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

Технології розробки програмного забезпечення

«Патерни проєктування»

«Веб-браузер»

Виконав:
студент групи ІА-33
Мартинюк Ю.Р.

Перевірив:
Мягкий Михайло Юрійович

Київ 2025

Тема: Патерни проєктування

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

Зміст

Завдання

• Ознайомитись з короткими теоретичними відомостями.

• Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

• Реалізувати один з розглянутих шаблонів за обраною темою.

• Реалізувати не менше 3-х класів відповідно до обраної теми.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

Теоретичні відомості

Composite (Компонувальник)

Цей структурний патерн вирішує проблему побудови складних деревоподібних структур, дозволяючи працювати з ними через єдиний інтерфейс. Ключова ідея полягає в тому, що клієнтський код не повинен розрізняти прості

об'єкти (листки) та складені об'єкти (контейнери). Усі елементи дерева реалізують спільний інтерфейс, який зазвичай містить методи для виконання основної бізнес-логіки. Для простих компонентів метод просто виконує дію, а для контейнерів він проходить по списку всіх дочірніх елементів, викликає цей метод для кожного з них і підсумовує або агрегує результат. Це дозволяє будувати рекурсивні структури будь-якої глибини вкладеності.

На практиці цей патерн ідеально підходить для реалізації графічних редакторів (де фігури можна групувати в складніші об'єкти), файлових систем (де папки можуть містити файли та інші папки) або меню програмного забезпечення. Головною перевагою є спрощення клієнтського коду, оскільки зникає необхідність перевіряти типи об'єктів перед викликом методів. Однак, недоліком може стати занадто загальний інтерфейс: іноді доводиться додавати методи управління дітьми (додати/видалити) в інтерфейс листка, де вони не мають сенсу, що порушує принцип розділення інтерфейсу.

Flyweight (Легковаговик)

Основна мета цього структурного патерну — економія оперативної пам'яті в системах із величезною кількістю об'єктів. Патерн досягає цього шляхом розділення стану об'єкта на дві частини: внутрішній (intrinsic) та зовнішній (extrinsic). Внутрішній стан — це дані, які є спільними для багатьох об'єктів і не змінюються (наприклад, форма кулі, текстура дерева, шрифт літери). Зовнішній стан — це контекстні дані, унікальні для кожного екземпляра (координати, колір, розмір), які передаються об'єкту лише в момент виконання методу.

Легковаговик — це незмінний об'єкт, який ініціалізується один раз і використовується в різних контекстах. Зазвичай створення таких об'єктів контролюється спеціальною фабрикою, яка кешує вже створені екземпляри: якщо об'єкт із потрібним внутрішнім станом вже існує, фабрика повертає його, замість створення нового. Це критично важливо для ігрових рушіїв (відображення лісу, армії солдатів) або текстових редакторів (де кожна літера є об'єктом). Головний мінус — ускладнення коду програми та необхідність перерахунку зовнішніх даних кожного разу при зверненні до об'єкта, що може незначно вплинути на

процесорний час заради економії пам'яті.

Interpreter (Інтерпретатор)

Цей поведінковий патерн використовується для проектування мовних інтерпретаторів. Він визначає граматику простої мови та будує інтерпретатор для речень цієї мови. Кожне правило граматики (наприклад, змінні, числа, операції додавання чи віднімання) перетворюється на окремий клас. Речення мови представляється у вигляді дерева об'єктів (Абстрактне Синтаксичне Дерево), де вузли — це операції, а листки — термінальні вирази (значення). Щоб виконати програму або обчислити вираз, інтерпретатор рекурсивно проходить по дереву, викликаючи метод interpret для кожного вузла.

Патерн найкраще підходить для специфічних, вузьконаправлених задач: розбору математичних виразів, SQL-подібних запитів, регулярних виразів або конфігураційних файлів. Проте він стає вкрай неефективним для складних мов програмування з великою кількістю правил, оскільки кількість класів зростає лавиноподібно, а дерево стає надто громіздким для обробки. У таких випадках зазвичай використовують повноцінні парсери та компілятори, а не патерн Інтерпретатор.

Visitor (Відвідувач)

Цей поведінковий патерн дозволяє відокремити алгоритми від структури об'єктів, над якими вони оперують. Це особливо корисно, коли у вас є стабільна ієрархія класів (наприклад, типи вузлів у дереві документа: текст, картинка, таблиця), яка рідко змінюється, але вам часто потрібно додавати нові операції над цими даними (експорт у XML, генерація звіту, підрахунок статистики). Замість того, щоб додавати новий метод у кожен клас даних (що порушує принцип відкритості/закритості), ви створюєте окремий клас — Відвідувач, який містить методи обробки для кожного типу елемента.

Технічною основою патерну є механізм подвійної диспетчеризації (Double Dispatch). Кожен елемент структури має метод accept, який приймає об'єкт

відвідувача і викликає у нього відповідний метод (наприклад, visitor.visitTable(this)). Таким чином, вибір потрібного методу залежить і від типу відвідувача, і від типу елемента. Це дозволяє додавати скільки завгодно нових операцій, просто створюючи нові класи відвідувачів, не чіпаючи існуючий код класів даних. Недоліком є те, що при додаванні нового типу елемента в структуру (наприклад, "Відео"), доведеться оновлювати всі існуючі класи відвідувачів.

Тема проєкту

6. Web-browser (proxy, chain of responsibility, factory method, template method, visitor, p2p) Веб-браузер повинен мати можливість зробити наступне: мати адресний рядок для введення адреси сайту, переміщатися і відображати структуру html документа, переглядати підключений javascript та css файли, перегляд всіх підключених ресурсів (зображень), коректна обробка відповідей з сервера (коди відповідей HTTP) – переходи при перенаправленнях, відображення сторінок 404 і 502/503.

Патерн: Visitor

Посилання на Github: https://github.com/masonabor/web-browser-lab

Хід роботи

Діаграма класів

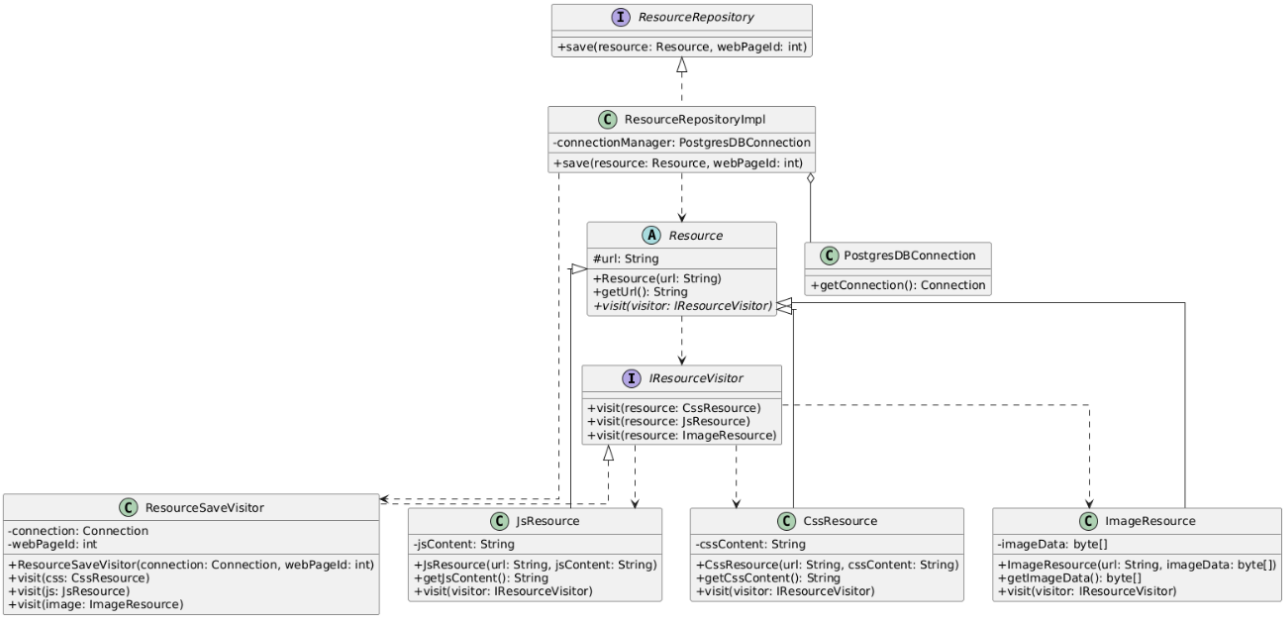


Рис.1 – Діаграма класів для ResourceRepository (патерн Visitor)

Опис діаграми класів

Ця діаграма класів ілюструє, як операцію збереження даних винесено за межі класів сутностей, відокремлюючи алгоритм від структури об'єктів за допомогою патерну "Відвідувач" (Visitor). Замість того, щоб кожен клас ресурсу (JsResource, CssResource тощо) містив логіку роботи з базою даних, ця відповідальність делегована окремому класу-відвідувачу, а ресурси лише надають метод для "прийняття" цього відвідувача.

Учасники патерну "Відвідувач"

На діаграмі чітко видно всіх ключових учасників патерну та допоміжні класи:

1. Інтерфейс Відвідувача (Visitor Interface)

- **Клас:** IResourceVisitor
- **Роль:** Визначає контракт для взаємодії з елементами. Він оголошує набір методів visit(...), по одному для кожного конкретного типу ресурсу, що існує в системі.
- **Ключова деталь:** Це дозволяє додавати нові типи операцій (наприклад, експорт у XML) просто створивши нову реалізацію цього інтерфейсу, не змінюючи класи ресурсів.

2. Конкретний Відвідувач (Concrete Visitor)

- **Клас:** ResourceSaveVisitor
- **Роль:** Реалізує специфічну поведінку, визначену в інтерфейсі — у даному випадку це логіка збереження (SQL INSERT).

- **Взаємодія:**
    - Він зберігає контекст, необхідний для виконання операції (поле connection та webPageId).
    - Він реалізує методи visit(js: JsResource), visit(css: CssResource) тощо, витягуючи специфічні дані з переданих об'єктів та записуючи їх у базу.

3. Елемент (Element)

- **Клас:** Resource (абстрактний)
- **Роль:** Базовий клас для структури об'єктів. Він визначає абстрактний метод visit(visitor: IResourceVisitor).
- **Ключова деталь:** Цей метод є точкою входу для відвідувача.

4. Конкретні Елементи (Concrete Elements)

- **Класи:** JsResource, CssResource, ImageResource
- **Роль:** Це класи даних. Вони зберігають контент (jsContent, imageData) та реалізують метод visit.
- **Взаємодія:** Реалізація методу visit у цих класах використовує техніку "подвійної диспетчеризації" (Double Dispatch): вони викликають метод відвідувача, передаючи себе як аргумент (visitor.visit(this)), що дозволяє точно визначити тип об'єкта під час виконання.

5. Клієнт (Client)

- **Клас:** ResourceRepositoryImpl
- **Роль:** Ініціатор процесу. Цей клас відповідає за створення відвідувача та запуск ланцюжка обробки.
- **Взаємодія:**
    - Використовує PostgresDBConnection для отримання з'єднання з БД.
    - Інстанціює ResourceSaveVisitor.
    - Викликає метод resource.visit(visitor), передаючи створеного відвідувача ресурсу.

Як працює потік (The Flow)

1. Клієнт (ResourceRepositoryImpl) отримує на вхід абстрактний Resource та webPageId.
2. Клієнт створює екземпляр ResourceSaveVisitor, передаючи йому активне з'єднання з БД.
3. Клієнт викликає метод resource.visit(visitor).
4. Завдяки поліморфізму виконується метод visit конкретного класу (наприклад, ImageResource).
5. Всередині цього методу об'єкт ImageResource викликає метод

visitor.visit(this). Оскільки this є типом ImageResource, викликається відповідний метод visit(image: ImageResource) у класі ResourceSaveVisitor.

6. ResourceSaveVisitor бере дані з ImageResource (байти картинки) і виконує SQL-запит для їх збереження.

Частина коду програми з використанням патерну Visitor

IResourceVisitor.java

```
package com.edu.web.restservicewebbrowser.visitor.resource;

import com.edu.web.restservicewebbrowser.domain.resource.CssResource;
import com.edu.web.restservicewebbrowser.domain.resource.ImageResource;
import com.edu.web.restservicewebbrowser.domain.resource.JsResource;

public interface IResourceVisitor {
    void visit(CssResource resource) throws Exception;
    void visit(JsResource resource) throws Exception;
    void visit(ImageResource resource) throws Exception;
}
```

ResourceSaveVisitor.java

```
package com.edu.web.restservicewebbrowser.visitor.resource.impl;

import com.edu.web.restservicewebbrowser.domain.resource.CssResource;
import com.edu.web.restservicewebbrowser.domain.resource.ImageResource;
import com.edu.web.restservicewebbrowser.domain.resource.JsResource;
import com.edu.web.restservicewebbrowser.visitor.resource.IResourceVisitor;

import java.sql.Connection;
import java.sql.PreparedStatement;

public class ResourceSaveVisitor implements IResourceVisitor {

    private Connection connection;
    private int webPageId;

    public ResourceSaveVisitor(Connection connection, int webPageId) {
        this.connection = connection;
        this.webPageId = webPageId;
```

```
    }

    @Override
    public void visit(CssResource css) throws Exception {
        System.out.println("VISITOR: Збереження CssResource в таблицю
CssResources...");

        String sql = "INSERT INTO CssResources (web_page_id, css_content, url)
VALUES (?, ?, ?)";
        try (PreparedStatement stmt = connection.prepareStatement(sql)) {
            stmt.setInt(1, this.webPageId);
            stmt.setString(2, css.getCssContent());
            stmt.setString(3, css.getUrl());
            stmt.executeUpdate();
        }
    }

    @Override
    public void visit(JsResource js) throws Exception {
        System.out.println("VISITOR: Збереження JsResource в таблицю
JsResources...");

        String sql = "INSERT INTO JsResources (web_page_id, js_content, url)
VALUES (?, ?, ?)";
        try (PreparedStatement stmt = connection.prepareStatement(sql)) {
            stmt.setInt(1, this.webPageId);
            stmt.setString(2, js.getJsContent());
            stmt.setString(3, js.getUrl());
            stmt.executeUpdate();
        }
    }

    @Override
    public void visit(ImageResource image) throws Exception {
        System.out.println("VISITOR: Збереження ImageResource в таблицю
ImageResources...");

        String sql = "INSERT INTO ImageResources (web_page_id, image_data, url)
VALUES (?, ?, ?)";
        try (PreparedStatement stmt = connection.prepareStatement(sql)) {
            stmt.setInt(1, this.webPageId);
```

```
            stmt.setBytes(2, image.getImageData());
            stmt.setString(3, image.getUrl());
            stmt.executeUpdate();
        }
    }
}
```

Resource.java

```
package com.edu.web.restservicewebbrowser.domain.resource;

import com.edu.web.restservicewebbrowser.visitor.resource.IResourceVisitor;

public abstract class Resource {
    protected String url;

    public abstract void visit(IResourceVisitor visitor) throws Exception;

    public Resource(String url) {
        this.url = url;
    }

    public String getUrl() {
        return url;
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName() + " [url=" + url + "]";
    }
}
```

JsResource.java

```
package com.edu.web.restservicewebbrowser.domain.resource;

import com.edu.web.restservicewebbrowser.visitor.resource.IResourceVisitor;

public class JsResource extends Resource {
    private final String jsContent;

    public JsResource(String url, String jsContent) {
```

```
        super(url);
        this.jsContent = jsContent;
    }

    public String getJsContent() {
        return jsContent;
    }

    @Override
    public void visit(IResourceVisitor visitor) throws Exception {
        visitor.visit(this);
    }
}
```

ImageResource.java

```
package com.edu.web.restservicewebbrowser.domain.resource;

import com.edu.web.restservicewebbrowser.visitor.resource.IResourceVisitor;

public class ImageResource extends Resource {
    private final byte[] imageData;

    public ImageResource(String url, byte[] imageData) {
        super(url);
        this.imageData = imageData;
    }

    public byte[] getImageData() {
        return imageData;
    }

    @Override
    public void visit(IResourceVisitor visitor) throws Exception {
        visitor.visit(this);
    }
}
```

CssResource.java

```
package com.edu.web.restservicewebbrowser.domain.resource;
```

```
import com.edu.web.restservicewebbrowser.visitor.resource.IResourceVisitor;

public class CssResource extends Resource {

    private final String cssContent;

    public CssResource(String url, String cssContent) {
        super(url);
        this.cssContent = cssContent;
    }

    public String getCssContent() {
        return cssContent;
    }

    @Override
    public void visit(IResourceVisitor visitor) throws Exception {
        visitor.visit(this);
    }
}
```

ResourceRepositoryImpl.java

```
package com.edu.web.restservicewebbrowser.repository.impl;


import com.edu.web.restservicewebbrowser.config.db.PostgresDBConnection;
import com.edu.web.restservicewebbrowser.domain.resource.Resource;
import com.edu.web.restservicewebbrowser.repository.ResourceRepository;
import
com.edu.web.restservicewebbrowser.visitor.resource.impl.ResourceSaveVisitor;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;

import java.sql.Connection;

@ApplicationScoped
public class ResourceRepositoryImpl implements ResourceRepository {

    @Inject
    private PostgresDBConnection connectionManager;
```

```
    public void save(Resource resource, int webPageId) {
        try (Connection conn = connectionManager.getConnection()) {

            ResourceSaveVisitor saveVisitor = new ResourceSaveVisitor(conn,
webPageId);

            resource.visit(saveVisitor);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Опис коду з використання патерну Chain of responsibility

Цей код демонструє архітектурне рішення для збереження різнотипних веб-ресурсів (CSS, JS, зображення) у реляційну базу даних. Замість порушення принципу єдиної відповідальності (SRP) шляхом додавання SQL-запитів безпосередньо в класи даних (JsResource, ImageResource), логіка персистентності (збереження) була повністю винесена в окремий клас-відвідувач. Це дозволяє додавати нові операції над ресурсами без зміни їхньої структури.

Учасники патерну "Visitor"

Реалізація спирається на механізм **подвійної диспетчеризації** (Double Dispatch), де виконання потрібного методу залежить від типу двох об'єктів: самого ресурсу та відвідувача.

1. Інтерфейс Відвідувача (Visitor Interface)

- **Клас:** IResourceVisitor
- **Роль:** Оголошує набір методів visit для кожного конкретного типу ресурсу, що існує в системі.
- **Призначення:** Забезпечує контракт, який гарантує, що будь-який відвідувач (чи то для збереження в БД, чи для експорту в файл) вміє працювати з усіма типами контенту: CssResource, JsResource та ImageResource.

2. Конкретний Відвідувач (Concrete Visitor)

- **Клас:** ResourceSaveVisitor
- **Роль:** Реалізує специфічну бізнес-логіку — збереження даних у базу даних.
- **Деталі реалізації:**
    - Клас зберігає стан контексту (Connection та webPageId), необхідний

для виконання SQL-запитів.
    - Для кожного методу visit(...) реалізовано унікальний INSERT запит у відповідну таблицю (CssResources, JsResources, ImageResources).
    - Це дозволяє інкапсулювати всю роботу з JDBC в одному місці, залишаючи доменні об'єкти "чистими".

3. Елемент (Element)

- **Клас:** Resource (абстрактний).
- **Роль:** Базовий клас для всіх ресурсів, який оголошує абстрактний метод visit(IResourceVisitor visitor). У класичній термінології патерну цей метод часто називають accept.
- **Призначення:** Гарантує, що будь-який ресурс у системі може "прийняти" відвідувача.

4. Конкретні Елементи (Concrete Elements)

- **Класи:** CssResource, JsResource, ImageResource.
- **Роль:** Об'єкти даних, що містять специфічний контент (текст скриптів, байтовий масив зображень).
- **Взаємодія (Ключовий момент):** У кожному класі метод visit реалізовано ідентично: visitor.visit(this);. Однак, завдяки поліморфізму, ключове слово this передає відвідувачу конкретний тип поточного об'єкта. Це дозволяє Java автоматично вибрати правильне перевантаження методу visit у класі ResourceSaveVisitor (наприклад, visit(ImageResource image) для зображення), уникаючи довгих ланцюжків if (obj instanceof ...).

5. Клієнт (Client)

- **Клас:** ResourceRepositoryImpl.
- **Роль:** Ініціює процес обробки.
- **Взаємодія:**
    1. Отримує з'єднання з БД.
    2. Створює екземпляр конкретного відвідувача ResourceSaveVisitor.
    3. Викликає метод resource.visit(saveVisitor) у переданого об'єкта ресурсу. Клієнту не потрібно знати, який саме підтип Resource йому передали — патерн Visitor сам скерує виконання до потрібного SQL-запиту.

Переваги архітектурного підходу

1. **Розділення обов'язків (Separation of Concerns):** Класи ресурсів відповідають лише за зберігання даних (URL, контент), а клас ResourceSaveVisitor — виключно за взаємодію з базою даних.
2. **Легкість розширення операцій:** Якщо в майбутньому знадобиться,

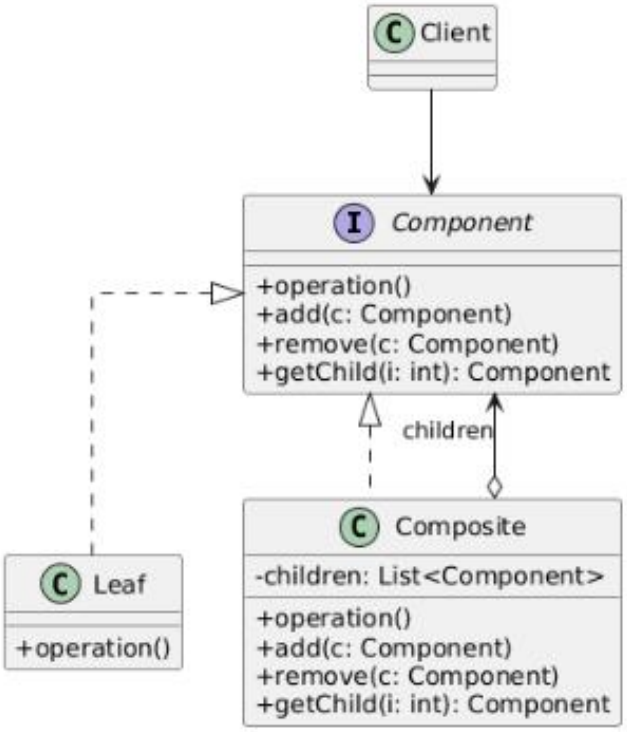
наприклад, генерувати XML-звіт по ресурсах, достатньо буде створити новий клас ResourceXmlVisitor, не змінюючи жодного рядка коду в класах JsResource чи ImageResource.

3. **Типізація:** Всі перевірки типів відбуваються на етапі компіляції завдяки перевантаженню методів, що надійніше, ніж приведення типів (casting) у рантаймі.

Контрольні запитання

1. Яке призначення шаблону «Композит»? Призначення структурного шаблону «Композит» полягає в тому, щоб дозволити клієнтському коду працювати зі складними деревоподібними структурами об'єктів так само просто, як і з одиничними об'єктами. Головна мета полягає в групуванні об'єктів у структури типу «частина-ціле», де кожен вузол може бути або простим елементом, або контейнером, що містить інші елементи. Це усуває необхідність постійно перевіряти тип об'єкта перед викликом методу, оскільки всі елементи дерева реалізують спільний інтерфейс, що значно спрощує код клієнта при роботі з рекурсивними даними, такими як файлові системи, графічні сцени або меню програм.

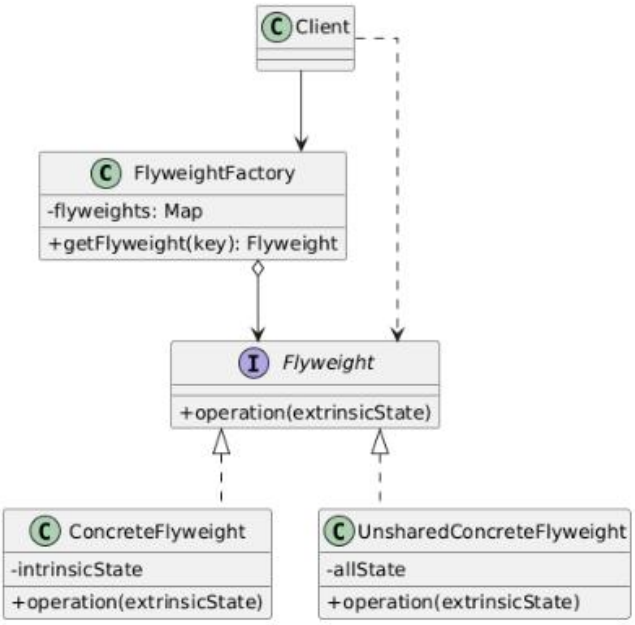
2. Нарисуйте структуру шаблону «Композит».



3. Які класи входять в шаблон «Композит», та яка між ними взаємодія? До складу шаблону входять три основні типи класів. По-перше, це Компонент, який є абстракцією (інтерфейсом або абстрактним класом) і визначає спільні операції для всіх об'єктів у композиції, як простих, так і складних. По-друге, це Листок (Leaf), який представляє кінцевий об'єкт дерева, що не має підлеглих елементів, і виконує основну роботу. По-третє, це Композит (Composite) — контейнер, який зберігає колекцію дочірніх компонентів і реалізує методи інтерфейсу, делегуючи виконання своїм нащадкам. Взаємодія відбувається наступним чином: клієнт викликає метод через спільний інтерфейс Компонента. Якщо виклик надходить до Листка, він просто виконує операцію. Якщо ж виклик отримує Композит, він проходить по списку своїх дітей і перенаправляє запит кожному з них, часто додаючи власну логіку обробки результатів, таким чином рекурсивно проходячи по всьому дереву.

4. Яке призначення шаблону «Легковаговик»? Призначення структурного шаблону «Легковаговик» полягає в ефективній підтримці великої кількості дрібних об'єктів за рахунок економії оперативної пам'яті. Це досягається шляхом відмови від зберігання повного стану в кожному об'єкті. Замість цього стан розділяється на внутрішній, який є спільним для багатьох об'єктів і зберігається в одному екземплярі, та зовнішній, який є унікальним для кожного контексту і передається об'єкту лише в момент виконання операцій. Цей підхід критично важливий для систем, де створення тисяч унікальних об'єктів призвело б до переповнення пам'яті, наприклад, у текстових редакторах для відображення символів або в іграх для відображення частинок.

5. Нарисуйте структуру шаблону «Легковаговик».

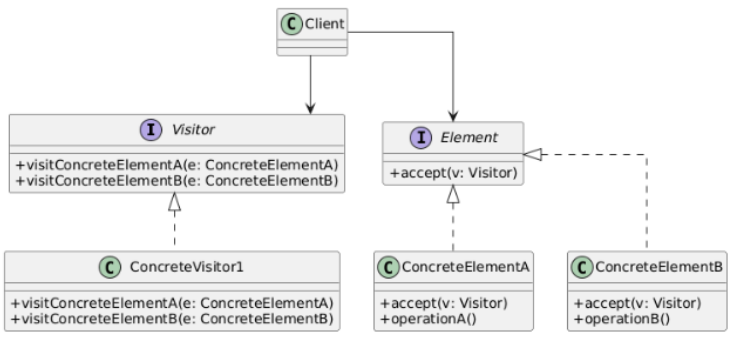


6. Які класи входять в шаблон «Легковаговик», та яка між ними взаємодія? Основним елементом є інтерфейс Легковаговика (Flyweight), який визначає методи, що приймають зовнішній стан як параметри. Конкретний Легковаговик (ConcreteFlyweight) реалізує цей інтерфейс і зберігає незмінний внутрішній стан, який розділяється між багатьма клієнтами. Ключову роль відіграє Фабрика Легковаговиків (FlyweightFactory), яка керує пулом об'єктів: вона перевіряє, чи існує вже потрібний легковаговик, і повертає його, або створює новий, якщо такого ще немає. Клієнтський код зберігає або обчислює зовнішній стан (контекст) і звертається до фабрики за об'єктами. Взаємодія полягає в тому, що клієнт отримує посилання на спільний об'єкт від фабрики і викликає його методи, передаючи унікальні дані контексту безпосередньо в аргументах методу, що дозволяє одному об'єкту обслуговувати різні частини програми.

7. Яке призначення шаблону «Інтерпретатор»? Призначення поведінкового шаблону «Інтерпретатор» полягає у визначенні граматики для простої мови та побудові механізму для інтерпретації речень цієї мови. Він використовується для задач, де дані можна представити у вигляді ієрархічної структури виразів, наприклад, для розбору математичних формул, регулярних виразів або логічних запитів. Патерн перетворює кожне правило граматики на окремий клас, що дозволяє складати нові вирази, комбінуючи об'єкти цих класів у деревоподібну структуру (абстрактне синтаксичне дерево), та обчислювати їх шляхом рекурсивного обходу.

8. Яке призначення шаблону «Відвідувач»? Призначення поведінкового шаблону «Відвідувач» — відокремити алгоритм від структури об'єктів, над якими він оперує. Це дозволяє додавати нові операції до існуючої ієрархії класів без необхідності змінювати код цих класів, що відповідає принципу відкритості/закритості. Патерн часто застосовується, коли структура об'єктів (наприклад, типи вузлів у дереві документа) стабільна і змінюється рідко, але бізнес-логіка обробки цих об'єктів (експорт, аналіз, перевірка) постійно розширюється і потребує додавання нових функцій.

9. Нарисуйте структуру шаблону «Відвідувач».



10. Які класи входять в шаблон «Відвідувач», та яка між ними взаємодія? До структури входять два паралельні набори ієрархій. Перша — це Елементи (Element), які оголошують метод прийняття відвідувача (accept), та Конкретні Елементи, що реалізують цей метод. Друга — це Інтерфейс Відвідувача (Visitor), який оголошує набір методів для відвідування кожного конкретного типу елемента, та Конкретні Відвідувачі, які містять реалізацію бізнес-логіки. Взаємодія базується на механізмі подвійної диспетчеризації: клієнт викликає метод accept на елементі та передає туди об'єкт відвідувача. Елемент, знаючи свій тип, викликає відповідний метод у відвідувача (наприклад, visitConcreteElementA), передаючи себе як аргумент. Таким чином, вибір потрібного алгоритму залежить і від типу відвідувача, і від фактичного типу елемента, що дозволяє коректно обробити дані без використання чисельних перевірок типів.

Висновки

У ході виконання лабораторної роботи було закріплено навички проєктування гнучких програмних архітектур на прикладі системи обробки веб-ресурсів. Практична реалізація засвідчила доцільність комбінування структурних та поведінкових шаблонів для оптимізації роботи зі складними ієрархіями даних. Зокрема, впровадження патерну Composite дозволило уніфікувати роботу з деревоподібними структурами (DOM-деревом або ієрархією UI-елементів), забезпечивши єдиний інтерфейс управління як простими, так і складеними компонентами. Застосування Flyweight вирішило проблему надмірного споживання оперативної пам’яті шляхом виокремлення та спільного використання незмінного стану для великої кількості однотипних об'єктів. У свою чергу, патерн Visitor дозволив відділити алгоритми обробки даних (такі як збереження в БД, експорт чи рендеринг) від самих класів сутностей, реалізувавши механізм подвійної диспетчеризації. У результаті було створено оптимізовану та розширювану систему, яка відповідає принципам SOLID, забезпечує чітке розділення відповідальностей та дозволяє додавати нові операції без модифікації існуючого коду.