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

ЛАБОРАТОРНА РОБОТА №8

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування»

Виконав:

студент групи ІА-33

Грицай Андрій

Перевірив:

асистент кафедри ІСТ

Мягкий Михайло Юрійович

Київ - 2025

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

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.

Теоретичні відомості

Компонувальник (Composite)

Шаблон "Компонувальник" дозволяє працювати з групами об'єктів так само, як з одним об'єктом, організовуючи їх у деревоподібну структуру. Використовується, коли потрібно представити ієрархію частина-ціле і надати єдиний інтерфейс для роботи з окремими об'єктами та їх групами. У Java цей шаблон реалізується шляхом створення базового інтерфейсу або абстрактного класу для всіх об'єктів у структурі та підкласів для представлення як "листків" (окремих об'єктів), так і "гілок" (груп об'єктів). Це дозволяє клієнтському коду працювати з усією структурою без перевірок її деталей.

Легковаговик (Flyweight)

Шаблон "Легковаговик" оптимізує використання пам'яті шляхом спільного використання об'єктів, які поділяють однаковий стан. Використовується, коли система створює велику кількість однотипних об'єктів, що можуть розділяти загальний внутрішній стан. У Java цей шаблон зазвичай реалізується через фабрику, яка управляє пулом спільних об'єктів. Зовнішній стан об'єктів зберігається поза ними, щоб уникнути надмірного споживання пам'яті. Це дозволяє значно знизити накладні витрати в системах із великою кількістю об'єктів.

Інтерпретатор (Interpreter)

Шаблон "Інтерпретатор" визначає спосіб представлення граматики ієрархії мови і надає інтерпретатор для її виконання. Використовується для роботи з мовами, які мають чітко визначену граматику, наприклад, для створення калькуляторів або розбору виразів. У Java цей шаблон реалізується через створення класів, які представляють правила граматики, та методу interpret(), що виконує операції. Це забезпечує зручність у роботі з граматикою, проте може бути неефективним для складних мов через зростання кількості класів.

Відвідувач (Visitor)

Шаблон "Відвідувач" дозволяє визначати нові операції для об'єктів без зміни їхніх класів. Використовується, коли необхідно виконати кілька різних операцій над об'єктами складної структури, але їх класи не можна змінювати. У Java цей шаблон реалізується шляхом створення інтерфейсу Visitor, який має методи для кожного типу елементів, та їх реалізацій для виконання конкретних операцій. Об'єкти структури реалізують метод accept(Visitor visitor), який викликає відповідний метод відвідувача. Це дозволяє додавати нові операції, зберігаючи класи об'єктів незмінними.

Хід роботи

У цій роботі використано Патерн "Відвідувач" (Visitor), обраний як ключове архітектурне рішення для відокремлення алгоритмів обробки даних від структури об'єктів, над якими вони виконуються.

У процесі роботи над проєктом стало зрозуміло, що над об'єктами музичних треків необхідно виконувати різні операції (наприклад, розрахунок загальної тривалості плейлиста, збір статистики, експорт метаданих). Хоча структура об'єкта Track є стабільною, набір операцій над ним може змінюватися або розширюватися.

Без використання патерну "Відвідувач", додавання нової логіки обробки безпосередньо в клас Track призвело б до:

Порушення принципу єдиної відповідальності (SRP): Клас Track був би перевантажений методами бізнес-логіки, не пов'язаними безпосередньо зі зберіганням інформації про трек.

Порушення принципу відкритості/закритості (OCP): Кожна нова операція вимагала б зміни існуючого коду класу Track та його перекомпіляції, що підвищує ризик внесення помилок.

Патерн "Відвідувач" елегантно вирішує цю проблему, дозволяючи додавати нові операції над об'єктами без зміни самих класів цих об'єктів, використовуючи механізм подвійної диспетчеризації.

У моїй реалізації, відображеній на UML-схемі, цей підхід реалізовано наступними класами:

1. DurationVisitor (Конкретний Відвідувач / Concrete Visitor)
    - Логіка операції: Реалізує специфічний алгоритм обходу — підрахунок сумарної тривалості треків. Зберігає проміжний стан (totalDurationInSeconds, trackCount).

    - Метод visitTrack(): Отримує доступ до публічного стану об'єкта Track (його тривалості) та додає його до загальної суми, не змінюючи сам об'єкт.
2. Visitor (Інтерфейс Відвідувача)
    - Контракт: Оголошує метод відвідування visitTrack(Track track), який має бути реалізований усіма конкретними алгоритмами. Це дозволяє легко створювати нові типи відвідувачів (наприклад, ExportVisitor) без змін у структурі треків.
3. Track (Елемент / Concrete Element)
    - Інтерфейс Visitable: Реалізує метод accept(Visitor visitor).
    - Делегування: Метод accept не виконує логіку самостійно, а викликає відповідний метод відвідувача (visitor.visitTrack(this)), передаючи посилання на себе. Це дозволяє відвідувачу отримати доступ до даних треку.
4. MusicPlayerController (Клієнтський код)
    - Оркестрація: Створює екземпляр DurationVisitor та ініціює процес обходу списку треків у плейлисті. Для кожного треку викликається метод accept. Після завершення обходу клієнт отримує готовий результат (відформатовану тривалість) безпосередньо з об'єкта DurationVisitor.

Діаграма класів

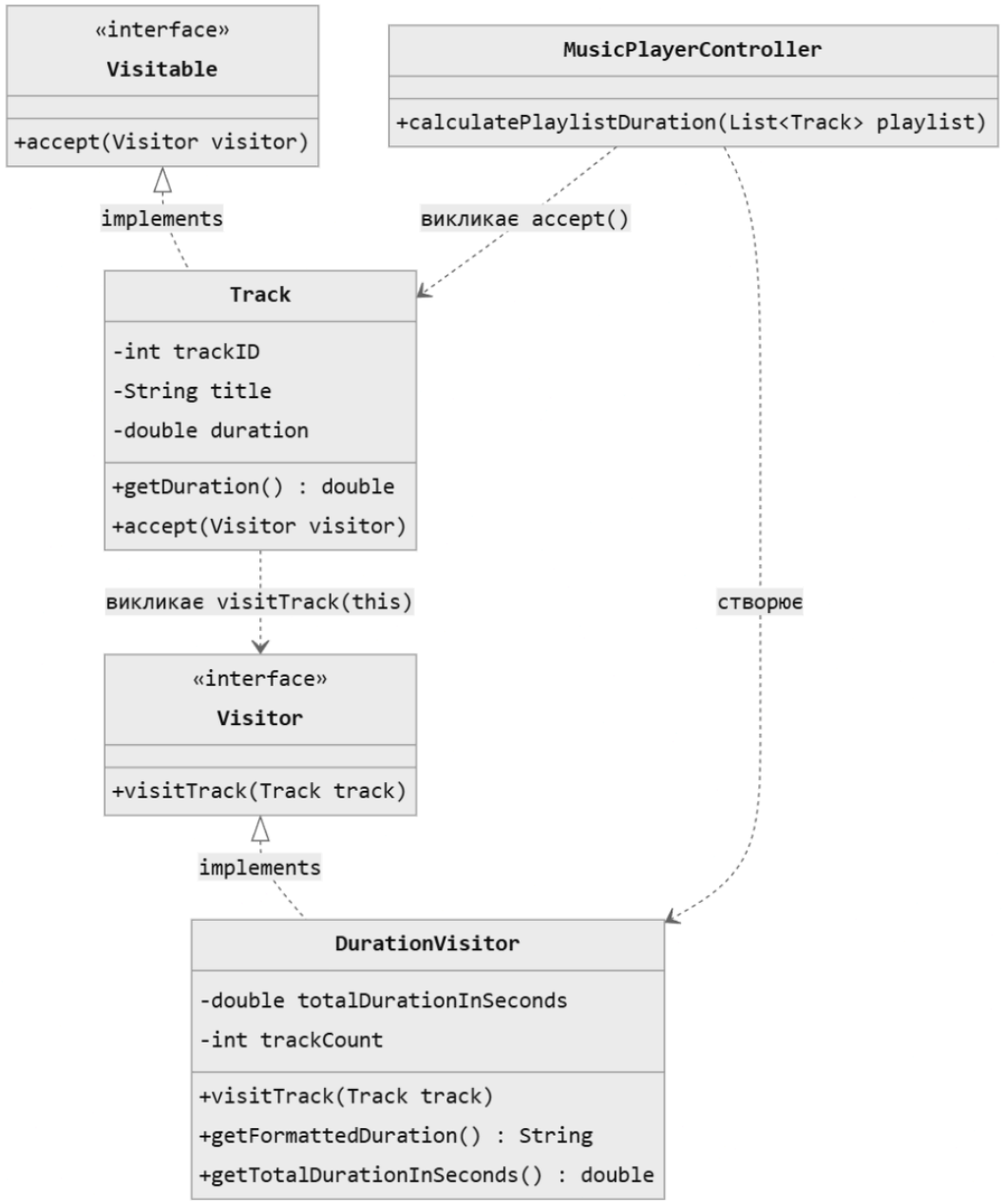


Рисунок 1 – Діаграма класів, яка представляє використання патерну Visitor

Код класів

Visitor:

```
public interface Visitor {

    void visitTrack(Track track);

}
```

Visitable:

```
public interface Visitable {

    void accept(Visitor visitor);
```

```
}
```

DurationVisitor:

```
public class DurationVisitor implements Visitor {

    private double totalDurationInSeconds = 0;

    private int trackCount = 0;


    @Override

    public void visitTrack(Track track) {

        if (track != null) {

            totalDurationInSeconds += track.getDuration();

            trackCount++;

        }

    }


    public double getTotalDurationInSeconds() {

        return totalDurationInSeconds;

    }


    public String getFormattedDuration() {

        int hours = (int) (totalDurationInSeconds / 3600);

        int remainder = (int) (totalDurationInSeconds % 3600);

        int minutes = remainder / 60;

        int seconds = remainder % 60;


        if (hours > 0) {

            return String.format("%d год %d хв %d сек", hours, minutes, seconds);

        } else {

            return String.format("%d хв %d сек", minutes, seconds);

        }

    }
```

```
    public int getTrackCount() {
        return trackCount;
    }
}
```

Track:

```
public class Track implements Visitable, Serializable {
    private int trackID;
    private String title;
    private String artist;
    private String album;
    private double duration;
    private String filePath;

    public Track() {
    }

    public Track(int trackID, String title, String artist, String album, double
duration, String filePath) {
        this.trackID = trackID;
        this.title = title;
        this.artist = artist;
        this.album = album;
        this.duration = duration;
        this.filePath = filePath;
    }

    public int getTrackID() { return trackID; }
    public void setTrackID(int trackID) { this.trackID = trackID; }

    public String getTitle() { return title; }
```

```
public void setTitle(String title) { this.title = title; }

public String getArtist() { return artist; }
public void setArtist(String artist) { this.artist = artist; }

public String getAlbum() { return album; }
public void setAlbum(String album) { this.album = album; }

public double getDuration() { return duration; }
public void setDuration(double duration) { this.duration = duration; }

public String getFilePath() { return filePath; }
public void setFilePath(String filePath) { this.filePath = filePath; }

@Override
public void accept(Visitor visitor) {
    visitor.visitTrack(this);
}

@Override
public String toString() {
    return title;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Track track = (Track) o;
    return trackID == track.trackID;
```

```
    }


    @Override
    public int hashCode() {
        return Objects.hash(trackID);
    }
}
```

MusicPlayerController (Використання):

```
view.setDurationCalculator(playlist -> {
    DurationVisitor visitor = new DurationVisitor();
    List<Integer> trackIds = playlist.getTracks();


    for (Integer id: trackIds) {
        Track track = systemFacade.getTrackService().getTrackByID(id);
        if (track != null) {
            track.accept(visitor);
        }
    }
    return visitor.getFormattedDuration();
});
```

Висновок: Виконуючи цю лабораторну роботу, я ознайомився з такими патернами, як «Composite», «Flyweight», «Interpreter» та «Visitor». Особливу увагу я приділив реалізації шаблону "Відвідувач" (Visitor), детально описавши його основну логіку та функціональність у контексті мого проєкту музичного програвача. Цей патерн було обрано як архітектурне рішення для відокремлення алгоритмів обробки даних від самих об'єктів, щоб уникнути перевантаження класу треку зайвою логікою (наприклад, розрахунком загальної тривалості плейлиста).

Під час реалізації шаблону "Відвідувач" я зрозумів, наскільки цей патерн елегантно вирішує проблему розширюваності системи при роботі зі структурами об'єктів. Він дозволяє додавати нові операції над класами, не

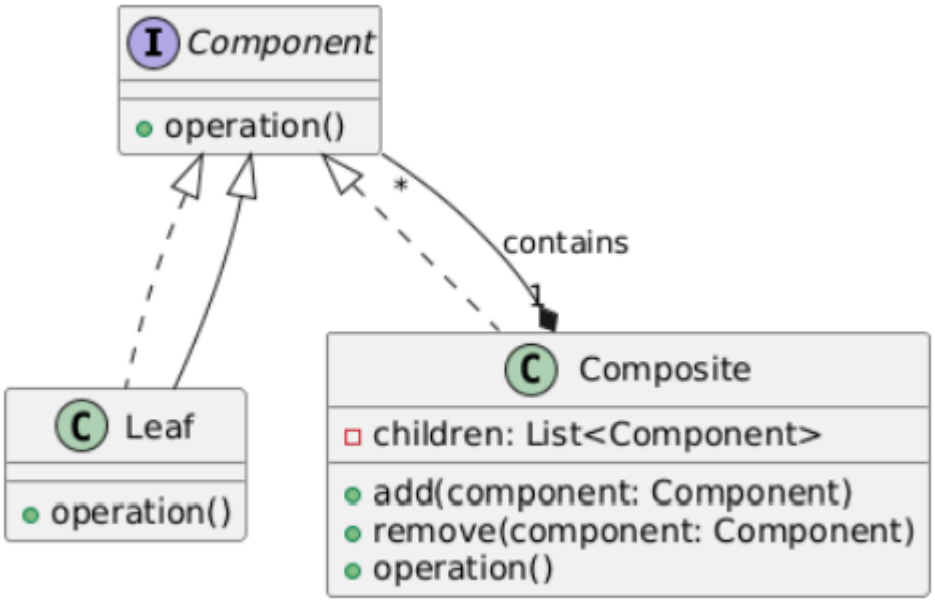
змінюючи їхній вихідний код, завдяки використанню механізму подвійної диспетчеризації. Це робить архітектуру застосунку значно гнучкішою та зручнішою для супроводу, оскільки нова функціональність (наприклад, експорт даних або збір статистики) інкапсулюється в окремих класах-відвідувачах, не порушуючи принцип єдиної відповідальності сутностей.

Контрольні запитання

1. Яке призначення шаблону «Композит»?

Шаблон «Композит» (Composite) призначений для того, щоб єдиним способом працювати як з окремими об’єктами, так і з групами об’єктів. Він дозволяє будувати деревоподібні структури (наприклад, елементи інтерфейсу, файлову систему, шари зображення) і викликати операції однаково для листків та компонувальників. Тобто клієнт не думає, має він справу з одним елементом чи з цілим набором елементів — інтерфейс у них спільний.

2. Нарисуйте структуру шаблону «Композит».



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

Шаблон «Композит» складається з трьох основних класів: Компонент, Листок і Компоновальник.

Компонент – це базовий інтерфейс або абстрактний клас, який визначає спільні операції для всіх елементів структури.

Листок – це простий елемент без дочірніх об’єктів, який просто виконує свою частину роботи.

Компоновальник — це елемент, який може містити інші компоненти. Він зберігає список дітей і викликає їхні операції, коли виконується його власна операція.

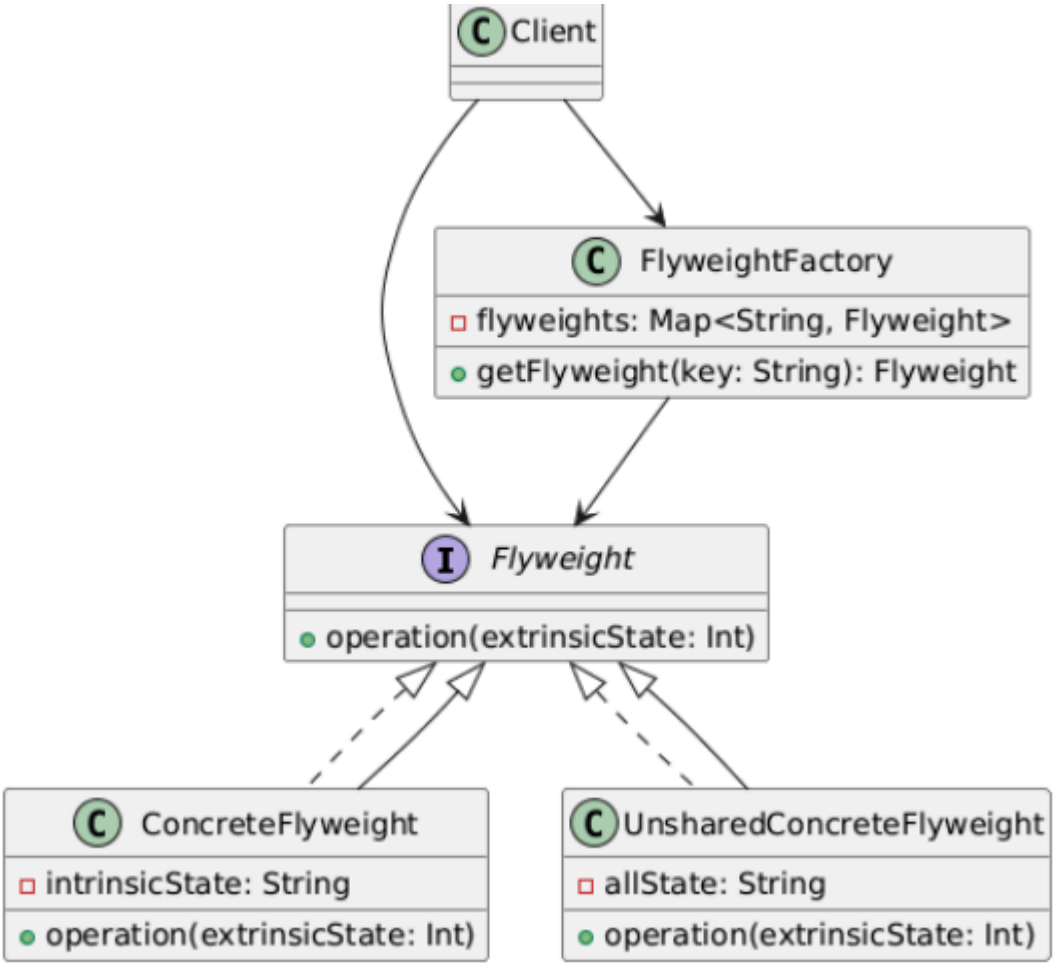
Взаємодія між ними така: клієнт звертається до всіх елементів через тип «Компонент». Листки обробляють запити напряму, а компоновальники

передають їх своїм дочірнім елементам. Завдяки цьому клієнт не відрізняє, працює він з одним елементом чи з цілою групою.

4. Яке призначення шаблону «Легковаговик»?

Шаблон «Легковаговик» (Flyweight) призначений для зменшення використання пам’яті, коли у програмі потрібно створити дуже багато однотипних об’єктів. Замість того щоб створювати тисячі однакових об’єктів, спільні дані виносять у один розділюваний об’єкт, а унікальні дані зберігають окремо. Це дозволяє значно економити ресурси та прискорювати роботу програми.

5. Нарисуйте структуру шаблону «Легковаговик».



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

У шаблон «Легковаговик» входять такі основні класи:

- Flyweight (Легковаговик) – об’єкт, який містить внутрішній стан (той, що спільний і не змінюється). Ці об’єкти можна багаторазово використовувати.
- ConcreteFlyweight – конкретна реалізація легковаговика з фіксованими внутрішніми даними.
- FlyweightFactory – створює та зберігає легковаговики. Якщо об’єкт з такими даними уже є, фабрика повертає існуючий, замість створення нового.
- Client – використовує легковаговики, передаючи їм зовнішній стан (унікальні дані, які не можна зберігати всередині легковаговика). Клієнт звертається до

фабрики, фабрика повертає вже існуючий легковаговик або створює новий. Легковаговик містить лише спільні дані, а унікальні дані клієнт передає йому під час використання. Це дозволяє значно економити пам’ять.

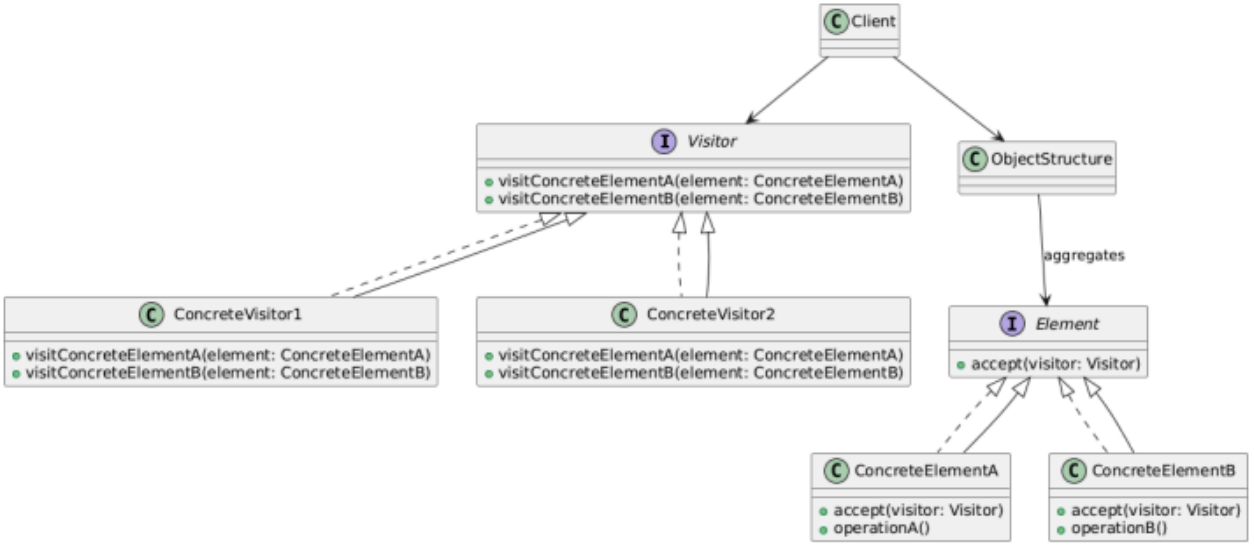
7. Яке призначення шаблону «Інтерпретатор»?

Шаблон «Інтерпретатор» (Interpreter) призначений для опису граматики мови та інтерпретації виразів цієї мови. Він дозволяє створити невелику мову (наприклад, для формул, фільтрів, команд) та написати набір класів, які можуть читати та виконувати вирази цієї мови.

8. Яке призначення шаблону «Відвідувач»?

Шаблон «Відвідувач» (Visitor) призначений для того, щоб додавати нові операції до об’єктів складної структури, не змінюючи їхні класи. Тобто він дозволяє винести логіку обробки об’єктів у окремий клас-відвідувач і “проходити” ним по елементах структури, виконуючи потрібні дії.

9. Нарисуйте структуру шаблону «Відвідувач».



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

Шаблон «Відвідувач» складається з таких основних класів:

- Visitor (Відвідувач) Оголошує методи для відвідування кожного типу елементів у структурі. Кожен метод відповідає конкретному класу елементів.
- ConcreteVisitor Реалізує конкретні операції, які виконуються над елементами. Наприклад: підрахунок, вивід, перевірка, експорт тощо.
- Element (Елемент) Базовий інтерфейс або абстрактний клас для всіх об’єктів структури. Містить метод accept(Visitor).
- ConcreteElement Конкретні елементи, які реалізують метод accept, передаючи себе відвідувачу (visitor.visit(this)).
- ObjectStructure Колекція або дерево елементів, які можна “обійти” за допомогою відвідувача.

Кожен елемент має метод accept, який приймає відвідувача. Відвідувач заходить у елемент, а елемент викликає відповідний метод відвідувача. Так відвідувач може виконувати нові операції над елементами, не змінюючи їхні класи.