Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

Лабораторна робота №5

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 «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype» 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

1. Adapter

Шаблон Adapter дозволяє узгодити роботу класів із різними інтерфейсами. Його суть полягає в тому, що створюється окремий клас-посередник — адаптер, який “перекладає” виклики одного інтерфейсу в інший. Наприклад, якщо клієнт очікує метод getData(), а наявний клас має лише readInfo(), адаптер реалізує getData(), але всередині викликає readInfo(). Таким чином, змінювати код клієнта або стороннього класу не потрібно. Часто застосовується, коли потрібно інтегрувати старі або сторонні бібліотеки у нову систему, забезпечуючи зворотну сумісність без зміни логіки роботи.

2. Builder

Builder використовується для створення складних об’єктів крок за кроком, коли їх ініціалізація передбачає багато параметрів або варіантів конфігурації. Створення об’єкта поділяється на кілька етапів — наприклад, побудову компонентів, налаштування властивостей, завершення конструкції. Є “директор”, який визначає порядок викликів методів будівельника, і “конкретний будівельник”, який реалізує ці кроки. Результатом є об’єкт, готовий до використання. Такий підхід спрощує контроль за створенням складних структур,

робить код зрозумілішим і дозволяє створювати різні варіації об’єкта, використовуючи один і той самий алгоритм побудови.

3. **Command**

Command інкапсулює дію у вигляді окремого об’єкта, який містить всю необхідну інформацію для виконання операції — метод, параметри, отримувача. Замість того щоб викликати метод напряму, програма створює об’єкт команди з методом execute(). Це дає можливість ставити команди у чергу, скасовувати або повторювати виконання. Наприклад, у текстовому редакторі кожна дія користувача (копіювання, вставлення, видалення) реалізується як окрема команда. Коли користувач натискає “Undo”, програма просто викликає undo() у потрібній команді. Такий підхід роз’єднує відправника і виконавця дії та спрощує керування операціями.

4. **Chain of Responsibility**

У шаблоні Chain of Responsibility кілька об’єктів з’єднуються у ланцюг, через який передається запит. Кожен об’єкт-обробник вирішує, чи може він обробити запит, і якщо ні — передає його далі. Така структура дозволяє створити гнучку систему, у якій додавання нового обробника не потребує змін у коді клієнта. Наприклад, у веб-сервері ланцюг може складатися з фільтрів авторизації, кешування, логування тощо. Якщо один із фільтрів не здатен виконати запит, він передає його наступному у черзі. Це спрощує розширення системи й мінімізує зв’язність між компонентами.

5. **Prototype**

Prototype базується на створенні нових об’єктів шляхом клонування вже існуючих. Замість створення об’єкта через конструктор, система копіює готовий екземпляр-прототип із заданими параметрами. Це особливо зручно, коли створення об’єкта є складним або ресурсомістким процесом, наприклад, при генерації великих структур даних або графічних елементів. Кожен клас-прототип реалізує метод clone(), який повертає новий об’єкт з тими ж властивостями. Такий підхід дає змогу створювати нові об’єкти під час виконання програми, не

знаючи наперед, якого саме типу вони будуть, і полегшує динамічну зміну або копіювання станів.

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. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону

Проблема:

Процес формування відповіді у HTTP-сервері складається з багатьох елементів – статус-коду, заголовків, тіла відповіді, службових полів тощо. Формування кожного з них відбувається поетапно, залежно від типу запиту (наприклад, успішна відповідь, помилка, створення ресурсу тощо). Якщо створювати такі об’єкти безпосередньо, код стає складним для розширення, а будь-які зміни у структурі відповіді призводять до модифікації всієї логіки обробки запитів.

Рішення:

Для спрощення та поетапного створення складних HTTP-відповідей застосовано шаблон проєктування **Builder**. Він дозволяє відокремити процес побудови об’єкта HttpResponse від його кінцевого представлення. Завдяки цьому формується стандартна структура відповіді, що містить статус-код, заголовки та тіло. Клас HttpResponseBuilder відповідає за створення частин відповіді, а HttpResponseDirector визначає типові сценарії побудови – Ok, NotFound, InternalServerError, BadRequest. Такий підхід забезпечує узгодженість усіх HTTP-відповідей, спрощує додавання нових типів і підвищує гнучкість сервера.

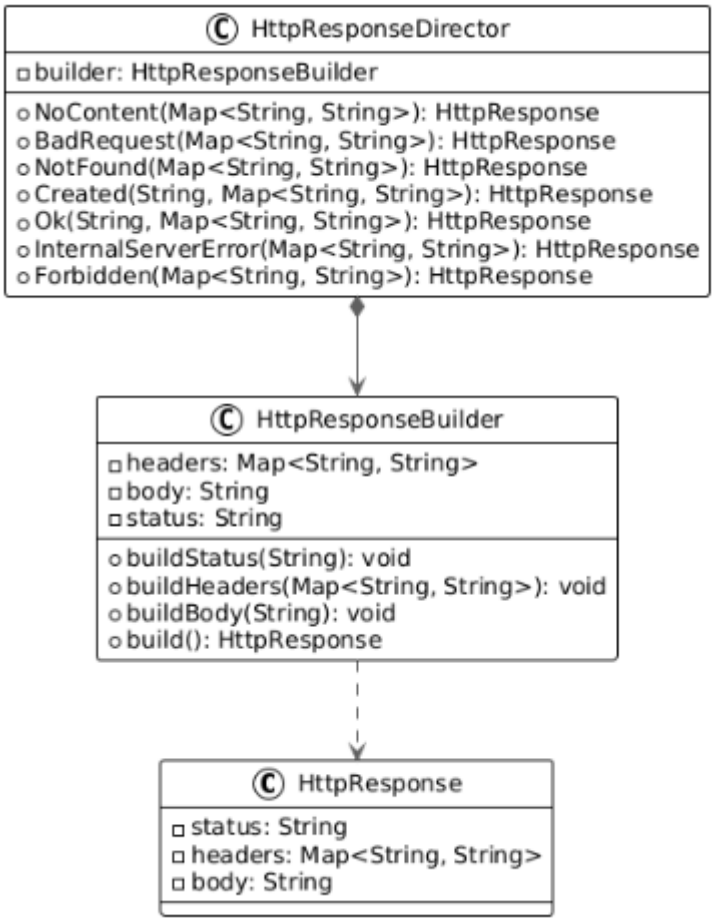


Рисунок 1 – Структура патерну проєктування «Builder»

Висновок: У даному проєкті застосування шаблону проєктування Builder дало змогу впорядкувати процес створення HTTP-відповідей та зробити його гнучким і розширюваним. Завдяки відокремленню процесу побудови від кінцевого представлення об’єкта забезпечується єдина структура відповіді незалежно від її типу (успіх, помилка, перенаправлення тощо).

Таке рішення спрощує підтримку коду, підвищує зрозумілість логіки обробки запитів і дозволяє легко додавати нові типи HTTP-відповідей без зміни існуючої архітектури сервера. Отже, використання патерну Builder є доцільним для формування складних багатокомпонентних об’єктів у серверних застосунках, де важлива стандартизація та повторне використання коду.

```
public class HttpResponse {

        private final String status;
        private final Map<String, String> headers;
        private final String body;

        public HttpResponse(String status, Map<String, String> headers, String body) {
            this.status = status;
            this.headers = headers;
            this.body = body;
        }
}


import java.util.Map;
public class HttpResponseDirector {

    private static final HttpResponseBuilder builder = new HttpResponseBuilder();

    public static HttpResponse Ok(String body, Map<String, String> headers) {
        builder.buildStatus("200 OK");
        builder.buildHeaders(headers);
        builder.buildBody(body);
        return builder.build();
    }

    public static HttpResponse NotFound(Map<String, String> headers) {
        builder.buildStatus("404 Not Found");
        builder.buildHeaders(headers);
        builder.buildBody(null);
        return builder.build();
    }

    public static HttpResponse InternalServerError(Map<String, String> headers) {
        builder.buildStatus("500 Internal Server Error");
        builder.buildHeaders(headers);
        builder.buildBody(null);
        return builder.build();
    }

    public static HttpResponse BadRequest(Map<String, String> headers) {
        builder.buildStatus("400 Bad Request");
```

```
        builder.buildHeaders(headers);
        builder.buildBody(null);
        return builder.build();
    }

    public static HttpResponse Created(String body, Map<String, String> headers) {
        builder.buildStatus("201 Created");
        builder.buildHeaders(headers);
        builder.buildBody(body);
        return builder.build();
    }

    public static HttpResponse NoContent(Map<String, String> headers) {
        builder.buildStatus("204 No Content");
        builder.buildHeaders(headers);
        builder.buildBody(null);
        return builder.build();
    }

    public static HttpResponse Forbidden(Map<String, String> headers) {
        builder.buildStatus("403 Forbidden");
        builder.buildHeaders(headers);
        builder.buildBody(null);
        return builder.build();
    }
    public static HttpResponse ServiceUnavailable(String body, Map<String,String> headers) {
        builder.buildStatus("503 Service Unavailable");
        builder.buildHeaders(headers);
        builder.buildBody(body);
        return builder.build();
    }
}


public class HttpResponseBuilder {
    private String status;
    private Map<String, String> headers;
    private String body;

    public void buildStatus(String status) {
        this.status = status;
    }

    public void buildHeaders(Map<String, String> headers) {
        this.headers = headers;
    }
```

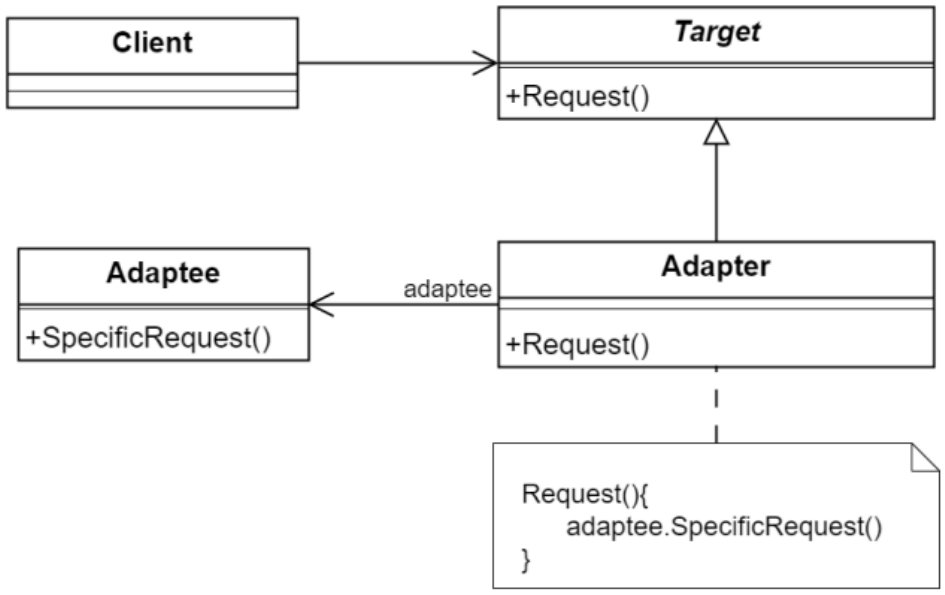
```
    public void buildBody(String body) {
        this.body = body;
    }

    public HttpResponse build() {
        return new HttpResponse(status, headers, body);
    }
}
```

Контрольні питання:

1. Яке призначення шаблону «Адаптер»?

Призначення патерну: Шаблон "Adapter" (Адаптер) використовується для адаптації інтерфейсу одного об'єкту до іншого . Наприклад, існує декілька бібліотек для роботи з принтерами, проте кожна має різний інтерфейс (хоча однакові можливості і призначення). Має сенс розробити уніфікований інтерфейс (сканування, асинхронне сканування, двостороннє сканування, потокове сканування і тому подібне), і реалізувати відповідні адаптери для приведення бібліотек до уніфікованого інтерфейсу. Це дозволить в програмі звертатися до загального інтерфейсу, а не приводити різні сценарії роботи залежно від способу реалізації бібліотеки. Адаптери також називаються "wrappers" (обгортками).

2. Нарисуйте структуру шаблону «Адаптер».



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

- Client – викликає методи через інтерфейс, який він знає.
- Target – інтерфейс, який очікує клієнт.
- Adaptee – існуючий клас із несумісним інтерфейсом.

- Adapter – клас, що реалізує Target і всередині містить об’єкт типу Adaptee.

Клієнт звертається до адаптера через інтерфейс Target. Адаптер приймає виклик, перетворює його у форму, зрозумілу Adaptee, і передає далі. Adaptee виконує операцію та повертає результат через адаптер клієнту.

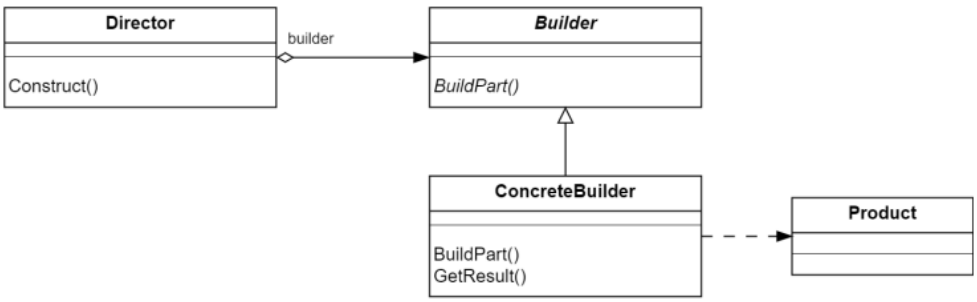
4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?

Object Adapter використовує композицію — адаптер містить об’єкт Adaptee і делегує йому виклики. Гнучкий, бо може працювати з різними підкласами.
Class Adapter використовує успадкування — адаптер наслідує Target і Adaptee. Менш гнучкий, але швидший і простіший у реалізації.

5. Яке призначення шаблону «Будівельник»?

Призначення патерну: Шаблон «Builder» використовується для відділення процесу створення об'єкту від його представлення. Це доречно у випадках, коли об'єкт має складний процес створення (наприклад, Webсторінка як елемент повної відповіді web- сервера) або коли об'єкт повинен мати декілька різних форм створення (наприклад, при конвертації тексту з формату у формат).

6. Нарисуйте структуру шаблону «Будівельник»



7. Які класи входять в шаблон «Будівельник», та яка між ними взаємодія?

- Builder – інтерфейс, який визначає кроки створення об’єкта.
- ConcreteBuilder – реалізує конкретні кроки побудови та формує об’єкт.
- Director – керує послідовністю викликів методів будівельника.
- Product – кінцевий об’єкт, який створюється.

Клієнт звертається до Director, той викликає методи Builder, які реалізовані у ConcreteBuilder, і поступово створюється Product. Після завершення будівництва клієнт отримує готовий об’єкт від будівельника.

8. У яких випадках варто застосовувати шаблон «Будівельник»"?

Шаблон «Будівельник» варто застосовувати, коли:

- Об’єкт має багато параметрів або його створення відбувається у кілька кроків.
- Потрібно створювати різні варіації одного об’єкта, використовуючи один і той самий алгоритм побудови.
- Код створення об’єкта складний і його потрібно відокремити від логіки бізнес-процесу.
- Необхідно мати контроль над процесом побудови — визначати порядок і обов’язкові кроки.

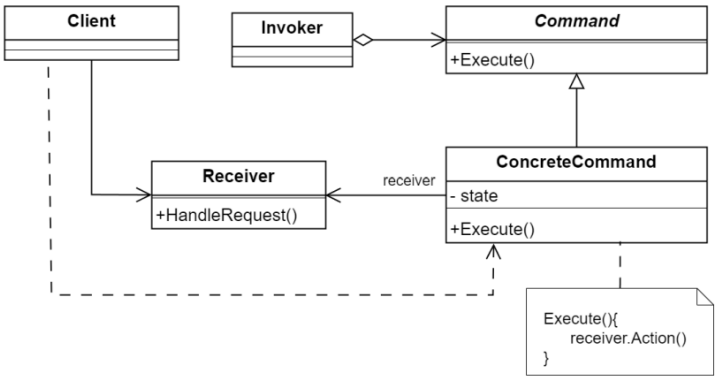
Наприклад побудова складних структур, як-от документа, автомобіля чи GUI-вікна, де є безліч деталей і налаштувань.

9. Яке призначення шаблону «Команда»

   Шаблон "command" (команда) перетворить звичайний виклик методу в клас. Таким чином дії в системі стають повноправними об'єктами. Це зручно в наступних випадках:

- Коли потрібна розвинена система команд – відомо, що команди будуть добавлятися;
- Коли потрібна гнучка система команд – коли з'являється необхідність додавати командам можливість відміни, логування і інш.;
- Коли потрібна можливість складання ланцюжків команд або виклику команд в певний час.

10. Нарисуйте структуру шаблону «Команда»



11. Які класи входять в шаблон «Команда», та яка між ними взаємодія?

- Command (Інтерфейс або абстрактний клас) — визначає єдиний метод execute(), який виконує дію.
- ConcreteCommand (Конкретна команда) — реалізує інтерфейс Command, зберігає посилання на об’єкт Receiver і викликає його методи для виконання операції.
- Receiver (Одержувач) — містить безпосередню логіку виконання дії (наприклад, збереження файлу, відправлення запиту, тощо).
- Invoker (Ініціатор або Виконавець) — зберігає об’єкт команди і викликає його метод execute() у потрібний момент.
- Client (Клієнт) — створює об’єкт команди, передаючи в неї посилання на потрібного одержувача, і передає команду виконавцю.

Клієнт створює команду, пов’язану з конкретним одержувачем, і передає її виконавцю. Коли виконавець викликає метод execute(), команда звертається до одержувача, який фактично виконує дію. Таким чином, виконавець не знає деталей реалізації операції — він просто викликає команду.

12. Розкажіть як працює шаблон «Команда».

Шаблон Command інкапсулює запит у вигляді об’єкта, відокремлюючи відправника запиту від отримувача. Це дозволяє виконувати дії пізніше, ставити їх у чергу, скасовувати або повторювати.

Принцип роботи такий:

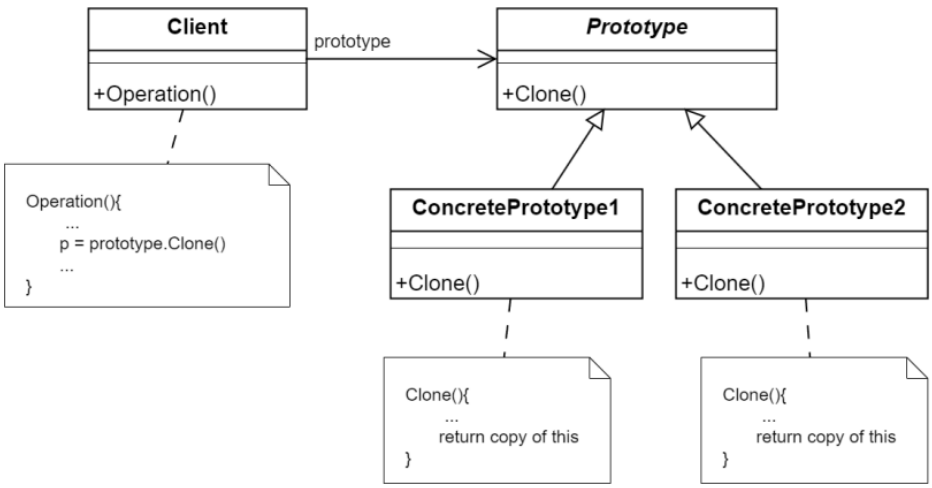
1. Клієнт створює команду (ConcreteCommand), зв’язану з конкретним одержувачем (Receiver).
2. Команда зберігає всю інформацію, необхідну для виконання дії (наприклад, параметри або посилання на об’єкт).
3. Коли виконавець (Invoker) викликає метод execute(), команда звертається до одержувача, який безпосередньо виконує дію.
4. Завдяки цьому Invoker не знає, яку саме дію виконує команда — він лише активує її.

Цей підхід спрощує реалізацію історії дій (undo/redo), планувальників завдань і систем макрокоманд, де дії потрібно виконувати послідовно або відкладено.

13. Яке призначення шаблону «Прототип»?

Шаблон «Prototype» (Прототип) використовується для створення об'єктів за «шаблоном» (чи «кресленням», «ескізом») шляхом копіювання шаблонного об'єкту, який називається прототипом. Для цього визначається метод «клонувати» в об'єктах цього класу. Цей шаблон зручно використати, коли заздалегідь відомо як виглядатиме кінцевий об'єкт (мінімізується кількість змін до об'єкту шляхом створення шаблону), а також для видалення необхідності створення об'єкту – створення відбувається за рахунок клонування, і самій програмі немає необхідності знати, як створювати об'єкт.

14. Нарисуйте структуру шаблону «Прототип».



15. Які класи входять в шаблон «Прототип», та яка між ними взаємодія?

- Prototype (інтерфейс або абстрактний клас). Містить метод clone(), який повертає копію об’єкта.
- ConcretePrototype (конкретний прототип). Реалізує метод clone() і створює копію самого себе. Тут реалізується логіка поверхневого або глибокого копіювання.
- Client (клієнт). Використовує об’єкти-прототипи для створення нових екземплярів шляхом виклику clone() замість звичайного new.

Взаємодія:

Клієнт не створює об’єкти напряму. Він звертається до прототипу (ConcretePrototype), викликаючи clone(), і отримує новий екземпляр. Таким чином клієнт не залежить від конкретних класів та їх складних конструкторів.

16. Які можна привести приклади використання шаблону «Ланцюжок відповідальності»?

Шаблон Chain of Responsibility (Ланцюжок відповідальності) застосовується, коли потрібно передавати запит по ланцюжку об’єктів, де кожен об’єкт або обробляє його, або передає далі. Наприклад:

1. Обробка HTTP-запитів у веб-фреймворках. Middleware-ланцюжок: логування → авторизація → маршрутизація → рендер.
2. Системи логування. Логи передаються кільком рівням: INFO → DEBUG → WARN → ERROR.
3. Валідація даних. Кожен валідатор перевіряє свою умову й передає запит далі, якщо перевірка пройдена.
4. UI події (натискання кнопки). Подія йде від кнопки → контейнера → головного вікна, поки хтось не обробить її.