



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 №4  
**Технологія розроблення програмного забезпечення**  
«ШАБЛОНИ «SINGLETON»,  
«ITERATOR», «PROXY», «STATE»,  
«STRATEGY» »  
Варіант 11

Виконала  
студентка групи ІА-24  
Тильна Марія Сергіївна

Перевірив:  
Мягкий М.Ю.

Київ 2024р.

## **Зміст**

1. Завдання	2
2. Теоретичні відомості	2
3. Хід роботи	5
4. Реалізація шаблону	6
5. Структура шаблону	8
6. Висновок	9

**Тема:** Шаблони «SINGLETON», «ITERATOR», «PROXY», «STATE», «STRATEGY»

### **Завдання.**

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
3. Застосування одного з розглянутих шаблонів при реалізації програми.

#### **..11 Web crawler (proxy, chain of responsibility, memento, template method, composite, p2p)**

Веб-сканер повинен вміти розпізнавати структуру сторінок сайту, переходити за посиланнями, збирати необхідну інформацію про зазначений термін, видаляти не семантичні одиниці (рекламу, об'єкти javascript і т.д.), зберігати знайдені дані у вигляді структурованого набору html файлів вести статистику відвіданих сайтів і метадані.

<https://github.com/mashunchik/Webcrawler.git>

### **Теоретичні відомості**

### **Патерни проєктування**

Патерн проєктування — це типовий спосіб вирішення певної проблеми, що часто зустрічається при проєктуванні архітектури програм.

На відміну від готових функцій чи бібліотек, патерн не можна просто взяти й скопіювати в програму. Патерн являє собою не якийсь конкретний код, а загальний принцип вирішення певної проблеми, який майже завжди треба підлаштовувати для потреб тієї чи іншої програми.

Патерни часто плутають з алгоритмами, адже обидва поняття описують типові рішення відомих проблем. Але якщо алгоритм — це чіткий набір дій, то патерн — це високорівневий опис рішення, реалізація якого може відрізнятися у двох різних програмах.

Якщо провести аналогію, то алгоритм — це кулінарний рецепт з чіткими кроками, а патерн — інженерне креслення, на якому намальовано рішення без конкретних кроків його отримання.

Найбільш низькорівневі та прості патерни — *ідіоми*. Вони не дуже універсальні, позаяк мають сенс лише в рамках однієї мови програмування.

Найбільш універсальні — архітектурні патерни, які можна реалізувати

практично будь-якою мовою. Вони потрібні для проектування всієї програми, а не окремих її елементів.

Крім цього, патерни відрізняються і за призначенням. Існують три основні групи патернів:

- **Породжуючі патерни** піклуються про гнучке створення об'єктів без внесення в програму зайвих залежностей.
- **Структурні патерни** показують різні способи побудови зв'язків між об'єктами.
- **Поведінкові патерни** піклуються про ефективну комунікацію між об'єктами.

### Одинак (Singleton)

Шаблон проектування "Одинак" гарантує, що клас матиме лише один екземпляр, і забезпечує глобальну точку доступу до цього екземпляра. Це корисно, коли потрібно контролювати доступ до деяких спільних ресурсів, наприклад, підключення до бази даних або конфігураційного файлу. Основна ідея полягає у тому, щоб закрити доступ до конструктора класу і створити статичний метод, що повертає єдиний екземпляр цього класу. У мовах, які підтримують багатопоточність, також важливо синхронізувати метод доступу, щоб уникнути створення кількох екземплярів в різних потоках. Один із способів реалізації одинака у Java – використання статичної ініціалізації, яка автоматично забезпечує безпечність у багатопоточному середовищі. Деякі розробники вважають одинак антипатерном, оскільки він створює глобальний стан програми, що ускладнює тестування та підтримку. Використання цього шаблону має бути обґрунтованим і обмеженим певними обставинами.

### Ітератор (Iterator)

Шаблон "Ітератор" надає спосіб послідовного доступу до елементів колекції без розкриття її внутрішньої структури. Він особливо корисний для обходу різних типів колекцій, таких як списки, множини або деревовидні структури, незалежно від того, як вони реалізовані. Ітератор інкапсулює поточний стан перебору, тому може зберігати інформацію про те, який елемент є наступним. Цей шаблон дозволяє відокремити логіку роботи з колекцією від логіки обходу, що робить код більш гнучким і модульним. У Java цей шаблон реалізований у вигляді інтерфейсу `Iterator`, який надає методи `hasNext()` та `next()` для послідовного перебору елементів. Ітератор також можна використовувати для видалення елементів під час обходу колекції. Завдяки цьому шаблону можна використовувати поліморфізм для однакового доступу до елементів різних колекцій.

## Проксі (Proxy)

Шаблон "Проксі" створює замісник або посередника для іншого об'єкта, що контролює доступ до цього об'єкта. Проксі може виконувати додаткову роботу перед передачею викликів реальному об'єкту, як-от перевірку прав доступу або відкладену ініціалізацію. Існує декілька типів проксі, серед яких захисний проксі, який контролює доступ, і віртуальний проксі, який затримує створення об'єкта, поки він не буде потрібен. У Java цей шаблон часто використовується для створення динамічних проксі за допомогою інтерфейсів, де проксі-клас реалізує той самий інтерфейс, що й реальний об'єкт. Проксі ефективний для оптимізації роботи з ресурсами або для контролю доступу до важких у створенні об'єктів. Це дозволяє зберігати оригінальний об'єкт захищеним і надає додатковий шар для маніпуляцій.

## Стан (State)

Шаблон "Стан" дозволяє об'єкту змінювати свою поведінку при зміні внутрішнього стану, надаючи йому різні стани для різних контекстів. Він ефективно інкапсулює різні стани об'єкта як окремі класи і делегує дії поточному стану. Наприклад, кнопка може мати різні дії залежно від того, чи вона активована чи деактивована. Це дозволяє замість довгих умовних операторів використовувати поліморфізм, де кожен клас стану реалізує свою поведінку, визначену інтерфейсом стану. У Java цей шаблон може бути реалізований як клас з інтерфейсом для станів, де кожен стан є окремим підкласом, що відповідає за певну поведінку. Це полегшує масштабування та тестування, оскільки додавання нового стану не вимагає змін у вихідному коді.

## Стратегія (Strategy)

Шаблон "Стратегія" дозволяє вибирати алгоритм або поведінку під час виконання, забезпечуючи взаємозамінність різних алгоритмів для конкретного завдання. Він передбачає інкапсуляцію різних варіантів поведінки в окремих класах, які реалізують один інтерфейс, що спрощує заміну і додавання алгоритмів. Клас контексту отримує об'єкт стратегії і викликає відповідні методи, не знаючи деталей реалізації конкретної стратегії. Наприклад, клас сортування може мати кілька стратегій: швидке сортування, сортування вставкою чи сортування вибором, і залежно від контексту обирається відповідний метод. У Java шаблон реалізується через інтерфейс стратегії, який мають різні класи конкретних стратегій.

## Хід роботи

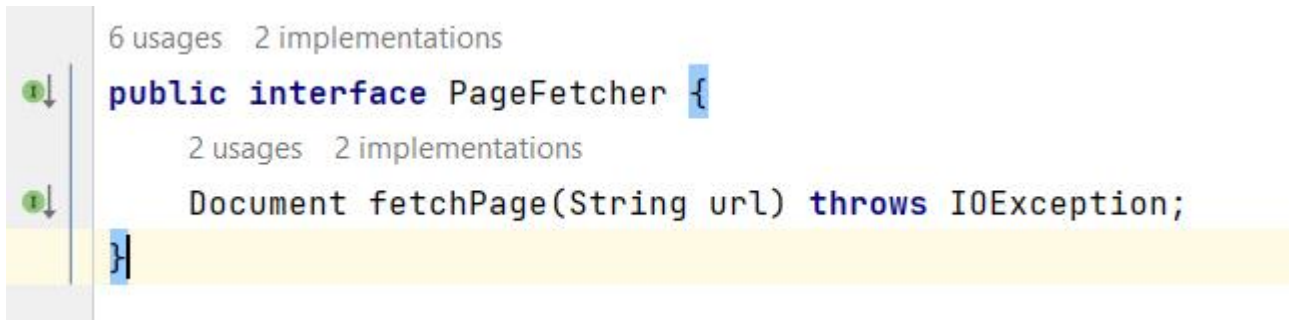
У цій лабораторній роботі я реалізувала шаблон Proxy. Шаблон Proxy у цьому проєкті забезпечує проміжний рівень між сканером і виконанням HTTP-запитів. Він додає кешування для зменшення повторних запитів, обмеження швидкості для сканування без перевантаження серверів. Це робить систему більш ефективною, безпечною та легкою для розширення.

Він обробляє всі складні веб-запити, зберігаючи основний код сканера чистим і зосередженим на своєму основному завданні – обробці веб-сторінок.

### Як працює шаблон у Webcrawler:

1. Веб-сканер або сервіс викликає метод `fetchPage(url)` у проксі `PageFetcherProxy`.
2. `PageFetcherProxy`:
  1. Перевіряє, чи є сторінка в кеші:
    - 1) Якщо є — повертає з кешу.
    - 2) Якщо **немає** — звертається до `PageFetcherImpl` для завантаження.
  2. Зберігає сторінку в кеші для подальших викликів.
3. `PageFetcherImpl`:
  - 1) Завантажує сторінку з Інтернету за допомогою **Jsoup**.
  - 2) Повертає об'єкт `Document`.

## Структура проекту



```
6 usages 2 implementations
public interface PageFetcher {
    2 usages 2 implementations
    Document fetchPage(String url) throws IOException;
}
```

Рис1. Інтерфейс

Це інтерфейс, який визначає основний контракт (метод) для завантаження сторінок за URL-адресою. Дає можливість створювати різні реалізації завантаження сторінок без прив'язки до конкретного механізму. Оголошує метод `fetchPage(String url)`, який має повертати документ (`Document`) із завантаженим HTML-контентом. Не містить логіки — лише описує, як мають виглядати класи, які реалізують цей інтерфейс.



```
2 usages
public class PageFetcherImpl implements PageFetcher {
    2 usages
    @Override
    public Document fetchPage(String url) throws IOException {
        return Jsoup.connect(url).get();
    }
}
```

Рис2. Код класу PageFetcherImp

Реалізація інтерфейсу `PageFetcher` для фактичного завантаження HTML-сторінок з Інтернету.

```

public class PageFetcherProxy implements PageFetcher {

    2 usages
    private final PageFetcher pageFetcher;

    3 usages
    private final Map<String, Document> cache = new HashMap<>();

    2 usages
    private static final long REQUEST_DELAY = 1000; // 1 запит в секунду

    3 usages
    private long lastRequestTime = 0;

    1 usage
    public PageFetcherProxy(PageFetcher pageFetcher) {
        this.pageFetcher = pageFetcher;
    }
}

```

Рис.3.1 Код класу PageFetcherProxy

```

@Override
public Document fetchPage(String url) throws IOException {
    // Кешування
    if (cache.containsKey(url)) {
        System.out.println("Page fetched from cache: " + url);
        return cache.get(url);
    }

    // Контроль швидкості
    long currentTime = System.currentTimeMillis();
    if (currentTime - lastRequestTime < REQUEST_DELAY) {
        try {
            Thread.sleep( millis: REQUEST_DELAY - (currentTime - lastRequestTime));
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    // Завантаження сторінки
    Document document = pageFetcher.fetchPage(url);
    cache.put(url, document);
    lastRequestTime = System.currentTimeMillis();

    System.out.println("Page fetched from web: " + url);
    return document;
}
}

```

Рис.3.2 Код класу PageFetcherProxy



Клас реалізує шаблон Проху для контролю доступу до ресурсу (веб-сторінки) і оптимізації за допомогою кешування.

Додає логіку кешування до механізму завантаження сторінок. Зберігає посилання на реальний об'єкт PageFetcher: це дозволяє делегувати завантаження сторінок класу PageFetcherImpl. Перевіряє кеш: якщо URL вже є в кеші (cache.containsKey(url)), повертає збережений результат, а не виконує повторний HTTP-запит. Якщо сторінки немає в кеші, вона завантажується з Інтернету і зберігається в локальному сховищі (cache). А також повідомляє в консоль, чи сторінка завантажена з кешу, чи з Інтернету.

Структура шаблону Проху

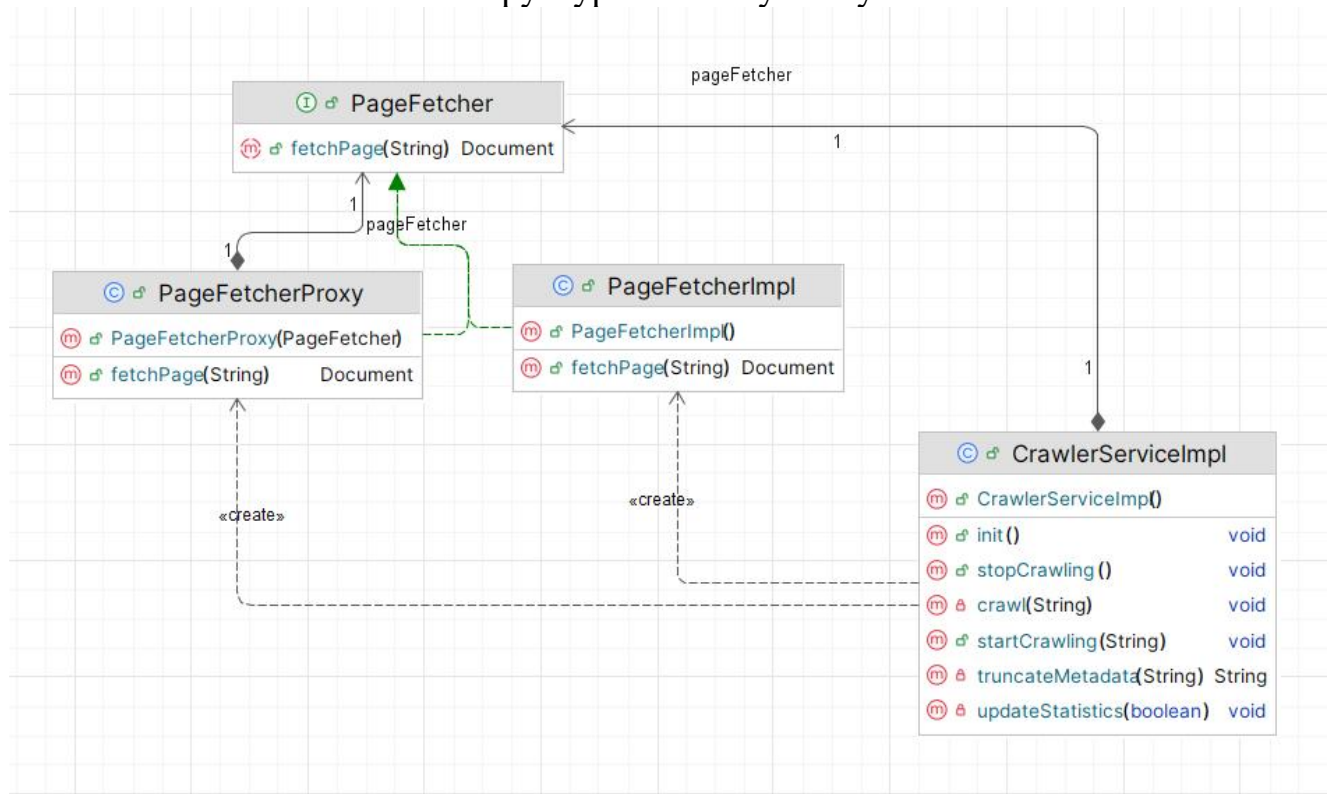


Рис. 4 Діаграма шаблону Проху

**Посилання на репозиторій:**

<https://github.com/mashunchik/Webcrawler>

**Висновок :** виконуючи цю лабораторну роботу, я ознайомилась з такими патернами, як Singleton, Iterator, Proxy, State та Strategy. Кожен із цих патернів має свої переваги та обмеження, а їх правильне застосування дозволяє значно підвищити ефективність процесу розробки програмного забезпечення.

На основі отриманих знань було реалізовано проксі-клас PagefetcherProxy, що використовує шаблон Proxy. Це рішення забезпечило безпечну та ефективну роботу сканера без перевантажень. Використання шаблону Proxy дозволило чітко розділити відповідальність між попереднім скануванням сторінок та основною бізнес-логікою, що позитивно вплинуло на підтримуваність і масштабованість коду.



