Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

Лабораторна робота №6

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 «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator» 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:

- **Abstract Factory**

Патерн *Abstract Factory* визначає інтерфейс для створення сімейства пов’язаних об’єктів без уточнення їх конкретних класів. Він дозволяє змінювати цілі групи продуктів, не змінюючи клієнтський код. Кожна конкретна фабрика створює набір продуктів, які логічно належать до однієї платформи або конфігурації. Це забезпечує узгодженість створюваних об’єктів і спрощує підтримку системи. Abstract Factory часто застосовується, коли необхідно підтримувати кілька варіантів інтерфейсу, теми, або конфігурацій. Така структура робить систему масштабованою та легко змінюваною.

- **Factory Method**

Патерн *Factory Method* дозволяє делегувати створення об’єктів підкласам шляхом визначення фабричного методу в базовому класі. Клієнт працює з абстракцією і не залежить від конкретних реалізацій продуктів. Кожен підклас фабрики перевизначає метод створення, повертаючи свій тип продукту. Завдяки цьому система легко розширюється додаванням нових типів об’єктів без змін у клієнтському коді. Factory Method усуває жорстке зв’язування між творцем і продуктом та підсилює дотримання принципу відкритості/закритості. Його часто

застосовують у фреймворках або системах із змінною логікою створення об’єктів.

- **Memento**

Патерн *Memento* дозволяє зберігати та відновлювати внутрішній стан об’єкта без порушення інкапсуляції. Об’єкт Originator створює знімок свого стану, який зберігається у Memento. Керування колекцією збережених станів здійснює Caretaker, не маючи доступу до внутрішніх деталей. Завдяки цьому можна повертатися до попередніх версій об’єкта, реалізовуючи «undo/redo» або контрольні точки. Патерн корисний у редакторах, іграх, системах керування транзакціями. Він зберігає інкапсуляцію та дозволяє оперувати історією змін без ризику порушити стан об’єкта.

- **Observer**

Патерн *Observer* формує залежність «один до багатьох», коли зміна стану одного об’єкта автоматично повідомляє всіх його підписників. Об’єкт видавець (Subject) підтримує список спостерігачів (Observers) та викликає їх при оновленні стану. Це забезпечує слабке зв’язування між об’єктами і дозволяє динамічно додавати або видаляти підписників. Патерн часто використовують у подієвих системах, інтерфейсах користувача та реактивних механізмах. Система стає більш гнучкою та здатною реагувати на зміни в реальному часі. Observer дозволяє розподілити відповідальність між компонентами без прямої залежності.

- **Decorator**

Патерн *Decorator* дозволяє динамічно додавати нові функції об’єктам без зміни їхнього класу. Він обгортає базовий об’єкт у додатковий клас-декоратор, який розширює його поведінку. Такий підхід зберігає інтерфейс об’єкта та дозволяє комбінувати кілька декораторів у стек. Це робить систему значно гнучкішою, ніж використання спадкування. Патерн особливо корисний, коли потрібно додавати поведінку не всім підкласам, а лише окремим екземплярам. Decorator застосовується у потоках вводу/виводу, форматуванні, логуванні та всіх випадках, де поведінку потрібно розширювати «на льоту».

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: HTTP-сервер повинен обробляти запити різними типами обробників (Handler), але не може бути жорстко прив’язаний до конкретних класів або до логіки їх створення. Без цього сервер стає складним у розширенні та порушує принципи SOLID.

Рішення: застосування патерну *Factory Method* у вигляді абстрактного класу Middleware, який містить метод createHandler(). Конкретні підкласи Middleware (InternalMiddleware) перевизначають фабричний метод та створюють власні типи обробників (InternalHandler). Це дозволяє ізолювати логіку створення Handler від механізму обробки HTTP-запиту, спрощує розширення системи та забезпечує слабке зв’язування.

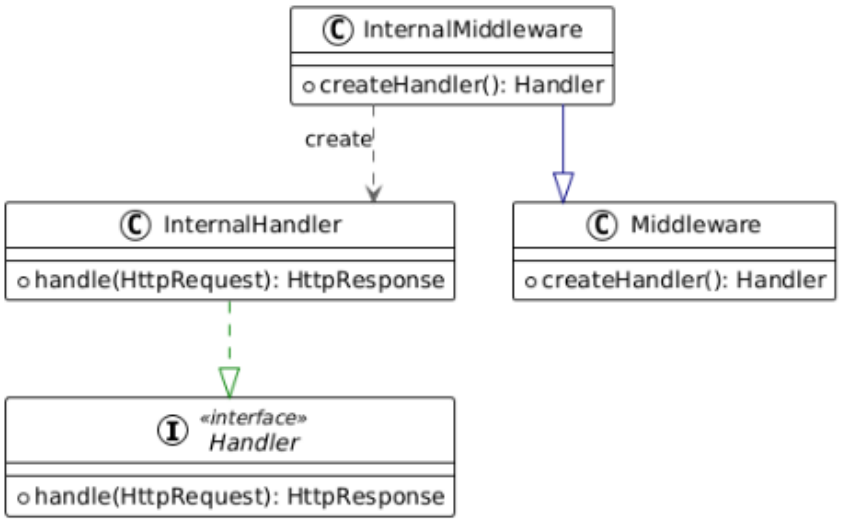


Рисунок 1 – Структура патерну проєктування «Factory Method»

Висновок: У ході виконання лабораторної роботи було досліджено структуру та принципи роботи шаблонів проєктування. Зокрема **Factory Method**, його було також застосовано його для побудови гнучкої архітектури програмної системи. У межах системи фабричний метод дав змогу винести логіку створення обробників HTTP-запитів (Handler) у окремий компонент – Middleware, що усунуло жорстке зв'язування між сервером та конкретними реалізаціями обробників. Завдяки цьому сервер працює виключно з абстракцією Handler, а вибір конкретного типу обробника делегується підкласам Middleware. Такий підхід спростив розширення функціональності: для додавання нового типу обробника достатньо створити новий Middleware із власною реалізацією фабричного методу, не змінюючи код основного сервера. Запровадження Factory Method покращило модульність, забезпечило слабке зв'язування між компонентами та дотримання принципів SOLID, зокрема інверсії залежностей та відкритості/закритості. У результаті архітектура HTTP-сервера стала більш гнучкою, масштабованою та зручною для подальшої модифікації.

Лістинг коду

```
public abstract class Middleware {
    public abstract Handler createHandler();
}


package src.main;

import java.util.Map;

public class InternalMiddleware extends Middleware{
    RouteComponent routes;
    public InternalMiddleware(RouteComponent routes){
        this.routes = routes;
    }
    @Override
    public Handler createHandler() {
        return new InternalHandler(routes);
    }
}
```

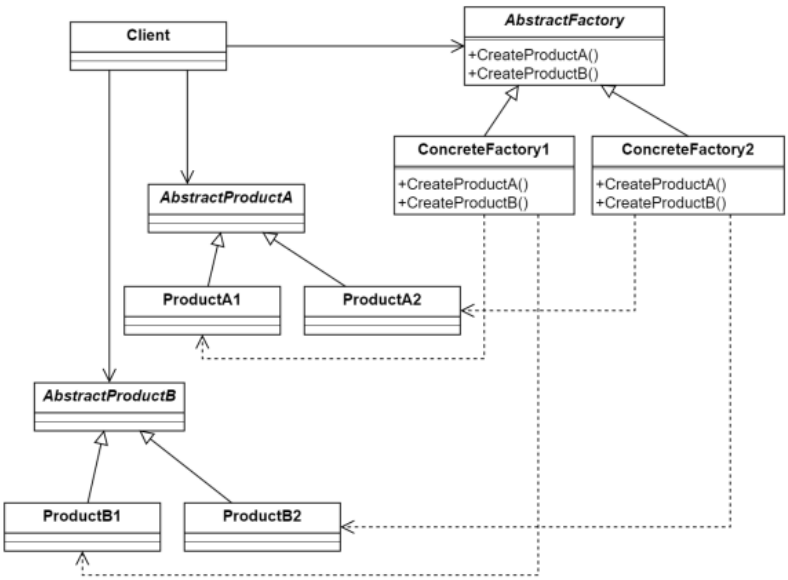
```
public interface Handler {
   public HttpResponse handle(HttpRequest request);
}


package src.main;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Map;

public class InternalHandler implements Handler {


   private final Logger logger = new Logger();

   private final FileHandler fileHandler = new FileHandler();

   RouteComponent routes;
   public InternalHandler(RouteComponent routes){
      this.routes=routes;
   }
   @Override
   public HttpResponse handle(HttpRequest request) {
      HttpResponse response;
      logger.info("Handling request for path: " +
            request.getPath());
      response = routes.handle(request);
      String path = request.getPath();
      if (response!=null) {
         logger.info("Route found for path: " + path);
      }else {
         if(path.equals("/")) {
            path = "/index.html";
         }
         try{
            String fileContent =
                  fileHandler.readFile("src/main/resources" + path);
            logger.info("File found for path: " + path);
            response = HttpResponseDirector.Ok(fileContent,
                  request.getHeaders());
         }
         catch (IOException fileNotFoundException) {
            logger.warning("File not found for path: " + path);
            response =
                  HttpResponseDirector.NotFound(request.getHeaders());
         }
      }
      return response;
   }
}
```

Контрольні запитання

1. **Яке призначення шаблону «Абстрактна фабрика»?**

Шаблон «Абстрактна фабрика» використовується для створення сімейств об'єктів без вказівки їх конкретних класів. Для цього виноситься загальний інтерфейс фабрики (AbstractFactory) і створюються його реалізації для різних сімейств продуктів.

2. **Нарисуйте структуру шаблону «Абстрактна фабрика».**



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

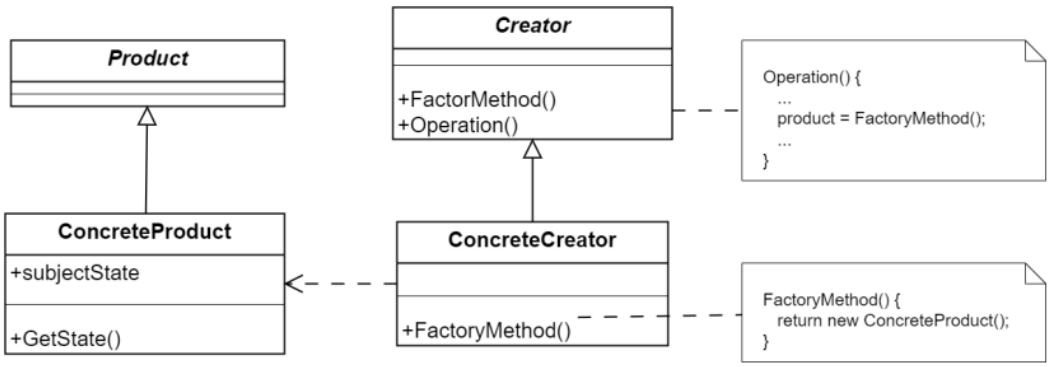
- **AbstractFactory** – абстрактний інтерфейс, який оголошує набір методів для створення групи пов’язаних продуктів.
- **ConcreteFactory** – конкретні фабрики, які реалізують методи AbstractFactory та створюють конкретні варіанти продуктів певного сімейства.
- **AbstractProduct** – абстракції продуктів, які визначають загальний інтерфейс для об’єктів одного типу.
- **ConcreteProduct** – конкретні продукти, які створюють ConcreteFactory і які належать до одного сімейства.
- **Client** – використовує AbstractFactory та оперує лише абстракціями продуктів, не знаючи про їх конкретні реалізації.

Клієнт звертається до AbstractFactory, щоб отримати об’єкти через фабричні методи. Конкретна фабрика повертає відповідні конкретні продукти, які сумісні між собою, але клієнт працює лише з абстрактними інтерфейсами. Завдяки цьому можна замінювати цілі сімейства продуктів, змінюючи лише фабрику, без змін у клієнтському коді.

4. **Яке призначення шаблону «Фабричний метод»?**

Шаблон «Фабричний метод» визначає інтерфейс для створення об'єктів певного базового типу. Це зручно, коли хочеться додати можливість створення об'єктів не базового типу, а деякого дочірнього. Фабричний метод у такому разі є зачіпкою для впровадження власного конструктора об'єктів. Основна ідея полягає саме в заміні об'єктів їх підтипами, що при цьому зберігає ту ж функціональність; інша частина поведінки об'єктів не є інтерфейсною (AnOperation) і дозволяє взаємодіяти із створеними об'єктами як з об'єктами базового типу.

5. **Нарисуйте структуру шаблону «Фабричний метод».**



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

- **Product** — абстрактний продукт, що задає інтерфейс об’єктів, які створює фабрика.
- **ConcreteProduct** — конкретні реалізації продукту, які повертаються фабричним методом.

- **Creator** (або *Factory*) — базовий клас, який оголошує фабричний метод createProduct(). Він може містити базовий алгоритм, що використовує продукт.
- **ConcreteCreator** — підкласи, які перевизначають фабричний метод, створюючи конкретні продукти.

Клієнт працює з Creator та викликає його методи, не знаючи, який саме ConcreteProduct буде створено. Creator використовує фабричний метод createProduct() для створення об’єкта Product, а конкретний тип продукту визначається ConcreteCreator. Таким чином, логіка створення об’єктів делегується підкласам, що забезпечує гнучкість і дозволяє додавати нові типи продуктів без зміни клієнтського коду.

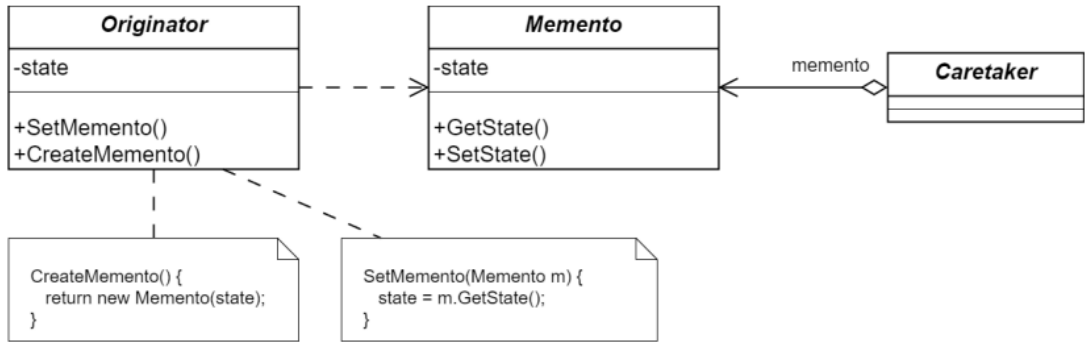
7. **Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»?**

- Абстрактна фабрика створює сімейства пов’язаних продуктів, тоді як Фабричний метод створює один продукт певного типу.
- У Абстрактній фабриці є кілька фабричних методів, кожен з яких створює свій продукт; у Фабричному методі – зазвичай один фабричний метод, який підкласи перевизначають.
- Абстрактна фабрика забезпечує повну заміну сімейства продуктів, змінюючи лише конкретну фабрику; Фабричний метод дозволяє замінювати лише один конкретний продукт.
- Абстрактна фабрика працює на рівні об’єктних сімейств, а Фабричний метод – на рівні класа та його підкласів.
- У Фабричному методі клієнт зазвичай розширює Creator, щоб створити продукт; у Абстрактній фабриці клієнт просто використовує фабрику через інтерфейс, без успадкування.
- Фабричний метод – це механізм розширення; Абстрактна фабрика – це механізм організації родин взаємопов’язаних продуктів.

8. Яке призначення шаблону «Знімок»?

Шаблон використовується для збереження і відновлення стану об'єктів без порушення інкапсуляції. Об'єкт «Memento» служить виключно для збереження змін над початковим об'єктом (Originator). Лише початковий об'єкт має можливість зберігати і отримувати стан об'єкту «Memento» для власних цілей, цей об'єкт є «порожнім» для кого-небудь ще. Об'єкт «Caretaker» використовується для передачі і зберігання мементо об'єктів в системі.

9. Нарисуйте структуру шаблону «Знімок».



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

- **Originator** – об’єкт, стан якого потрібно зберігати та відновлювати. Він створює знімки свого стану та вміє відновлюватися з них.
- **Memento** – об’єкт-знімок, який інкапсулює внутрішній стан Originator у певний момент часу. Його структура прихована від інших класів, щоб не порушувати інкапсуляцію.
- **Caretaker** – керує збереженням та історією Memento. Він зберігає знімки, але не має доступу до їхнього внутрішнього стану.

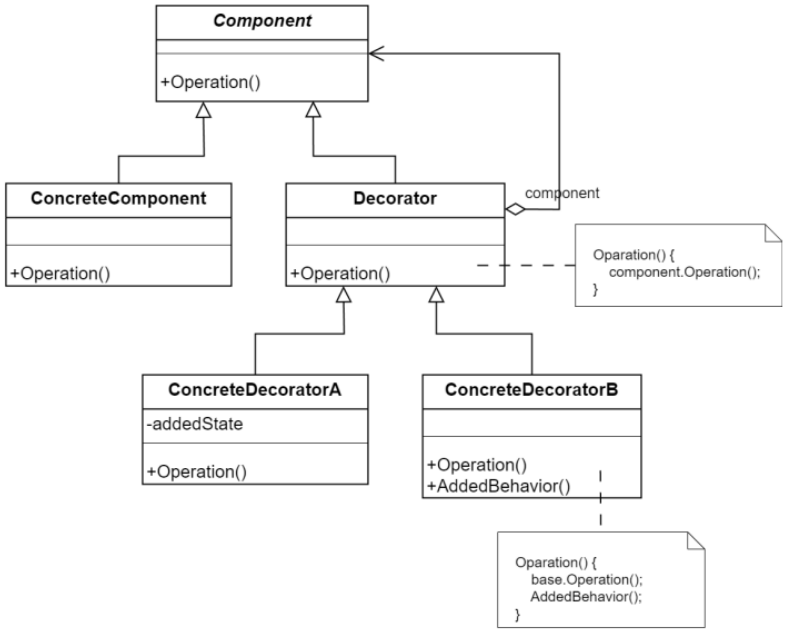
Originator створює Memento, яке зберігається Caretaker. Коли потрібно повернутися до попереднього стану, Caretaker передає відповідний Memento назад Originator, а той відновлює свій стан за даними знімка. Таким чином Caretaker управляє історією, Originator – станом, а Memento – гарантує інкапсуляцію.

11. Яке призначення шаблону «Декоратор»?

Шаблон призначений для динамічного додавання функціональних можливостей об'єкту під час роботи програми. Декоратор деяким чином

«обертає» (за рахунок агрегації) початковий об'єкт зі збереженням його функцій, проте дозволяє додати додаткові дії. Такий шаблон надає гнучкіший спосіб зміни поведінки об'єкту чим просте спадкоємство, оскільки початкова функціональність зберігається в повному об'ємі. Більше того, таку поведінку можна застосовувати до окремих об'єктів, а не до усієї системи в цілому.

12. Нарисуйте структуру шаблону «Декоратор».



13. Які класи входять в шаблон «Декоратор», та яка між ними взаємодія?

- **Component** — абстрактний інтерфейс або базовий клас, який визначає спільні операції для всіх об’єктів, що можуть бути розширені.
- **ConcreteComponent** — конкретний об’єкт, чию функціональність потрібно динамічно розширювати.
- **Decorator** — абстрактний клас-обгортка, який реалізує інтерфейс Component і містить посилання на інший об’єкт Component.
- **ConcreteDecorator** — конкретні декоратори, які додають нову поведінку до об’єкта, викликаючи базову реалізацію та доповнюючи її власною логікою.

Клієнт працює з об’єктами через інтерфейс Component, не знаючи, чи об’єкт є звичайним компонентом, чи обгорнутим у декоратори. ConcreteComponent

забезпечує основну поведінку, а Decorator обгортає його і передає виклики, додаючи нові функції до або після базової операції. Багато декораторів можуть обгортати один одного, утворюючи стек розширень і забезпечуючи динамічне комбінування поведінки без зміни класів.

14. Які є обмеження використання шаблону «декоратор»?

- **Ускладнення структури системи.** Через велику кількість дрібних декораторів код стає важчим для розуміння, а ієрархія — заплутаною.
- **Важко відстежити порядок застосування декораторів.** Оскільки декоратори обгортають один одного, складно визначити, яка саме комбінація поведінок була застосована.
- **Не можна гарантувати правильну композицію всіх декораторів.** Деякі декоратори можуть потребувати конкретну послідовність або не поєднуватися один з одним.
- **Зміна інтерфейсу компонента ускладнюється.** Якщо базовий інтерфейс Component змінюється, доведеться переробляти всі декоратори, що порушує принцип відкритості/закритості.
- **Перевірка типів стає складнішою.** Через вкладеність декораторів важче визначати реальний тип об’єкта або виконувати спеціалізовані операції.
- **Можлива надмірна кількість об’єктів у пам’яті.** Кожен рівень декоратора створює новий об’єкт-обгортку, що збільшує навантаження на систему.