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 №5

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 “Adapter”, “Builder”, “Command”, “Chain of responsibility”, “Prototype” 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

Адаптер (Adapter)

Патерн Адаптер — це, по суті, "перехідник" або "перекладач". Його головне завдання — змусити два об'єкти працювати разом, навіть якщо їхні інтерфейси абсолютно несумісні. Уявіть, що у вас є ноутбук з сучасним портом USB-C, але

ваша улюблена стара мишка має штекер USB-A. Ви не можете підключити їх напряму. Адаптер стає посередником, який "перекладає" сигнали з одного інтерфейсу на інший, дозволяючи їм працювати разом. У програмуванні це виглядає так само: у вас є новий клас, який очікує отримати об'єкт з методом doWork(), але у вас є стара бібліотека, де схожий клас має метод executeLegacyTask(). Ви створюєте клас-Адаптер, який реалізує потрібний інтерфейс doWork(), але всередині себе він просто викликає метод executeLegacyTask() старого об'єкта. Таким чином, ви "адаптуєте" старий код до нового, не змінюючи сам старий код, що особливо корисно при роботі зі сторонніми бібліотеками.

Будівельник (Builder)

Патерн Будівельник вирішує проблему створення складних об'єктів. Уявіть, що вам потрібно створити об'єкт "Комп'ютер". У нього може бути 10-15 характеристик: процесор, відеокарта, об'єм пам'яті, тип диска, блок живлення, материнська плата тощо. Створювати такий об'єкт через один гігантський конструктор (new Computer("i9", "RTX4090", 32, ...)), де половина параметрів може бути не потрібна (наприклад, немає дискретної відеокарти), — це просто жахіття. Будівельник пропонує інший підхід: він дозволяє "збирати" об'єкт крок за кроком. Ви створюєте окремий об'єкт-Будівельник і даєте йому послідовні команди: builder.setCPU("i9"), builder.setRAM(32), builder.setGPU("RTX4090"). Кожен з цих методів повертає самого себе, дозволяючи створювати ланцюжки викликів. Коли ви налаштували все, що хотіли, ви викликаєте фінальний метод build(), і Будівельник віддає вам готовий, сконструйований об'єкт "Комп'ютер". Це робить код набагато чистішим і дозволяє створювати різні конфігурації об'єктів, використовуючи той самий процес збірки.

Команда (Command)

Патерн Команда — це геніальний спосіб перетворити дію або запит на окремий об'єкт. Замість того, щоб один об'єкт напряму викликав метод іншого (editor.pasteText()), ви створюєте об'єкт-Команду, наприклад PasteCommand. Цей

об'єкт "знає" все, що потрібно для виконання дії (наприклад, посилання на редактор і текст, який треба вставити). Найкраща аналогія — замовлення в ресторані. Ви (клієнт) не йдете на кухню і не кажете кухарю, що робити. Ви даєте офіціанту "Команду" (замовлення на папірці). Офіціант ставить його в чергу. Кухар бере команди з черги і виконує їх. Це дає неймовірні переваги: ви можете ставити команди в чергу, відкладати їх виконання, зберігати історію команд і, найголовніше, — реалізувати функцію "Скасувати/Повторити" (Undo/Redo). Адже якщо кожна команда знає, як себе виконати (execute()), вона так само може знати, як скасувати свою дію (undo()).

Ланцюжок Обов'язків (Chain of Responsibility)

Цей патерн дозволяє уникнути жорсткої прив'язки відправника запиту до його одержувача. Ідея в тому, щоб дати можливість обробити запит кільком об'єктам. Ви вибудовуєте ці об'єкти в "ланцюжок". Коли надходить запит, він передається першому об'єкту в ланцюжку. Цей об'єкт дивиться на запит і вирішує: "Це моя робота? Можу я це обробити?". Якщо так, він його обробляє, і на цьому все. Якщо ні, він передає запит *наступному* об'єкту в ланцюжку, і так далі. Відправник запиту навіть не знає, хто саме в кінцевому підсумку обробить його запит. Це дуже схоже на роботу банкомата: ви просите 480 гривень. Перший обробник ("відповідальний" за 200-гривневі купюри) видає дві й передає залишок 80 гривень далі. Наступний (для 100) нічого не робить. Наступний (для 50) видає одну і передає 30 гривень. І так, доки запит не буде виконано повністю. Це часто використовується у веб-серверах для фільтрів: перевірка автентифікації, потім логування, потім кешування — кожен є ланкою в ланцюжку.

Прототип (Prototype)

Патерн Прототип вирішує проблему створення об'єктів, коли їх "будівництво" з нуля є дуже дорогим або складним процесом. Замість того, щоб щоразу створювати об'єкт через new, цей патерн пропонує створити новий об'єкт шляхом копіювання (клонування) вже існуючого. Уявіть, що у вас є складний об'єкт "НалаштуванняГри", на створення якого ви витратили 5 секунд,

завантаживши дані з файлів та бази даних. Якщо вам потрібен точно такий же об'єкт, але з однією маленькою зміною, набагато "дешевше" створити повну копію (клон) першого об'єкта за мілісекунди і змінити лише одне поле, ніж знову 5 секунд створювати його "з нуля". Ви створюєте один об'єкт-зразок, або "прототип", а всі наступні об'єкти такого типу отримуєте, просто викликаючи у прототипа метод clone(). Це особливо корисно, коли вам потрібно створити багато однакових або майже однакових об'єктів.

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: Chain of responsibility

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

Хід роботи

Діаграма класів

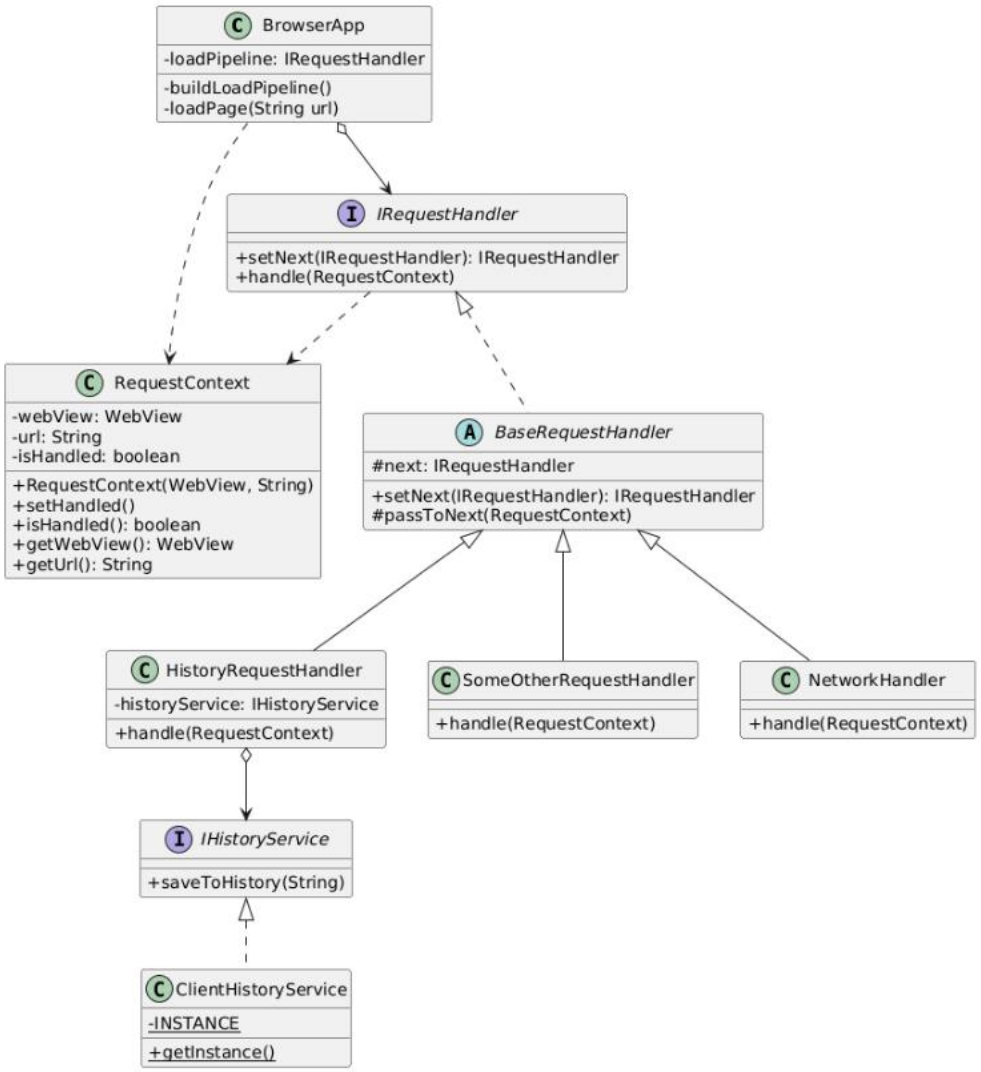


Рис.1 – Діаграма класів для RequestHandler (патерн Chain of responsibility)

Опис діаграми класів

Ця діаграма класів ілюструє, як процес завантаження вебсторінки у BrowserApp розбитий на серію послідовних кроків за допомогою патерну "Ланцюжок Обов'язків".

Замість того, щоб один метод loadPage() виконував всю логіку (перевірку кешу, збереження історії, завантаження з мережі), ця відповідальність делегується ланцюжку об'єктів-обробників.

Учасники патерну "Ланцюжок Обов'язків"

На діаграмі чітко видно всіх ключових учасників патерну:

1. Клієнт (Client)

- Клас: BrowserApp

- Роль: Це ініціатор запиту. Він не виконує логіку завантаження сам, а натомість створює "ланцюжок" (через метод buildLoadPipeline()) і зберігає посилання на його початок у полі loadPipeline.
- Взаємодія: Коли користувач вводить URL, метод loadPage() створює об'єкт RequestContext і передає його першому обробнику в ланцюжку (loadPipeline.handle(...)).

2. Об'єкт Запиту (Request Object)
   - Клас: RequestContext
   - Роль: Це об'єкт, який передається *вздовж* ланцюжка від одного обробника до іншого. Він містить усі необхідні дані для обробки (WebView, url), а також, що дуже важливо, прапорець isHandled.
   - Ключова деталь: Метод setHandled() дозволяє будь-якому обробнику (наприклад, NetworkHandler або обробнику кешу) зупинити подальше проходження запиту по ланцюжку.
3. Інтерфейс Обробника (Handler)
   - Інтерфейс: IRequestHandler
   - Роль: Це спільний "контракт" для всіх ланок ланцюжка. Він визначає два ключові методи:
     - handle(RequestContext): Метод, що виконує логіку обробки.
     - setNext(...): Метод, що дозволяє з'єднати обробники один з одним, формуючи ланцюжок.
4. Базовий Обробник (Base Handler)
   - Абстрактний клас: BaseRequestHandler
   - Роль: Це допоміжний клас, який реалізує IRequestHandler. Він надає спільну логіку для всіх конкретних обробників:
     - Зберігає посилання на наступний обробник у полі next.
     - Надає захищений метод passToNext(), який перевіряє, чи запит ще не був оброблений (!requestContext.isHandled()) і чи існує наступний обробник.
5. Конкретні Обробники (Concrete Handlers)
   - Класи: HistoryRequestHandler, SomeOtherRequestHandler, NetworkHandler
   - Роль: Це реальні "робочі" ланки ланцюжка. Кожен з них успадковує BaseRequestHandler і реалізує свою вузькоспеціалізовану логіку:
     - HistoryRequestHandler: Отримує запит, виконує свою дію (зберігає URL в історію, використовуючи IHistoryService) і передає запит далі через passToNext().
     - SomeOtherRequestHandler: Виконує іншу проміжну логіку

(наприклад, перевірку кешу або автентифікацію) і передає запит далі.

- NetworkHandler: Є кінцевою ланкою. Він виконує реальне завантаження сторінки в WebView і викликає requestContext.setHandled(), щоб зупинити ланцюжок.

Як працює потік (The Flow)

1. BrowserApp у методі loadPage() створює RequestContext.
2. Він викликає handle() у loadPipeline (який вказує на HistoryRequestHandler).
3. HistoryRequestHandler робить свою роботу (зберігає історію) і викликає passToNext().
4. Запит "перетікає" до SomeOtherRequestHandler, який робить свою роботу і викликає passToNext().
5. Запит доходить до NetworkHandler, який завантажує URL в WebView і позначає запит як оброблений.
6. Ланцюжок завершується.

Частина коду програми з використанням патерну Chain of responsibility

BrowserApp.java

```
package com.edu.web.browserapp;

import com.edu.web.browserapp.requestHandle.IRequestHandler;
import com.edu.web.browserapp.requestHandle.RequestContext;
import com.edu.web.browserapp.requestHandle.handler.HistoryRequestHandler;
import com.edu.web.browserapp.requestHandle.handler.NetworkHandler;
import com.edu.web.browserapp.requestHandle.handler.SomeOtherRequestHandler;
import javafx.application.Application;
import javafx.concurrent.Worker;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.input.KeyCode;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
```

```
import javafx.scene.web.WebView;
import javafx.stage.Stage;

public class BrowserApp extends Application {

    private WebView webView;
    private TextField addressBar;
    private Label statusLabel;
    private IRequestHandler loadPipeline;

    @Override
    public void start(Stage stage) {

        webView = new WebView();
        addressBar = new TextField("https://www.google.com");
        statusLabel = new Label("Готовий");

        buildLoadPipeline();

        addressBar.setOnKeyPressed(e -> {
            if (e.getCode() == KeyCode.ENTER) {
                loadPage(addressBar.getText());
            }
        });


        webView.getEngine().getLoadWorker().stateProperty().addListener((_, _,
newState) -> {
            statusLabel.setText(newState.toString());

            if (newState == Worker.State.SUCCEEDED) {
                String loadedUrl = webView.getEngine().getLocation();
                statusLabel.setText("Завантажено: " + loadedUrl);
            }
        });

        HBox topBar = new HBox(10, new Label("URL:"), addressBar);
        topBar.setPadding(new Insets(10));
        addressBar.prefWidthProperty().bind(topBar.widthProperty().subtract(50));
```

```
        BorderPane root = new BorderPane();
        root.setTop(topBar);
        root.setCenter(webView);
        root.setBottom(statusLabel);
        BorderPane.setMargin(statusLabel, new Insets(5));

        loadPage(addressBar.getText());

        Scene scene = new Scene(root, 1024, 768);
        stage.setTitle("Web Browser");
        stage.setScene(scene);
        stage.show();
    }

    private void loadPage(String url) {
        if (!url.startsWith("https://")) {
            url = "https://" + url;
        }

        var requestContext = new RequestContext(webView, url);

        loadPipeline.handle(requestContext);
    }

    private void buildLoadPipeline() {
        IRequestHandler historyHandler = new HistoryRequestHandler();
        IRequestHandler someOtherHandler = new SomeOtherRequestHandler();
        IRequestHandler networkHandler = new NetworkHandler();

        historyHandler.setNext(someOtherHandler)
                .setNext(networkHandler);

        loadPipeline = historyHandler;
    }
}
```

RequestContext.java

```
package com.edu.web.browserapp.requestHandle;

import javafx.scene.web.WebView;
```

```
public class RequestContext {
    private final WebView webView;
    private final String url;
    private boolean isHandled = false;

    public RequestContext(WebView webView, String url) {
        this.webView = webView;
        this.url = url;
    }

    public WebView getWebView() {
        return this.webView;
    }

    public String getUrl() {
        return this.url;
    }

    public boolean isHandled() {
        return this.isHandled;
    }

    public void setHandled() {
        this.isHandled = true;
    }

}
```

IRequestHandler.java

```
package com.edu.web.browserapp.requestHandle;

public interface IRequestHandler {
    IRequestHandler setNext(IRequestHandler IRequestHandler);
    void handle(RequestContext requestContext);
}
```

BaseRequestHandler.java

```
package com.edu.web.browserapp.requestHandle.handler;
```

```
import com.edu.web.browserapp.requestHandle.RequestContext;
import com.edu.web.browserapp.requestHandle.IRequestHandler;

abstract class BaseRequestHandler implements IRequestHandler {
   protected IRequestHandler next;

   @Override
   public IRequestHandler setNext(IRequestHandler IRequestHandler) {
      this.next = IRequestHandler;
      return IRequestHandler;
   }

   protected void passToNext(RequestContext  requestContext) {
      if (next != null && !requestContext.isHandled()) {
         next.handle(requestContext);
      }
   }
}
```

HistoryRequestHandler.java

```
package com.edu.web.browserapp.requestHandle.handler;

import com.edu.web.browserapp.requestHandle.RequestContext;
import com.edu.web.browserapp.service.IHistoryService;
import com.edu.web.browserapp.service.impl.ClientHistoryService;

public class HistoryRequestHandler extends BaseRequestHandler {

   private final IHistoryService historyService = ClientHistoryService.getInstance();

   @Override
   public void handle(RequestContext requestContext) {
      this.historyService.saveToHistory(requestContext.getUrl());
      passToNext(requestContext);
   }
}
```

NetworkHandler.java

```
package com.edu.web.browserapp.requestHandle.handler;

import com.edu.web.browserapp.requestHandle.RequestContext;
```

```
public class NetworkHandler extends BaseRequestHandler {

   @Override
   public void handle(RequestContext requestContext) {
      requestContext.getWebView()
            .getEngine()
            .load(requestContext.getUrl());

      requestContext.setHandled();
   }
}
```

SomeOtherRequestHandler.java

```
package com.edu.web.browserapp.requestHandle.handler;

import com.edu.web.browserapp.requestHandle.RequestContext;

public class SomeOtherRequestHandler extends BaseRequestHandler {

   @Override
   public void handle(RequestContext requestContext) {
      passToNext(requestContext);
   }
}
```

Опис коду з використання патерну Chain of responsibility

Цей підхід перетворює процес завантаження сторінки з одного монолітного методу на гнучкий "конвеєр". Замість того, щоб loadPage() робив усе сам, він тепер лише ставить "заготовку" (RequestContext) на початок цього конвеєра (loadPipeline). Кожен обробник на конвеєрі виконує одну малу операцію і передає заготовку далі.

Ролі учасників патерну у коді

"Клієнт" (Client)

- Клас: BrowserApp
- Роль: Це ініціатор запиту. Він відповідає за створення ланцюжка *один раз* при запуску (у методі buildLoadPipeline()) та збереження посилання на його початок у полі loadPipeline.

- Взаємодія: Коли користувач натискає Enter, метод loadPage() не виконує жодної логіки. Він лише:
    1. Готує об'єкт RequestContext, що містить url та WebView.
    2. Запускає процес, викликавши loadPipeline.handle(requestContext) — тобто, передає запит *першій ланці* ланцюжка.

"Об'єкт Запиту" (Request Object)

- Клас: RequestContext
- Роль: Це об'єкт, який "подорожує" по ланцюжку. Він несе в собі всі дані, необхідні для обробки (webView, url), а також стан (isHandled).
- Ключова деталь: Прапорець isHandled є критично важливим. Він дозволяє будь-якому обробнику (у вашому випадку NetworkHandler) "завершити" запит і зупинити його подальшу передачу по ланцюжку.

"Інтерфейс Обробника" (Handler Interface)

- Інтерфейс: IRequestHandler
- Роль: Це загальний "контракт" для всіх ланок ланцюжка. Він змушує кожен обробник мати два методи:
    1. handle(RequestContext): Метод, що виконує роботу.
    2. setNext(IRequestHandler): Метод, що дозволяє з'єднати ланки між собою.

"Базовий Обробник" (Base Handler)

- Клас: BaseRequestHandler
- Роль: Це абстрактний клас, який спрощує створення нових обробників. Він виконує "чорнову" роботу:
    1. Зберігає посилання на наступну ланку (protected IRequestHandler next).
    2. Реалізує setNext().
    3. Надає захищений метод passToNext(). Цей метод дуже важливий: він передає запит далі тільки якщо next існує і якщо запит ще не був оброблений (!requestContext.isHandled()).

"Конкретні Обробники" (Concrete Handlers)

- Класи: HistoryRequestHandler, SomeOtherRequestHandler, NetworkHandler
- Роль: Це реальні "робочі" на конвеєрі. Кожен має одну чітку відповідальність:
    - HistoryRequestHandler: Це перша ланка. Він виконує свою роботу (зберігає URL в історію) і завжди викликає passToNext(), оскільки збереження історії не повинно переривати завантаження сторінки.
    - SomeOtherRequestHandler: Це "заглушка", яка показує, наскільки

легко розширювати ланцюжок. Він нічого не робить і просто передає запит далі. Сюди в майбутньому можна додати логіку кешування.

- NetworkHandler: Це остання ланка. Він виконує фінальну дію — реальне завантаження сторінки у WebView. Після цього він викликає requestContext.setHandled(), щоб повідомити ланцюжку, що робота завершена і далі нічого передавати не потрібно.

Переваги підходу

Ця архітектура є чудовим прикладом застосування патерну. BrowserApp (клієнт) тепер повністю відокремлений від логіки завантаження. Якщо завтра знадобиться додати CacheHandler або AuthenticationHandler, не доведеться чіпати BrowserApp — достатньо буде лише додати нову ланку в ланцюжок у методі buildLoadPipeline().

Скриншот застосунку

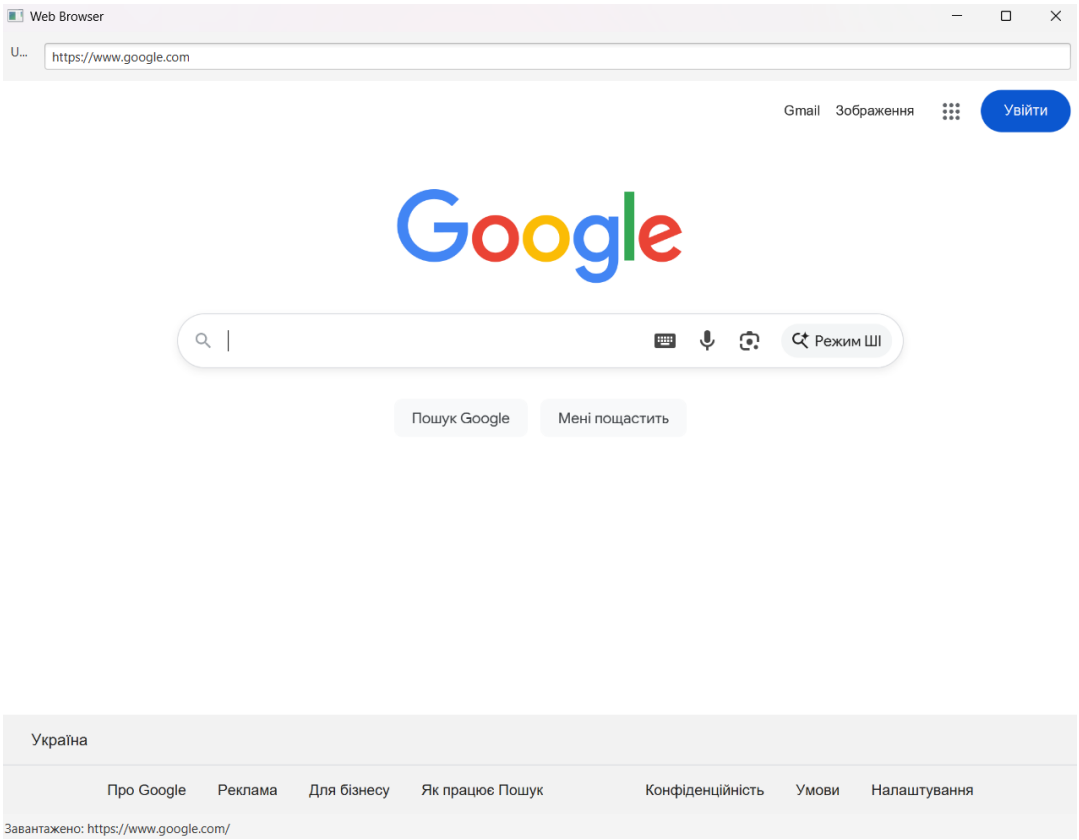


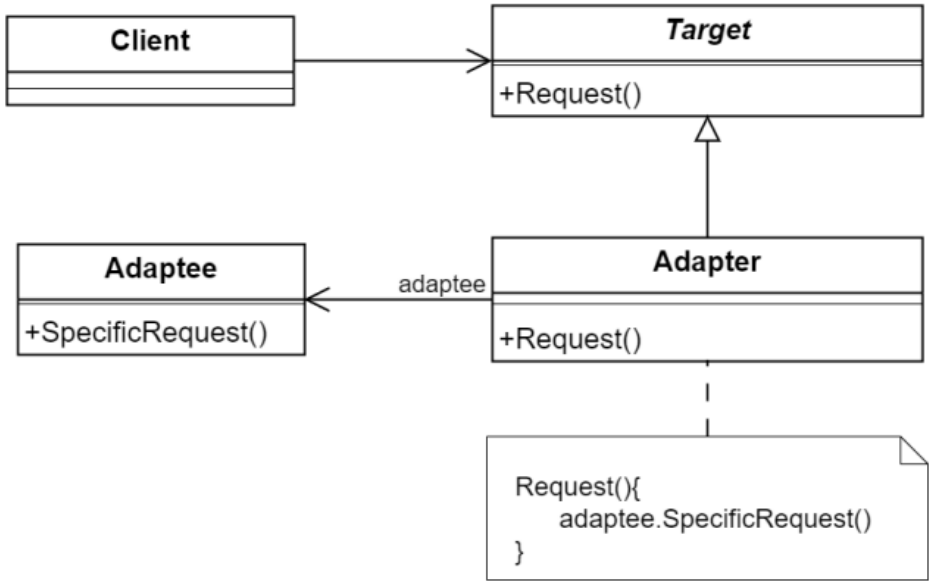
Рис.2 – Скриншот застосунку BrowserApp

Контрольні запитання

1. Яке призначення шаблону «Адаптер»?

Призначення шаблону «Адаптер» полягає в тому, щоб "подружити" два об'єкти з абсолютно несумісними інтерфейсами. Він діє як перекладач або перехідник. Уявіть, що у вас є ноутбук з портом USB-C, але стара мишка має штекер USB-A. Адаптер стає посередником, який дозволяє їм працювати разом. У коді це клас-обгортка, який приймає виклики через один (зрозумілий клієнту) інтерфейс і "перенаправляє" їх, трансформуючи у виклики іншого (несумісного) інтерфейсу.

2. Нарисуйте структуру шаблону «Адаптер».



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

В шаблон «Адаптер» входять: Клієнт (той, хто хоче використовувати об'єкт), Цільовий інтерфейс (той, який Клієнт розуміє), Об'єкт, що адаптується (корисний клас, але з "неправильним" інтерфейсом) та сам Адаптер. Взаємодія така: Клієнт викликає метод у Адаптера, оскільки Адаптер реалізує Цільовий інтерфейс. Адаптер, отримавши цей виклик, перетворює його на виклик методу (або методів) Об'єкта, що адаптується, який він тримає всередині себе, і повертає результат у зрозумілому для Клієнта форматі.

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

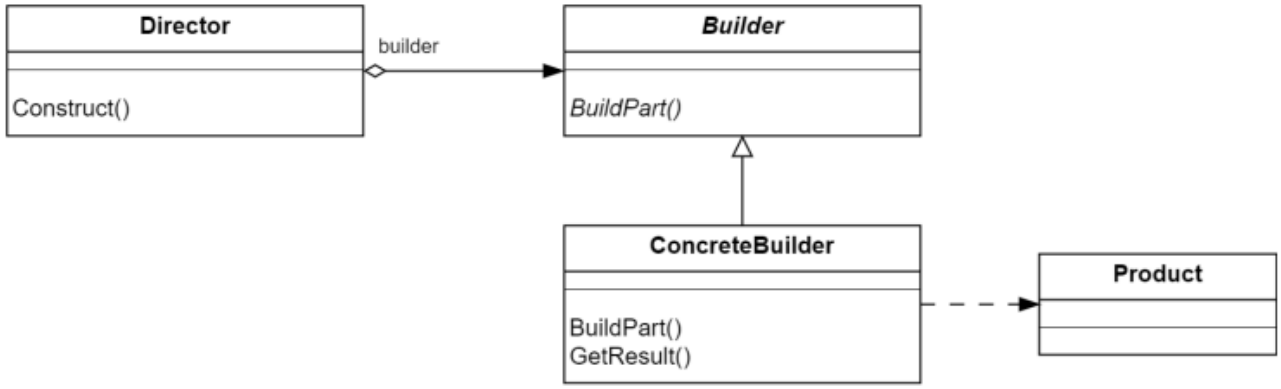
Різниця між реалізацією на рівні об'єктів та класів полягає в тому, як Адаптер досягає своєї мети. Адаптер об'єктів (найпоширеніший) використовує *композицію*: він містить у собі екземпляр об'єкта, що адаптується. Адаптер

класів використовує *успадкування*: він успадковує реалізацію від одного класу (Adaptee) і водночас реалізує інтерфейс іншого (Target). Адаптер об'єктів гнучкіший, оскільки дозволяє адаптувати цілу ієрархію класів (будь-якого нащадка Adaptee), тоді як адаптер класів жорстко прив'язаний до конкретного класу.

5. Яке призначення шаблону «Будівельник»?

Призначення шаблону «Будівельник» — вирішити проблему "телескопічного конструктора", коли для створення об'єкта потрібно передати величезну кількість параметрів. Він дозволяє створювати складні об'єкти крок за кроком. Замість одного жахливого конструктора new Pizza(size, dough, cheese, pepperoni, mushrooms, olives...) ви отримуєте чистий процес: builder.setSize("large").addCheese().addPepperoni().build(). Це відокремлює процес конструювання об'єкта від його кінцевого вигляду.

6. Нарисуйте структуру шаблону «Будівельник».



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

В шаблон «Будівельник» входять: Продукт (складний об'єкт), Інтерфейс Будівельника (описує кроки збірки), Конкретний Будівельник (реалізує кроки і зберігає проміжний результат) та Директор (керує процесом збірки). Взаємодія така: Клієнт створює об'єкт Конкретного Будівельника і передає його Директору. Директор викликає методи Будівельника у потрібній послідовності (buildWalls(), buildRoof()). Сам Будівельник збирає частини Продукту. Після завершення роботи Директора, Клієнт забирає готовий Продукт у Будівельника.

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

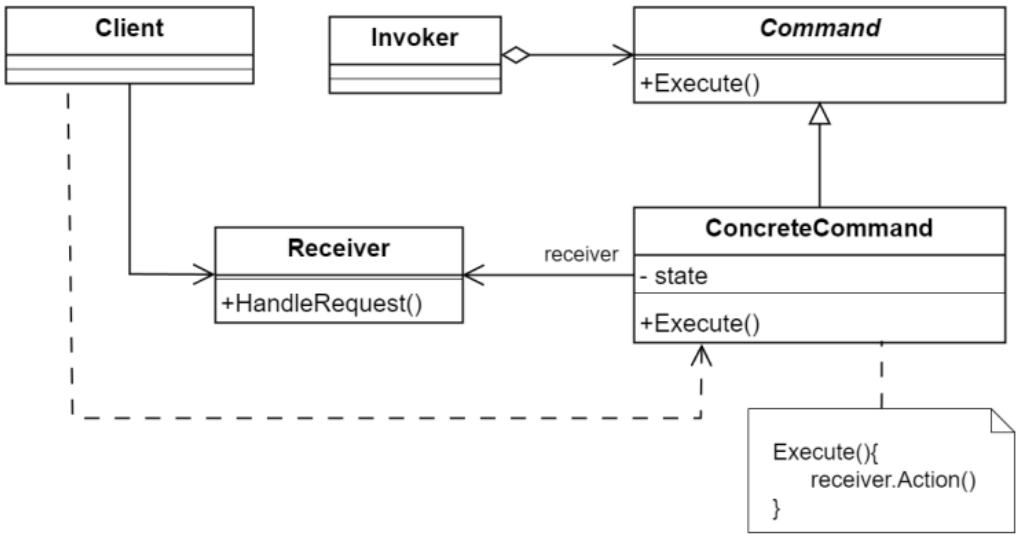
Шаблон «Будівельник» варто застосовувати у кількох випадках. По-перше, коли ваш об'єкт має дуже багато параметрів у конструкторі, більшість з яких опціональні. По-друге, коли процес створення об'єкта складний і складається з

кількох етапів, і ви хочете зробити цей процес чітким та покроковим. По-третє, коли вам потрібно створити *різні версії* або *представлення* об'єкта, використовуючи той самий процес конструювання (наприклад, buildCar() і buildTruck() можуть використовувати того самого Директора, але з різними Будівельниками).

9. Яке призначення шаблону «Команда»?

Призначення шаблону «Команда» — перетворити запит (або дію) на самостійний об'єкт. Замість того, щоб один об'єкт напряму викликав метод іншого, ми "загортаємо" цей виклик в окремий об'єкт-Команду. Це дає величезну гнучкість: ви можете ставити ці команди в чергу, скасовувати їх (Undo/Redo), логувати або передавати по мережі.

10. Нарисуйте структуру шаблону «Команда».



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

До шаблону «Команда» входять: Інтерфейс Команди (з методом execute()), Конкретна Команда (реалізує execute(), знає, кого викликати), Одержувач (Receiver, об'єкт, що робить реальну роботу, наприклад, TextEditor), Ініціатор (Invoker, наприклад, Button або MenuItem), та Клієнт (створює команду і пов'язує її з одержувачем). Взаємодія: Клієнт створює PasteCommand (Конкретна Команда) і передає їй TextEditor (Одержувач). Потім Клієнт прикріплює цю команду до PasteButton (Ініціатор). Коли користувач натискає кнопку, Ініціатор викликає command.execute(). Команда, у свою чергу, викликає textEditor.paste().

12. Розкажіть як працює шаблон «Команда».

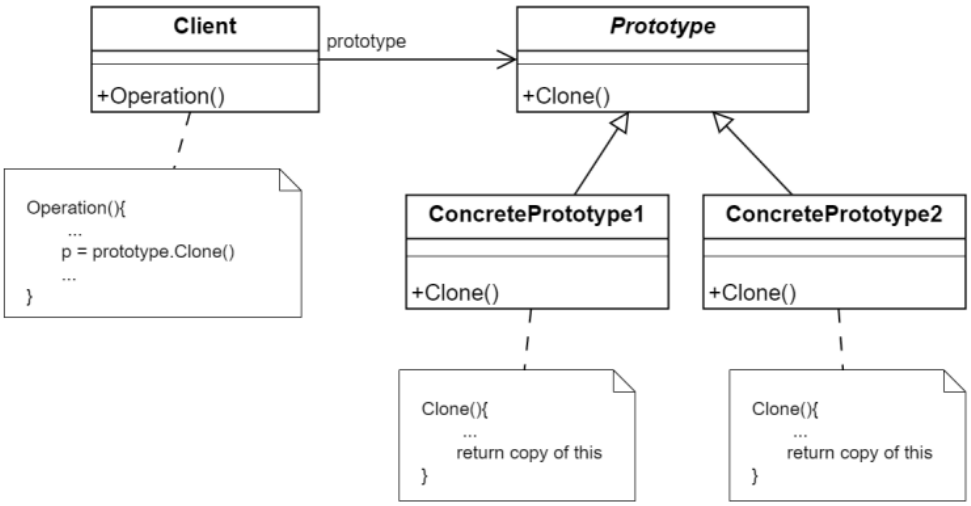
Шаблон «Команда» працює, створюючи рівень абстракції між тим, *хто*

ініціює дію, і тим, *хто* її виконує. Ініціатор (наприклад, кнопка "Зберегти") не знає нічого про те, як відбувається збереження. Він знає лише, що у нього є об'єкт-Команда з методом execute(). Коли кнопку натискають, вона просто викликає цей метод. А вже сам об'єкт-Команда (наприклад, SaveCommand) знає, що для виконання цієї дії потрібно викликати метод saveDocument() у об'єкта Document. Це дозволяє мати багато різних кнопок, які виконують ту саму команду, або одну кнопку, яка в різний час виконує різні команди.

13. Яке призначення шаблону «Прототип»?

Призначення шаблону «Прототип» — надати можливість створювати нові об'єкти шляхом копіювання (клонування) вже існуючого об'єкта, який називається прототипом. Це використовується, коли процес створення об'єкта "з нуля" (через конструктор new) є дуже дорогим або складним, наприклад, вимагає завантаження даних з бази даних або складних обчислень.

14. Нарисуйте структуру шаблону «Прототип».



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

В шаблон «Прототип» входять Інтерфейс Прототипу (який має метод clone()) та Конкретні класи Прототипи, які реалізують цей метод. Взаємодія така: Клієнт, замість того, щоб писати new MyObject(config), бере вже існуючий об'єкт-прототип (наприклад, defaultConfigObject) і викликає у нього myNewObject = defaultConfigObject.clone(). Цей метод створює повну копію об'єкта. Це значно швидше, ніж створювати об'єкт з нуля, якщо його початкова конфігурація складна.

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

Прикладів використання шаблону «Ланцюжок відповідальності» дуже багато. Класичний приклад — обробка подій в графічному інтерфейсі. Коли ви клікаєте мишкою на кнопку, спочатку подія "клік" відправляється самій кнопці. Якщо кнопка її не обробляє, подія "спливає" вгору до контейнера (панелі), на якій лежить кнопка. Якщо панель не обробляє, подія йде далі до вікна програми. Це і є ланцюжок. Інший чудовий приклад — фільтри у вебсервері (middleware), як у вашому проєкті. Запит проходить через ланцюжок: перший обробник перевіряє автентифікацію, другий — логує запит, третій — перевіряє кеш, і лише останній реально виконує запит. Також цей патерн використовують банкомати для видачі готівки: перший обробник видає купюри по 500, другий — по 200, і так далі по ланцюжку.

Висновки

Під час виконання даної лабораторної роботи я ознайомився та вивчив такі патерни програмування: adapter, builder, command, chain of responsibility, prototype. На власному застосунку я виконав реалізаіцію патерну chain of responsibility при обробці запиту користувача до зовнішнього веб-ресурсу. Я створив ланцюг відповідальностей, який складає збереження в історії та рендер HTML.