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

Технології розробки програмного забезпечення

«Вступ до патернів проектування»

«Веб-браузер»

Виконав:
студент групи ІА-33
Мартинюк Ю.Р.

Перевірив:
Мягкий Михайло Юрійович

Київ 2025

Тема: Вступ до патернів проектування

Мета: Вивчити структуру шаблонів «Singleton», «Iterator», «Proxy», «State», «Strategy» 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

Будь-який патерн проєктування, використовуваний при розробці інформаційних систем, являє собою формалізований опис, який часто зустрічається в завданнях проєктування, вдале рішення даної задачі, а також рекомендації по застосуванню цього рішення в різних ситуаціях. Крім того, патерн проєктування обов’язково має загальновживане найменування. Правильно сформульований патерн проєктування дозволяє, відшукавши одного разу вдале

рішення, користуватися ним знову і знову. Варто підкреслити, що важливим початковим етапом при роботі з патернами є адекватне моделювання розглянутої предметної області. Це є необхідним як для отримання належним чином формалізованої постановки задачі, так і для вибору відповідних патернів проєктування.

Відповідне використання патернів проєктування дає розробнику ряд незаперечних переваг. Наведемо деякі з них. Модель системи, побудована в межах патернів проєктування, фактично є структурованим виокремленням тих елементів і зв'язків, які значимі при вирішенні поставленого завдання. Крім цього, модель, побудована з використанням патернів проєктування, більш проста і наочна у вивченні, ніж стандартна модель. Проте, не дивлячись на простоту і наочність, вона дозволяє глибоко і всебічно опрацювати архітектуру розроблюваної системи з використанням спеціальної мови.

Застосування патернів проєктування підвищує стійкість системи до зміни вимог та спрощує неминуче подальше доопрацювання системи. Крім того, важко переоцінити роль використання патернів при інтеграції інформаційних систем організації. Також слід зазначити, що сукупність патернів проєктування, по суті, являє собою єдиний словник проєктування, який, будучи уніфікованим засобом, незамінний для спілкування розробників один одним.

Таким чином шаблони представляють собою, підтверджені роками розробок в різних компаніях і на різних проєктах, «ескізи» архітектурних рішень, які зручно застосовувати у відповідних обставинах.

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: Proxy

Посилання на Github: https://github.com/masonabor/web-browser-lab/tree/main/lab4

Хід роботи

Діаграма класів

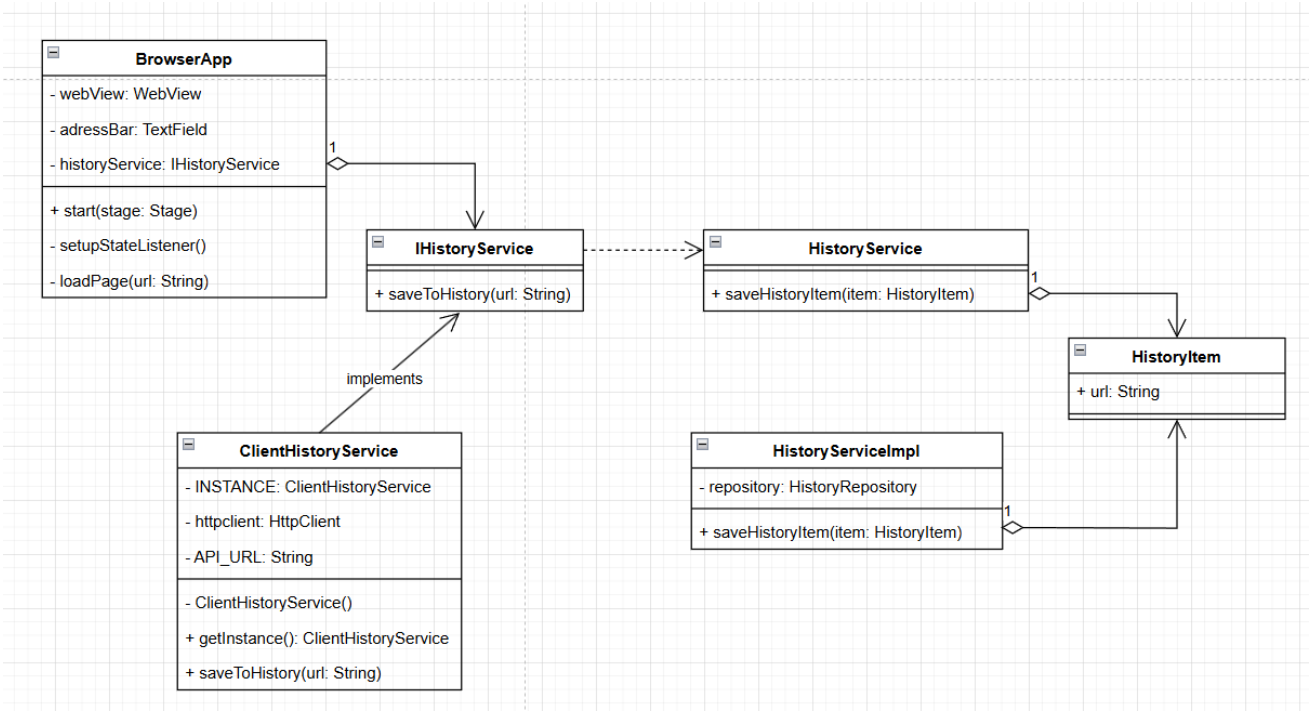


Рис.1 – Діаграма класів для HistoryService (патерн Proxy)

Опис Діаграми Класів

Ця діаграма демонструє архітектуру клієнт-серверної взаємодії для функції збереження історії браузера. Діаграма чітко розділена на дві логічні частини:

- Клієнтська сторона (ліворуч): Компоненти, що працюють у BrowserApp.
- Серверна сторона (праворуч): Компоненти, що представляють віддалений бекенд-сервіс.

Клієнтська сторона:

- BrowserApp: Це головний клас UI-додатку. Він керує вікном, містить WebView для відображення сторінок та TextField (adressBar). Найголовніше — він має залежність (1) від інтерфейсу IHistoryService.
- IHistoryService: Це інтерфейс (контракт) на стороні клієнта. Він визначає єдиний метод saveToHistory(url: String). BrowserApp працює лише з цим інтерфейсом, не знаючи про деталі його реалізації.
- ClientHistoryService: Це конкретна реалізація IHistoryService. Цей клас реалізований як Singleton (має приватний конструктор та статичний метод getInstance()). Він містить HttpClient та API_URL, що прямо вказує на його призначення — виконувати мережеві запити.

Серверна сторона (концептуально):

- HistoryService: Це представлення (можливо, інтерфейс) сервісу на бекенді.
- HistoryServiceImpl: Це реальна реалізація сервісу, яка містить HistoryRepository (не показаний) і виконує фактичну бізнес-логіку збереження.
- HistoryItem: Це об'єкт даних (модель домену), який використовується на сервері для представлення запису історії.

Пояснення Патерну «Proxy» (Замісник)
Ця діаграма є класичним прикладом патерну Remote Proxy (Віддалений Замісник). Ось як він тут реалізований:

1. Інтерфейс (The Subject): IHistoryService Це спільний контракт, який визначає, *що* має робити сервіс. BrowserApp (клієнт) знає лише про цей інтерфейс.
2. Реальний Об'єкт (The Real Subject): HistoryServiceImpl Це справжній сервіс, який виконує роботу (зберігає в базу даних). Він "важкий" і, що найголовніше, знаходиться на іншому комп'ютері (на сервері). BrowserApp не може звернутися до нього напряму.
3. Замісник (The Proxy): ClientHistoryService Це ключовий елемент. Цей клас:
    - Реалізує той самий інтерфейс IHistoryService.
    - "Прикидається" справжнім сервісом.
    - Для BrowserApp виклик historyService.saveToHistory(...) виглядає як звичайний локальний виклик методу.

Як це працює на практиці:

1. BrowserApp хоче зберегти історію і викликає метод saveToHistory("google.com") у свого об'єкта historyService.
2. Насправді, historyService — це екземпляр ClientHistoryService (Proxy).
3. ClientHistoryService не зберігає нічого сам. Натомість він бере отриманий url, використовує свій HttpClient та API_URL, формує HTTP-запит (наприклад, POST-запит з JSON) і відправляє його по мережі на сервер.
4. HistoryServiceImpl (Реальний Об'єкт) на сервері отримує цей запит, виконує всю складну роботу (валідацію, збереження в БД) і повертає відповідь.
5. ClientHistoryService (Proxy) отримує відповідь і завершує роботу.

Перевага: Патерн Proxy повністю приховує (інкапсулює) всю складність мережевої взаємодії від BrowserApp. Ваш BrowserApp залишається чистим і не знає нічого про HTTP, JSON чи IP-адреси, роблячи код значно простішим у підтримці.

Частина коду програми з використанням патерну Proxy

BrowserApp.java

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

import com.edu.web.browserapp.service.IHistoryService;
```

```
import com.edu.web.browserapp.service.impl.ClientHistoryService;
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 IHistoryService historyService;

    @Override
    public void start(Stage stage) {
        historyService = ClientHistoryService.getInstance();

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

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


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

            if (newState == Worker.State.SUCCEEDED) {
```

```
            String loadedUrl = webView.getEngine().getLocation();
            statusLabel.setText("Завантажено: " + loadedUrl);

            historyService.saveToHistory(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("http://") && !url.startsWith("https://")) {
      url = "https://" + url;
    }
    webView.getEngine().load(url);
  }
}
```

IHistoryService.java

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

public interface IHistoryService {
  void saveToHistory(String url);
}
```

ClientHistoryService.java

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

import com.edu.web.browserapp.service.IHistoryService;

import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

public class ClientHistoryService implements IHistoryService {

   private static final ClientHistoryService INSTANCE = new ClientHistoryService();

   private final HttpClient httpClient;
   private final String API_URL = "http://localhost:8080/browser-service-
1.0/api/history"; // Оновіть назву .war

   private ClientHistoryService() {
      this.httpClient = HttpClient.newBuilder()
            .version(HttpClient.Version.HTTP_1_1)
            .build();
   }

   public static ClientHistoryService getInstance() {
      return INSTANCE;
   }

   @Override
   public void saveToHistory(String url) {
      try {
         String jsonPayload = "{\"url\":\"" + url.replace("\"", "\\\"") + "\"}";

         HttpRequest request = HttpRequest.newBuilder()
               .uri(URI.create(API_URL))
               .header("Content-Type", "application/json")
               .POST(HttpRequest.BodyPublishers.ofString(jsonPayload))
               .build();

         httpClient.sendAsync(request, HttpResponse.BodyHandlers.ofString())
```

```
                .thenAccept(this::handleResponse)
                .exceptionally(this::handleError);

        } catch (Exception e) {
            System.err.println("[Proxy] Error during request: " + e.getMessage());
        }
    }

    private void handleResponse(HttpResponse<String> response) {
        if (response.statusCode() == 201) {
            System.out.println("[Proxy] History successfully saved");
        } else {
            System.err.println("[Proxy] Server error" + response.statusCode());
        }
    }

    private Void handleError(Throwable ex) {
        System.err.println("[Proxy] Error during connecting to History Service: " +
ex.getMessage());
        return null;
    }

}
```

HistoryService.java

```
package com.edu.web.restservicewebbrowser.service;

import com.edu.web.restservicewebbrowser.domain.HistoryItem;

import java.sql.SQLException;

public interface HistoryService {
    void saveHistoryItem(HistoryItem historyItem) throws SQLException;
}
```

HistoryServiceImpl.java

```
package com.edu.web.restservicewebbrowser.service.impl;

import com.edu.web.restservicewebbrowser.domain.HistoryItem;
import com.edu.web.restservicewebbrowser.repository.HistoryRepository;
```

```
import com.edu.web.restservicewebbrowser.service.HistoryService;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;

import java.net.URI;
import java.net.URISyntaxException;
import java.sql.SQLException;
import java.util.Set;

@ApplicationScoped
public class HistoryServiceImpl implements HistoryService {

    private static final Set<String> IGNORED_PROTOCOLS = Set.of(
            "about",
            "chrome",
            "file",
            "data"
    );

    private static final int MAX_URL_LENGTH = 2048;

    @Inject
    private HistoryRepository repository;

    public void saveHistoryItem(HistoryItem item) throws SQLException {

        if (item == null || item.url() == null || item.url().isBlank()) {
            throw new IllegalArgumentException("History item або URL не може бути
порожнім.");
        }

        String urlString = item.url().trim();
        URI uri;

        try {
            uri = new URI(urlString);

            if (!uri.isAbsolute()) {
                throw new URISyntaxException(urlString, "URL повинен мати протокол
(scheme).");
            }
```

```
        } catch (URISyntaxException e) {
            throw new IllegalArgumentException("Некоректний формат URL: " +
e.getMessage(), e);
        }

        String protocol = uri.getScheme();

        if (protocol == null ||
IGNORED_PROTOCOLS.contains(protocol.toLowerCase())) {
            System.out.println("Ігнорування URL з протоколом: " + protocol);
            return;
        }

        if (urlString.length() > MAX_URL_LENGTH) {
            throw new IllegalArgumentException("URL занадто довгий (макс " +
MAX_URL_LENGTH + " символів).");
        }

        repository.save(item);
    }
}
```

HistoryItem.java

```
package com.edu.web.restservicewebbrowser.domain;

public record HistoryItem(String url) {
}
```

Опис коду з використанням Proxy

Цей набір класів є канонічним прикладом шаблону проєктування Remote Proxy (Віддалений Замісник).

Ось детальний опис того, як ваш код реалізує цей патерн, та яку роль відіграє кожен клас.

Загальна ідея: "Офіціант та Кухня"

Уявіть, що BrowserApp — це клієнт у ресторані. HistoryServiceImpl — це кухня, де виконується складна робота (приготування їжі / збереження в БД).

Клієнт (BrowserApp) не йде сам на кухню. Він кличе ClientHistoryService (Офіціанта/Проксі), який має такий самий "інтерфейс" (IHistoryService) і

приймає просте замовлення. Офіціант (Проксі) бере на себе всю складну роботу: біжить на кухню (робить HTTP-запит), перекладає замовлення на мову кухарів (пакує в JSON) і приносить відповідь.

Ролі учасників патерну у вашому коді

1. "Клієнт" (Client)
   - Клас: BrowserApp.java
   - Роль: Це ініціатор запиту. BrowserApp поняття не має про те, що існує якийсь сервер, HttpClient чи JSON. Він знає лише про одне: у нього є об'єкт historyService, який реалізує простий інтерфейс IHistoryService.
   - Ключовий рядок: historyService.saveToHistory(loadedUrl);
     - BrowserApp просто просить свого "помічника" зберегти URL. Він не знає, *як* той це зробить.
2. "Інтерфейс" (Subject)
   - Клас: IHistoryService.java
   - Роль: Це спільний "контракт" або "мова", яку розуміють і Клієнт, і Проксі. Він гарантує, що BrowserApp знає, який метод викликати, а ClientHistoryService знає, який метод реалізувати.
3. "Замісник" (Proxy)
   - Клас: ClientHistoryService.java
   - Роль: Це "офіціант" — ключовий елемент патерну. Він видає себе за реальний сервіс, реалізуючи IHistoryService.
   - Що він робить:
     1. Отримує простий виклик saveToHistory("http://google.com").
     2. Приховує всю складність: Замість того, щоб зберігати дані, він перетворює String на jsonPayload.
     3. Використовує HttpClient та API_URL для надсилання даних по мережі.
     4. Асинхронно обробляє відповідь (handleResponse / handleError).
   - Перевага: Вся логіка мережевої взаємодії інкапсульована (захована) тут.
4. "Реальний Об'єкт" (Real Subject)
   - Клас: HistoryServiceImpl.java (на сервері)
   - Роль: Це "кухня". Цей клас отримує запит, що надійшов по мережі від Проксі.
   - Що він робить:
     1. Отримує HistoryItem (розпарсений з JSON).
     2. Виконує реальну бізнес-логіку: проводить складну валідацію (перевіряє IGNORED_PROTOCOLS, MAX_URL_LENGTH тощо).

3. Викликає repository.save(item) для фактичного збереження в базу даних.

Скриншот застосунку

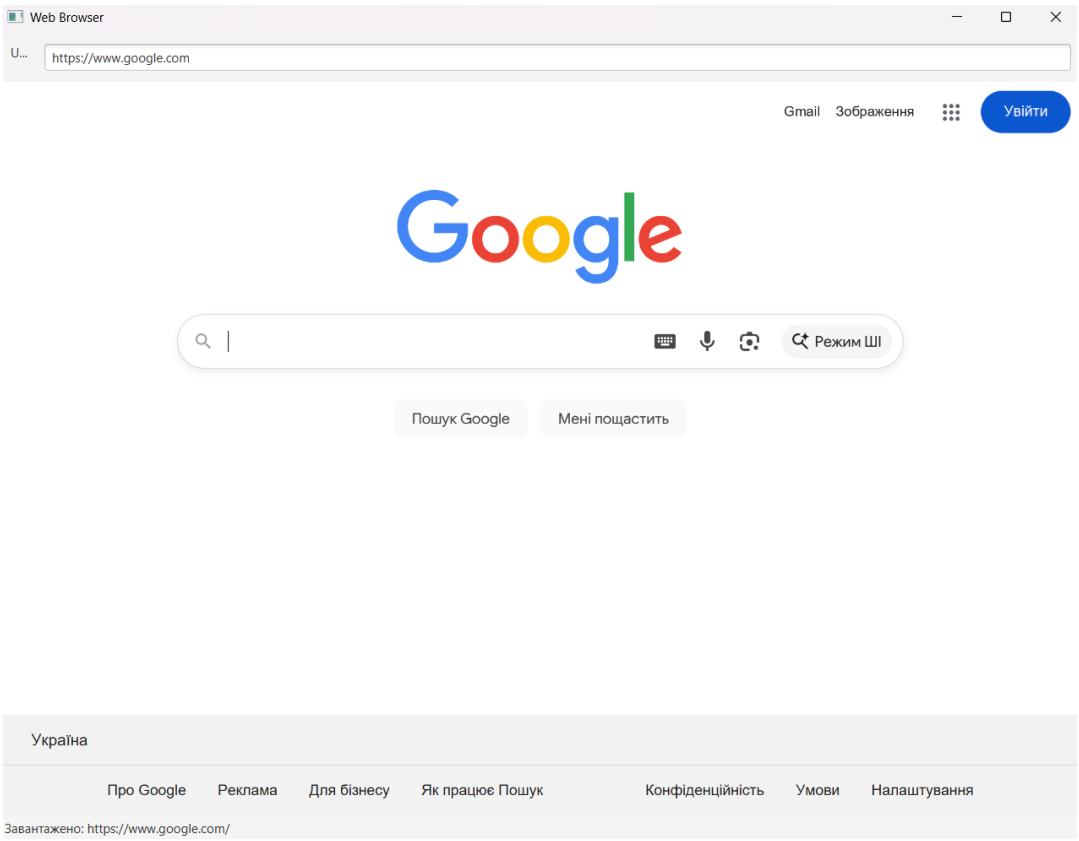


Рис.2 – Скриншот застосунку BrowserApp

Контрольні запитання

1. Що таке шаблон проєктування?

Шаблон проєктування (або "дизайн-патерн") — це, по суті, готовий рецепт для вирішення типових проблем, з якими програмісти стикаються щодня.

Уявіть, що ви будуєте будинок. Вам не потрібно щоразу "винаходити" двері, вікна чи сходи. Ви берете перевірені часом конструкції. Так само і в програмуванні: патерни — це перевірені та надійні способи організації коду для таких завдань, як створення об'єктів, керування їхньою поведінкою або взаємодія між ними.

2. Навіщо використовувати шаблони проєктування?

Використання патернів робить життя програміста набагато простішим:

- Не треба винаходити велосипед: Ви берете готове, надійне

рішення, яке вже довело свою ефективність. Це економить час і зменшує кількість помилок.

- Спільна мова: Це як професійний сленг. Коли один розробник каже: "Давай тут використаємо 'Одинака'", інший одразу розуміє, про що йдеться, без довгих пояснень.
- Код легше читати: Коли ви бачите знайому структуру патерну в чужому коді, ви набагато швидше розумієте, що цей код робить і чому він написаний саме так.
- Гнучкість: Патерни роблять ваш код гнучкішим, тобто його легше змінювати чи доповнювати в майбутньому, не ламаючи все, що вже працює.

3. Яке призначення шаблону «Стратегія»?

«Стратегія» призначена для того, щоб мати сімейство взаємозамінних алгоритмів (стратегій) і дозволяти вибирати потрібний під час виконання програми.

Уявіть, що ви створюєте навігатор. Користувач може захотіти прокласти маршрут.

Кожен із них — це окрема стратегія прокладання маршруту. Сам навігатор (контекст) не знає, як саме будується маршрут, він просто каже: "Поточна стратегіє, побудуй маршрут!" Це дозволяє легко додавати нові стратегії, не змінюючи сам навігатор.

4. Нарисуйте структуру шаблону «Стратегія».

Структура досить проста і складається з трьох ключових частин:

- Контекст (Context): Це головний клас, який використовує стратегію (наш Навігатор).
- Інтерфейс Стратегії (IStrategy): Це загальний "контракт", який описує, що має вміти кожна стратегія (наприклад, метод buildRoute()).
- Конкретні Стратегії (ConcreteStrategy A, B): Це реалізації алгоритмів (CarStrategy, WalkStrategy).

5. Які класи входять в шаблон «Стратегія», та яка між ними взаємодія?

Як ми бачили вище, це три типи класів:

1. Контекст (Context): Це клас, поведінку якого ми хочемо змінювати (наш Навігатор). Він має посилання на один об'єкт стратегії.
2. Інтерфейс Стратегії (Strategy): Оголошує один або декілька методів (напр., execute()), які будуть реалізовані всіма стратегіями.

3. Конкретна Стратегія (ConcreteStrategy): Це класи, що реалізують інтерфейс стратегії, кожен по-своєму (напр., CarStrategy шукає дороги, а WalkStrategy — стежки).

Клієнтський код дає Контексту потрібну йому Конкретну Стратегію. Коли клієнт викликає метод у Контексту (напр., navigator.createRoute()), Контекст просто делегує (передає) цей виклик тій стратегії, яку він наразі тримає.

6. Яке призначення шаблону «Стан»?

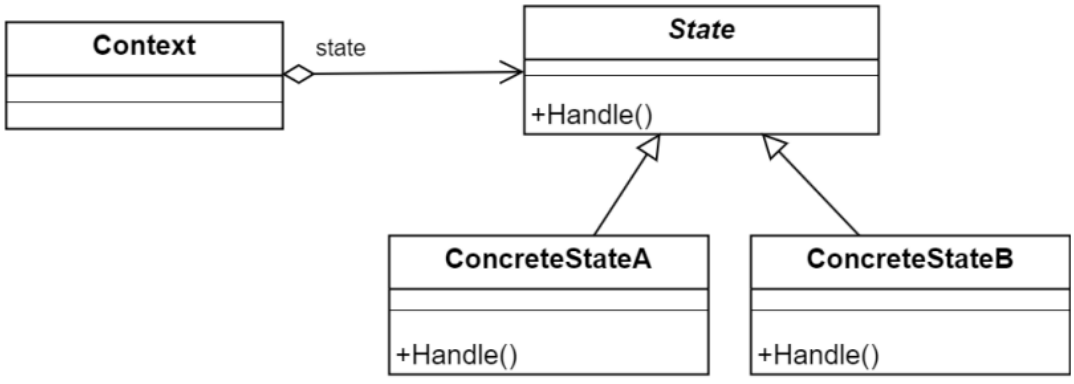
«Стан» використовується, коли об'єкт має кардинально змінювати свою поведінку залежно від свого внутрішнього стану.

Проста аналогія: Уявіть торговий автомат.

- Коли він у стані "Немає грошей", натискання кнопки нічого не робить.
- Але якщо ви кинете монету, автомат перейде у стан "Є гроші".
- Тепер натискання тієї ж самої кнопки видасть вам напій і поверне автомат у стан "Немає грошей".

Поведінка методу pressButton() повністю залежить від поточного стану. Патерн "Стан" дозволяє "винести" всю цю логіку поведінки для кожного стану в окремі класи.

7. Нарисуйте структуру шаблону «Стан».



8. Які класи входять в шаблон «Стан», та яка між ними взаємодія?

1. Контекст (Context): (наш ТорговийАвтомат) Зберігає посилання на свій поточний об'єкт-стан.
2. Інтерфейс Стану (State): Описує, що може робити кожен стан.
3. Конкретний Стан (ConcreteState): (напр., NoMoneyState) Реалізує

поведінку для конкретного стану.

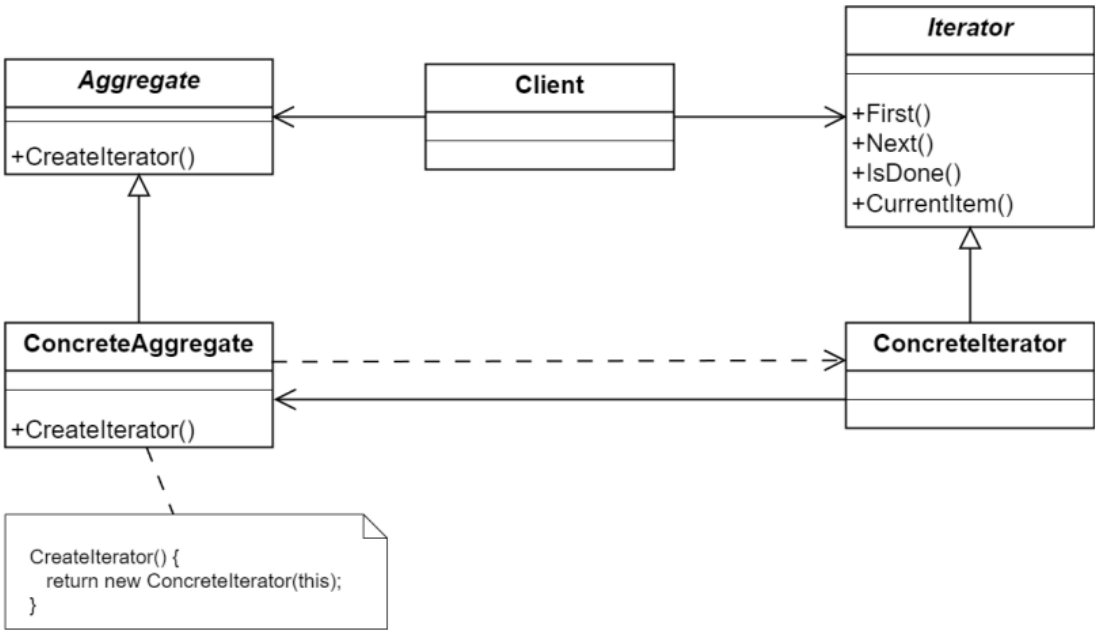
Взаємодія: Контекст делегує всі виклики (напр., insertCoin()) своєму поточному стану. Але найголовніше — самі об'єкти-стани вирішують, коли і в який стан переходити. Наприклад, NoMoneyState у своєму методі insertCoin() сам скаже Контексту: "Все, гроші отримано, тепер твій новий стан — HasMoneyState".

9. Яке призначення шаблону «Ітератор»?

«Ітератор» дозволяє отримати послідовний доступ до всіх елементів складної колекції (списку, дерева, хеш-таблиці), не розкриваючи, як ця колекція влаштована всередині.

Проста аналогія: Це як пульт від телевізора. У вас є кнопки "Наступний канал" (next()) і "Попередній канал". Вам не потрібно знати, як телевізор всередині зберігає ці канали (в пам'яті, на супутнику). Ітератор дає вам простий набір кнопок (hasNext? і next()) для перебору будь-якої колекції.

10. Нарисуйте структуру шаблону «Ітератор».



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

1. Інтерфейс Колекції (Aggregate): Оголошує метод createIterator().
2. Конкретна Колекція: (MyArrayList): Клас, що зберігає елементи. Він реалізує createIterator(), повертаючи новий екземпляр свого ітератора.
3. Інтерфейс Ітератора (Iterator): Оголошує методи, зазвичай boolean

hasNext() (чи є ще елементи?) та Object next() (повернути наступний елемент).

4. Конкретний Ітератор: (MyArrayListIterator): Знає про внутрішню структуру MyArrayList і зберігає поточну позицію (індекс) при переборі.

Взаємодія: Клієнтський код просить у Колекції її Ітератор. Отримавши його, клієнт працює тільки з Ітератором, викликаючи hasNext() та next() у циклі, доки елементи не закінчаться.

12. В чому полягає ідея шаблона «Одинак»?

«Одинак» — це шаблон, який гарантує, що у класу може бути лише один-єдиний екземпляр (об'єкт) на всю програму, і надає єдину (глобальну) точку доступу до нього.

Проста аналогія: Це як уряд у країні. Уряд (зазвичай) буває лише один. Якщо будь-яка частина програми (громадянин) хоче звернутися до уряду, вона звертається до цього єдиного, загальновідомого екземпляра. Це часто використовують для підключення до бази даних, налаштувань програми або ведення логів.

13. Чому шаблон «Одинак» вважають «анти-шаблоном»?

Незважаючи на свою корисність, "Одинак" має погану репутацію, тому що:

- Це по суті глобальна змінна: Будь-яка частина коду може отримати до нього доступ і змінити його стан. Це може спричинити хаос і помилки, які дуже важко відстежити.
- Ускладнює тестування: Коли ви пишете тести (юніт-тести), ви хочете тестувати класи в ізоляції. Але ви не можете "підмінити" справжнього Одинака на фальшивого (мок-об'єкт) під час тестування.
- Приховані залежності: Клас може виглядати простим, але всередині себе він може викликати Singleton.getInstance(). Це прихована залежність, яка робить код менш гнучким.

14. Яке призначення шаблону «Проксі»?

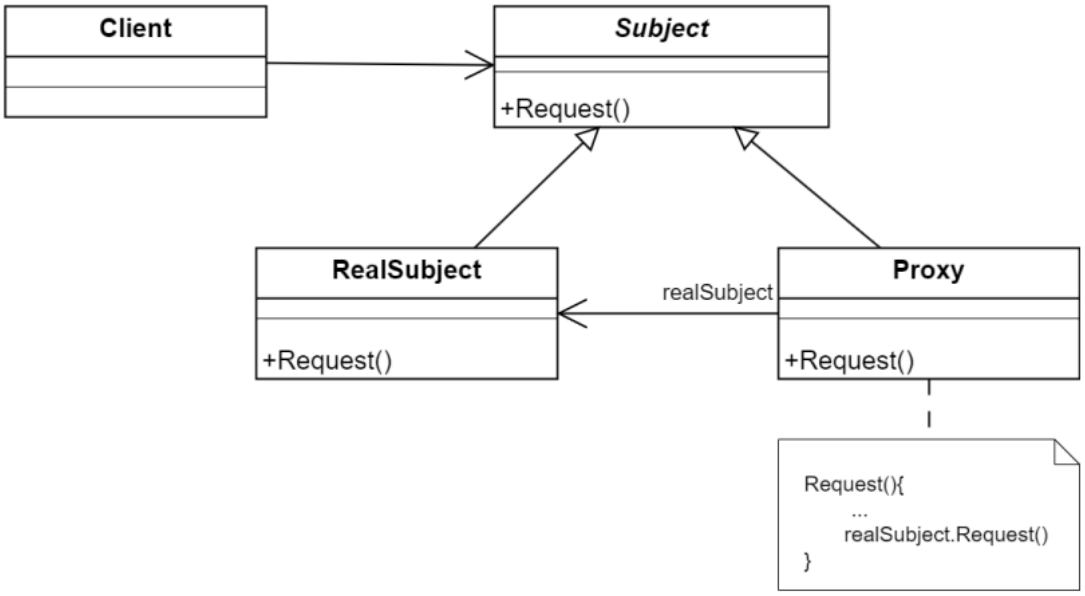
«Проксі» — це об'єкт-"замісник", який видає себе за інший об'єкт. Він виступає як посередник, що дозволяє контролювати доступ до реального об'єкта.

Проста аналогія: Це як охоронець на вході до VIP-клубу.

- Ви (клієнт) підходите до охоронця (Проксі).

- Охоронець має той самий "інтерфейс", що й клуб (ви можете попросити його "увійти").
- Але він додає свою логіку:
    1. Перевірка доступу: "Чи є ви у списках?"
    2. Ліниве завантаження: "Клуб ще не відкрився, зачекайте" (Проксі "створить" реальний об'єкт, лише коли він справді потрібен).
    3. Віддалений доступ: (Як у вашому проєкті) Охоронець може бути на зв'язку з клубом по рації, передаючи ваші запити.

15. Нарисуйте структуру шаблону «Проксі».



16. Які класи входять в шаблон «Проксі», та яка між ними взаємодія?

1. Інтерфейс (Subject): Описує спільні операції (saveToHistory()).
2. Реальний Об'єкт (RealSubject): Виконує основну, корисну роботу (HistoryServiceImpl, що реально лізе в базу даних).
3. Проксі (Proxy): (ClientHistoryService у вашому випадку).

Взаємодія: Клієнт (ваш BrowserApp) нічого не знає про RealSubject. Він створений так, що працює тільки з Інтерфейсом (IHistoryService).

1. Клієнту дають екземпляр Проксі.
2. Клієнт викликає метод на Проксі (напр., proxy.saveToHistory(.)).
3. Проксі перехоплює цей виклик. Він виконує свою додаткову логіку (у вашому випадку — пакує дані в JSON, відправляє HTTP-запит на сервер).
4. (Якщо потрібно) Проксі викликає той самий метод на Реальному

Об'єкті (у вашому випадку, він викликає його через мережу).

5. Проксі повертає результат клієнту.

Висновки

Під час виконання даної лабораторної роботи я ознайомився з такими патернами проектування як Singleton, Strategy, Iterator, Proxy та State. На власному застосунку я застосував патерн Proxy для HistoryService, а саме імплементація сервісу IHistoryService на стороні клієнта є об`єктом-замінником справжнього сервісу для збереження історії браузера HistoryService на стороні сервера, даний клас надсилає дані справжньому сервісу на стороні сервера, а застосунок браузера як клієнт використовує проксі.