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 проєктування»

«2. HTTP-сервер»

Виконала:

студентка групи - ІА-34
Мартинюк Т.В.

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.

Тема проєкту: HTTP-сервер (state, builder, factory method, mediator, composite, p2p) Сервер повинен мати можливість розпізнавати вхідні запити і формувати коректні відповіді (згідно протоколу HTTP), надавати сторінки chtml (html сторінки з додаванням найпростіших C# конструкцій на розсуд студента), вести статистику вхідних запитів, обробку запитів у багатопотоковому/подієвому режимах.

Теоретичні відомості:

- **Composite**

Патерн Composite використовується для побудови ієрархічних деревоподібних структур, у яких окремі елементи й цілі групи елементів мають однаковий інтерфейс. Це дозволяє клієнтському коду працювати з будь-яким елементом – чи то листом, чи композитом – однаковим способом. Composite спрощує обробку рекурсивних структур, дозволяє об’єднувати об’єкти у складні композиції та забезпечує легке розширення системи без змін у коді клієнта.

- **Flyweight (Пристосуванець)**

Патерн Flyweight застосовується для економії пам’яті у випадках, коли в системі існує велика кількість однотипних об’єктів. Він відокремлює спільний внутрішній стан, роблячи його розділюваним між багатьма екземплярами, а унікальний зовнішній стан зберігається окремо. Такий підхід значно зменшує витрати ресурсів і дозволяє ефективно працювати з великою кількістю логічних об’єктів. Flyweight широко використовується в графічних підсистемах, кешах, ігрових рушіях та текстових редакторах.

- **Interpreter**

Патерн Interpreter призначений для опису граматики певної мови та побудови інтерпретатора, що може обробляти та виконувати вирази цієї мови. Вирази організовуються у вигляді дерева, де кожен тип вузла представляє окреме граматичне правило. Interpreter дозволяє легко розширювати мову новими виразами та правилами, не змінюючи існуючої структури. Цей патерн застосовується в мовах конфігурацій, фільтрації, математичних та логічних обчислень, а також у доменних мовах (DSL)

- **Visitor**

Патерн Visitor дозволяє додавати нові операції до структури об’єктів, не змінюючи самі класи цих об’єктів. Він відокремлює алгоритм обробки від даних: структура залишається стабільною, а нові дії реалізуються у вигляді окремих відвідувачів. Завдяки цьому можна легко додавати нові операції без зміни основної моделі, що сприяє гнучкості й підтримуваності системи. Visitor широко використовується в компіляторах, аналізаторах синтаксичних дерев, системах валідації та інструментах обробки моделей.

Хід роботи:

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.

Проблема: у HTTP-сервері виникла потреба підтримувати вкладені маршрути та групи маршрутів, які повинні оброблятися однаковим чином через єдиний інтерфейс. Без структурного підходу система маршрутизації стає складною, важко розширюваною та вимагає від сервера знань про конкретну структуру URL-шляхів. Це призводить до дублювання коду, сильного зв’язування та порушення принципів SOLID, зокрема відкритості/закритості та інверсії залежностей.

Рішення: застосування патерну Composite, у якому абстракція RouteComponent визначає спільний інтерфейс для всіх маршрутів. Кінцеві маршрути реалізуються у вигляді RouteLeaf, а групи маршрутів — RouteComposite, що може містити інші компоненти та делегувати їм обробку. Такий підхід дозволяє представляти маршрути у вигляді дерева, ізолює логіку пошуку та обробки запитів, спрощує додавання нових гілок маршрутизації та забезпечує слабке зв'язування між Server і конкретною структурою URL.

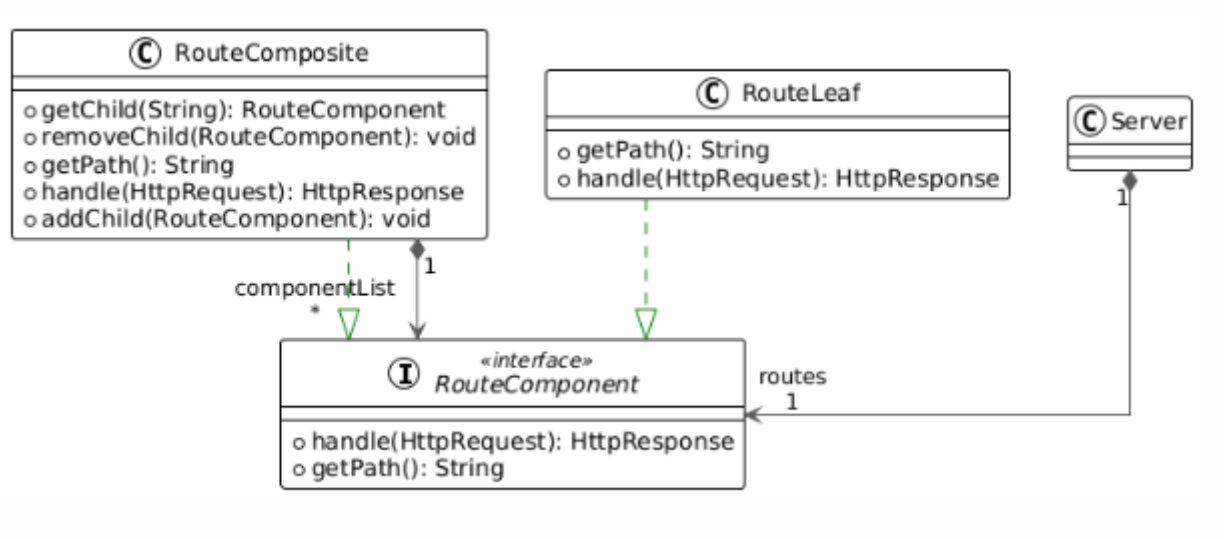


Рисунок 1 – Структура патерну проєктування «Composite»

Висновок: Під час виконання даної лабораторної роботи ми ознайомились з структурою та принципами роботи шаблонів проєктування . У даній системі було застосовано патерн «Composite», оскільки маршрутизація HTTP-запитів природно утворює ієрархічну структуру, де окремі кінцеві маршрути та цілі групи маршрутів повинні оброблятися однаковим способом. Без цього підходу серверу довелося б підтримувати складні вкладені конструкції вручну, працюючи з різними типами маршрутів окремо, що ускладнювало б розширення системи та призводило до дублювання логіки.

Використання Composite дозволило представити маршрути у вигляді дерева, де RouteLeaf і RouteComposite реалізують спільний інтерфейс RouteComponent, завдяки чому сервер працює з маршрутизацією абстрактно, не знаючи про деталі її структури. Це забезпечило гнучке рекурсивне делегування

обробки, спростило додавання нових маршрутів та груп, а також підвищило модульність і відповідність принципам SOLID.

Лістинг коду

```
public interface RouteComponent {
   public HttpResponse handle (HttpRequest request);
   public String getPath();
}


public class RouteComposite implements RouteComponent{
   List<RouteComponent> componentList = new ArrayList<>();
   String path;

   public RouteComposite(String path){
      this.path = path;
   }

   @Override
   public HttpResponse handle(HttpRequest request){
      String requestPath = request.getPath();

      if(!requestPath.startsWith(path)){
         return null;
      }

      String remainingPath = requestPath.substring(path.length());

      for (RouteComponent component : componentList) {
         if (remainingPath.startsWith(component.getPath())) {
            return component.handle(
                  new HttpRequest(
                        request.getMethod(),
                        remainingPath,
                        request.getHeaders(),
                        request.getBody()
                  )
            );
         }
      }
      return null;
   }
   @Override
   public String getPath() {
      return path;
   }

   public void addChild(RouteComponent component) {
      if (componentList.stream().anyMatch(c -> c.getPath() == component.getPath())) {
         throw new IllegalArgumentException("Path is already taken");
```

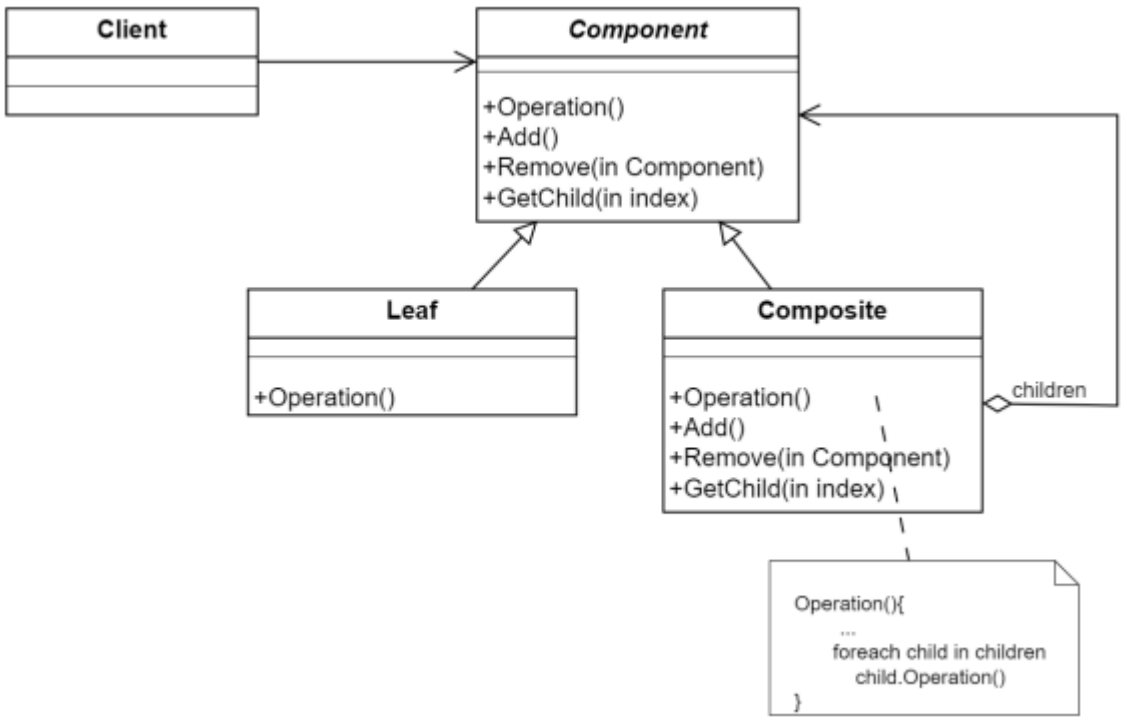
```
        }
        componentList.add(component);
    }

    public void removeChild(RouteComponent component) {
        componentList.remove(component);
    }

    public RouteComponent getChild(String path) {
        return componentList.stream()
                .filter(c -> c.getPath() == path)
                .findFirst()
                .orElse(null);
    }
}



public class RouteLeaf implements RouteComponent {
    String path;
    HttpRoute route;
    public RouteLeaf(String path,  HttpRoute route) {
        this.path = path;
        this.route = route;
    }
    @Override
    public HttpResponse handle(HttpRequest request) {
        if (route == null ||
                !request.getPath().replaceAll("/", "")
                        .equals(path.replaceAll("/", ""))) {
            return null;
        }
        return route.execute(request);
    }
    @Override
    public String getPath() {
        return path;
    }
}
```

Контрольні запитання

1. **Яке призначення шаблону «Композит»?**

Шаблон використовується для складання об'єктів в деревоподібну структуру для подання ієрархій типу «частина цілого». Даний шаблон дозволяє уніфіковано обробляти як поодинокі об'єкти, так і об'єкти з вкладеністю.

2. **Нарисуйте структуру шаблону «Композит».**



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

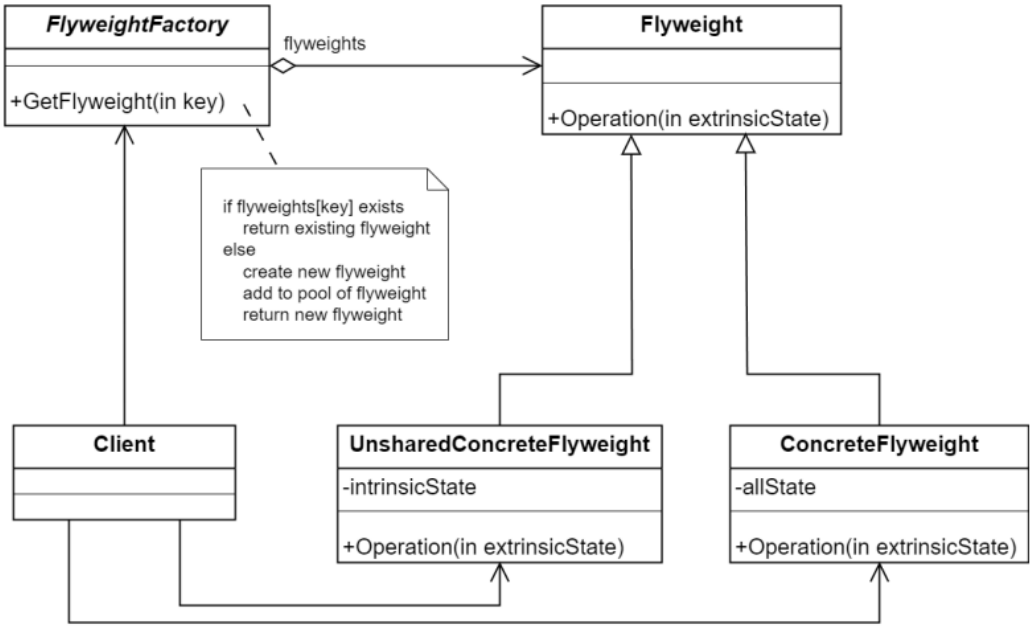
- **Component** – абстрактний інтерфейс або базовий клас, який визначає спільні операції для всіх елементів структури (як простих, так і складених).
- **Leaf** – кінцевий елемент дерева, який не містить інших компонентів і реалізує базову поведінку.
- **Composite** – складений елемент, що містить інші об’єкти Component та реалізує операції додавання, видалення і доступу до дочірніх елементів. Composite делегує роботу своїм дітям і може організовувати їх у деревоподібну структуру.
- **Client** – працює з усіма елементами через інтерфейс Component, не розрізняючи, має він справу з Leaf чи Composite.

Клієнт викликає операції через Component. Якщо об’єкт є Leaf, методи виконуються безпосередньо. Якщо це Composite, він передає виконання відповідним дочірнім елементам, часто рекурсивно. Завдяки спільному інтерфейсу Leaf і Composite обробляються однаково, що дозволяє будувати й опрацьовувати складні деревоподібні структури без ускладнення коду клієнта.

4. **Яке призначення шаблону «Легковаговик»?**

Шаблон використовується для зменшення кількості об'єктів в додатку шляхом поділу цих об'єктів між ділянками додатку. Flyweight являє собою поділюваний об'єкт. Дуже важливою є концепція «внутрішнього» і «зовнішнього» станів. Внутрішній стан відображає дані, характерні саме поділюваному об'єкту 92 (наприклад, код букви); зовнішній стан несе інформацію про його застосування в додатку (наприклад, рядок і стовпчик). Внутрішній стан зберігається в самому поділюваному об'єкті, зовнішній – в об'єктах додатку (контексту використання поділюваного об'єкта)

5. **Нарисуйте структуру шаблону «Легковаговик».**



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

- **Flyweight** — інтерфейс або абстрактний клас, який визначає операції для об’єктів, що можуть розділяти спільний внутрішній стан.
- **ConcreteFlyweight** — конкретна реалізація легковагового об’єкта, який містить *внутрішній (інваріантний)* стан, спільний для багатьох логічних об’єктів.
- **UnsharedConcreteFlyweight** (необов’язковий) — об’єкти, які не розділяють свій стан, але можуть використовувати Flyweight у своїй структурі.

- **FlyweightFactory** — фабрика, яка створює та керує Flyweight-об’єктами. Вона забезпечує повторне використання вже існуючих легковагових об’єктів замість створення нових.
- **Client** — зберігає *зовнішній (змінний)* стан і використовує Flyweight для виконання операцій.

Клієнт звертається до FlyweightFactory, щоб отримати легковаговий об’єкт. Фабрика повертає або новий ConcreteFlyweight, або вже існуючий, якщо він підходить для повторного використання. Клієнт додає до нього свій зовнішній стан і викликає операції Flyweight. Таким чином внутрішній стан розділяється багатьма об’єктами, а змінний зовнішній передається під час виконання, що значно скорочує використання пам’яті.

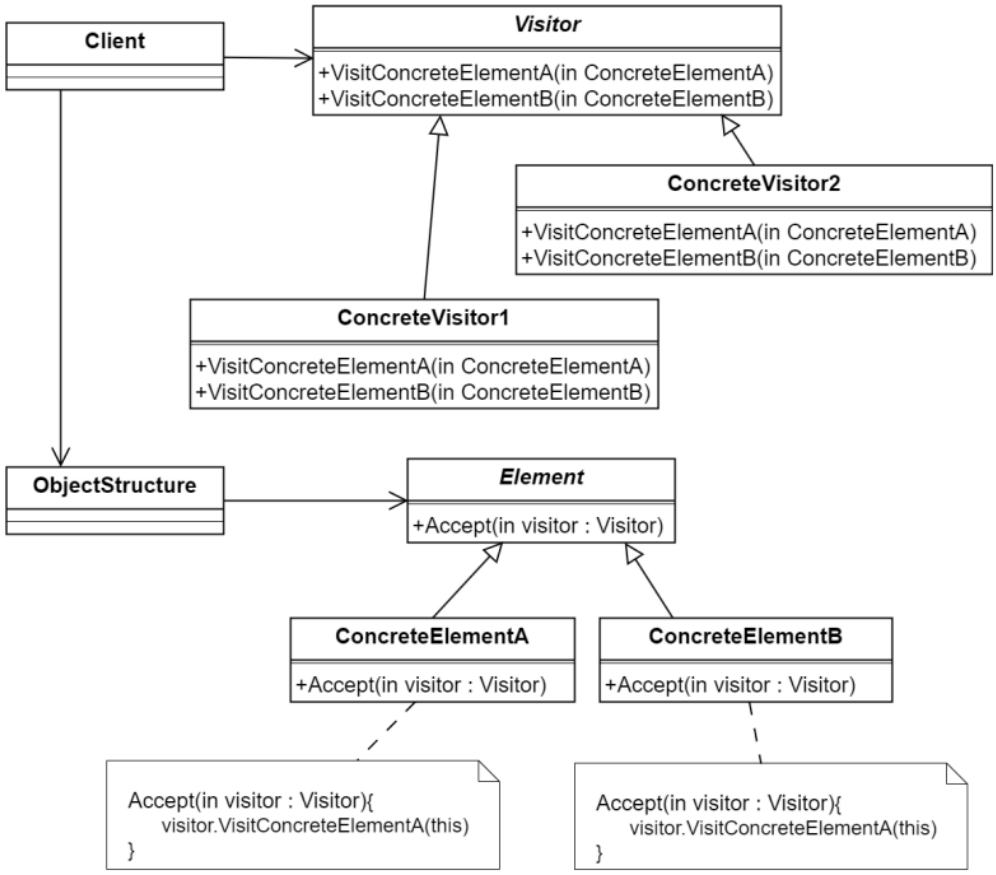
7. Яке призначення шаблону «Інтерпретатор»?

Даний шаблон використовується для подання граматики і інтерпретатора для вибраної мови (наприклад, скриптової). Граматика мови представлена термінальними і нетермінальними символами, кожен з яких інтерпретується в контексті використання. Клієнт передає контекст і сформовану пропозицію в використовувану мову в термінах абстрактного синтаксичного дерева (деревоподібна структура, яка однозначно визначає ієрархію виклику підвиразів), кожен вираз інтерпретується окремо з використанням контексту. У разі наявності дочірніх виразів, батьківський вираз інтерпретує спочатку дочірні (рекурсивно), а потім обчислює результат власної операції.

8. Яке призначення шаблону «Відвідувач»?

Шаблон відвідувач дозволяє вказувати операції над елементами без зміни структури конкретних елементів. 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.

9. Нарисуйте структуру шаблону «Відвідувач».



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

- **Visitor** – інтерфейс або абстрактний клас, який оголошує набір методів visitX(), по одному для кожного типу елементів, які може обробляти відвідувач.
- **ConcreteVisitor** – конкретні реалізації відвідувачів, що містять різні операції, які виконуються над елементами структури. Кожен відвідувач може реалізувати іншу поведінку, не змінюючи самих елементів.
- **Element –** інтерфейс для елементів структури, який містить метод accept(Visitor visitor).
- **ConcreteElement** – конкретні елементи структури, які реалізують метод accept() і передають себе відвідувачу, викликаючи відповідний метод visit.
- **ObjectStructure** – сукупність елементів, яким можна передавати відвідувача для обходу або обробки (наприклад, колекція вузлів дерева).

Кожен елемент структури має метод accept(Visitor), який приймає відвідувача та викликає відповідний метод visitX(this) у ньому. Таким чином логіка обробки виноситься у відвідувача, а структура елементів залишається незмінною. Клієнт створює ConcreteVisitor і передає його у всі елементи структури через ObjectStructure. Це дозволяє легко додавати нові операції без модифікації класів елементів, що особливо корисно для складних і стабільних структур даних.