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 №7

Технології розроблення програмного забезпечення

Тема роботи: « ШАБЛОН «MEDIATOR», «FACADE», «BRIDGE», «TEMPLATE METHOD»*»*

Виконав студент групи ІА-23:

Каширов Д. О.

Перевірив:

Мягкий Михайло Юрійович

Київ 2024

Мета роботи: полягає в освоєнні патерну проектування Mediator та його застосуванні для організації взаємодії між компонентами без безпосереднього зв'язку між ними, що дозволяє покращити масштабованість і зменшити складність програмної системи.

Хід роботи

Варіант:

..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.

Завдання.

1. Ознайомитися з короткими теоретичними відомостями

2. Реалізувати частину функціонала робочої програми у вигляді класів і їх взаємодій для досягнення конкретних функціональних можливостей.

3. Застосування одного з даних шаблонів при реалізації програми

1. **Ознайомлення з патерном Mediator:** Спочатку було здійснене вивчення патерну проектування **Mediator**, який дозволяє організувати взаємодію між різними компонентами програми через посередника, зменшуючи прямі залежності між ними. Це дозволяє забезпечити більш гнучку та масштабовану архітектуру.
2. **Реалізація класу Mediator:** Створено клас **Mediator**, який містить методи для реєстрації обробників подій (метод register) та для сповіщення зареєстрованих слухачів про події (метод notify). Клас реалізує централізовану точку взаємодії між різними компонентами.
3. **Інтеграція класу Mediator в систему:** У коді було використано клас **Mediator** для організації взаємодії між компонентами програми. Прикладом є обробка запитів на завантаження сторінок та взаємодія між проксі-сервером і браузером, де кожен компонент отримує лише необхідні дані через **Mediator**.
4. **Тестування роботи Mediator:** Після реалізації класу **Mediator** було здійснено тестування його роботи, реєструючи обробники для різних подій та перевіряючи, чи коректно спрацьовують всі оповіщення. В результаті, всі компоненти успішно взаємодіяли через посередника, що підтвердило правильність реалізації.
5. **Оцінка ефективності:** Використання патерну Mediator значно спростило організацію взаємодії між компонентами, що призвело до зменшення зв'язності в програмі та полегшення її подальшого розширення.

Короткі теоретичні відомості:

1. **Шаблон «Mediator» (Посередник):** Патерн **Mediator** дозволяє організувати централізовану точку взаємодії між різними компонентами програми, усуваючи необхідність прямої взаємодії між ними. Це знижує залежність між компонентами, роблячи систему більш гнучкою та легкою для модифікацій. Основним елементом цього патерну є **Mediator** — клас, який приймає та передає повідомлення між компонентами. Замість того, щоб компоненти безпосередньо взаємодіяли між собою, вони надсилають повідомлення до **Mediator**, який у свою чергу обробляє це повідомлення та пересилає іншим компонентам.
2. **Шаблон «Facade» (Фасад):** Патерн **Facade** забезпечує спрощений інтерфейс для взаємодії з комплексною підсистемою. Його основне завдання — приховати складність системи, надаючи зручніші та більш абстрактні методи для користувачів. Завдяки **Facade** користувач або інші компоненти можуть взаємодіяти із системою через один клас, який обробляє запити і викликає необхідні компоненти підсистеми. Це дозволяє значно знизити складність взаємодії та покращити читабельність коду.
3. **Шаблон «Bridge» (Міст):** Патерн **Bridge** дозволяє розділяти абстракцію і реалізацію таким чином, щоб зміни в одній частині не впливали на іншу. Він використовує два основних компоненти: абстракцію та її реалізацію. Абстракція є основною логікою, яку використовує користувач, а реалізація надає конкретну реалізацію функціональності. Завдяки цьому патерну можна змінювати абстракцію або реалізацію окремо без необхідності переписувати інші частини системи, що сприяє зниженню зв'язності.
4. **Шаблон «Template Method» (Метод Шаблону):** Патерн **Template Method** визначає структуру алгоритму, залишаючи деякі кроки алгоритму реалізованими в підкласах. Це дозволяє спільно використовувати алгоритм у вигляді шаблону, одночасно дозволяючи підкласам змінювати деякі етапи цього алгоритму. Основна мета — створити загальну структуру алгоритму, зберігаючи можливість кастомізації окремих етапів для конкретних реалізацій. Патерн корисний, коли алгоритм має незмінну структуру, але деякі кроки можуть бути змінені в підкласах для досягнення різних результатів.

Діаграма реалізації паттерну Mediator:

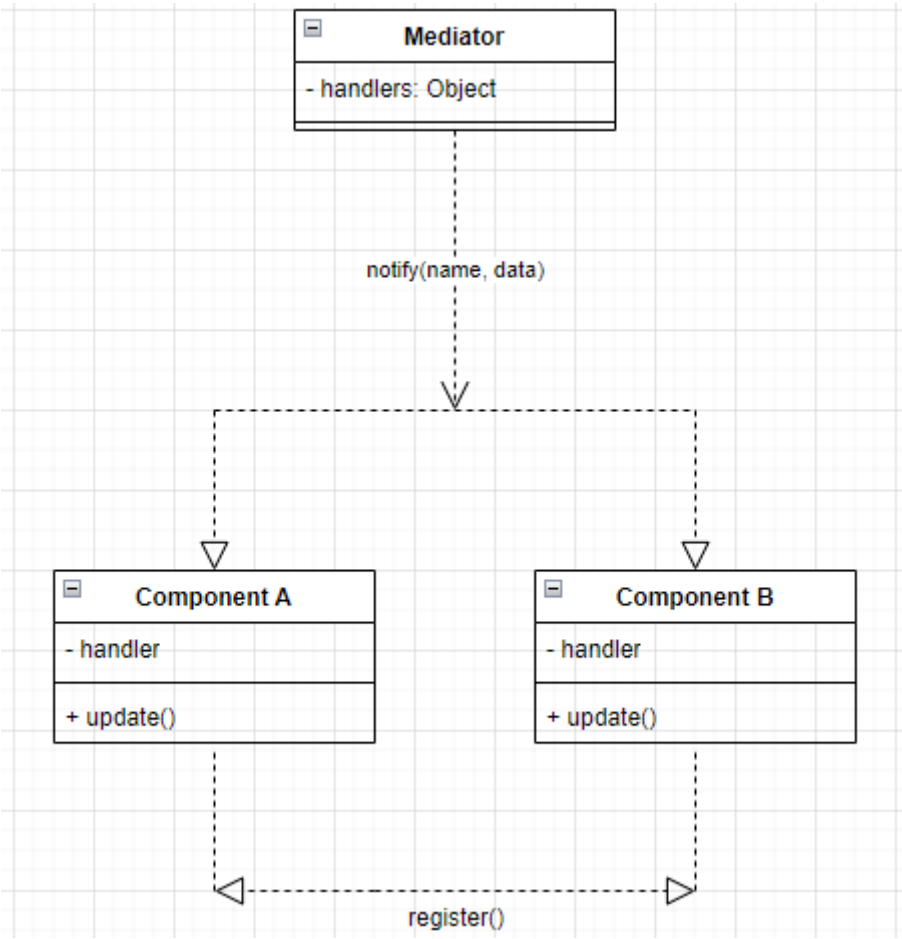

Рис 1- реалізації паттерну Mediator

Опис компонентів діаграми паттерну Mediator:

1. **Mediator (Посередник)**:
    - **Опис**: Це центральний клас, який відповідає за реєстрацію та взаємодію компонентів. Він слугує "посередником", через який компоненти можуть обмінюватися повідомленнями або даними, не знаючи один про одного.
    - **Методи**:
        - register(name, handler) — реєструє обробники для певного компонента під певним іменем.
        - notify(name, data) — сповіщає всі зареєстровані обробники для конкретного компонента про нові дані або події.
2. **Component A (Компонент А)**:

- **Опис**: Один з компонентів, який взаємодіє через Mediator. Компонент може бути будь-якою частиною програми (наприклад, користувацький інтерфейс, бізнес-логіка тощо).
- **Методи**:
  - update(data) — метод для оновлення або обробки даних, отриманих від Mediator. Кожен компонент має власний метод для обробки отриманих повідомлень або подій.

3. **Component B (Компонент B)**:
   - **Опис**: Інший компонент, який взаємодіє з Mediator, подібно до Компонента A. Обидва компоненти можуть бути різними частинами системи, і їх взаємодія контролюється через Mediator.
   - **Методи**:
     - update(data) — цей метод викликається, коли Mediator передає дані компоненту. Компонент обробляє ці дані відповідно до своєї логіки.

Взаємодія компонентів:

- **Mediator** реєструє компоненти через метод register(), де кожен компонент додає свій обробник (handler). Це дозволяє Mediator знати, які компоненти потрібно сповіщати про події.
- Коли **Component A** або **Component B** хоче повідомити інший компонент про подію або зміну стану, вони не викликають методи іншого компонента безпосередньо. Натомість вони викликають метод notify() в Mediator, який передає дані або повідомлення всім відповідним компонентам, зареєстрованим для цього.
- **Компоненти** обробляють отримані дані через метод update(), що дозволяє їм реагувати на зміни або події, не залежачи від того, які саме компоненти їх викликають. Це знижує рівень зв'язності в програмі.

Вхідні коди з проекту **LiteBrowse** з паттерном **Mediator:**

Створення файле для паттерну:

Файл Mediator.js


```
class Mediator {
    constructor() {
      this.handlers = {};
    }

    // Регистрируем обработчик
    register(name, handler) {
      if (!this.handlers[name]) {
        this.handlers[name] = [];
      }
      this.handlers[name].push(handler);
    }

    // Оповещаем всех слушателей
    notify(name, data) {
      if (this.handlers[name]) {
        this.handlers[name].forEach(handler => handler.update(data));
      }
    }
  }

  module.exports = Mediator;
```

Також додали підключення до класу mediator в файлах такі як: **app.js browserRoutes.js**

Файл browserRoutes.js

```
const Notifier = require('../observers/Notifier');
const Mediator = require('../observers/Mediator');
```

Файл app.js

```
const Logger = require('./observers/Logger');
const Notifier = require('./observers/Notifier');

// Mediator
const Mediator = require('./observers/Mediator'); // Правильний шлях
```

Файл Logger.js

```
const Observer = require('./Observer');

class Logger extends Observer {
  constructor(mediator) {
    super();
    this.mediator = mediator; // Посилання на Mediator
  }

  update(data) {
    console.log(`[Logger] Подія: ${data.event}, Деталі:
${JSON.stringify(data.details)}`);

    // Повідомлення Mediator про подію
    if (this.mediator) {
      this.mediator.notify('logEvent', data); // Повідомляємо Mediator про
логування події
    }
  }
}

module.exports = Logger;
```

Файл Notifier.js

```
const Observer = require('./Observer');

class Notifier extends Observer {
  constructor(mediator) {
    super();
    this.mediator = mediator; // Посилання на Mediator
  }

  update(data) {
    console.log(`[Notifier] Сповіщення: ${data.event}`);

    // Повідомлення Mediator про сповіщення
    if (this.mediator) {
      this.mediator.notify('notifyEvent', data); // Повідомляємо Mediator про
нове сповіщення
    }

    // Додаткова логіка, наприклад, надсилання email чи push-повідомлення
  }
}

module.exports = Notifier;
```

Посилання на Git Hub на гілку main1
[https://github.com/DimytroKashiroff/TRPZ_Kashirov_All-laboratory-work.git]

Висновок:

У процесі виконання лабораторної роботи було реалізовано та використано патерн проектування **Mediator** для організації взаємодії між компонентами в програмному середовищі. Використання цього патерну дозволяє зменшити залежність між компонентами, що важливо для підтримки масштабованості та гнучкості системи.

Патерн **Mediator** забезпечує централізоване управління обміном даними між різними частинами програми. Це дозволяє уникнути прямої взаємодії компонентів, знижуючи складність коду та полегшуючи його модифікацію. Патерн допомагає організувати чітку структуру взаємодії, де кожен компонент може реагувати на зміни, отримуючи повідомлення від посередника без необхідності знати подробиці про інші компоненти системи.

Завдяки такому підходу, програма стає більш гнучкою та зручною для розширення, оскільки додавання нових компонентів або зміна логіки взаємодії не вимагають істотних змін у вже існуючому коді.

У підсумку, реалізація патерну **Mediator** в проекті продемонструвала ефективність цього шаблону у створенні більш чистого та підтримуваного коду. Це дозволяє краще організовувати комунікацію між компонентами та покращує структуру програмного забезпечення.