



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

## ЛАБОРАТОРНА РОБОТА №8

з дисципліни "Технології розроблення програмного забезпечення"

Виконав

студент групи ІА–33:

Китченко Д.В.

Перевірив:

Мягкий М.Ю.

Київ 2025

## Зміст

Вступ.....	3
Мета роботи.....	4
Теоретичні відомості.....	5
Хід роботи .....	7
Висновки.....	14
Питання до лабораторної роботи .....	15

## Вступ

Патерни проектування є фундаментальними інструментами розробки програмного забезпечення, які надають перевірені рішення для типових архітектурних задач. Серед поведінкових патернів особливе місце займає патерн "Visitor" (Відвідувач), який дозволяє додавати нові операції до ієрархії класів без зміни самих класів.

У даній лабораторній роботі розглядається патерн Visitor у контексті системи архівації файлів. Під час розробки архіватора виникає потреба виконувати різні операції над елементами архіву: обчислення контрольних сум, перевірка цілісності, збір статистики, пошук файлів тощо. Якби ми додали всі ці операції прямо в класи ArchivedFile та ArchivedFolder, ці класи стали б занадто великими і складними. Вони б почали відповідати не тільки за зберігання даних, а й за складні обчислення, що є поганим тоном у програмуванні.

Патерн Visitor вирішує цю проблему, виносячи всю логіку нових операцій в окремі класи. Наші файли і папки залишаються "чистими", а вся складна робота виконується "відвідувачами" зовні. Замість додавання методів до класів елементів архіву, кожна операція реалізується як окремий клас-відвідувач. Елементи архіву лише надають можливість прийняти відвідувача через метод accept(), а вся логіка операцій міститься у відвідувачах. Це забезпечує гнучкість при додаванні нових операцій та дозволяє групувати пов'язану функціональність в окремих класах.

## Мета роботи

Вивчити структуру шаблонів "Composite", "Flyweight", "Interpreter", "Visitor" та навчитися застосовувати їх в реалізації програмної системи. Детально дослідити патерн "Visitor", зрозуміти його призначення, переваги та недоліки.

Реалізувати патерн Visitor в контексті системи архіватора для виконання операцій над елементами архіву: обчислення контрольних сум файлів та перевірки цілісності архіву. Створити мінімум три класи, що демонструють застосування патерну, та підготувати діаграму класів, яка представляє використання шаблону в реалізації системи.

## Теоретичні відомості

### Шаблон "Visitor"

Призначення: Шаблон відвідувач дозволяє вказувати операції над елементами без зміни структури конкретних елементів [6]. Таким чином вкрай зручно додавати нові операції, проте дуже важко додавати нові елементи в ієрархію (необхідно додавати відповідні методи для обробки їх відвідувань в кожного відвідувача).

Даний шаблон дозволяє групувати однотипні операції, що застосовуються над різнотипними об'єктами.

Проблема: Ви розробляєте онлайн-корзину інтернет магазину. Товари які представлені в магазині є різних типів, наприклад, електроніка, міцні напої, домашня хімія.

Логіка роботи з товарами в корзині є різна, наприклад, розрахунок вартості, формування замовлення.

Ми можемо всі ці методи зробити в товарах, але тоді ми ускладнюємо товари і змішуємо логіку (розрахунок вартості) з даними (товаром та його кількістю).

Якщо ми в подальшому необхідно буде додати ще логіку розрахунку вартості з врахуванням знижки, то потрібно буде додати ще цю логіку до товарів. А якщо буде ще сезонна знижка, то ми знову будемо додавати нову логіку до класів товарів.

Рішення: Основна ідея патерна "Відвідувач" це рознести логіку і дані в різні класи та ієрархії. Якщо так зробити для нашої онлайн-корзини, то в класі відвідувача ми маємо функції для обрахунку логіки для кожного типу, а об'єкти в корзині знають свій тип, отримують екземпляр відвідувача і в нього викликають метод відповідно до свого типу. В результаті ми робимо різні відвідувачі для розрахунку вартості: один для звичайних розрахунків, інший для розрахунку вартості зі знижкою, ще один для розрахунку сезонних знижок.

За рахунок того, що логіка відокремлена від наших товарів в корзині ми можемо реалізовувати за необхідності нові класи відвідувачі, а класи товарів та і корзини, в цілому, змінюватися не будуть.

Якщо розвивати далі, то можна зробити і клас відвідувача, який буде формувати замовлення з товарів в корзині в залежності від продавців і можливих варіантів доставки. Це дозволить формувати, наприклад, не одне замовлення, а два одна з самовивозом з магазину, а інше з доставкою Новою Поштою.

Приклад з життя: Прикладом може служити написання компілятора. Припустимо, існують різні об'єкти в синтаксисі мови програмування: виклики методів і умовні вирази. Компілятор перед генерацією коду повинен обійти всі вирази (і виклики методів, і умовні вирази) і перевірити безпеку типів, після чого згенерувати відповідний код. Відповідно буде два відвідувачі – для перевірки безпеки типів і для генерації коду. У кожного з них буде по 2 методи – для викликів методів і для умовних операцій. Таким чином при необхідності додавання нових кроків компіляції досить буде визначити нового "відвідувача" і викликати його у відповідний час.

#### 14. **Архіватор** (strategy, adapter, factory method, facade, visitor, p2p)

Архіватор повинен являти собою візуальний додаток з можливістю створення і редагування архівів різного типу (.tar.gz, .zip, .rar, .ace) – додавання/ видалення файлів / папок, редагування метаданих (по можливості), перевірка checksum архівів, тестування архівів на наявність пошкоджень, розбиття архівів на частини.

У процесі розробки системи архівації файлів виникла необхідність реалізувати дві важливі функції відповідно до вимог: перевірку контрольних сум архівів та тестування архівів на наявність пошкоджень. Існуюча система має ієрархію класів для представлення елементів архіву: абстрактний клас `ArchiveItem` та його нащадки `ArchivedFile` і `ArchivedFolder`.

Пряма реалізація функцій перевірки та тестування як методів у класах елементів призвела б до їх ускладнення та змішування різної функціональності. Для вирішення цієї проблеми було застосовано патерн `Visitor`.

Компоненти реалізації:

1. Інтерфейс `ArchiveVisitor` - це список обов'язкових дій, які повинен вміти виконувати будь-який "відвідувач". Це контракт для будь-якого майбутнього "відвідувача". Він вимагає, щоб кожен відвідувач умів робити дві речі: працювати з файлом і працювати з папкою.

2. `ChecksumVisitor` - конкретний відвідувач для обчислення загальної контрольної суми архіву. При відвідуванні файлу намагається розпарсити його контрольну суму як число та додати до загального підсумку. Якщо контрольна сума збережена не як число, використовується розмір файлу як альтернатива. При відвідуванні папки рекурсивно обходить всіх її нащадків. Відвідувач підтримує лічильник оброблених файлів для статистики.

3. `TestArchiveVisitor` - конкретний відвідувач для виявлення проблем у архіві. Підтримує два списки: критичних помилок та попереджень. При відвідуванні файлу перевіряє: чи не є розмір файлу нульовим (можливе

пошкодження), чи присутня контрольна сума, чи не перевищує стиснений розмір оригінальний. При відвідуванні папки перевіряє чи не є вона порожньою та рекурсивно обходить нащадків. Надає метод `getReport()` для форматowanego виведення знайдених проблем.

Модифікація існуючих класів:

Для інтеграції патерну Visitor в існуючу ієрархію класів елементів архіву були внесені мінімальні зміни. В абстрактному класі `ArchiveItem` додано абстрактний метод `accept(ArchiveVisitor visitor)`. Це змушує всі конкретні класи елементів реалізувати цей метод.

У класах `ArchivedFile` та `ArchivedFolder` метод `accept()` реалізовано однаково просто: `visitor.visit(this)`. Коли ми пишемо `visitor.visit(this)`, ми передаємо відвідувачу посилання на себе. Java автоматично розуміє, хто саме викликав цей метод (файл чи папка), і запускає відповідну, правильну частину коду відвідувача.

Принцип роботи:

Коли потрібно виконати операцію над архівом, створюється відповідний відвідувач та передається кореневому елементу архіву через метод `accept()`. Відвідувач автоматично обходить всю структуру архіву завдяки рекурсивним викликам у методі `visit(ArchivedFolder)`. Після завершення обходу результати можна отримати через методи відвідувача.

Для обчислення контрольної суми всього архіву достатньо створити `ChecksumVisitor`, викликати `root.accept(checksumVisitor)` та отримати результат через `getTotalChecksum()`. Аналогічно для перевірки цілісності створюється `TestArchiveVisitor`, викликається `accept()` та результати перевірки отримуються через `getReport()`.

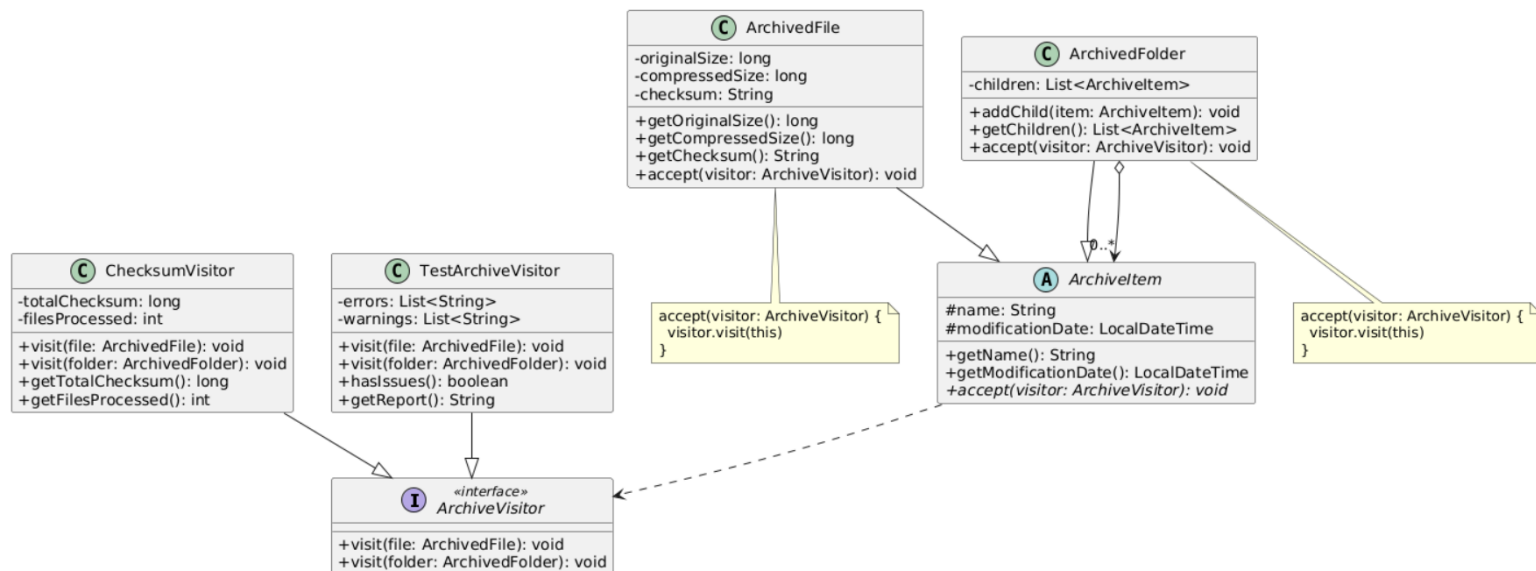


Рисунок 1. Діаграма класів для реалізації патерну Visitor

## Клас ArchiveVisitor (інтерфейс)

```

package visitor;

import model.ArchivedFile;
import model.ArchivedFolder;

public interface ArchiveVisitor {
    void visit(ArchivedFile file);
    void visit(ArchivedFolder folder);
}

```

## Клас ChecksumVisitor

```

package visitor;

import model.ArchivedFile;
import model.ArchivedFolder;
import model.ArchiveItem;

public class ChecksumVisitor implements ArchiveVisitor {

    private long totalChecksum = 0;
    private int filesProcessed = 0;

    @Override
    public void visit(ArchivedFile file) {
        System.out.println("ChecksumVisitor: Обробка файлу " + file.getName());

        try {
            // розписано чексум як число
            long fileChecksum = Long.parseLong(file.getChecksum());
            totalChecksum += fileChecksum;
        } catch (NumberFormatException e) {
            totalChecksum += file.getOriginalSize();
        }
        filesProcessed++;
    }

    @Override
    public void visit(ArchivedFolder folder) {

```

```

        System.out.println("ChecksumVisitor: Заходимо в папку " +
folder.getName());

        // рекурсивний обхід дочірніх елементів
        for (ArchiveItem item : folder.getChildren()) {
            item.accept(this);
        }
    }

    public long getTotalChecksum() {
        return totalChecksum;
    }

    public int getFilesProcessed() {
        return filesProcessed;
    }
}

```

## Клас TestArchiveVisitor

```

package visitor;

import model.ArchivedFile;
import model.ArchivedFolder;
import model.ArchiveItem;

import java.util.ArrayList;
import java.util.List;

public class TestArchiveVisitor implements ArchiveVisitor {

    private final List<String> errors = new ArrayList<>();
    private final List<String> warnings = new ArrayList<>();

    @Override
    public void visit(ArchivedFile file) {
        System.out.println("TestArchiveVisitor: Тестування файлу " +
file.getName());

        // перевірка 1 чи файл нульового розміру
        if (file.getOriginalSize() == 0) {
            warnings.add("ПОПЕРЕДЖЕННЯ: Файл '" + file.getName() + "' має нульовий розмір");
        }

        // перевірка 2 відсутність checksum
        if (file.getChecksum() == null || file.getChecksum().isEmpty()) {
            warnings.add("ПОПЕРЕДЖЕННЯ: Файл '" + file.getName() + "' не має контрольної суми");
        }

        // перевірка 3 стиснений розмір більший за оригінальний
        if (file.getCompressedSize() > file.getOriginalSize()) {
            errors.add("ПОМИЛКА: Файл '" + file.getName() + "' має стиснений розмір більший за оригінальний!");
        }
    }

    @Override
    public void visit(ArchivedFolder folder) {
        System.out.println("TestArchiveVisitor: Перевірка папки " +
folder.getName());
    }
}

```

```

        // рекурсивно тестуємо всіх дітей папки
        for (ArchiveItem item : folder.getChildren()) {
            item.accept(this);
        }
        // перевірка папка не має бути порожньою
        if (folder.getChildren().isEmpty() && !folder.getName().equals("/")) {
            warnings.add("ІНФО: Папка '" + folder.getName() + "' порожня");
        }
    }

    public boolean hasIssues() {
        return !errors.isEmpty() || !warnings.isEmpty();
    }

    public String getReport() {
        StringBuilder sb = new StringBuilder();
        sb.append("=== Звіт тестування ===\n");

        if (!hasIssues()) {
            sb.append("Помилки не виявлено. Архів виглядає цілісним\n");
        } else {
            if (!errors.isEmpty()) {
                sb.append("\nКРИТИЧНІ ПОМИЛКИ:\n");
                for (String err : errors) {
                    sb.append("    • ").append(err).append("\n");
                }
            }

            if (!warnings.isEmpty()) {
                sb.append("\nПОПЕРЕДЖЕННЯ:\n");
                for (String warn : warnings) {
                    sb.append("    • ").append(warn).append("\n");
                }
            }
        }

        return sb.toString();
    }
}

```

## Приклад використання у клієнтському коді

```

package visitor;

import model.*;

import java.time.LocalDateTime;

public class TestVisitorPattern {

    public static void main(String[] args) {

        ArchivedFolder root = new ArchivedFolder("archive.zip",
LocalDateTime.now());

        // додаємо нормальні файли
        root.addChild(new ArchivedFile("readme.txt", LocalDateTime.now(),
1024, 512, "12345"));
        root.addChild(new ArchivedFile("image.png", LocalDateTime.now(),
50000, 45000, "67890"));

        // додаємо пошкоджений файл (нульовий розмір, немає checksum)
        root.addChild(new ArchivedFile("corrupted.dat", LocalDateTime.now(),

```

```

        0, 0, null));

        // створюємо папку з файлами
        ArchivedFolder docs = new ArchivedFolder("documents",
LocalDateTime.now());
        docs.addChild(new ArchivedFile("doc1.pdf", LocalDateTime.now(),
            20000, 18000, "11111"));
        docs.addChild(new ArchivedFile("doc2.docx", LocalDateTime.now(),
            30000, 25000, "22222"));
        root.addChild(docs);

        // додаємо порожню папку
        ArchivedFolder emptyFolder = new ArchivedFolder("empty_folder",
LocalDateTime.now());
        root.addChild(emptyFolder);

        System.out.println("ТЕСТУВАННЯ ПАТЕРНУ VISITOR");
        System.out.println("=====\n");

        System.out.println("1. ПЕРЕВІРКА CHECKSUM:");
        System.out.println("-----");
        ChecksumVisitor checksumVisitor = new ChecksumVisitor();
        root.accept(checksumVisitor);
        System.out.println("\nРезультат:");
        System.out.println("Загальна checksum: " +
checksumVisitor.getTotalChecksum());
        System.out.println("Оброблено файлів: " +
checksumVisitor.GetFilesProcessed());

        System.out.println("\n\n2.ТЕСТУВАННЯ НА ПОШКОДЖЕННЯ:");
        System.out.println("-----");
        TestArchiveVisitor testVisitor = new TestArchiveVisitor();
        root.accept(testVisitor);
        System.out.println("\n" + testVisitor.getReport());

        System.out.println("=====");
        System.out.println("ТЕСТУВАННЯ ЗАВЕРШЕНО");

    }
}

```

## Тестування реалізації

Для перевірки коректності роботи реалізованого патерну Visitor було створено тестову структуру архіву з різними типами елементів та запущено обидва відвідувачі

### Крок 1: Створення тестової структури архіву

Було створено кореневу папку archive.zip з наступними елементами:

- readme.txt (1024 байт, checksum "12345")
- image.png (50000 байт, checksum "67890")
- corrupted.dat (0 байт, без checksum) - тестовий пошкоджений файл
- Папка documents з файлами doc1.pdf та doc2.docx

- Порожня папка empty\_folder

## Крок 2: Тестування ChecksumVisitor

Викликано метод root.accept(checksumVisitor). Відвідувач автоматично обійшов всю структуру архіву, включаючи вкладені папки. Для кожного файлу була розпарсена контрольна сума та додана до загального підсумку. Для файлу без checksum використано його розмір як альтернативу.

## Крок 3: Тестування TestArchiveVisitor

Викликано метод root.accept(testVisitor). Відвідувач перевіряв кожен файл на наявність проблем. Також перевірено всі папки на наявність вмісту.

Результати тестування:

### ТЕСТУВАННЯ ПАТЕРНУ VISITOR

=====

#### 1. ПЕРЕВІРКА CHECKSUM:

-----

ChecksumVisitor: Заходимо в папку archive.zip

ChecksumVisitor: Обробка файлу readme.txt

ChecksumVisitor: Обробка файлу image.png

ChecksumVisitor: Обробка файлу corrupted.dat

ChecksumVisitor: Заходимо в папку documents

ChecksumVisitor: Обробка файлу doc1.pdf

ChecksumVisitor: Обробка файлу doc2.docx

ChecksumVisitor: Заходимо в папку empty\_folder

Результат:

Загальна checksum: 113568

Оброблено файлів: 5

#### 2.ТЕСТУВАННЯ НА ПОШКОДЖЕННЯ:

-----

TestArchiveVisitor: Перевірка папки archive.zip

TestArchiveVisitor: Тестування файлу readme.txt

TestArchiveVisitor: Тестування файлу image.png

TestArchiveVisitor: Тестування файлу corrupted.dat

TestArchiveVisitor: Перевірка папки documents

TestArchiveVisitor: Тестування файлу doc1.pdf

TestArchiveVisitor: Тестування файлу doc2.docx

TestArchiveVisitor: Перевірка папки empty\_folder

=== Звіт тестування ===

#### ПОПЕРЕДЖЕННЯ:

- ПОПЕРЕДЖЕННЯ: Файл 'corrupted.dat' має нульовий розмір
- ПОПЕРЕДЖЕННЯ: Файл 'corrupted.dat' не має контрольної суми
- ІНФО: Папка 'empty\_folder' порожня

=====

#### ТЕСТУВАННЯ ЗАВЕРШЕНО

Process finished with exit code 0

#### Висновки

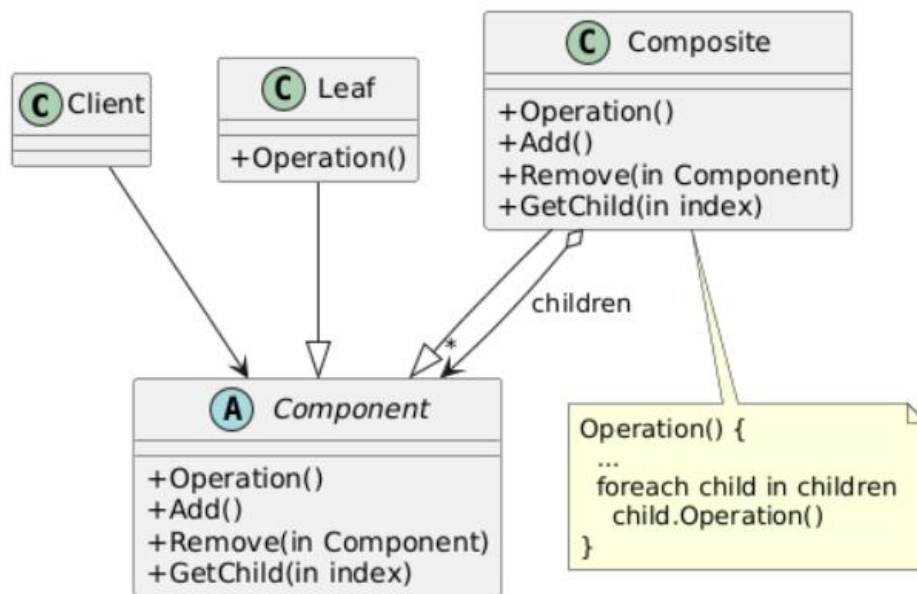
В ході виконання даної лабораторної роботи було детально вивчено та успішно застосовано на практиці поведінковий патерн проєктування Visitor. Було реалізовано три класи: інтерфейс ArchiveVisitor та два конкретні відвідувачі - ChecksumVisitor для обчислення контрольних сум та TestArchiveVisitor для перевірки цілісності архіву. Патерн Visitor продемонстрував свою основну перевагу - можливість додавання нових операцій до ієрархії класів без зміни самих класів. Класи ArchivedFile та ArchivedFolder залишились простими контейнерами даних, а вся логіка операцій була винесена в окремі класи-відвідувачі. Це забезпечило чітке розділення відповідальностей та спростило підтримку коду.

## Питання до лабораторної роботи

### 1. Яке призначення шаблону "Композит"?

Дозволяє працювати з деревоподібними структурами так, ніби це один об'єкт. Клієнт не знає чи він працює з листком чи з гілкою - для нього всі однакові. Класика: файли і папки в провіднику Windows.

### 2. Нарисуйте структуру шаблону "Композит".



### 3. Які класи входять в шаблон "Композит", та яка між ними взаємодія?

Component - базовий інтерфейс/клас з операціями (operation())

Leaf - кінцевий елемент, не має нащадків, просто виконує operation()

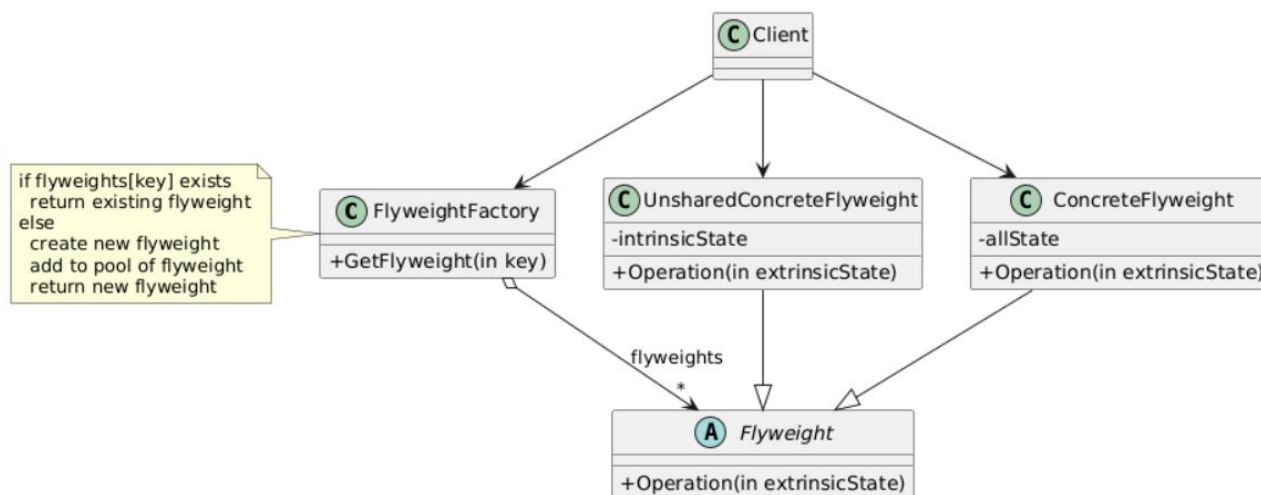
Composite - контейнер, містить children: List<Component>. При виклику operation() викликає operation() у всіх дітей (цикл)

Взаємодія: Клієнт викликає operation() на будь-якому Component. Якщо це Leaf - просто виконується. Якщо Composite - рекурсивно викликається у всіх дітей.

### 4. Яке призначення шаблону "Легковаговик"?

Економить пам'ять коли треба створити дофіга однакових об'єктів. Розділяє стан на intrinsic (спільний для всіх, зберігається один раз) і extrinsic (унікальний, передається ззовні). Приклад: символи в текстовому редакторі - шрифт і розмір спільні (intrinsic), позиція унікальна (extrinsic).

5. Нарисуйте структуру шаблону "Легковаговик".



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

Flyweight - інтерфейс з Operation(extrinsicState)

ConcreteFlyweight - розділюваний легковаговик. Зберігає intrinsicState (не міняється, спільний для багатьох клієнтів). Отримує extrinsicState як параметр у Operation().

UnsharedConcreteFlyweight - НЕрозділюваний легковаговик. Зберігає allState (весь стан всередині, не економить пам'ять). Використовується коли об'єкт не можна розділити, але треба дотримати єдиний інтерфейс.

FlyweightFactory - кеш розділюваних легковаговиків. Перевіряє чи є ConcreteFlyweight у pool за ключем. Якщо є - повертає існуючий, якщо ні - створює новий і додає в pool. НЕ створює UnsharedConcreteFlyweight!

Client - для розділюваних об'єктів запитує Factory.GetFlyweight(key), для нерозділюваних створює UnsharedConcreteFlyweight() напряму. Викликає Operation(extrinsicState) на отриманих об'єктах.

Взаємодія:

Розділювані: Client → Factory → перевіряє pool → повертає існуючий/новий ConcreteFlyweight → Client викликає Operation(extrinsicState)

Нерозділювані: Client → new UnsharedConcreteFlyweight() → викликає Operation(extrinsicState)

Фішка: Обидва типи реалізують Flyweight, тому Client може працювати з ними однаково, але економія пам'яті тільки на ConcreteFlyweight.

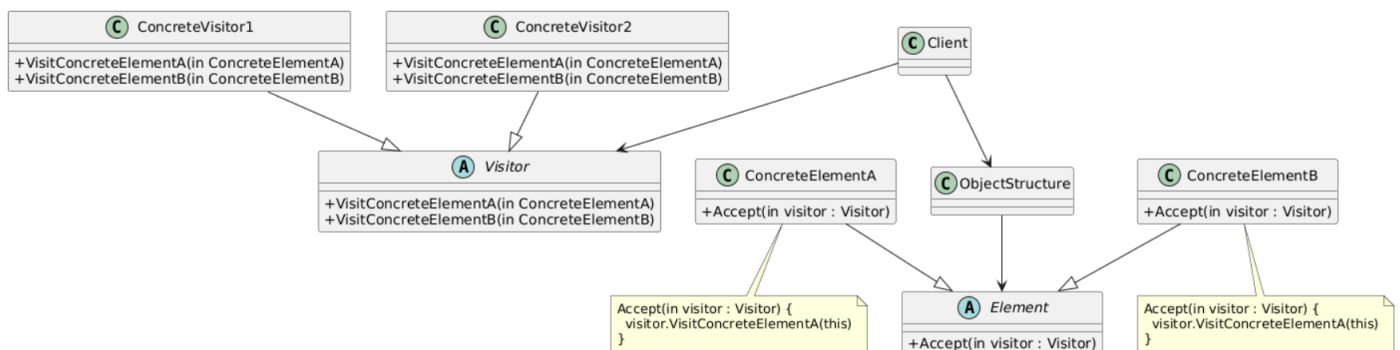
## 7. Яке призначення шаблону "Інтерпретатор"?

Використовується для створення своєї власної міні-мови в програмі. Він дозволяє визначити граматику цієї мови та написати інтерпретатор, який буде виконувати "речення" цією мовою. Корисно для складних пошукових запитів або математичних виразів.

## 8. Яке призначення шаблону "Відвідувач"?

Додавати нові операції класам, не змінюючи їхній код. Це ідеально, коли у вас є стабільна структура класів (як файли і папки), але постійно виникають нові задачі для них (порахувати статистику, перевірити на віруси, заархівувати).

## 9. Нарисуйте структуру шаблону "Відвідувач".



## 10. Які класи входять в шаблон "Відвідувач", та яка між ними взаємодія?

Visitor - інтерфейс з visit() для кожного типу елемента

ConcreteVisitor - реалізує логіку операції для кожного типу

Element - має метод accept(Visitor v), щоб "впустити" до себе відвідувача

ConcreteElement - реалізує accept як visitor.visit(this)

Взаємодія:

Клієнт: element.accept(visitor)

Element: visitor.visit(this) <-- this має конкретний тип!

Java: бачить тип this, викликає правильний visit()

Visitor: виконує логіку для цього типу елемента

Фішка: Без instanceof! Без switch! Типобезпечно на рівні компіляції. Додати нову операцію = створити новий Visitor, не чіпаючи Element.