Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Технології розроблення програмного забезпечення

Лабораторна робота №8
«ШАБЛОНИ «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR».»

Виконала:
Студентка групи ІА-22
Микитенко Ірина

Перевірив:
Мягкий Михайло Юрійович

Київ 2024

Зміст

Тема: шаблони «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR».

Мета: ознайомитися з шаблонами проектування «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR» та набути практичних навичок їх застосування. Реалізувати частину функціоналу програми за допомогою одного з розглянутих шаблонів для досягнення конкретних функціональних можливостей та забезпечення ефективної взаємодії між класами.

Хід роботи

..1 Музичний програвач (iterator, command, memento, facade, visitor, client-server)

Музичний програвач становить собою програму для програвання музичних файлів або відтворення потокової музики з можливістю створення, запам'ятовування і редагування списків програвання, перемішування/повторення (shuffle/repeat), розпізнавання різних аудіо-форматів, еквалайзер.

1. Короткі теоритичні відомості

Шаблони роботи з БД при розробці корпоративних додатків: Існує набір сучасних шаблонів, які використовуються для розробників широко-масштабних корпоративних додатків - з кількістю користувачів понад 100 і великою кількістю взаємопов'язаних програм.

Такі додатки вимагають окремих підходів до організації доступу до даних, оскільки кількість даних, що зберігаються, класів і клієнтів, що звертаються, зростає.

Шаблон «Active Record»:
Поєднує дані й логіку в одному об'єкті, що представляє рядок БД. Широко використовується через простоту, але складність класу зростає зі збільшенням запитів, тому логіку часто виносять окремо.

Шаблон «Table Data Gateway»:
Окремий клас для взаємодії з БД для кожного класу даних. Вся логіка запитів зосереджена в ньому, що підвищує гнучкість і тестованість. Часто абстрагується в базовий клас для зменшення дублювання.

Шаблон «Data Mapping»:
Перетворює об'єкти даних у формат, що приймається БД чи мережею. Маппер пов'язує властивості об'єкта з колонками таблиці, вирішуючи невідповідності типів.

Шаблон «Composite»:

Дозволяє об’єднувати об’єкти в деревоподібну структуру для подання ієрархій типу «частина-ціле» і обробляти їх уніфіковано.

Структура шаблону «Composite»:

1. Компонент (Component)
    - Загальний інтерфейс для всіх об’єктів у структурі (наприклад, метод getPrice()).
    - Може мати стандартну реалізацію деяких методів.
2. Лист (Leaf)
    - Конкретний об’єкт без вкладених елементів (наприклад, Продукт).
    - Реалізує методи інтерфейсу (наприклад, повертає свою ціну).
3. Контейнер (Composite)
    - Об’єкт, що містить інші компоненти (наприклад, Коробка).
    - Зберігає колекцію дочірніх елементів і викликає їхні методи.
4. Клієнт (Client)
    - Працює з компонентами через загальний інтерфейс, не залежачи від їхньої конкретної реалізації.

Використання:

- Використовується, коли модель програми можна представити у вигляді дерева.
- Продукт і Коробка мають спільний інтерфейс із методом для отримання вартості.
- Коробка підсумовує ціни продуктів і вкладених коробок.

Переваги:

- Спрощує роботу з деревоподібними структурами.
- Полегшує додавання нових компонентів.

Недоліки:

- Занадто загальний дизайн класів.

Шаблон «Flyweight»

Призначення:

Шаблон використовується для оптимізації пам'яті в програмах з великою кількістю схожих об'єктів шляхом поділу загальних (внутрішніх) даних між цими об'єктами. Унікальні (зовнішні) дані зберігаються окремо в контексті.

Проблема:

Коли додаток створює багато схожих об'єктів, це може призвести до надмірного використання оперативної пам’яті. Наприклад, в грі тисячі частинок з однаковим кольором і спрайтом займають багато місця через дублювання цих даних.

Рішення:

Розділити об'єкти на:

1. **Внутрішні (загальні) дані:** спільні для всіх екземплярів (наприклад, спрайт, колір).
2. **Зовнішні (унікальні) дані:** специфічні для кожного екземпляра (наприклад, координати, швидкість).

Об'єкти зберігають посилання на загальні дані, а унікальні — отримують із контексту.

Приклад:

1. У грі частинки мають спільні властивості (спрайт, колір) у класі ParticleType, тоді як унікальні дані (позиція, швидкість) залишаються в контексті.
2. У базі ветеринарної клініки для котів:
    - Загальні дані (порода, колір) зберігаються у класі CatBreed.
    - Унікальні дані (ім’я, вік, власник) — у класі Cat.

Структура:

1. **Flyweight (Легковаговик):**
   Оголошує інтерфейс для спільних даних.
2. **ConcreteFlyweight (Конкретний Легковаговик):**
   Реалізує внутрішній стан і забезпечує доступ до спільних даних.
3. **FlyweightFactory (Фабрика Легковаговиків):**
   Відповідає за створення і управління спільними об'єктами.
   Перевіряє, чи існує потрібний об'єкт, і створює його за необхідності.
4. **Client (Клієнт):**
   Використовує об'єкти Flyweight, додаючи до них зовнішні дані.

Переваги:

- Зменшує використання оперативної пам’яті завдяки спільним об'єктам.

Недоліки:

- Підвищує складність коду через введення додаткових класів.
- Витрачає більше процесорного часу на обробку зовнішніх даних.

Шаблон «Interpreter»

Призначення:

Шаблон використовується для подання граматики та інтерпретатора вибраної мови. Граматика описується термінальними та нетермінальними символами, кожен з яких інтерпретується в заданому контексті.
Клієнт передає сформовану пропозицію в термінах абстрактного синтаксичного дерева, де кожен вузол виразу інтерпретується рекурсивно.

Проблема:

Часто виникає потреба реалізувати задачу, яка потребує регулярного оновлення чи складного розбору текстів або виразів, наприклад, пошук рядків за шаблоном.

Рішення:

Створити інтерпретатор для заданої мови, який складається з:

- **Абстрактного виразу:** Базовий клас, що оголошує метод інтерпретації.
- **Термінальних виразів:** Описують кінцеві символи граматики.
- **Нетермінальних виразів:** Описують правила граматики, які можуть включати інші термінальні та нетермінальні вирази.
- **Контексту:** Зберігає глобальну інформацію для інтерпретації.

Клієнт будує абстрактне синтаксичне дерево, що складається з термінальних і нетермінальних вузлів. Дерево обчислює результат за допомогою рекурсивного виклику операції Розібрати().

Приклад:

Реалізація автоматичного розбору поставлених завдань із розбивкою на прості інструкції, зрозумілі для системи.

Структура:

1. **AbstractExpression (Абстрактний Вираз):**
   Оголошує метод Інтерпретувати(Context).
2. **TerminalExpression (Термінальний Вираз):**
   Реалізує метод для інтерпретації термінального символу граматики.

3. **NonterminalExpression (Нетермінальний Вираз):**
   Реалізує метод для інтерпретації нетермінального символу, що може включати інші вирази.
4. **Context (Контекст):**
   Зберігає дані для інтерпретації виразів (глобальна інформація).
5. **Client (Клієнт):**
   Створює синтаксичне дерево та ініціює інтерпретацію.

Переваги:

- Легко змінювати та розширювати граматику.
- Простота додавання нових способів обчислення виразів.

Недоліки:

- Ускладнення підтримки для граматики з великою кількістю правил.

Шаблон «Visitor»

Призначення:

Шаблон дозволяє визначати нові операції для класів ієрархії без зміни їх структури. Використовується для групування однотипних операцій, що застосовуються до різнотипних об'єктів, зберігаючи відкритість для розширення.

Переваги:

- Легке додавання нових операцій.
- Локалізація операцій у відвідувачах.

Недоліки:

- Складність додавання нових елементів у ієрархію (потрібно оновлювати всіх відвідувачів).
- Збільшення кількості методів у відвідувачах для підтримки нових елементів.

Проблема:

Необхідно реалізувати експорт даних графа з вузлами різного типу (міста, пам’ятки тощо) у формат XML.
Однак класи вузлів є стабільними і не можуть бути змінені. Це унеможливлює додавання в них нових методів для виконання операцій.

Рішення:

Виділити нову поведінку в окремі класи-відвідувачі. Об’єкти вузлів передаються до методів відвідувача, який виконує необхідні дії залежно від типу вузла.

Структура:

1. **Visitor (Відвідувач):**
   Інтерфейс або базовий клас, що декларує методи для відвідування кожного типу елементів.
2. **ConcreteVisitor (Конкретний Відвідувач):**
   Реалізує методи для виконання операцій над конкретними типами елементів.
3. **Element (Елемент):**
   Інтерфейс або базовий клас, що визначає метод accept(Visitor).
4. **ConcreteElement (Конкретний Елемент):**
   Реалізує метод accept(), викликаючи відповідний метод відвідувача.
5. **ObjectStructure (Структура Об'єктів):**
   Контейнер для елементів, який дозволяє зручно перебирати їх та передавати відвідувачу.

Приклад з життя:

У компіляторі є об'єкти різних синтаксичних конструкцій (виклики методів, умовні вирази тощо). Для кожного з них реалізовано:

1. **Відвідувача перевірки типів.**
2. **Відвідувача генерації коду.**
   Нові етапи компіляції додаються шляхом створення додаткових відвідувачів.

Переваги:

- Легкість у додаванні нових операцій (наприклад, експорту в JSON).
- Зручність групування поведінки за різними відвідувачами.

Недоліки:

- Додавання нових типів вузлів вимагає змін у всіх існуючих відвідува

2. Реалізація не менше 3-х класів відповідно до обраної теми.

Структура проекту з реалізованими класами зображена на рисунку 1.

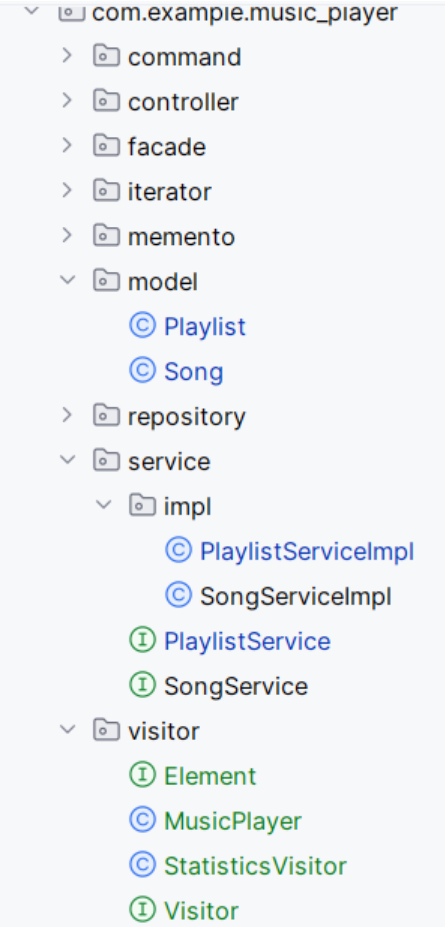

Рисунок 1 – структура проекту

На основі діаграми класів, наданої у зображенні, система музичного плеєра має наступну архітектуру, яка ілюструє використання патерна *Visitor* для роботи з елементами системи, такими як Song та Playlist.

1. Інтерфейс Visitor

- **Visitor**
  Інтерфейс, який визначає методи для відвідування об'єктів типів Song та Playlist.
  Методи:
  - visit(Song) — для обробки об'єктів класу Song.
  - visit(Playlist) — для обробки об'єктів класу Playlist.

2. Конкретна реалізація Visitor

- **StatisticsVisitor**
  Реалізує інтерфейс Visitor для збору статистики:
  - totalSongs — змінна для зберігання загальної кількості пісень.

- totalDuration — змінна для обчислення загальної тривалості всіх пісень.
  Методи:
- visit(Song) — збільшує тривалість (totalDuration) та кількість пісень (totalSongs) для кожного елемента Song.
- visit(Playlist) — обробляє плейлисти, викликаючи методи для їх пісень.

3. Інтерфейс Element

- **Element**
  Інтерфейс, який реалізують класи Song та Playlist.
  Метод:
  - accept(Visitor) — забезпечує можливість приймати об'єкт Visitor, дозволяючи виконувати операції над поточним елементом.

4. Елементи системи

- **Song**
  Представляє пісню із такими атрибутами:
  - title — назва пісні.
  - artist — виконавець.
  - duration — тривалість пісні у секундах.
  - fileData — бінарні дані файлу пісні.
    Реалізує метод accept(Visitor), що передає об'єкт Visitor для виконання над ним операцій.
- **Playlist**
  Представляє плейлист, який містить список пісень (songs).
  Атрибути:
  - name — назва плейлиста.
  - songs — список пісень, що входять до плейлиста.
    Методи:
  - accept(Visitor) — викликає метод відвідувача для роботи з плейлистом.
  - Додатково містить методи для управління піснями у плейлисті.

5. MusicPlayer

- **MusicPlayer**
  Клас, який агрегує список елементів (List<Element>), таких як пісні або

плейлисти, та надає метод acceptVisitor(Visitor), що викликає методи візитора для кожного елемента у списку.

6. Сервіс для плейлистів

- **PlaylistServiceImpl**
  Клас, який надає функціонал для управління плейлистами:
  - addSongToPlaylist() — додає пісню до плейлиста.
  - removeSongFromPlaylist() — видаляє пісню із плейлиста.
  - calculateStatistics() — використовує *Visitor* для збору статистики по плейлистах.
  - savePlaylistState() і restorePlaylistState() — реалізують збереження та відновлення стану плейлистів (імплементація патерна *Memento*).

7. Зв'язки між класами

- **MusicPlayer** асоціюється із списком елементів (Element) через агрегацію.
- **Visitor** зв'язаний із класами Song та Playlist, дозволяючи виконувати над ними операції через методи visit().
- Класи Song та Playlist реалізують інтерфейс Element, що забезпечує підтримку відвідувача.
- **StatisticsVisitor** є конкретною реалізацією Visitor, що додає функціонал для збору статистики.

3. Реалізація шаблону за обраною темою

У ході виконання лабораторної роботи ми застосували патерн **Visitor** для обробки об'єктів музичного плеєра, таких як пісні та плейлисти. Патерн **Visitor** дозволяє додавати нові операції до існуючих об'єктів без зміни їх класів. Це зручно для виконання різних обчислень чи операцій, таких як підрахунок статистики чи зміна властивостей, без необхідності змінювати самі об'єкти.

Кроки реалізації патерну:

1. **Інтерфейс Element:**

• Призначення: Це інтерфейс, який визначає метод accept, який дозволяє візитору взаємодіяти з об'єктами цього типу. Клас Song та Playlist реалізують цей інтерфейс, що дає можливість відправляти їх до візитора для обробки.

Даний інтерфейс представлений на рисунку 2:

```
public interface Element {  9 usages  2 implementations  new *
    void accept(Visitor visitor);  2 usages  2 implementations  new *
}
```

Рисунок 2 - Інтерфейс Element

2. **Клас Song:**

• Призначення: Клас представляє пісню в системі. Він містить атрибути, такі як назва пісні, виконавець, тривалість і формат.

• Роль в патерні: Реалізує інтерфейс Element, дозволяючи візитору взаємодіяти з об'єктом пісні, наприклад, для підрахунку статистики.

Даний клас представлений на рисунку 3:

```
@Override  2 usages  new *
public void accept(Visitor visitor) {
    visitor.visit( song: this);
}
```

Рисунок 3 – клас Song

3. **Клас Playlist:**

• Призначення: Клас, що представляє плейлист пісень. Має атрибути для зберігання назви плейлиста та списку пісень. Плейлист реалізує інтерфейс Element, що дозволяє передавати його в візитора для виконання операцій.

• Роль в патерні: Також реалізує метод accept, щоб взаємодіяти з візиторами, такими як StatisticsVisitor.

Даний клас представлений на рисунку 4:

```
@Override  2 usages  new *
public void accept(Visitor visitor) {
    visitor.visit( playlist: this);
    for (Song song : songs) {
        song.accept(visitor);
    }
}
```

Рисунок 4 - Клас Playlist

4.**Інтерфейс Visitor**:

• Призначення: Це інтерфейс, який визначає методи для обробки різних типів елементів (в даному випадку пісень і плейлистів). Клас StatisticsVisitor є конкретною реалізацією цього інтерфейсу.

• Методи:

- visit(Song song): здійснює операцію на пісні (наприклад, додає її до загальної статистики).
- visit(Playlist playlist): здійснює операцію на плейлисті, обробляючи всі пісні в ньому.

Даний інтерфейс представлений на рисунку 5:

```
public interface Visitor {  7 usages  1 implementation  new *
    void visit(Song song);  2 usages  1 implementation  new *
    void visit(Playlist playlist);  1 usage  1 implementation  new *
}
```

Рисунок 5 - Інтерфейс Visitor

5. **Клас StatisticsVisitor:**

• Призначення: Цей клас є реалізацією патерну Візитор і призначений для збору статистики щодо пісень та плейлистів.

• Атрибути: Містить змінні для збереження загальної кількості пісень та їхньої тривалості.

• Методи:

- visit(Song song): підраховує кількість пісень та додає їх тривалість.
- visit(Playlist playlist): викликає visit(Song) для кожної пісні в плейлисті, щоб додати її статистику.

Даний клас представлений на рисунку 6:

```java
@Getter  3 usages  new *
public class StatisticsVisitor implements Visitor {
    private int totalSongs = 0;
    private long totalDuration = 0;

    @Override  2 usages  new *
    public void visit(Song song) {
        totalSongs++;
        totalDuration += (long) song.getDuration();
    }

    @Override  1 usage  new *
    public void visit(Playlist playlist) {
        for (Song song : playlist.getSongs()) {
            visit(song);
        }
    }
}
```

Рисунок 6 - Клас StatisticsVisitor

6. **Клас MusicPlayer:**

• Призначення: Це клас, який зберігає список елементів (пісень або плейлистів) і передає їх в візитора для обробки. Він надає метод для прийому візитора і виклику відповідних операцій на елементах.

• Методи:

- acceptVisitor(Visitor visitor): проходимо по всіх елементах (наприклад, плейлисти або пісні) та передаємо їх в візитора для обробки.

Даний клас представлений на рисунку 7:

```
@Getter  3 usages  new *
public class StatisticsVisitor implements Visitor {
    private int totalSongs = 0;
    private long totalDuration = 0;

    @Override  2 usages  new *
    public void visit(Song song) {
        totalSongs++;
        totalDuration += (long) song.getDuration();
    }

    @Override  1 usage  new *
    public void visit(Playlist playlist) {
        for (Song song : playlist.getSongs()) {
            visit(song);
        }
    }
}
```

Рисунок 7 - Клас MusicPlayer

7. **Клас PlaylistServiceImpl (Клієнт):**

• Призначення: Клас PlaylistServiceImpl виступає клієнтом у нашій реалізації патерну Візитор. У методі calculateStatistics клієнт створює список елементів (плейлистів) і передає їх до візитора StatisticsVisitor. Візитор обробляє кожен елемент, обчислюючи загальну кількість пісень і тривалість. Це дозволяє розширювати функціонал обчислень без змін у класах об'єктів.

Даний клас представлений на рисунку 8:

```
@Override  1 usage  new *
public void calculateStatistics(Long playlistId) {
    Playlist playlist = getPlaylistById(playlistId);

    if (playlist != null) {
        List<Element> elements = new ArrayList<>();
        elements.add(playlist);
        MusicPlayer musicPlayer = new MusicPlayer(elements);

        musicPlayer.acceptVisitor(statisticsVisitor);
    }
}
```

Рисунок 8 - Клас PlaylistServiceImpl

4. Діаграма класів для паттерну Visitor

Діаграма класів, які реалізовують паттерн Memento зображена на рисунку 9

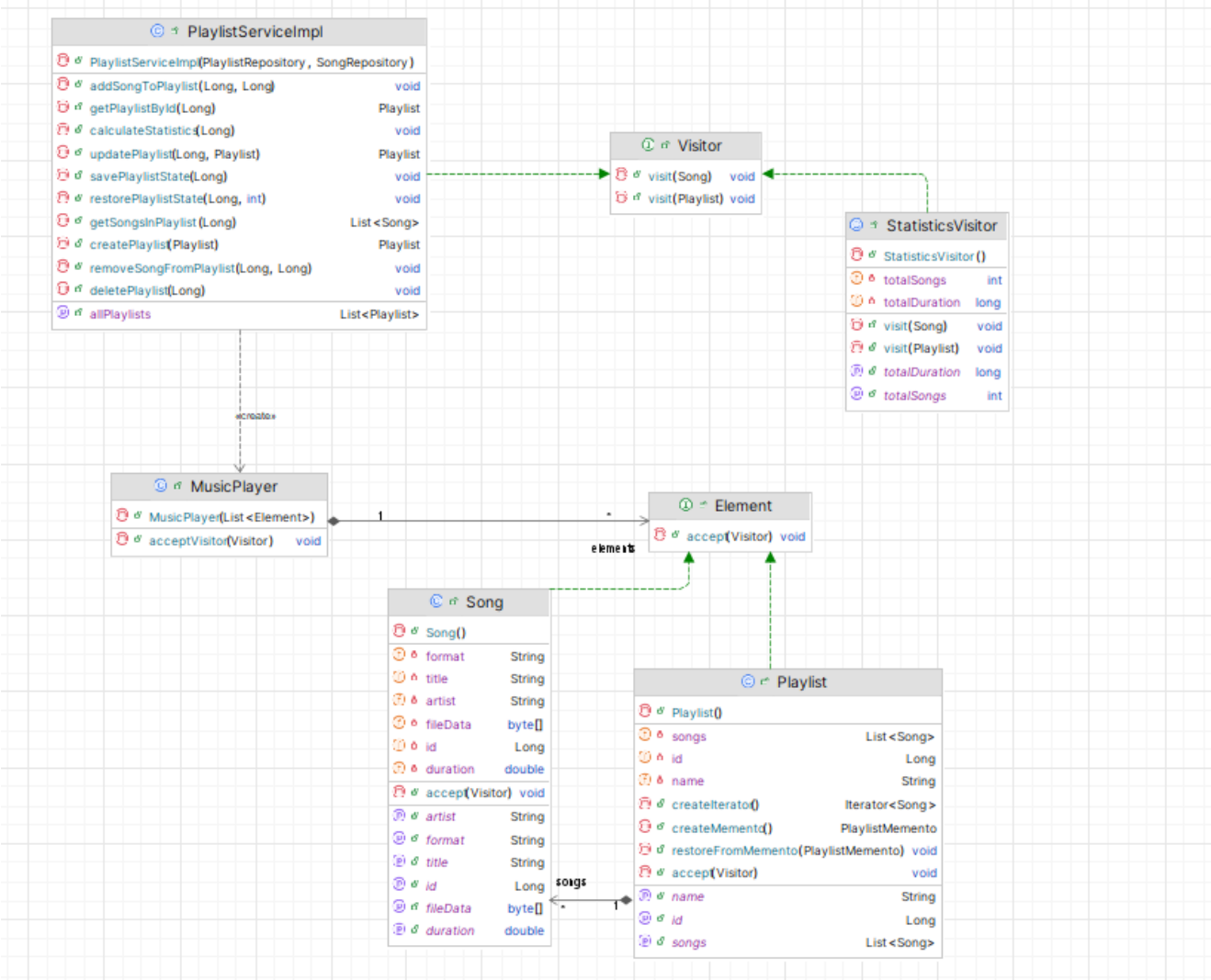

Рисунок 9 – діаграма класів

Проблема, яку допоміг вирішити шаблон Visitor

У нашій системі музичного плеєра виникла проблема при роботі з різними об'єктами, такими як пісні та плейлисти, які мають спільний функціонал для обчислення статистики (кількість пісень, загальна тривалість) та інших операцій. Для обчислення статистики кожен об'єкт потребував окремої реалізації, а додавання нового функціоналу (наприклад, аналізу метаданих) вимагало змін у класах об'єктів. Це ускладнювало підтримку коду та порушувало принцип відкритості/закритості (OCP).

Шаблон Visitor допоміг вирішити цю проблему наступним чином:

- Він виділив операції, які виконуються над об'єктами, у окремі класи (відвідувачі). Це дозволило зосередити функціонал, наприклад, для обчислення статистики, в одному місці.
- Він дозволив додавати нові операції без зміни класів пісень і плейлистів. Це спрощує масштабування системи.
- Visitor забезпечив гнучкість у роботі з різними об'єктами, зберігаючи їх внутрішню структуру незмінною.

Переваги застосування шаблону Visitor

1. **Відокремлення логіки від об'єктів:**
   Шаблон дозволив винести логіку обчислення статистики та інших операцій у окремий клас StatisticsVisitor. Це зменшило складність класів об'єктів (Song, Playlist) і зробило їх код більш зрозумілим.
2. **Легкість додавання нового функціоналу:**
   Завдяки Visitor ми можемо додати нові операції (наприклад, аналіз жанру або визначення тривалості найкоротшої пісні) шляхом створення нового відвідувача, без змін у класах пісень чи плейлистів.
3. **Спрощення підтримки:**
   Код, пов’язаний з операціями над об'єктами, централізований у відвідувачах. Це спрощує підтримку та налагодження системи, оскільки вся логіка зосереджена в одному місці.
4. **Полегшення розширення системи:**
   Нові об'єкти або класи, що реалізують інтерфейс Element, можуть легко взаємодіяти з існуючими відвідувачами. Це підвищує гнучкість і масштабованість системи.
5. **Мінімальні зміни в класах об'єктів:**
   Класи Song і Playlist були доповнені лише методом accept(Visitor visitor), що дозволяє взаємодіяти з відвідувачем, без значних змін у внутрішній логіці цих класів.

Висновок

У даній лабораторній роботі я реалізувала патерн проектування **Visitor** для музичного плеєра, що дозволило виділити операції над піснями та плейлистами в окремі класи-відвідувачі. Це спростило підтримку та масштабування системи, зменшило складність логіки класів об'єктів та забезпечило можливість легко додавати нові операції без зміни основного коду. Реалізація цього патерну зробила систему більш організованою та зручною для подальшого розвитку, підвищивши її ефективність і гнучкість.