драйвери рисування) незалежно.
- Підвищує гнучкість та спрощує супровід коду.

Недоліки:

- Підвищує загальну складність системи через введення додаткових проміжних рівнів.

Шаблон «Template Method»

Призначення:

Шаблон "Template Method" (Шаблонний метод) дозволяє визначити "кістяк" алгоритму в абстрактному класі, але залишити реалізацію деяких кроків цього алгоритму підкласам.

Проблема:

Виникає, коли різні, але схожі алгоритми (наприклад, обробка відеофайлів MPEG-4, MPEG-2, MPEG-1) мають багато спільного коду (наприклад, 70% алгоритму однакове). Якщо реалізувати все в одному класі, він переповнюється умовами (if-else для кожного формату), що робить код складним для читання та внесення змін.

Рішення**:**

Загальний алгоритм (шаблонний метод) реалізується в базовому абстрактному класі. Ті кроки алгоритму, які відрізняються для різних форматів (наприклад, читання кадрів, обробка аудіодоріжок), оголошуються як абстрактні або віртуальні методи. Потім для кожного формату (MPEG-4, MPEG-2) створюється свій дочірній клас, який перевизначає ці віртуальні методи, надаючи специфічну реалізацію.

Переваги:

- Полегшує повторне використання коду, оскільки загальна логіка знаходиться в одному місці.

Недоліки:

- Підкласи жорстко обмежені "кістяком" алгоритму, визначеним у базовому класі.
- Можна порушити принцип підстановки Барбари Лісков, якщо підклас кардинально змінює поведінку одного з кроків.
- Алгоритм стає складно підтримувати, якщо він має забагато віртуальних методів.

Хід роботи:

1. Діаграма класів.

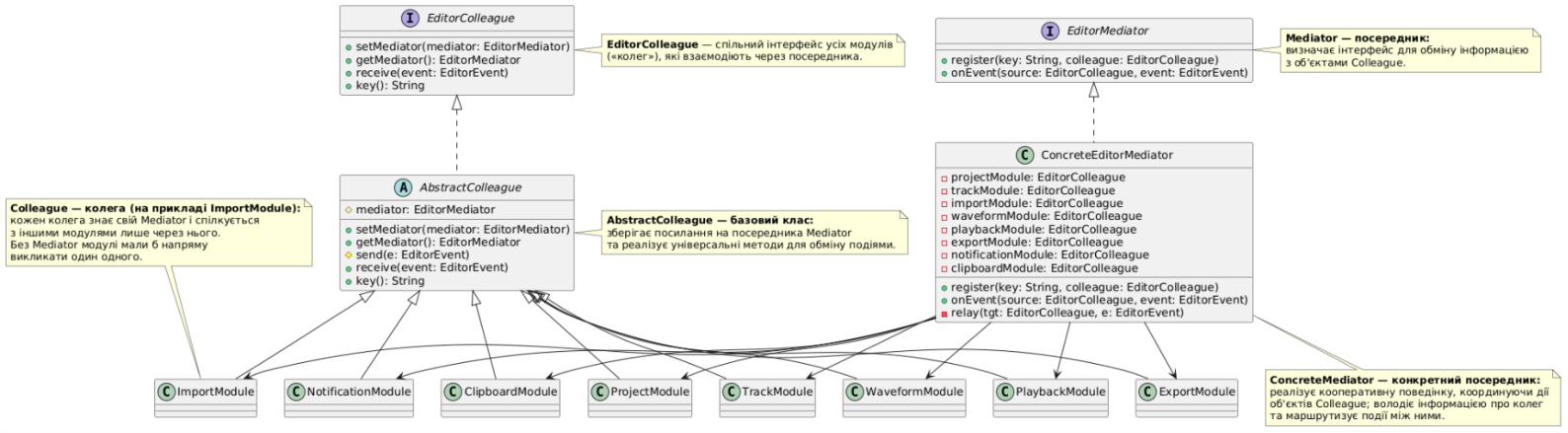


Рисунок 1.1 – Представлення шаблону проєктування «Посередник» на діаграмі класів

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

Шаблон Mediator (Посередник) використовується для централізації взаємодії між об’єктами, які інакше мали б численні прямі зв’язки один з одним. Його основна ідея полягає у тому, що об’єкти («колеги») не взаємодіють безпосередньо, а надсилають повідомлення через окремий об’єкт — посередника (Mediator), який координує їхню співпрацю.

Це дозволяє зменшити кількість залежностей між компонентами системи та спростити підтримку коду.

Проблема:

Під час розроблення системи Audio Editor логіка взаємодії між модулями стала надмірно складною. Більшість функціональних процесів — створення проєкту, імпорт файлів, оновлення хвильової форми, керування треками, відтворення, експорт та показ повідомлень — виконувалися через безпосередні виклики між окремими компонентами або концентрувалися у головному класі EditorView.

Подібна архітектура призводила до високої зв’язності компонентів і ускладнювала підтримку системи. Наприклад, коли користувач імпортував новий трек, потрібно було вручну викликати оновлення кількох частин інтерфейсу – WaveformView, TrackList, NotificationPanel. У випадку зміни логіки хоча б одного з модулів доводилося змінювати код одразу в кількох місцях, що збільшувало ризик появи помилок і ускладнювало тестування.

Подібна ситуація аналогічна до проблеми складних візуальних форм, де десятки елементів інтерфейсу впливають один на одного. Якщо при активації одного контролу потрібно змінити стан десятків інших компонентів, кількість можливих зв’язків між елементами може сягати тисяч, що робить систему заплутаною та складною для розширення.

Рішення:

Для усунення надмірної зв’язаності та централізації управління взаємодіями між компонентами було застосовано шаблон Mediator (Посередник).

Цей шаблон передбачає створення спеціального класу-посередника, який бере на себе всю логіку координації об’єктів, що раніше взаємодіяли напряму. Кожен компонент тепер знає лише про посередника, а не про інші модулі. Коли один компонент викликає подію, він повідомляє про це посередника, який у свою чергу ініціює потрібні зміни в інших модулях.

У системі Audio Editor реалізація шаблону виглядає так:

- EditorView продовжує формувати події через AudioEditor.notifyObservers().
- Один зі спостерігачів — MediatorObserver — отримує події й передає їх у ConcreteEditorMediator.
- ConcreteEditorMediator аналізує тип події й викликає потрібні модулі (наприклад, при імпорті аудіо — ImportModule, WaveformModule, TrackModule і NotificationModule).
- Модулі (колеги) реалізують інтерфейс EditorColleague і взаємодіють лише через методи send() та receive(), не знаючи про структуру інших частин системи.

Основна перевага цього підходу полягає у значному зниженні зв’язності між компонентами: тепер для додавання нового функціоналу достатньо зареєструвати новий модуль у медіаторі, не змінюючи код решти системи.

Архітектура рішення:

Інтерфейс **EditorMediator**: Визначає основні методи взаємодії між модулями:

- register(String key, EditorColleague colleague) — реєструє модулі (колег) у системі;
- onEvent(EditorColleague source, EditorEvent event) — приймає події від одного модуля та передає їх іншим.

Посередник виступає центральною ланкою, через яку здійснюється вся комунікація між компонентами програми.

Клас **ConcreteEditorMediator**: Реалізує роль посередника. Він містить посилання на всі основні модулі системи — ProjectModule, TrackModule, ImportModule, WaveformModule, PlaybackModule, ExportModule, NotificationModule, ClipboardModule. Після отримання події посередник визначає, які саме модулі мають на неї реагувати.

Наприклад:

- при події AUDIO_IMPORTED активуються WaveformModule, TrackModule та NotificationModule;
- при події PLAYBACK_START — PlaybackModule і WaveformModule;
- при події EXPORT_REQUEST — ExportModule. Таким чином, ConcreteEditorMediator координує взаємодію між компонентами без їх прямої залежності.

Інтерфейс **EditorColleague**: Задає контракт для всіх модулів (колег) системи.

Він містить методи:

- setMediator(EditorMediator mediator) — встановлення посередника;
- receive(EditorEvent event) — отримання подій;
- key() — повернення унікального ідентифікатора модуля.
  Кожен модуль взаємодіє з іншими лише через медіатор, не знаючи про їхню внутрішню реалізацію.

Клас **AbstractColleague**: Є базовим класом для всіх колег і містить спільну логіку роботи з посередником. Він має поле mediator та метод send(EditorEvent e), який дозволяє надсилати події до посередника без прямого звернення до інших модулів. Це забезпечує слабке зв’язування між компонентами системи.

Клас **EditorEvent**: Інкапсулює тип події (**EditorEventType**) та супровідні параметри. Об’єкти цього класу використовуються для обміну інформацією між колегами через посередника.

Конкретні модулі (Colleagues): Кожен модуль реалізує інтерфейс EditorColleague та має власну логіку реакції на події:

- **ProjectModule** — створення та вибір проєктів;
- **TrackModule** — управління треками;
- **ImportModule** — імпорт аудіо;
- **WaveformModule** — побудова хвильової форми;
- **PlaybackModule** — відтворення аудіо;
- **ExportModule** — експорт у різні формати;
- **NotificationModule** — показ повідомлень користувачу;

- **ClipboardModule** — операції копіювання, вирізання та вставки.

Клас **EditorView**: Тепер виконує лише роль інтерфейсу користувача. Він створює події та передає їх у AudioEditor, який через механізм Observer сповіщає ConcreteEditorMediator. Посередник визначає, які модулі мають відреагувати, що дозволяє відокремити інтерфейс від бізнес-логіки та спростити розширення функціональності системи.

2.2. Код реалізації шаблона «Mediator».

2.2.1. Інтерфейс EditorMediator.

```
public interface EditorMediator {
    void register(String key, EditorColleague colleague);
    void onEvent(EditorColleague source, EditorEvent event);
}
```

2.2.2. Інтерфейс EditorColleague.

```
public interface EditorColleague {

    void setMediator(EditorMediator mediator);

    EditorMediator getMediator();

    void receive(EditorEvent event);

    String key();

}
```

2.2.3. Абстрактний клас AbstractColleague.

```
public abstract class AbstractColleague implements EditorColleague {
    protected EditorMediator mediator;
```

```
    @Override public void setMediator(EditorMediator mediator) { this.mediator = mediator; }
    @Override public EditorMediator getMediator() { return mediator; }

    protected final void send(EditorEvent e) {
        if (mediator != null) mediator.onEvent(this, e);
    }

    @Override public abstract void receive(EditorEvent event);
    @Override public abstract String key();
}
```

2.2.4. Клас ConcreteEditorMediator.

```
public class ConcreteEditorMediator implements EditorMediator {

    private EditorColleague projectModule;

    private EditorColleague trackModule;

    private EditorColleague importModule;

    private EditorColleague waveformModule;

    private EditorColleague playbackModule;

    private EditorColleague exportModule;

    private EditorColleague notificationModule;

    private EditorColleague clipboardModule;


    @Override

    public void register(String key, EditorColleague colleague) {

        colleague.setMediator(this);

        switch (key) {

            case "Project" -> this.projectModule = colleague;

            case "Track" -> this.trackModule = colleague;

            case "Import" -> this.importModule = colleague;

            case "Waveform" -> this.waveformModule = colleague;

            case "Playback" -> this.playbackModule = colleague;
```

```
            case "Export" -> this.exportModule = colleague;
            case "Notification" -> this.notificationModule = colleague;
            case "Clipboard" -> this.clipboardModule = colleague;
            default -> System.err.println("Unknown colleague key: " + key);
        }
    }

    @Override
    public void onEvent(EditorColleague source, EditorEvent e) {
        switch (e.type) {
            case PROJECT_CREATE_REQUEST, PROJECT_SELECTED -> relay(projectModule, e);
            case TRACK_ADD_REQUEST, TRACKS_REFRESH_REQUEST -> relay(trackModule, e);
            case IMPORT_REQUEST -> relay(importModule, e);
            case AUDIO_IMPORTED -> {
                relay(waveformModule, e);
                relay(trackModule, e);
                relay(notificationModule, e.with("info", "Audio imported"));
            }
            case PLAYBACK_START, PLAYBACK_STOP -> {
                relay(playbackModule, e);
                relay(trackModule, e);
                relay(waveformModule, e);
            }
            case EXPORT_REQUEST -> relay(exportModule, e);
            case NOTIFY_INFO, NOTIFY_WARN, NOTIFY_ERROR -> relay(notificationModule, e);
            case CLIPBOARD_COPY, CLIPBOARD_CUT, CLIPBOARD_PASTE ->
relay(clipboardModule, e);
            default -> { /* no action */ }
        }
```

```
    }

    private void relay(EditorColleague target, EditorEvent e) {
        if (target != null) target.receive(e);
    }
}
```

2.2.5. Конкретні модулі (Colleagues).

ProjectModule:

https://github.com/ivannakotyk/SDT/blob/main/AudioEditor/audio-editor-intellij-2023_3_4/client/src/main/java/com/ivanka/audioeditor/client/core/modules/ProjectModule.java

TrackModule:

https://github.com/ivannakotyk/SDT/blob/main/AudioEditor/audio-editor-intellij-2023_3_4/client/src/main/java/com/ivanka/audioeditor/client/core/modules/TrackModule.java

ClipboardModule:

https://github.com/ivannakotyk/SDT/blob/main/AudioEditor/audio-editor-intellij-2023_3_4/client/src/main/java/com/ivanka/audioeditor/client/core/modules/ClipboardModule.java

ExportModule:

https://github.com/ivannakotyk/SDT/blob/main/AudioEditor/audio-editor-intellij-2023_3_4/client/src/main/java/com/ivanka/audioeditor/client/core/modules/ExportModule.java

ImportModule:

https://github.com/ivannakotyk/SDT/blob/main/AudioEditor/audio-editor-intellij-2023_3_4/client/src/main/java/com/ivanka/audioeditor/client/core/modules/ImportModule.java

NotificationModule:

https://github.com/ivannakotyk/SDT/blob/main/AudioEditor/audio-editor-intellij-2023_3_4/client/src/main/java/com/ivanka/audioeditor/client/core/modules/NotificationModule.java

PlaybackModule:

https://github.com/ivannakotyk/SDT/blob/main/AudioEditor/audio-editor-intellij-2023_3_4/client/src/main/java/com/ivanka/audioeditor/client/core/modules/PlaybackModule.java

WaveformModule:

https://github.com/ivannakotyk/SDT/blob/main/AudioEditor/audio-editor-intellij-2023_3_4/client/src/main/java/com/ivanka/audioeditor/client/core/modules/WaveformModule.java

Загальний приклад реалізації конкретних модулів колег:

```
public class ExampleModule extends AbstractColleague {

    @Override
    public String key() {
        // Унікальний ідентифікатор для реєстрації в Mediator
        return "Example";
    }

    @Override
    public void receive(EditorEvent e) {
        // Отримання події від Mediator
        switch (e.type) {
            case ACTION_ONE -> onActionOne(e);
            case ACTION_TWO -> onActionTwo(e);
            default -> { /* подія не обробляється цим модулем */ }
        }
    }

    // --- Конкретні методи обробки подій ---

     private void onActionOne(EditorEvent e) {
        // Тут мала би бути логіка обробки події ACTION_ONE
        /...
        // При необхідності модуль може ініціювати іншу подію через Mediator:
        send(new EditorEvent(e.type)); // приклад виклику
    }

    private void onActionTwo(EditorEvent e) {
        // Тут могла би бути логіка для ACTION_TWO
    }
}
```

3. Пояснення коду реалізації шаблону «Mediator».

У застосунку Audio Editor шаблон «Посередник» (Mediator) використано для впорядкування взаємодії між численними модулями системи, такими як Project, Track, Import, Export, Playback, Waveform та Notification. Його мета –усунути надмірну зв’язаність між компонентами та централізувати обмін подіями через єдиний координаційний об’єкт.

Раніше кожен модуль напряму звертався до інших або через громіздку логіку в класі EditorView. Такий підхід ускладнював підтримку і масштабування коду. Після впровадження шаблону Mediator усі модулі спілкуються лише через єдиний об’єкт-посередник — ConcreteEditorMediator.

Інтерфейс EditorMediator визначає спільні методи register() і onEvent(), які забезпечують реєстрацію модулів (колег) та обробку подій. Інтерфейс EditorColleague описує поведінку колег і містить методи setMediator(), receive() і key(), що дозволяють кожному модулю знати лише свій посередник, але не інші компоненти.

Абстрактний клас AbstractColleague реалізує частину спільної логіки для всіх колег. Він зберігає посилання на EditorMediator і надає метод send(EditorEvent e) — через нього модуль надсилає подію посереднику, не знаючи, хто її отримає. Кожен конкретний модуль перевизначає метод receive(EditorEvent e) для обробки отриманих подій.

Клас ConcreteEditorMediator виконує роль центрального координатора. У його методі register() усі колеги реєструються за унікальним ключем, після чого Mediator зберігає їхні посилання. Метод onEvent() визначає логіку маршрутизації подій між модулями: наприклад, подія AUDIO_IMPORTED від ImportModule передається до WaveformModule, TrackModule та NotificationModule, а подія

EXPORT_REQUEST — до ExportModule. Таким чином, логіка взаємодії між частинами системи повністю ізольована в одному класі.

Кожен модуль (ProjectModule, ImportModule, ExportModule, PlaybackModule, WaveformModule тощо) реалізує власну поведінку, але спілкується з іншими лише через Mediator. Наприклад, після імпорту аудіофайлу ImportModule надсилає подію AUDIO_IMPORTED, яку Mediator пересилає до модулів, що мають її обробити.

Клас EditorView залишається відповідальним лише за графічний інтерфейс користувача. Він створює події (через AudioEditor.notifyObservers(...)), які потрапляють до ConcreteEditorMediator, де відбувається подальша маршрутизація.

Таким чином, шаблон Mediator забезпечив:

- централізоване керування взаємодією між підсистемами;
- зменшення зв’язності між модулями;
- можливість легкого розширення — новий модуль можна додати без зміни існуючих;
- покращену масштабованість і зрозумілу архітектуру коду.

4. Взаємодія Mediator із шаблоном Observer

У системі Audio Editor шаблони Observer та Mediator працюють спільно, утворюючи цілісну подієво-орієнтовану архітектуру. Шаблон Observer використовується для розповсюдження подій, тоді як Mediator — для адресної маршрутизації між окремими модулями системи.

Послідовність обробки події:

1. Користувач натискає кнопку Import Audio у вікні редактора.

2. Клас EditorView формує подію IMPORT_REQUEST та викликає метод «editor.notifyObservers(event);» Ця подія надсилається у систему спостереження.

3. Клас AudioEditor (реалізація шаблону *Observer – Subject*) розсилає подію всім підписаним спостерігачам.

4. Серед спостерігачів є MediatorObserver, який перехоплює цю подію і передає її посереднику через виклик «mediator.onEvent(null, event);»

5. ConcreteEditorMediator отримує подію IMPORT_REQUEST і визначає, якому модулю (колезі) її потрібно передати. У цьому випадку викликається «importModule.receive(event);».

6. ImportModule виконує логіку імпорту аудіофайлу. Після успішного завершення імпорту він генерує нову подію AUDIO_IMPORTED і відправляє її знову через медіатор методом «send(new EditorEvent(EditorEventType.AUDIO_IMPORTED));».

7. Посередник ConcreteEditorMediator отримує цю подію та надсилає її кільком модулям, яким вона стосується:

- WaveformModule — для оновлення відображення хвильової форми;
- TrackModule — для оновлення інформації про трек;
- NotificationModule — для виведення повідомлення користувачу про успішний імпорт.

У результаті EditorView не має знань про те, які саме модулі виконують імпорт чи обробку звуку. Всі зв’язки між компонентами здійснюються через ConcreteEditorMediator, який координує їхню взаємодію. Такий підхід усуває пряму залежність між компонентами, зменшує зв’язаність системи, робить код гнучкішим, легшим для модифікації та розширення.

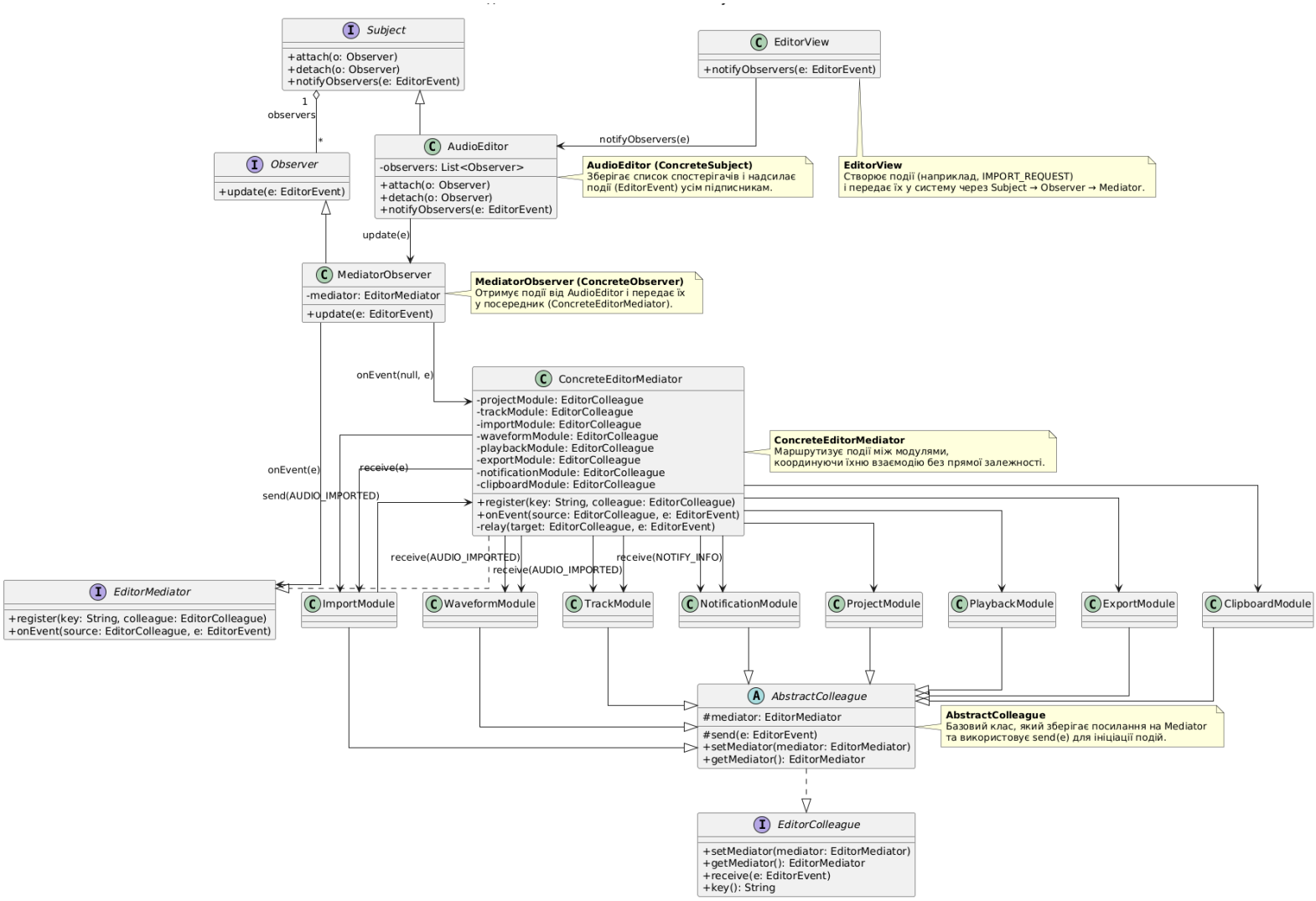


Рисунок 1.1 – Представлення взаємодії шаблонів проєктування «Посередник» та «Спостерігач» на діаграмі класів

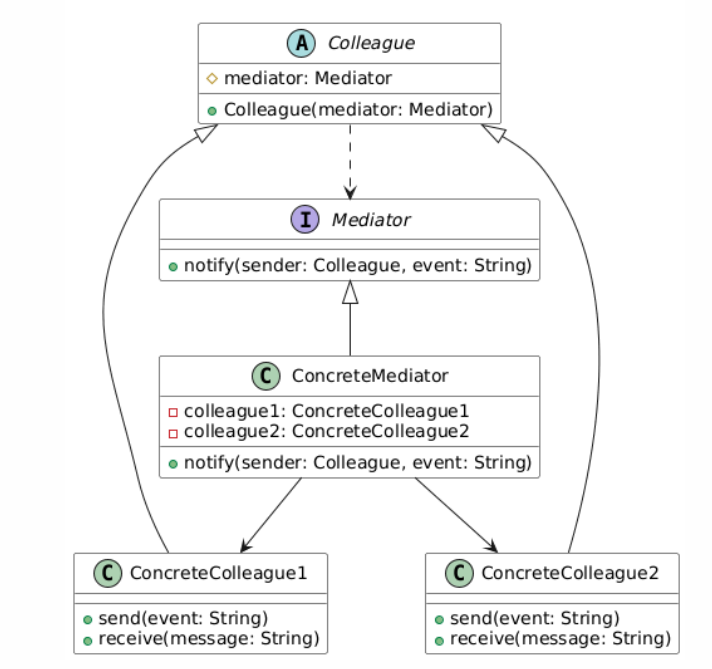
Відповіді на контрольні запитання:

1. Яке призначення шаблону «Посередник»?

Шаблон «Mediator» (посередник) використовується для визначення взаємодії об'єктів за допомогою іншого об'єкта (замість зберігання посилань один на одного). Даний шаблон схожий на шаблон «команда», проте в даному випадку

замість зберігання даних про конкретну дію, зберігаються дані про взаємодії між компонентами.

2. Нарисуйте структуру шаблону «Посередник».



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

Класи, що входять в шаблон «Посередник»:

- *Mediator – визначає інтерфейс для обміну інформацією з об'єктами Colleague;*
- *ConcreteMediator – реалізує кооперативну поведінку, координуючи дії об'єктів Colleague; володіє інформацією про колег, та підраховує їх;*
- *Класи Colleague – кожному класу Colleague відомо про свій об'єкт Mediator; усі колеги обмінюються інформацією виключно через посередника, інакше за його відсутності їм довелося б спілкуватися між собою напряму.*

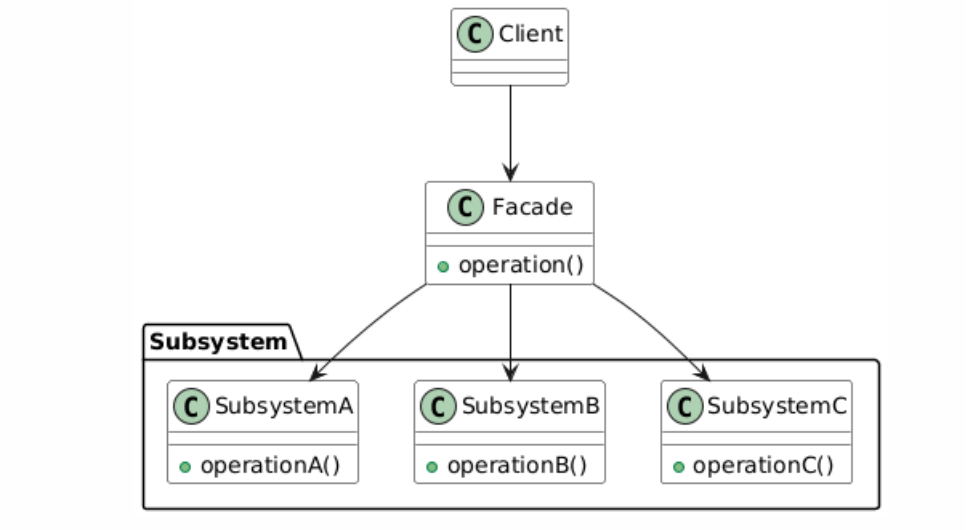
Взаємодія між класами:

Колеги посилають запити посередникові та отримують запити від нього. Посередник реалізує кооперативну поведінку шляхом переадресації кожного запиту відповідному колезі (або декільком з них).

4. Яке призначення шаблону «Фасад»?

Шаблон «Facade» (фасад) передбачає створення єдиного уніфікованого способу доступу до підсистеми без розкриття внутрішніх деталей підсистеми. Оскільки підсистема може складатися з безлічі класів, а кількість її функцій – не більше десяти, то щоб уникнути створення «спагеті коду» (коли все тісно пов'язано між собою) виділяють один загальний інтерфейс доступу, здатний правильним чином звертатися до внутрішніх деталей.

5. Нарисуйте структуру шаблону «Фасад».



Шаблон «Фасад» передбачає використання окремого класу Facade, який надає клієнту спрощений і узагальнений інтерфейс для взаємодії зі складною

підсистемою. Клас Facade інкапсулює роботу підсистеми та делегує виклики відповідним класам, що знаходяться всередині неї. Підсистема представлена сукупністю класів (SubsystemA, SubsystemB, SubsystemC), які виконують основну функціональну логіку. Клієнтський код звертається лише до об'єкта фасаду, не маючи прямої залежності від класів підсистеми.

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

До шаблону входять:

- *Facade (Фасад) – єдиний клас, що надає уніфікований, спрощений інтерфейс до підсистеми.*
- *Subsystem (Підсистема) – набір класів, що реалізують складну функціональність, але є прихованими (наприклад, internal).*

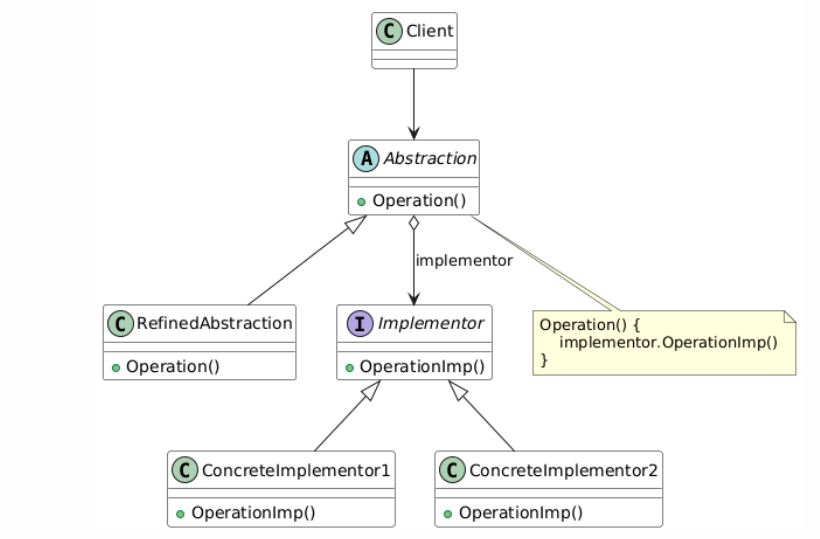
Взаємодія:

Клієнти (зовнішній код) взаємодіють лише з об'єктом Facade. Facade звертається до внутрішніх класів підсистеми, щоб виконати запит клієнта. Це інкапсулює внутрішню структуру та дозволяє її змінювати, не впливаючи на клієнтський код.

7. Яке призначення шаблону «Міст»?

Шаблон «Bridge» (Міст) використовується для розділення інтерфейсу (абстракції) та його реалізації таким чином, щоб перше та друге можна було змінювати незалежно одне від одного.

8. Нарисуйте структуру шаблону «Міст».



9. Які класи входять в шаблон «Міст», та яка між ними взаємодія?

Класи, що входять в шаблон «Посередник»:

- *Abstraction – визначає інтерфейс абстракції; зберігає посилання на об'єкт типу Implementor;*
- *RefinedAbstraction – розширює інтерфейс, означений абстракцією Abstraction;*
- *Implementor – визначає інтерфейс для класів реалізації. Він не зобов'язаний точно відповідати інтерфейсу класу Abstraction. Насправді обидва інтерфейси можуть бути зовсім різними. Зазвичай, інтерфейс класу Implementor надає тільки примітивні операції, а клас Abstraction визначає операції більш високого рівня, що базуються на цих примітивах;*
- *ConcreteImplementor – містить конкретну реалізацію інтерфейсу класу Implementor.*

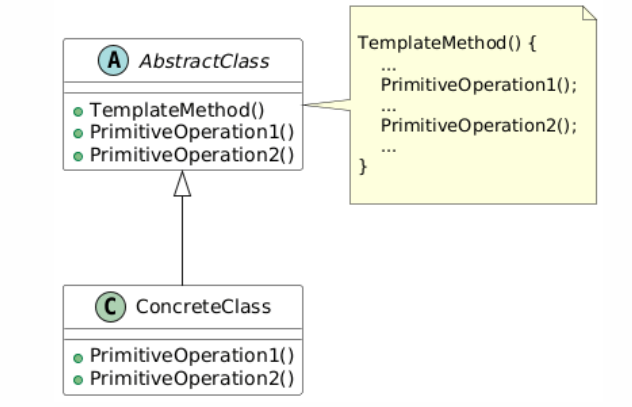
Взаємодія між класами:

Об'єкт Abstraction містить посилання на Implementor і делегує йому виконання запитів клієнта. RefinedAbstraction може розширювати поведінку базової абстракції, але також передає реалізацію методів об'єкту Implementor. Інтерфейс Implementor визначає базові операції, а класи ConcreteImplementor забезпечують їх конкретну реалізацію.

10. Яке призначення шаблону «Шаблоний метод»?

Шаблон Шаблонний метод (Template Method) визначає загальний алгоритм у базовому класі, дозволяючи підкласам змінювати окремі кроки алгоритму, не змінюючи його структури.

11. Нарисуйте структуру шаблону «Шаблоний метод».



12. Які класи входять в шаблон «Шаблоний метод», та яка між ними взаємодія?

Класи, що входять в шаблон «Посередник»:

- *AbstractClass – визначає абстрактні примітивні операції, що заміщуються у конкретних підкласах для реалізації кроків алгоритму та реалізує шаблонний метод, що визначає скелет алгоритму.*

Шаблонний метод викликає примітивні операції, а також операції, означенні у класі AbstractClass, чи в інших об'єктах;

- *ConcreteClass – реалізує примітивні операції, що виконують кроки алгоритму у спосіб, котрий залежить від підкласу;*

Взаємодія між класами:

ConcreteClass припускає, що незмінні кроки алгоритму реалізовані в AbstractClass, тоді як змінні кроки підкласи перевизначають за потреби.

13. Чим відрізняється шаблон «Шаблонний метод» від «Фабричного методу»?

Шаблон «Шаблонний метод» визначає загальну структуру алгоритму в базовому класі та дозволяє підкласам змінювати окремі його кроки без зміни послідовності виконання. Натомість «Фабричний метод» визначає інтерфейс для створення об’єктів, перекладаючи рішення про те, який саме об’єкт створити, на підкласи.

«Шаблонний метод» змінює кроки алгоритму, а «Фабричний метод» — спосіб створення об’єктів.

14. Яку функціональність додає шаблон «Міст»?

Шаблон «Міст» забезпечує розділення абстракції та її реалізації так, щоб їх можна було змінювати незалежно одна від одної. Це дозволяє легко розширювати ієрархії класів як з боку абстракції, так і з боку реалізації, не спричиняючи взаємної залежності.

Висновки:

У ході лабораторної роботи було реалізовано поведінковий шаблон проєктування «Посередник» на прикладі застосунку Audio Editor. Метою впровадження цього шаблону було усунення надмірної зв’язаності між окремими модулями системи та спрощення взаємодії між ними.

У процесі виконання роботи створено центральний компонент — посередник (ConcreteEditorMediator), який координує обмін подіями між незалежними модулями програми. Кожен модуль реалізує роль «колеги» (Colleague) і спілкується лише через посередника, не маючи прямих залежностей від інших частин системи. Такий підхід дозволив централізувати логіку обміну подіями та забезпечити чітке розділення відповідальностей між компонентами.

Завдяки впровадженню шаблону «Посередник» структура програми стала більш гнучкою, зрозумілою та масштабованою. Додавання нових функціональних модулів або зміна поведінки існуючих тепер не потребує втручання в інші частини коду.