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

**Лабораторна робота № 4**

з дисципліни «Технології розроблення програмного забезпечення»

Тема лабораторної роботи: «Вступ до паттернів проектування»

Тема проєкта: «Аудіо редактор»

Виконала:
студентка групи ІА-33
Котик Іванна

Перевірив:
асистент кафедри ІСТ
Мягкий Михайло Юрійович

Київ 2025

# Зміст

**Тема:** Вступ до паттернів проектування.

**Мета:** Вивчити структуру шаблонів «Singleton», «Iterator», «Proxy», «State», «Strategy» 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:

### Поняття шаблону проєктування

Будь-який патерн проєктування, використовуваний при розробці інформаційних систем, являє собою формалізований опис, який часто зустрічається в завданнях проєктування, вдале рішення даної задачі, а також рекомендації по застосуванню цього рішення в різних ситуаціях. Крім того, патерн проєктування обов'язково має загальновживане найменування.

Правильно сформульований патерн проєктування дозволяє, відшукавши одного разу вдале рішення, користуватися ним знову і знову. Варто підкреслити, що важливим початковим етапом при роботі з патернами є адекватне моделювання розглянутої предметної області. Це є необхідним як для отримання належним чином формалізованої постановки задачі, так і для вибору відповідних патернів проєктування.

Відповідне використання патернів проєктування дає розробнику ряд незаперечних переваг. Наведемо деякі з них. Модель системи, побудована в межах патернів проєктування, фактично є структурованим виокремленням тих елементів і зв'язків, які значимі при вирішенні поставленого завдання. Крім цього, модель, побудована з використанням патернів проєктування, більш проста і наочна у вивченні, ніж стандартна модель. Проте, не дивлячись на простоту і наочність, вона дозволяє глибоко і всебічно опрацювати архітектуру розроблюваної системи з використанням спеціальної мови.

Застосування патернів проєктування підвищує стійкість системи до зміни вимог та спрощує неминуче подальше доопрацювання системи. Крім того, важко переоцінити роль використання патернів при інтеграції інформаційних систем організації. Також слід зазначити, що сукупність патернів проєктування, по суті, являє собою єдиний словник проєктування, який, будучи уніфікованим засобом, незамінний для спілкування розробників один одним.

Таким чином шаблони представляють собою, підтверджені роками розробок в різних компаніях і на різних проєктах, «ескізи» архітектурних рішень, які зручно застосовувати у відповідних обставинах.

**Шаблон «Singleton»**

Шаблон «Singleton» Призначення патерну: «Singleton» (Одинак) являє собою клас в термінах ООП, який може мати не більше одного об'єкта (звідси і назва «одинак»). Насправді, кількість об'єктів можна задати (тобто не можна створити більш n об'єктів даного класу). Даний об'єкт найчастіше зберігається як статичне поле в самому класі.

Проблема: Використання одинака виправдано для наступних випадків:

• може бути не більше N фізичних об'єктів, що відображаються в певних класах;

• необхідно жорстко контролювати всі операції, що проходять через даний клас.

Одинак вирішує відразу дві проблеми, порушуючи принцип єдиної відповідальності класу.

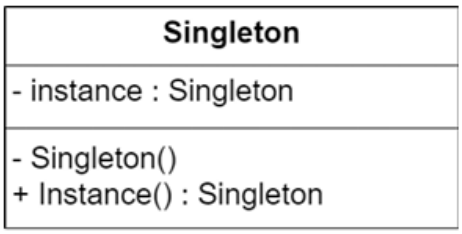



Рисунок 1.1. Структура патерну Одинак

Рішення:

Перший випадок легко продемонструвати. У кожної програми є певний набір налаштувань, який як правило зберігається в окремий файл (для сучасних комп'ютерних ігор це може бути .ini файл, для .net додатків –.xml файл). Цей файл – єдиний, і тому використання безлічі об'єктів для завантаження/запису даних – нераціональне рішення.

Для демонстрації другого випадку, розглянемо наступний приклад. Припустимо, існує дві взаємодіючі системи, між якими встановлено сеанс зв'язку. Накладені обмеження, що по даному сеансу зв'язку дані можуть йти в один момент часу лише в одну сторону. Таким чином, на кожен надісланий запит необхідно дочекатися відповіді, перш ніж відсилати новий запит. Об'єкт «одинак» дозволить не тільки містити рівно один сеанс зв'язку, а й ще реалізувати відповідну логіку перевірок на основі bool операторів про можливість відправки запиту і, можливо, деяку чергу запитів.

Переваги та недоліки: Однак слід зазначити, що в даний час патерн «Одинак» багато хто вважає т.зв. «анти-шаблоном», тобто поганою практикою проєктування. Це пов'язано з тим, що «одинаки» представляють собою глобальні дані (як глобальна змінна), що мають стан. Стан глобальних об'єктів важко відслідковувати і підтримувати коректно; також глобальні об'єкти важко тестуються і вносять складність в програмний код (у всіх ділянках коду виклик в одне єдине місце з «одинаком»; при зміні підходу доведеться змінювати масу коду).

При цьому реалізація контролю доступу можлива за допомогою статичних змінних, замикань, мютексов та інших спеціальних структур.

+ Гарантує наявність єдиного екземпляра класу.

+ Надає до нього глобальну точку доступу.

- Порушує принцип єдиної відповідальності класу.

- Маскує поганий дизайн.

**Шаблон «Iterator»**

Призначення: «Iterator» (Ітератор) являє собою шаблон реалізації об'єкта доступу до набору (колекції, агрегату) елементів без розкриття внутрішніх механізмів реалізації. Ітератор виносить функціональність перебору колекції елементів з самої колекції, таким чином досягається розподіл обов'язків: колекція відповідає за зберігання даних, ітератор – за прохід по колекції.

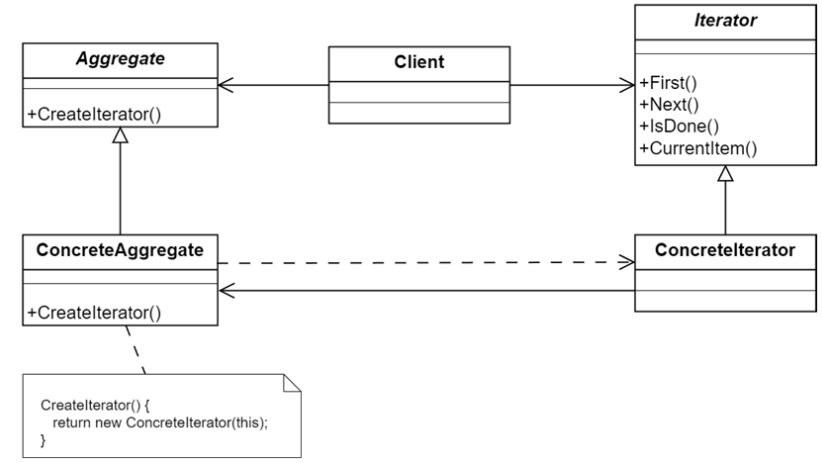



Рисунок 1.2. - Структура патерну Ітератор

При цьому алгоритм ітератора може змінюватися – при необхідності пройти в зворотньому порядку використовується інший ітератор. Можливо також написання такого ітератора, який проходить список спочатку по парних позиціях (2,4,6-й елементи і т.д.), потім по непарних. Тобто, шаблон ітератор дозволяє реалізовувати різноманітні способи проходження по колекції незалежно від виду і способу представлення даних в колекції.

Проблема: Більшість колекцій виглядають як звичайний список елементів. Але є й екзотичні колекції, побудовані на основі дерев, графів та інших складних структур даних.

Але як би не була структурована колекція, користувач повинен мати можливість послідовно обходити її елементи, щоб виробляти з ними якісь дії.

Але яким способом слід переміщатися по складній структурі даних? Наприклад, сьогодні може бути достатнім обхід дерева в глибину, але завтра буде потрібно можливість переміщатися по дереву в ширину. А на наступному тижні і того гірше – знадобиться обхід колекції у випадковому порядку.

Додаючи все нові алгоритми в код колекції, ви потроху розмиваєте її основне завдання, яке полягає в ефективному зберіганні даних.

Рішення: Ідея патерна Ітератор полягає в тому, щоб винести поведінку обходу колекції з самої колекції в окремий клас.

Об'єкт-ітератор буде відстежувати стан обходу, поточну позицію в колекції і скільки елементів ще залишилося обійти. Одну і ту ж колекцію зможуть одночасно обходити різні ітератори, а сама колекція не буде навіть знати про це.

До того ж, якщо вам знадобиться додати новий спосіб обходу, ви зможете створити окремий клас ітератора, не змінюючи існуючий код колекції.

Шаблонний ітератор містить:

• First() – установка покажчика перебору на перший елемент колекції;

• Next() – установка покажчика перебору на наступний елемент колекції;

• IsDone – булевське поле, яке встановлюється як true коли покажчик перебору досяг кінця колекції;

• CurrentItem – поточний об'єкт колекції.

Переваги та недоліки: Цей шаблон дозволяє уніфікувати операції проходження по наборам об'єктів для всіх наборів. Тобто, незалежно від реалізації (масив, зв'язаний список, незв'язаний список, дерево та ін.), кожен з наборів може використовувати будь-який з реалізованих ітераторів.

+ Дозволяє реалізувати різні способи обходу структури даних.

+ Спрощує класи зберігання даних.

- Не виправданий, якщо можна обійтися простим циклом.

**Шаблон «Proxy»**

Призначення: «Proxy» (Проксі) – об'єкти є об'єктами-заглушками або двійниками/замінниками для об'єктів конкретного типу. Зазвичай, проксі об'єкти вносять додатковий функціонал або спрощують взаємодію з реальними об'єктами.

Проксі об'єкти використовувалися в більш ранніх версіях інтернет браузерів, наприклад, для відображення картинки: поки картинка завантажується, користувачеві відображається «заглушка» картинки.

Проблема: Ви супроводжуєте систему, одна із частин якої працює з зовнішнім сервісом підписання документів, нарпиклад, DocuSign. Періодично клієнти в вашій системі формують звіти и форматі pdf, далі ваша система відправляє їх на підпис в сервіс DocuSign і потім періодично перевіряєте чи документ вже підписаний. Якщо документ підписаний, ви викачуєте його з сервісу і поміщаєте в своїй базі.

За останній рік кількість користувачів вашої системи виросло суттєво і почали приходити великі рахунки від DocuSign. Після аналізу ви розумієте, що рахунки зросли через велику кількість запитів з відправкою документів на підписання та запитів на перевірку чи документ вже підписаний. Після обговорення з бізнес-аналітиками та користувачам зрозуміли, що критичний інтервал доставки документів на підписання – 2 години і такий самий час критичності перевірки що документ підписаний і можна було б групувати всі запит на відправку та на отримання і відправляти пакетом раз на годину. На даний момент клієнтський код працює з класом DocSignManager через інтерфейс IDocSignManager.

Рішення: Для вирішення проблеми можна застосувати патерн "Замісник". Ви реалізовуєте клас замісник, який також реалізовує інтерфейс IDocSignManager, але він накопичує запити на відправку файлів на підписання і відправляє їх раз на годину, також він приблизно раз на годину отримує підписані документи, а на запити від клієнтів відповідає на основі інформації взятої з бази. Таким чином старий клас DocSignManager так само використовується для роботи з DocuSign сервісом, але вже набагато рідже, а клієнтський код взаємодіє з додатковим проміжним рівнем DocSignManagerProxy, хоча з точки зору клієнського коду нічого не змінилося і він працює з тим самим об'єктом IDocSignManager.

Реалізувавши такий підхід ви тепер економите 40% від попередньої вартості використання DocuSign сервіса і тепер основний вплив на вартість робить розмір файлів, що передаються на підпис, а не кількість запитів до служби.

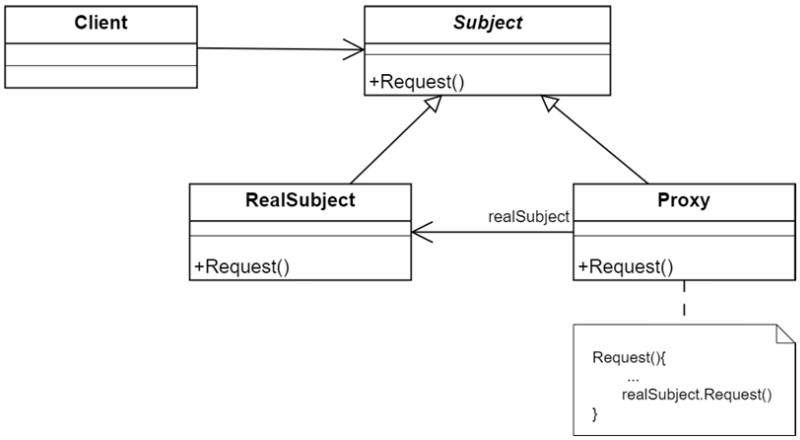



Рисунок 1.3. - Структура патерну Proxy

Переваги та недоліки:

\+ Легкість впровадження проміжного рівня без переробки клієнтського коду.

+ Додаткові можливості по керуванню життєвим циклом об’єкту. - Існує ризик падіння швидкості роботи через впровадження додаткових операцій.

- Існує ризик неадекватної заміни відповіді клієнтському коду.

**Шаблон «State»**

Призначення: Шаблон «State» (Стан) дозволяє змінювати логіку роботи об'єктів у випадку зміни їх внутрішнього стану. Наприклад, відсоток нарахованих на картковий рахунок грошей залежить від стану картки: Visa Electron, Classic, Platinum і т.д. Або обсяг послуг, які надані хостинг компанією, змінюється в залежності від обраного тарифного плану (стану членства – бронзовий, срібний або золотий клієнт). Реалізація даного шаблону полягає в наступному: пов'язані зі станом поля, властивості, методи і дії виносяться в окремий загальний інтерфейс (State); кожен стан являє собою окремий клас (ConcreteStateA, ConcreteStateB), які реалізують загальний інтерфейс.

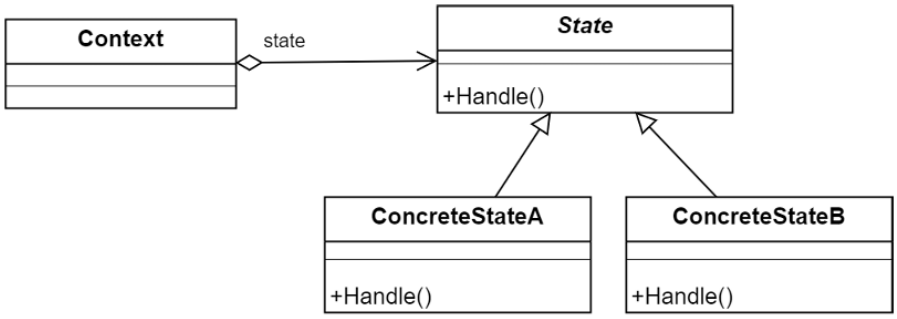



Рисунок 4.4. Структура патерну Стан

Об’єкти, що мають стан (Context), при зміні стану просто записують новий об'єкт в поле state, що призводить до повної зміни поведінки об'єкта.

Це дозволяє легко додавати в майбутньому і обробляти нові стани, відокремлювати залежні від стану елементи об'єкта в інших об'єктах, і відкрито проводити заміну стану (що має сенс у багатьох випадках).

Проблема: Ви розробляєте систему, яка складається з мобільних клієнтів та центрального сервера.

Для отримання запитів від клієнтів ви створюєте модуль Listener. Але вам потрібно щоб під час старту, поки сервер не запустився Listener не приймав запити від клієнтів, а після команди

shutdown, поки сервер зупиняється, Listener вже не приймав запити від клієнтів, але відповіді, якщо вони будуть готові, відправив.

Рішення: Тут явно видно три стани для Listener з різною поведінкою: initializing, open, closing. Тому для вирішення краще за все підходить патерн State.

Визначаємо загальний інтерфейс IListenerState з методами, які по різному працюють в різних станах. Під кожний стан створюємо клас з реалізацією цього інтерфейсе. Контекст Listener при цьому всі виклики буде перенаправляти на об'єкт стану простим делегуванням. При старті він (Listener) буде посилатися на об'єкт стану InitializingState, знаходячись в якому Listener, фактично, буде ігнорувати всі вхідні запити. Після того як система запуститься і завантажить всі базові дані для роботи, Listener буде переключено в робочий стан, наприклад, викликом методу Open(). Після цього Listener буде посилатися на об'єкт OpenState і буде відпрацьовувати всі вхідні повідомлення в звичайному режимі. При запуску виключення системи, Listener буде переключено в стан ClosingState, викликом методу Close().

Переваги та недоліки:

+ Код специфічний для окремого стану реалізується в класі стану.

+ Класи та об’єкти станів можна використовувати з різними контекстами, за рахунок чого збільшується гнучкість системи.

+ Код контексту простіше читати, тому що вся залежна від станів логіка винесена в інші класи.

+ Відносно легко додавати нові стани, головне правильно змінити переходи між станами.

- Клас контекст стає складніше через ускладнений механізм переключення станів.

**Шаблон «Strategy»**

Призначення: Шаблон «Strategy» (Стратегія) дозволяє змінювати деякий алгоритм поведінки об'єкта іншим алгоритмом, що досягає ту ж мету іншим способом. Прикладом можуть служити алгоритми сортування: кожен алгоритм має власну реалізацію і визначений в окремому класі; вони можуть бути взаємозамінними в об'єкті, який їх використовує.

Даний шаблон дуже зручний у випадках, коли існують різні «політики» обробки даних. По суті, він дуже схожий на шаблон «State» (Стан), проте використовується в абсолютно інших

цілях – незалежно від стану об'єкта відобразити різні можливі поведінки об'єкта (якими досягаються одні й ті самі або схожі цілі).

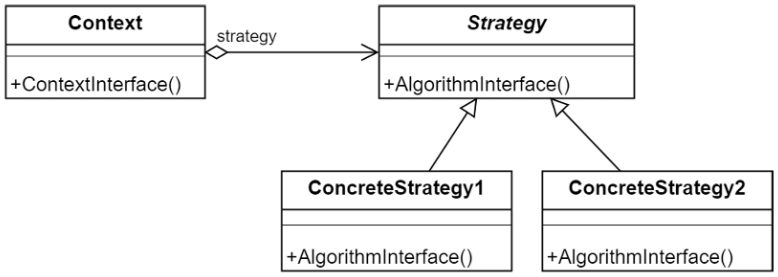



Рисунок 1.5. - Структура патерну Стратегія.

Рішення: Коли ви використовуєте патерн «Стратегія», то схожі алгоритми виносяться з класа контекста в конкретні стратегії, за рахунок чого клас контексту стає чистіше і його легше супроводжувати. Також одні і тіж самі стратегії можна використати з різними контекстами, що значно збільшує гнучкість вашої системи та зменшує кількість дублювань у коді.

Контекст при цьому містить посилання на конкретну стратегію, а коли стратегію потрібно замінити, то замінюється об’єкт стратегії в полі Context.strategy.

Важливою умовою є відносна простота інтерфейсу для алгоритмів стратегій. Якщо алгоритмам стратегій прийдеться передавати десятки параметрів, то це, скоріш за все, приведе до ускладнення системи та заплутаності коду. Якщо стратегії на вході будуть приймати об’єкт контексту, щоб отримувати з нього всі необхідні дані, то такі стратегії будуть прив’язані до конкретного контексту і їх не можна буде використати з іншим типом контексту.

Приклад з життя: Ви їдете на роботу. Можна доїхати на автомобілі, на метро, або йти пішки. Тут алгоритм, як ви добираєтеся на роботу, є стратегією. В залежності від поточної ситуації ви вибираєте стратегію, що найбільше підходить в цій ситуації, наприклад, на дорогах великі пробки тоді ви їдете на метро, або метро тимчасово не ходить тоді ви їдете на таксі, або ви знаходитеся в 5 хвилинах ходьби від місця роботи і простіше добратися пішки.

Переваги та недоліки:

+ Використовувані алгоритми можна змінювати під час виконання.

+ Реалізація алгоритмів відокремлюється від коду, що його використовує.

+ Зменшує кількість умовних операторів типу switch та if в контексті.

- Надмірна складність, якщо у вас лише кілька невеликих алгоритмів.

- Під час виклику алгоритму, клієнтський код має враховувати різницю між стратегіями.

## Хід роботи:

### 1. Діаграма класів.

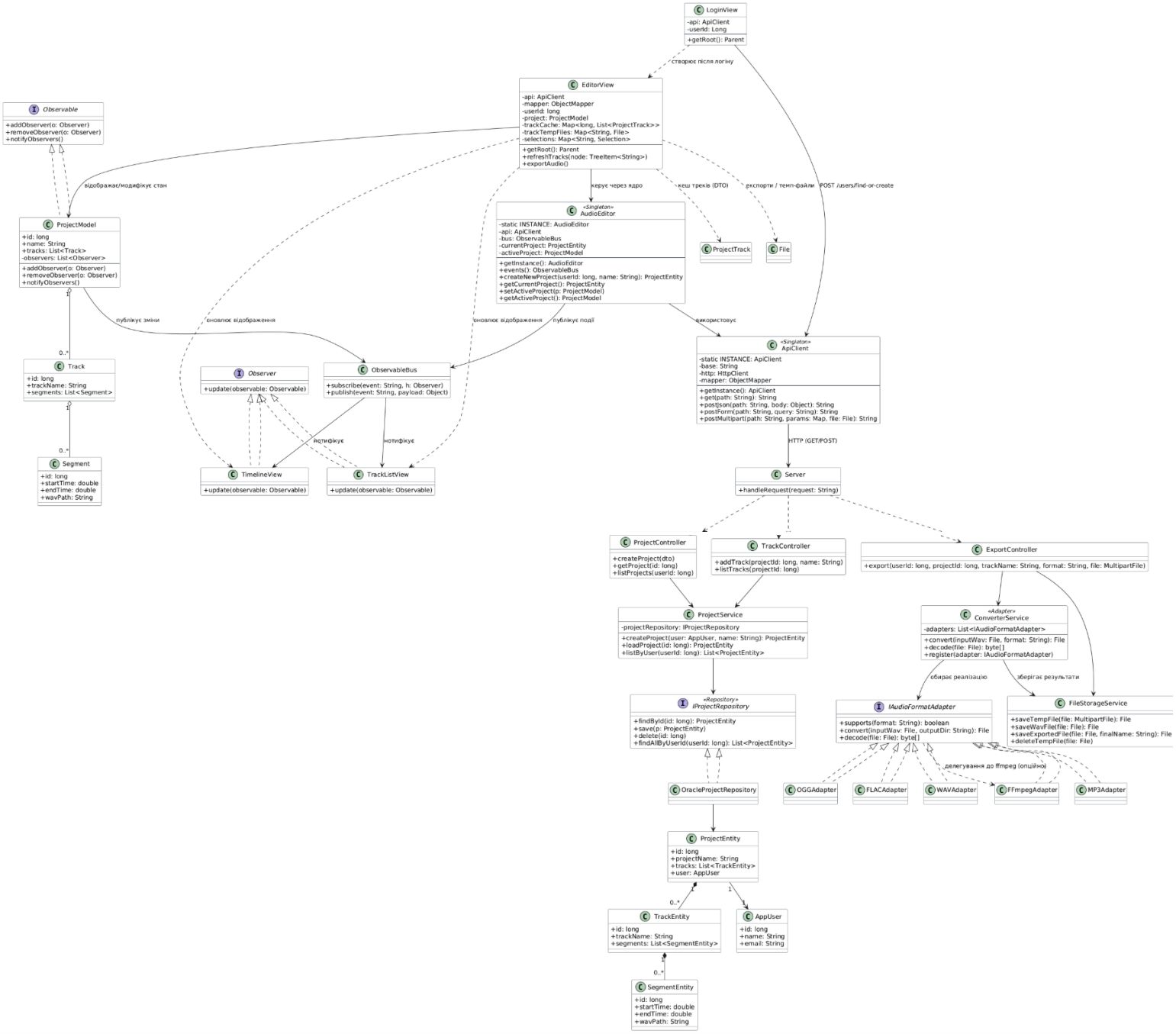



Рисунок 1.1 – Діаграма класів

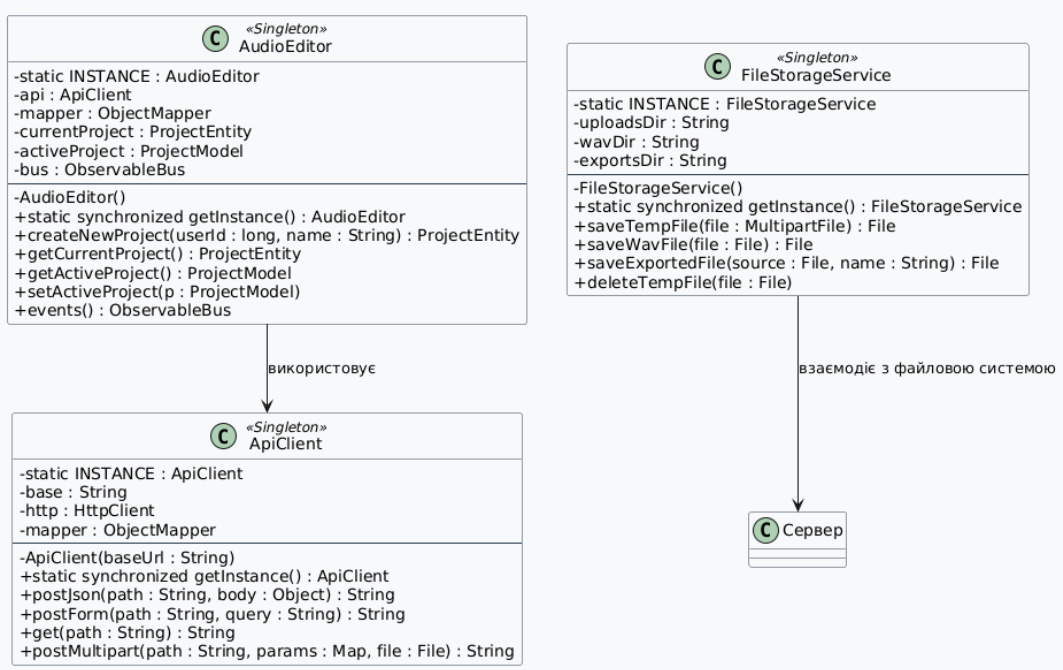



Рисунок 1.2 – Представлення патенту «Одинак» на діаграмі класів

## 2.1. Реалізація шаблону «Singleton» у системі для Audio Editor.

Патерн «Одинак» (Singleton) гарантує, що клас має лише один екземпляр і надає глобальну точку доступу до нього.

У системі Audio Editor він використовується для створення єдиного центру керування клієнтською частиною програми.

У застосунку Audio Editor є центральний об’єкт, який координує:

- створення та збереження проєктів;
- взаємодію з сервером через ApiClient;
- передачу подій між компонентами через ObservableBus;
- роботу з поточним активним проєктом.

Якщо б кожен компонент програми створював власний екземпляр AudioEditor, це призвело б до розсинхронізації стану системи — у кожного з’явився б свій «редактор», який зберігає різні проєкти, події та підключення.

Проблема:

У системі не можна дозволити існування кількох незалежних екземплярів класу AudioEditor, оскільки це призведе до дублювання даних, конфліктів між сеансами роботи з проєктами та втрати цілісності даних користувача.

Рішення:

Щоб уникнути таких проблем, у системі реалізовано патерн «Одинак» (Singleton). Він гарантує, що об’єкт AudioEditor створюється лише один раз за весь час роботи програми, а всі частини застосунку працюють саме з ним через статичний метод доступу getInstance().

Фрегмент коду реалізації шаблону «Одинак»:

```
public class AudioEditor {
   private static AudioEditor INSTANCE;
   private AudioEditor() {}
   public static synchronized AudioEditor getInstance() {
      if (INSTANCE == null) {
         INSTANCE = new AudioEditor();
      }
      return INSTANCE;
   }
}
```

## 2.2. Реалізація шаблону «Singleton» у системі для ApiClient.

У цьому проєкті такий підхід використовується для мережевого клієнта ApiClient, який відповідає за взаємодію з сервером через HTTP-запити.

Проблема:

Без цього шаблону кожен клас (LoginView, AudioEditor) створював би власний екземпляр ApiClient. Це могло призвести до розсинхронізації стану (наприклад,

різних токенів авторизації або помилок з’єднання) і збільшувало споживання пам’яті, бо кожен екземпляр мав власні об’єкти HttpClient та ObjectMapper.

Рішення:

Для забезпечення стабільності мережевих з’єднань і централізованого контролю було впроваджено шаблон Singleton у клас ApiClient.

Тепер у програмі існує лише один екземпляр клієнта, який створюється при першому виклику методу getInstance() та повторно використовується у всіх частинах застосунку.

Основні фрагменти реалізації:

```
public class ApiClient {

   private static ApiClient INSTANCE;

   private ApiClient(String baseUrl) {
      this.base = baseUrl.endsWith("/") ? baseUrl.substring(0, baseUrl.length() - 1) : baseUrl;
   }

   public static synchronized ApiClient getInstance() {
      if (INSTANCE == null) {
         INSTANCE = new ApiClient("http://localhost:8080/api");
      }
      return INSTANCE;
   }
```

Приклад використання у програмі:

```
ApiClient api = ApiClient.getInstance();
```

### 2.3. Реалізація шаблону «Singleton» у системі для FileStorageService

У серверній частині системи Audio Editor шаблон «Одинак» (Singleton) використовується для класу FileStorageService, який відповідає за всі операції збереження, копіювання, видалення та експорту аудіофайлів. Цей сервіс виконує роль єдиного централізованого файлового менеджера, який контролює роботу з усіма файлами користувачів.

Проблема:

- Якщо б для кожного запиту або користувача створювався окремий екземпляр FileStorageService, це могло б призвести до:
- конфліктів при одночасному записі у спільні директорії (uploads, wav, exports);
- дублювання об’єктів і надмірного споживання пам’яті;
- розсинхронізації шляху збереження або втрати даних при паралельному доступі.

Рішення:

Щоб уникнути таких проблем, реалізовано патерн Singleton. Клас FileStorageService має приватний конструктор і статичний метод getInstance(), який створює лише один екземпляр цього класу за час роботи серверу. Усі інші частини програми (наприклад, сервіси експорту або конвертації файлів) отримують доступ до файлової системи через цей єдиний екземпляр. Таким чином, забезпечується централізоване та безпечне керування файлами.

Фрагмент коду реалізації шаблону «Одинак»:

```
@Service
public class FileStorageService {
   private static FileStorageService INSTANCE;

   @Value("${storage.uploadsDir}")
   private String uploadsDir;
   @Value("${storage.wavDir}")
   private String wavDir;
   @Value("${storage.exportsDir}")
   private String exportsDir;

   private FileStorageService() { }

   public static synchronized FileStorageService getInstance() {
      if (INSTANCE == null) {
         INSTANCE = new FileStorageService();
      }
      return INSTANCE;
   }

   public File saveTempFile(MultipartFile file) throws IOException { ... }
   public File saveWavFile(File wavFile) throws IOException { ... }
   public File saveExportedFile(File source, String finalName) throws IOException { ... }
```

```
    public void deleteTempFile(File file) { ... }
}
```

Приклад використання у програмі:

```
FileStorageService storage = FileStorageService.getInstance();
File savedFile = storage.saveTempFile(uploadedFile);
```

**Відповіді на контрольні запитання:**

1. Що таке шаблон проєктування?

*Патерн проєктування — це узагальнене, перевірене часом архітектурне рішення типової задачі, що часто виникає під час проєктування програмних систем. Він описує структуру класів, об’єктів і зв’язків між ними, не прив’язуючись до конкретної мови програмування.*

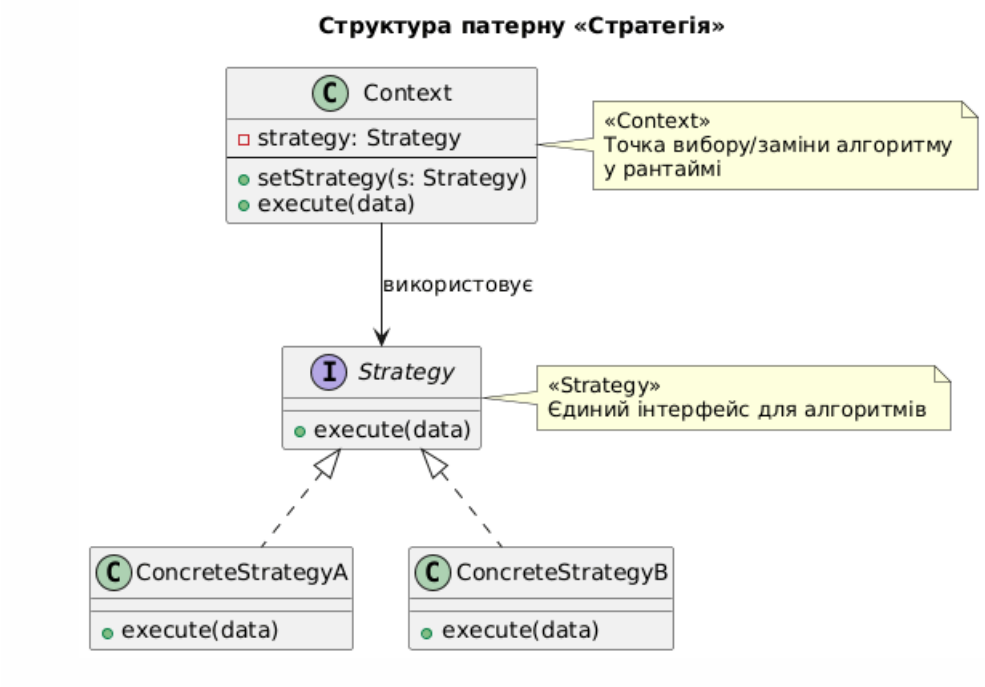
2. Навіщо використовувати шаблони проєктування?

*Призначення патернів — підвищити гнучкість, масштабованість і повторне використання коду. Вони дозволяють стандартизувати рішення, зробити архітектуру більш зрозумілою та зменшити кількість помилок під час розробки складних систем.*

3. Яке призначення шаблону «Стратегія»?

*Шаблон «Strategy» (Стратегія) дозволяє змінювати деякий алгоритм поведінки об'єкта іншим алгоритмом, що досягає ту ж мету іншим способом. Прикладом можуть служити алгоритми сортування: кожен алгоритм має власну реалізацію і визначений в окремому класі; вони можуть бути взаємозамінними в об'єкті, який їх використовує.*

4. Нарисуйте структуру шаблону «Стратегія».




5. Які класи входять в шаблон «Стратегія», та яка між ними взаємодія?

*До шаблону «Стратегія» (Strategy) входять такі класи:*

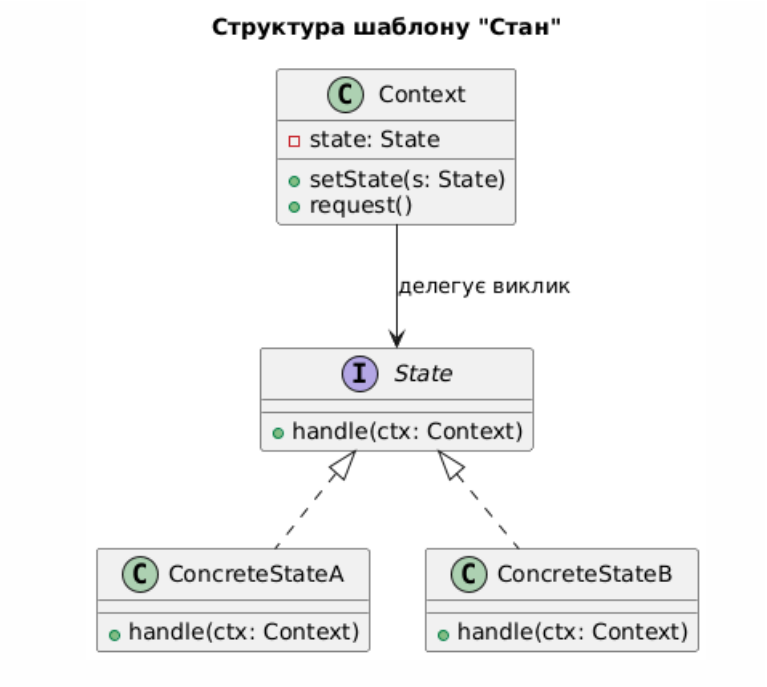
- *Context - зберігає посилання на об'єкт інтерфейсу Strategy і викликає його методи.*
- *Strategy - інтерфейс, який визначає спільну поведінку для всіх стратегій.*
- *ConcreteStrategyA / ConcreteStrategyB - конкретні реалізації алгоритмів.*

*Об'єкт Context отримує конкретну стратегію через метод setStrategy() і викликає її метод execute(). Завдяки цьому алгоритм можна змінювати «на льоту» без зміни коду контексту.*

6. Яке призначення шаблону «Стан»?

*Шаблон «Стан» (State) дозволяє об’єкту змінювати свою поведінку залежно від внутрішнього стану. Зовні здається, що об’єкт змінює свій клас. Він усуває великі if або switch блоки, що перевіряють стан.*

7. Нарисуйте структуру шаблону «Стан».



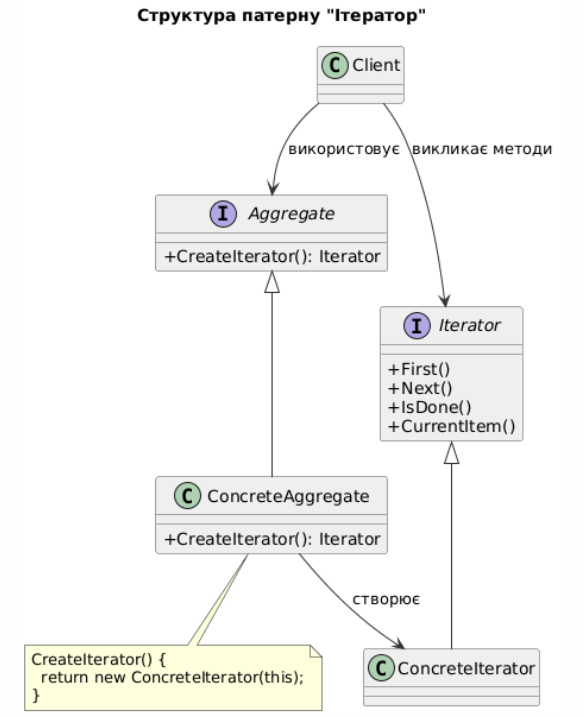

8. Які класи входять в шаблон «Стан», та яка між ними взаємодія?

*Пов'язані зі станом поля, властивості, методи і дії виносяться в окремий загальний інтерфейс (State); кожен стан являє собою окремий клас (ConcreteStateA, ConcreteStateB), які реалізують загальний інтерфейс. Об’єкти, що мають стан (Context), при зміні стану просто записують новий об'єкт в поле state, що призводить до повної зміни поведінки об'єкта.*

9. Яке призначення шаблону «Ітератор»?

*«Iterator» (Ітератор) являє собою шаблон реалізації об'єкта доступу до набору (колекції, агрегату) елементів без розкриття внутрішніх механізмів реалізації. Ітератор виносить функціональність перебору колекції елементів з самої колекції, таким чином досягається розподіл обов'язків: колекція відповідає за зберігання даних, ітератор – за прохід по колекції.*

10.Нарисуйте структуру шаблону «Ітератор».




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

- *Iterator - інтерфейс, який визначає методи навігації*
  - *(next() – установка покажчика перебору на наступний елемент колекції;*
  - *isDone() – булевське поле, яке встановлюється як true коли покажчик*
  - *перебору досяг кінця колекції;*
  - *First() – установка покажчика перебору на перший елемент колекції;*
  - *CurrentItem – поточний об'єкт колекції.).*

- *ConcreteIterator - реалізує інтерфейс Iterator, відстежуючи поточну позицію в колекції.*
- *Aggregate - інтерфейс, що створює ітератор.*
- *ConcreteAggregate - конкретна колекція, яка повертає об’єкт ConcreteIterator.*

*Клієнт отримує Iterator від Aggregate і використовує його методи для послідовного доступу до елементів колекції.*

12.В чому полягає ідея шаблона «Одинак»?

*«Singleton» (Одинак) являє собою клас в термінах ООП, який може мати не більше одного об'єкта (звідси і назва «одинак»). Насправді, кількість об'єктів можна задати (тобто не можна створити більш п об'єктів даного класу). Даний об'єкт найчастіше зберігається як статичне поле в самому класі.*

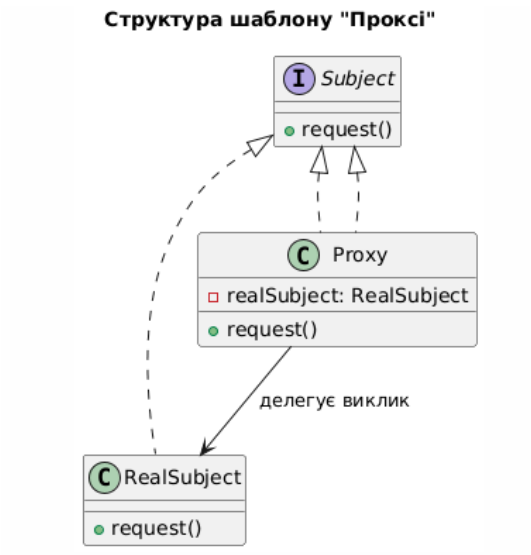
13.Чому шаблон «Одинак» вважають «анти-шаблоном»?

*Тому що він створює глобальний стан у програмі. Це ускладнює тестування, знижує гнучкість і призводить до прихованих залежностей між класами. Зміна єдиного екземпляра може непередбачувано вплинути на інші частини програми.*

14.Яке призначення шаблону «Проксі»?

*«Proxy» (Проксі) – об'єкти є об'єктами-заглушками або двійниками/замінниками для об'єктів конкретного типу. Зазвичай, проксі об'єкти вносять додатковий функціонал або спрощують взаємодію з реальними об'єктами.*

15.Нарисуйте структуру шаблону «Проксі».




16.Які класи входять в шаблон «Проксі», та яка між ними взаємодія?

- *Subject - спільний інтерфейс для справжнього об'єкта та проксі.*
- *RealSubject - реальний об'єкт, до якого здійснюється доступ.*
- *Proxy - обгортка, що контролює доступ до RealSubject.*

*Клієнт викликає метод у Proxy, який вирішує — виконати дію самостійно або делегувати її RealSubject. Це дозволяє додавати перевірку доступу, кешування чи логування без зміни логіки RealSubject.*

**Висновки:**

У ході виконання лабораторної роботи було вивчено та реалізовано основні шаблони проєктування, зокрема Singleton, Iterator, Proxy, State і Strategy. Їх використання в системі Audio Editor дозволило створити структуровану, гнучку та масштабовану архітектуру, підвищити якість коду та забезпечити ефективну взаємодію між компонентами програми.