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

Лабораторна робота № 6

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 «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator» 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 «Abstract Factory»

Призначення:

Шаблон застосовується для створення цілих сімейств взаємопов'язаних об'єктів без необхідності вказувати їх конкретні класи. Це дозволяє програмі бути незалежною від конкретних реалізацій, що використовуються.

Ідея:

Створюється інтерфейс, який визначає методи для створення різних типів продуктів. Конкретні фабрики реалізують цей інтерфейс, створюючи відповідні об'єкти. Таким чином, система може легко перемикатися між різними наборами об'єктів, не змінюючи код клієнта.

Основні елементи:

- AbstractFactory — оголошує інтерфейси для створення об'єктів певних типів.
- ConcreteFactory — реалізує методи для створення конкретних об'єктів.
- AbstractProduct — спільний інтерфейс для продуктів.
- ConcreteProduct — конкретна реалізація продукту.
- Client — використовує об'єкти, створені фабрикою, не знаючи їхніх конкретних класів.

Приклад:

Система створює інтерфейси користувача для різних ОС: Windows, Linux, macOS. Для кожної з них існує своя фабрика (WindowsFactory, MacFactory), яка створює відповідні кнопки, вікна та меню.

Переваги:

- Повна незалежність клієнтського коду від конкретних класів.
- Можливість легкої заміни сімейств продуктів.
- Централізоване створення об’єктів.

Недоліки:

- Ускладнює структуру програми через збільшення кількості класів.

Шаблон «Factory Method»

Призначення:

Замість створення об’єктів напряму через конструктор, цей шаблон делегує створення підкласам. Таким чином, програма може змінювати тип створюваного об’єкта, не змінюючи структуру основного коду.

Ідея:

Базовий клас містить метод, який повертає об’єкт певного типу. Підкласи перевизначають цей метод, створюючи об’єкти конкретних класів.

Структура:

- Creator — оголошує метод factoryMethod(), який створює об’єкт типу Product.
- ConcreteCreator — реалізує метод factoryMethod() для створення конкретного продукту.
- Product / ConcreteProduct — абстрактний і конкретний продукт.

Приклад:

У мережевій системі є класи TcpPacket та UdpPacket, які створюються через відповідні фабрики TcpCreator і UdpCreator. Базовий клас PacketCreator визначає спільну логіку відправки й отримання даних.

Переваги:

- Позбавляє клас залежності від конкретних реалізацій.
- Спрощує розширення системи.
- Централізує створення об’єктів.

Недоліки:

- Створює паралельні ієрархії класів.
- Ускладнює проєктування при великій кількості підкласів.

Шаблон «Memento»

Призначення:

Дозволяє зберігати та відновлювати попередній стан об’єкта без порушення інкапсуляції.

Основні учасники:

- Originator — об’єкт, стан якого потрібно зберегти.
- Memento — об’єкт, що зберігає стан Originator.
- Caretaker — керує збереженням і відновленням стану.

Ідея:

Об’єкт Originator створює Memento, який зберігає його поточний стан. Потім, у разі потреби, Originator може відновити стан із Memento. При цьому внутрішні деталі реалізації залишаються прихованими.

Приклад:

Функція “Undo” у текстовому редакторі: кожна зміна документа зберігає попередній стан у Memento, що дозволяє відновити текст.

Переваги:

- Збереження інкапсуляції.
- Простота реалізації “відкату” змін.
- Зменшення складності основного класу.

Недоліки:

- Велике споживання пам’яті при частих збереженнях.
- Може призвести до витоку пам’яті при неправильному керуванні старими станами.

Шаблон «Observer»

Призначення:

Встановлює залежність «один до багатьох» між об’єктами, коли зміна стану одного з них призводить до автоматичного оновлення всіх підписників.

Компоненти:

- Subject — зберігає список спостерігачів і повідомляє їх про зміни.
- Observer — визначає інтерфейс отримання оновлень.
- ConcreteSubject — конкретна реалізація об’єкта, який надсилає сповіщення.
- ConcreteObserver — реалізація підписника, який реагує на зміни.

Приклад:

У банківській системі, якщо змінюється баланс рахунку, усі клієнти, що підписані на нього, отримують оновлення.

У графічних інтерфейсах — оновлення вікна при зміні даних моделі (приклад архітектури MVC).

Переваги:

- Послаблює зв’язки між компонентами.
- Полегшує додавання нових типів підписників.
- Застосовується у подієвих системах.

Недоліки:

- Може призвести до каскадних оновлень.
- Ускладнює налагодження через велику кількість зв’язків.

Шаблон «Decorator»

Призначення:

Додає нову поведінку або функціональність об’єкту динамічно, без зміни його класу.

Суть:

Декоратор “обгортає” базовий об’єкт, реалізуючи той самий інтерфейс, і виконує додаткові дії до або після виклику базового методу.

Структура:

- Component — базовий інтерфейс або клас.
- ConcreteComponent — основна реалізація.
- Decorator — зберігає посилання на об’єкт Component і делегує йому виклики.
- ConcreteDecorator — додає нову поведінку.

Приклад:

Декорування графічних елементів (додавання рамки, тіні, кольору).

У потоковому введенні/виведенні — обгортання потоків (наприклад, BufferedReader над FileReader).

Переваги:

- Гнучке розширення поведінки без зміни коду класу.
- Можливість комбінування декораторів.
- Відповідає принципу відкритості/закритості.

Недоліки:

- Велика кількість дрібних об’єктів у складних системах.
- Складніше налагодження через вкладеність викликів.

Хід роботи:

1. Діаграма класів.

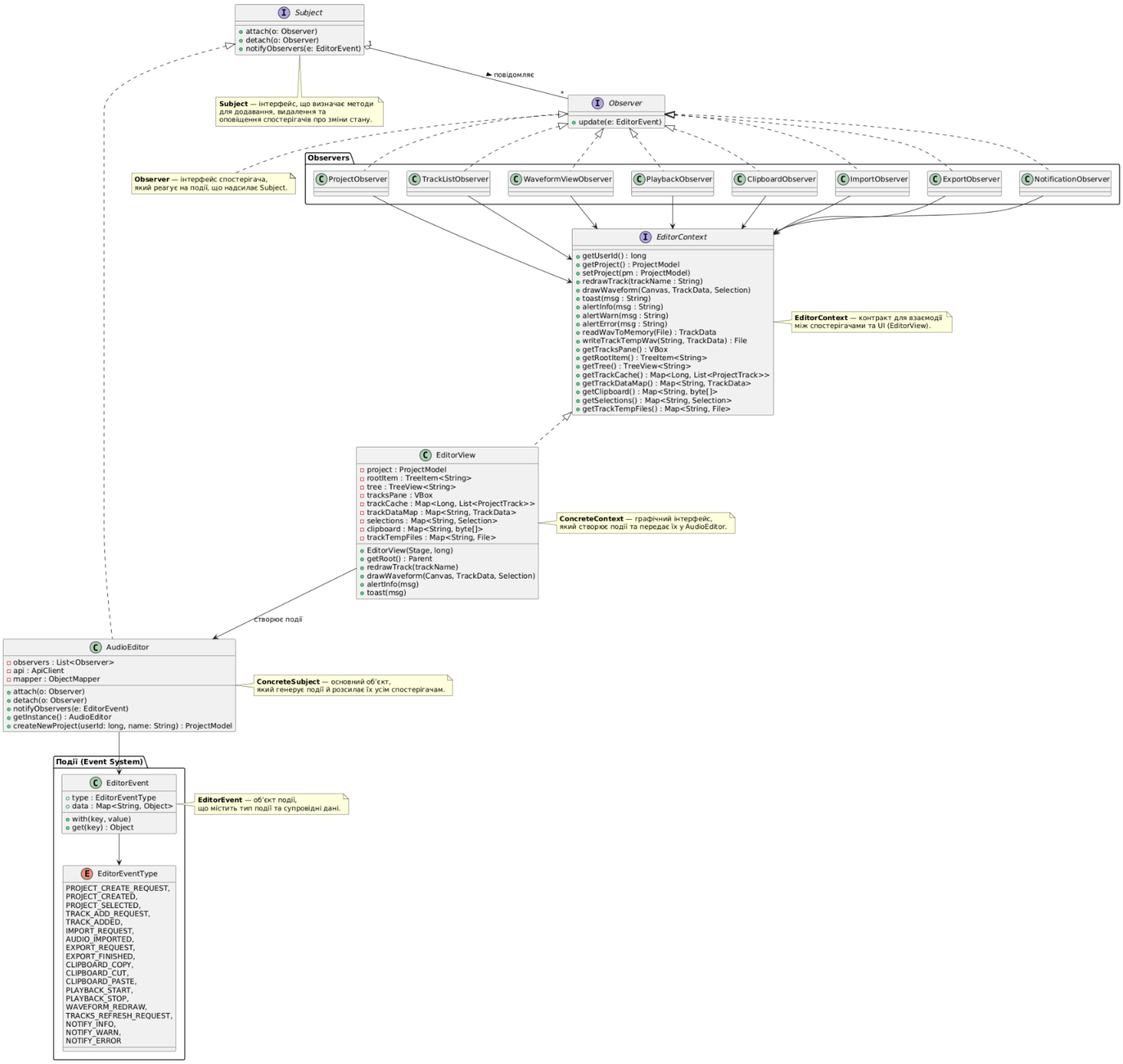


Рисунок 1.1 – Представлення шаблону проєктування «Спостерігач» на діаграмі класів

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

Шаблон Observer використовується для автоматичного сповіщення пов’язаних об’єктів про зміну стану певного об’єкта без створення жорсткої залежності між ними. Його суть полягає в тому, що є один суб’єкт, який зберігає список спостерігачів і повідомляє їх про події або зміни.

Проблема:

Початкова реалізація програми Audio Editor містила практично всю логіку в одному класі EditorView.
Саме в ньому виконувалися:

- створення нового проєкту;
- додавання треків;
- імпорт та експорт файлів;
- оновлення хвильової форми;
- керування відтворенням.

Усі ці дії оброблялися безпосередньо через умовні оператори та ручні виклики методів у межах одного класу.

Такий підхід призводив до надмірної зв’язаності — будь-яка зміна в логіці створення проєкту або імпорту вимагала внесення змін у EditorView, що ускладнювало розширення функціональності та знижувало стабільність коду.

Рішення:

Для усунення цих недоліків було застосовано шаблон Observer.
Його реалізація дозволила розділити монолітну логіку класу EditorView на

незалежні модулі, кожен з яких відповідає за власну підсистему (проєкти, треки, імпорт, відтворення тощо).

Архітектура рішення:

Інтерфейс Observer визначає метод «*void update(EditorEvent event);*», який викликається при надходженні нової події.

Клас AudioEditor — реалізує роль Subject . Він зберігає список спостерігачів і через метод notifyObservers() розсилає їм повідомлення про події, такі як створення проєкту, додавання треку, імпорт чи експорт аудіо.

Клас EditorEvent — інкапсулює тип події (EditorEventType) та супровідні параметри (наприклад, projectId, trackName, stage).

Конкретні спостерігачі:
Кожен із них реалізує власну логіку реакції на певні типи подій:

- ProjectObserver — створення й відображення проєктів;
- TrackListObserver — управління списком треків;
- WaveformViewObserver — відображення звукової хвилі;
- PlaybackObserver — керування програванням;
- ImportObserver / ExportObserver — імпорт та експорт файлів;
- NotificationObserver — показ повідомлень користувачу;
- ClipboardObserver — обробка операцій копіювання, вирізання, вставки аудіо в межах треку.

Клас EditorView тепер відповідає лише за інтерфейс — він створює події і передає їх у AudioEditor, який уже сповіщає всі підписані спостерігачі.

2.2. Код реалізації шаблона «Observer».

2.2.1. Інтерфейс Observer.

```
public interface Observer {
    void update(EditorEvent event);
}
```

2.2.2. Інтерфейс Subject.

```
public interface Subject {
    void attach(Observer o);
    void detach(Observer o);
    void notifyObservers(EditorEvent event);
}
```

2.2.3. Клас AudioEditor (Subject).

```
public class AudioEditor implements Subject {
    private static final AudioEditor INSTANCE = new AudioEditor();

    private final List<Observer> observers = new ArrayList<>();
    private final ApiClient api = ApiClient.getInstance();
    private final ObjectMapper mapper = new ObjectMapper();

    private AudioEditor() { }
```

```
    public static AudioEditor getInstance() { return INSTANCE; }

    @Override
    public synchronized void attach(Observer o) {
        if (!observers.contains(o)) observers.add(o);
    }

    @Override
    public synchronized void detach(Observer o) {
        observers.remove(o);
    }

    @Override
    public synchronized void notifyObservers(EditorEvent event) {
        List<Observer> snapshot = new ArrayList<>(observers);
        for (Observer o : snapshot) {
            try { o.update(event); } catch (Exception ex) { ex.printStackTrace(); }
        }
    }
...
}
```

2.2.4. Клас EditorEvent.

```
        public class EditorEvent {
    public final EditorEventType type;
    public final Map<String, Object> data = new HashMap<>();

    public EditorEvent(EditorEventType type) {
        this.type = type;
    }

    public EditorEvent with(String key, Object value) {
        data.put(key, value);
        return this;
    }

    @SuppressWarnings("unchecked")
    public <T> T get(String key) {
        return (T) data.get(key);
    }
}
```

2.2.5. Клас EditorEventType.

```
public enum EditorEventType {
    PROJECT_CREATE_REQUEST,
    PROJECT_CREATED,
    PROJECT_SELECTED,
    TRACK_ADD_REQUEST,
    TRACK_ADDED,
    IMPORT_REQUEST,
    AUDIO_IMPORTED,
    EXPORT_REQUEST,
    EXPORT_FINISHED,
    CLIPBOARD_COPY,
    CLIPBOARD_CUT,
    CLIPBOARD_PASTE,
    PLAYBACK_START,
    PLAYBACK_STOP,
    WAVEFORM_REDRAW,
    TRACKS_REFRESH_REQUEST,
    NOTIFY_INFO,
    NOTIFY_WARN,
    NOTIFY_ERROR ,
}
```

2.2.6. Клас EditorEvent.

```
public class EditorEvent {
    public final EditorEventType type;
```

```
    public final Map<String, Object> data = new HashMap<>();
    public EditorEvent(EditorEventType type) {
    this.type = type;
    }
    public EditorEvent with(String key, Object value) {
    data.put(key, value);
    return this;
    }
    @SuppressWarnings("unchecked")
    public <T> T get(String key) {
    return (T) data.get(key);
    }
}
```

2.2.7. Інтерфейс EditorContext.

```
public interface EditorContext {
    // Стан
    long getUserId();
    ProjectModel getProject();
    void setProject(ProjectModel pm);

    // UI вузли
    TreeItem<String> getRootItem();
    TreeView<String> getTree();
    VBox getTracksPane();
    void setCurrentProjectNode(TreeItem<String> node);
    TreeItem<String> getCurrentProjectNode();
```

```
// Кеш/мапи
Map<Long, List<ProjectTrack>> getTrackCache();
Map<String, TrackData> getTrackDataMap();
Map<String, Selection> getSelections();
Map<String, byte[]> getClipboard();
Map<String, File> getTrackTempFiles();

// Малювання / утиліти
void drawWaveform(Canvas c, TrackData td, Selection sel);
void drawEmptyBackground(Canvas c, String msg);
void redrawTrack(String trackName);
void toast(String msg);
void alertInfo(String msg);
void alertWarn(String msg);
void alertError(String msg);

// IO
TrackData readWavToMemory(File wav) throws Exception;
File ensureWav(File any) throws Exception;
File writeTrackTempWav(String trackName, TrackData td) throws Exception;
File writePcmToTempWav(AudioFormat fmt, byte[] pcm) throws Exception;
Stage getStage();

class TrackData {
    public AudioFormat format;
    public byte[] pcm;
    public int frameSize;
```

```
        public float frameRate;
        public long framesCount;
        public double durationSec;
    }

    class Selection {
        public double xStart = -1;
        public double xEnd = -1;
        public void clear() { xStart = -1; xEnd = -1; }
        public boolean isActive() { return xStart >= 0 && xEnd >= 0 && Math.abs(xEnd - xStart) > 1.5; }
        public double left() { return Math.min(xStart, xEnd); }
        public double right() { return Math.max(xStart, xEnd); }
        public double width() {
            return Math.abs(xEnd - xStart);
        }
    }
}
```

2.2.8. Фрагмент EditorView, який демонструє ініціацію події.

```
public class EditorView implements EditorContext {
    public EditorView(Stage stage, long userId) {
        var editor = AudioEditor.getInstance();
        editor.attach(new ClipboardObserver(this));
        editor.attach(new NotificationObserver(this));

        Button copyBtn = new Button("Copy");
        copyBtn.setOnAction(e ->
            editor.notifyObservers(
```

```
            new EditorEvent(EditorEventType.CLIPBOARD_COPY)
                .with("trackName", "Track 1"))
        );
    }
}
```

2.2.9. Класи конкретних спостерігачів.

ProjectObserver :

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

TrackListObserver:

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

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

PlaybackObserver:

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

ImportObserver :

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

ExportObserver:

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

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

ClipboardObserver:

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

2.3. Пояснення коду реалізації шаблону «Observer».

У системі Audio Editor шаблон «Спостерігач» використано для побудови подієво-орієнтованої архітектури, що дозволяє розділити логіку обробки подій на незалежні модулі. Завдяки цьому будь-яка зміна в одній частині програми не вимагає модифікації решти компонентів.

Центральним елементом є клас AudioEditor, який реалізує інтерфейс Subject. Він підтримує список зареєстрованих спостерігачів (attach/detach) і розсилає їм повідомлення (notifyObservers) про зміни стану у вигляді об’єкта EditorEvent, який інкапсулює тип події (EditorEventType) і супутні параметри.

Інтерфейс Observer визначає єдиний метод update(EditorEvent event), який реалізують усі конкретні спостерігачі. Кожен спостерігач відповідає за власну підсистему програми:

- ProjectObserver — створення та відображення проєктів;
- TrackListObserver — управління списком треків;
- WaveformViewObserver — оновлення та відображення хвильової форми;
- PlaybackObserver — керування процесом відтворення;
- ImportObserver та ExportObserver — імпорт і експорт аудіофайлів;
- ClipboardObserver — операції копіювання, вирізання, вставки;
- NotificationObserver — сповіщення користувача про події.

Клас EditorView тепер відповідає лише за графічний інтерфейс користувача. Він створює кнопки, панелі, елементи вибору і надсилає події у AudioEditor.
Після цього об’єкт AudioEditor сповіщає всі підписані спостерігачі, і кожен з них виконує власну реакцію.

Наприклад, WaveformViewObserver обробляє події AUDIO_IMPORTED та

WAVEFORM_REDRAW — оновлює PCM-дані треку, застосовує ефекти до виділеного сегмента і викликає метод redrawTrack() з EditorView, щоб перемалювати хвильову форму.

Інтерфейс EditorContext використовується як міст між спостерігачами та інтерфейсом користувача: він надає доступ до елементів інтерфейсу, а також до службових методів для малювання, роботи з файлами й відображення повідомлень.

Таким чином, шаблон Observer забезпечив:

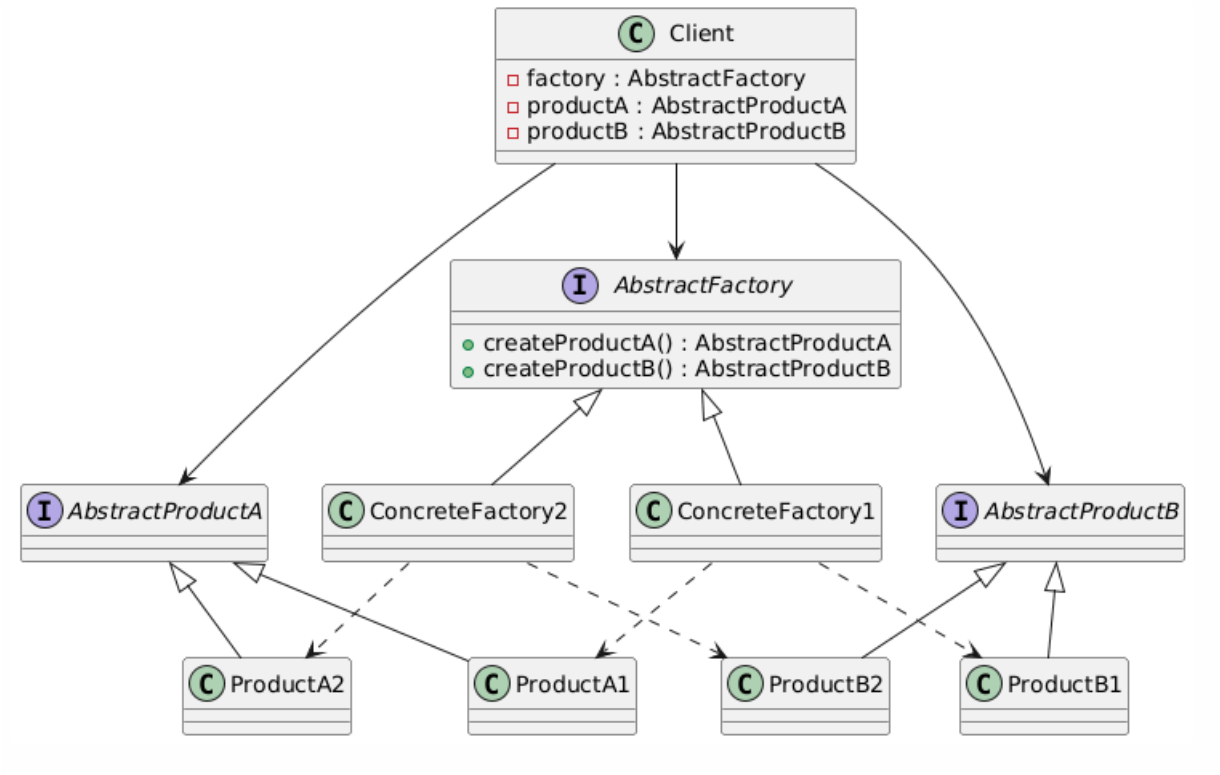
- розв’язання жорсткої зв’язаності між компонентами;
- чітке розділення відповідальностей;
- можливість масштабування — додавання нових типів подій або спостерігачів без змін у базовому коді.

Відповіді на контрольні запитання:

1. Яке призначення шаблону «Абстрактна фабрика»?

Шаблон «Абстрактна фабрика» призначений для створення сімейств взаємопов’язаних об’єктів без зазначення їх конкретних класів. Він дозволяє клієнтському коду працювати лише з інтерфейсами, не знаючи деталей реалізації конкретних класів, що забезпечує незалежність і розширюваність системи.

2. Нарисуйте структуру шаблону «Абстрактна фабрика».



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

Класи, що входять в шаблон «Абстрактна фабрика»:

- *AbstractFactory — оголошує інтерфейс для операцій, що створюють абстрактні об'єкти-продукти;*
- *ConcreteFactory — реалізує операції, що створюють конкретні об'єкти-продукти;*
- *AbstractProduct — оголошує інтерфейс для типу об'єкта-продукту;*
- *ConcreteProduct — визначає об'єкт-продукт, що створюється відповідною конкретною фабрикою та реалізує інтерфейс AbstractProduct;*
- *Client — користується виключно інтерфейсами, котрі оголошенні у класах AbstractFactory та AbstractProduct.*

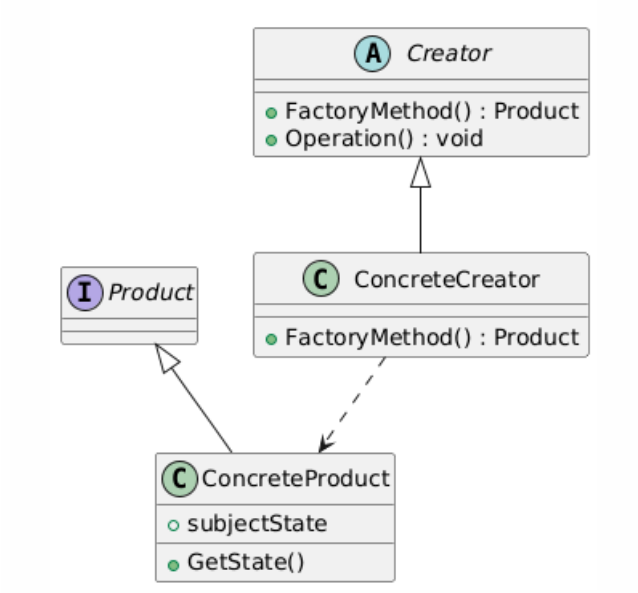
Взаємодія між класами:

Зазвичай під час виконання створюється єдиний екземпляр класу ConcreteFactory. Ця конкретна фабрика створює об'єкти продукти, що мають досить визначену реалізацію. Для створення інших видів об'єктів клієнт повинен користуватися іншою конкретною фабрикою. AbstractFactory передоручає створення об'єктів продуктів своєму підкласу ConcreteFactory.

3. Яке призначення шаблону «Фабричний метод»?

Шаблон «Фабричний метод» призначений для делегування створення об’єктів підкласам. Він дозволяє базовому класу визначати інтерфейс для створення об’єкта, а підкласи вирішують, який саме тип об’єкта створювати.

4. Нарисуйте структуру шаблону «Фабричний метод».



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

Класи, що входять в шаблон «Фабричний метод»:

- *Product — визначає інтерфейс об'єктів, що створюються фабричним методом;*
- *ConcreteProduct — реалізує інтерфейс Product;*
- *Creator — оголошує фабричний метод, що повертає об'єкт класу Product. Creator може також визначати реалізацію за умовчанням фабричного методу, що повертає об'єкт ConcreteProduct та може викликати фабричний метод для створення об'єкта Product;*
- *ConcreteCreator — заміщує фабричний метод, що повертає об'єкт ConcreteProduct.*

Взаємодія між класами:

Творець делегує створення об’єктів своїм підкласам, які повертають конкретні продукти через спільний інтерфейс.

7. Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»?

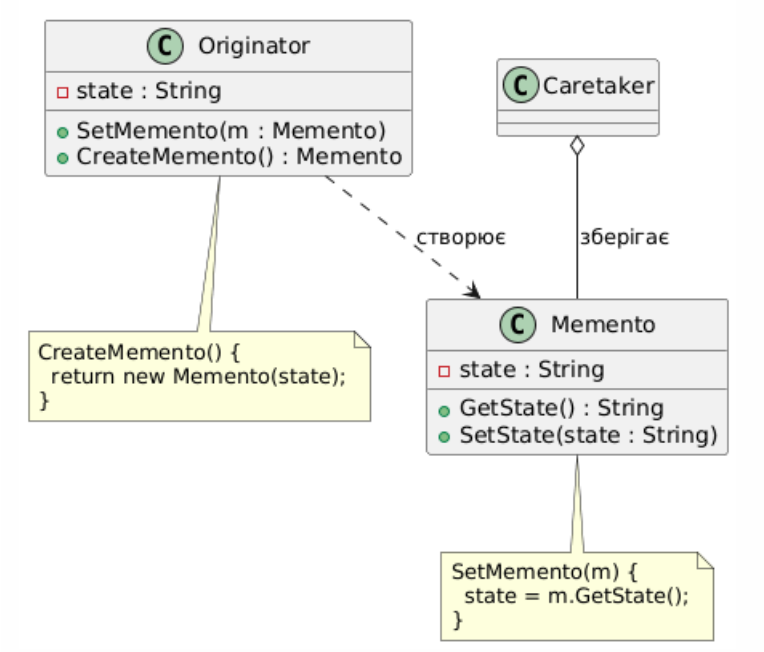
Шаблон «Абстрактна фабрика» створює сімейства взаємопов’язаних об’єктів без зазначення їх конкретних класів, тобто керує комбінацією пов’язаних продуктів.

Шаблон «Фабричний метод» делегує створення одного конкретного об’єкта підкласам, тобто керує одним конкретним типом продукту.

8. Яке призначення шаблону «Знімок»?

Шаблон «Знімок» використовується для збереження і відновлення попереднього стану об’єкта без порушення його інкапсуляції.

9. Нарисуйте структуру шаблону «Знімок».



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

Класи, що належать шаблону «Знімок»:

- *Memento — зберігає внутрішній стан об'єкта Originator. Обсяг інформації, що зберігається, може бути різним та визначається потребами хазяїна. Забороняє доступ усім іншим об'єктам окрім хазяїна. По суті знімок має два інтерфейси. Опікун Caretaker користується лише вузьким інтерфейсом знімку — він може лише передавати знімок іншим об'єктам. Навпаки, хазяїн користується широким інтерфейсом, котрий забезпечує доступ до всіх даних, необхідних для відтворення об'єкта (чи його частини) у попередньому стані. Ідеальний варіант — коли тільки хазяїну, що створив знімок, відкритий доступ до внутрішнього стану знімку;*
- *Originator — створює знімок, що утримує поточний внутрішній стан та використовує знімок для відтворення внутрішнього стану;*

- *CareTaker — відповідає за зберігання знімка та не проводить жодних операцій над знімком та не має уявлення про його внутрішній зміст.*

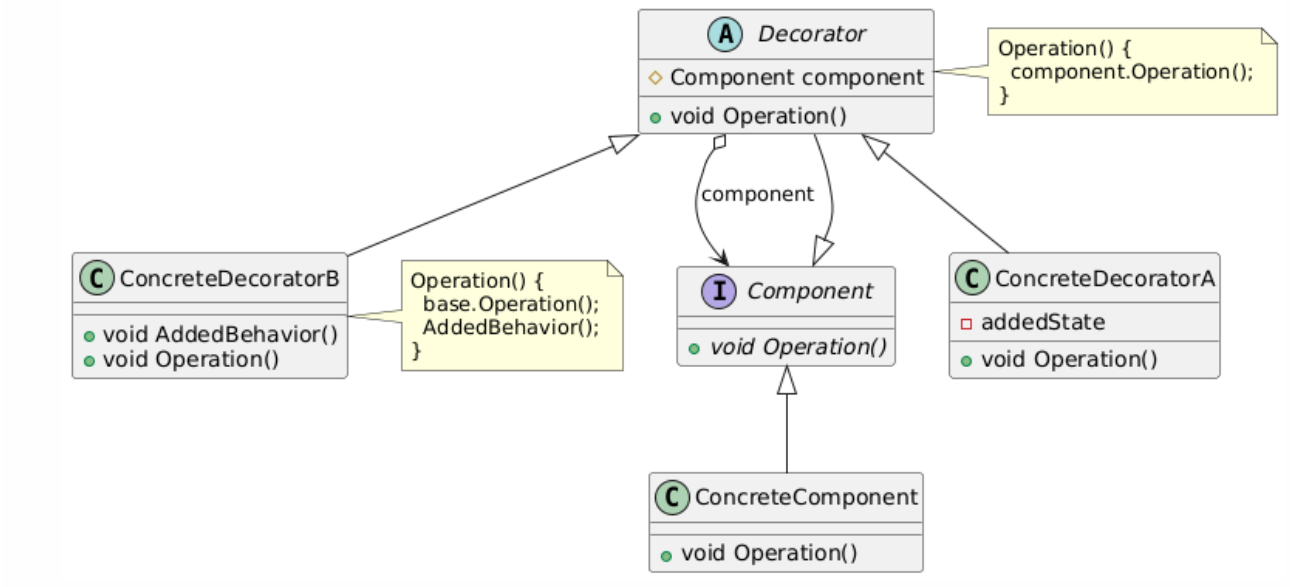
Взаємодія між класами:

Опікун запитує знімок у хазяїна, деякий час тримає його у себе, опісля повертає хазяїну. Іноді цього не відбувається, бо хазяїн не має необхідності відтворювати свій попередній стан. Знімки пасивні. Тільки хазяїн, що створив знімок, має доступ до інформації про стан.

11. Яке призначення шаблону «Декоратор»?

Шаблон «Декоратор» призначений для динамічного додавання нових функціональних можливостей до об'єкта під час виконання програми, не змінюючи його вихідний код.

12. Нарисуйте структуру шаблону «Декоратор».



13. Які класи входять в шаблон «Декоратор», та яка між ними взаємодія?

Класи, які входять в шаблон «Декоратор»:

- *Component — визначає базові операції.*
- *ConcreteComponent — основна реалізація.*
- *Decorator — має посилання на Component і делегує виклики.*
- *ConcreteDecorator — розширює поведінку.*

Взаємодія між класами:

Клієнт звертається до об'єкта через інтерфейс Component. У цей момент реальним об'єктом може бути як звичайний ConcreteComponent, так і ланцюг декораторів, які «обгортають» базовий компонент. Кожен декоратор виконує свою додаткову дію і передає виклик далі по ланцюгу.

14. Які є обмеження використання шаблону «декоратор»?

Велика кількість крихітних класів, і важко конфігурувати об'єкти, які загорнуто в декілька обгорток одночасно.

Висновки:

У лабораторній роботі було реалізовано шаблон «Спостерігач» у клієнтській частині системи Audio Editor. Його впровадження дозволило усунути надмірну зв'язаність компонентів, що існувала у початковій реалізації, де вся логіка була зосереджена в класі EditorView.

Після застосування патерну «Observer» систему розділено на незалежні модулі: AudioEditor виступає суб'єктом, а окремі класи-спостерігачі реагують на

зміни стану. Передача даних здійснюється за принципом розповсюдження за запитом и – спостерігачі самостійно отримують необхідну інформацію через EditorContext.

Завдяки цьому архітектура застосунку стала масштабованою та зручною для подальшого розширення.