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 №6

Технології розроблення програмного забезпечення

Тема роботи: « ШАБЛОНИ «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator» *»*

Виконав студент групи ІА-23:

Каширов Д. О.

Перевірив:

Мягкий Михайло Юрійович

Київ 2024

Мета роботи:

Метою цієї лабораторної роботи є вивчення та практичне застосування шаблону проектування **Observer** для організації системи сповіщень у програмному забезпеченні. Робота включає інтеграцію шаблону в існуючий проект **LiteBrowse** для забезпечення логування подій і реалізації сповіщень без внесення змін до основного коду.

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.

• Ознайомитися з короткими теоретичними відомостями.

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

• Застосування одного з даних шаблонів при реалізації програми

1. **Ознайомлення з теоретичними відомостями:**
    - Спочатку було вивчено теоретичні основи шаблону проектування **Observer**, який дозволяє організувати систему сповіщень або логування, що дає можливість спостерігати за змінами в одному об'єкті і автоматично сповіщати інші об'єкти про ці зміни.
    - Шаблон **Observer** складається з двох основних компонентів: **Subject** (суб'єкт) — об'єкт, який сповіщає про зміни, і **Observer** (спостерігач) — об'єкт, який отримує сповіщення та реагує на зміни.
2. **Інтеграція шаблону Observer в проект:**
    - Створено нову папку **observers** в проекті для зберігання відповідних файлів.
    - Реалізовано чотири основні класи:
        - **Subject.js** — клас, який зберігає список спостерігачів і повідомляє їх про події.
        - **Observer.js** — базовий інтерфейс для спостерігачів.
        - **Logger.js** — клас спостерігача, який виконує логування подій.
        - **Notifier.js** — клас спостерігача, який сповіщає про події.
3. **Модифікація middleware для Proxy:**
    - У файл **proxyMiddleware.js** було додано логіку для використання шаблону Observer.
    - Після обробки запиту до проксі, за допомогою методу **notify()** відправлялися сповіщення про виконання запиту або виникнення помилки.
4. **Інтеграція шаблону Observer в маршрути:**
    - У файлі **browserRoutes.js** було додано виклик **proxyMiddleware** для логування та сповіщення про обробку запитів.
    - Для кожного запиту до маршруту /browser/load, після обробки запиту, відправлялися повідомлення про його успішне виконання або помилку.
5. **Запуск та тестування проекту:**

- Після реалізації шаблону Observer та інтеграції його в проект, було проведено тестування.
- В результаті тестування, у консолі відображалися сповіщення про кожен запит і його результат: логування запитів, помилок, а також успішне завантаження сторінки.

6. **Аналіз результатів:**
   - Під час тестування було підтверджено, що система сповіщень працює коректно: кожен запит до серверу викликає відповідні сповіщення, які логуються та надсилаються за допомогою спостерігачів **Logger** і **Notifier**.

Короткі теоретичні відомості:

Шаблон проектування Observer є одним з найпоширеніших структурних патернів, який використовується для реалізації системи спостереження за змінами в одному об'єкті та автоматичної реакції на ці зміни в інших об'єктах. Цей шаблон дозволяє відокремити логіку спостереження за об'єктами від самих об'єктів, що спрощує їхнє використання і модифікацію.

Основні компоненти шаблону **Observer**:

1. **Subject (Суб'єкт)** — об'єкт, за яким спостерігають інші об'єкти. Він зберігає список спостерігачів і повідомляє їх про зміни.
2. **Observer (Спостерігач)** — об'єкти, які отримують повідомлення про зміни від суб'єкта. Спостерігачі виконують певну дію, реагуючи на події, які відбуваються в суб'єкті.

Основні принципи шаблону Observer:

- **Відсутність жорсткої залежності між суб'єктом та спостерігачами**. Спостерігачі не повинні знати, як працює суб'єкт, лише що він може змінити своє значення і що потрібно оновити спостерігачів.
- **Підписка і відписка**: Спостерігачі можуть підписуватися або відписуватися від отримання повідомлень від суб'єкта, що дозволяє динамічно змінювати систему сповіщень.
- **Реактивність**: Спостерігачі отримують повідомлення про зміни і автоматично виконують дії, не потребуючи прямого виклику їхньої логіки.

Шаблон **Observer** корисний, коли необхідно підтримувати синхронізацію між кількома об'єктами без прямої залежності між ними, а також коли існує потреба в централізованому управлінні повідомленнями та сповіщеннями. Це дозволяє зберігати гнучкість і масштабованість програмної системи.

У даній лабораторній роботі шаблон Observer було використано для реалізації системи логування та сповіщень про оброблені запити до сервера, що дозволяє легко додавати нові типи обробки подій, не змінюючи основний код.

Діаграма реалізації паттерну Observer:

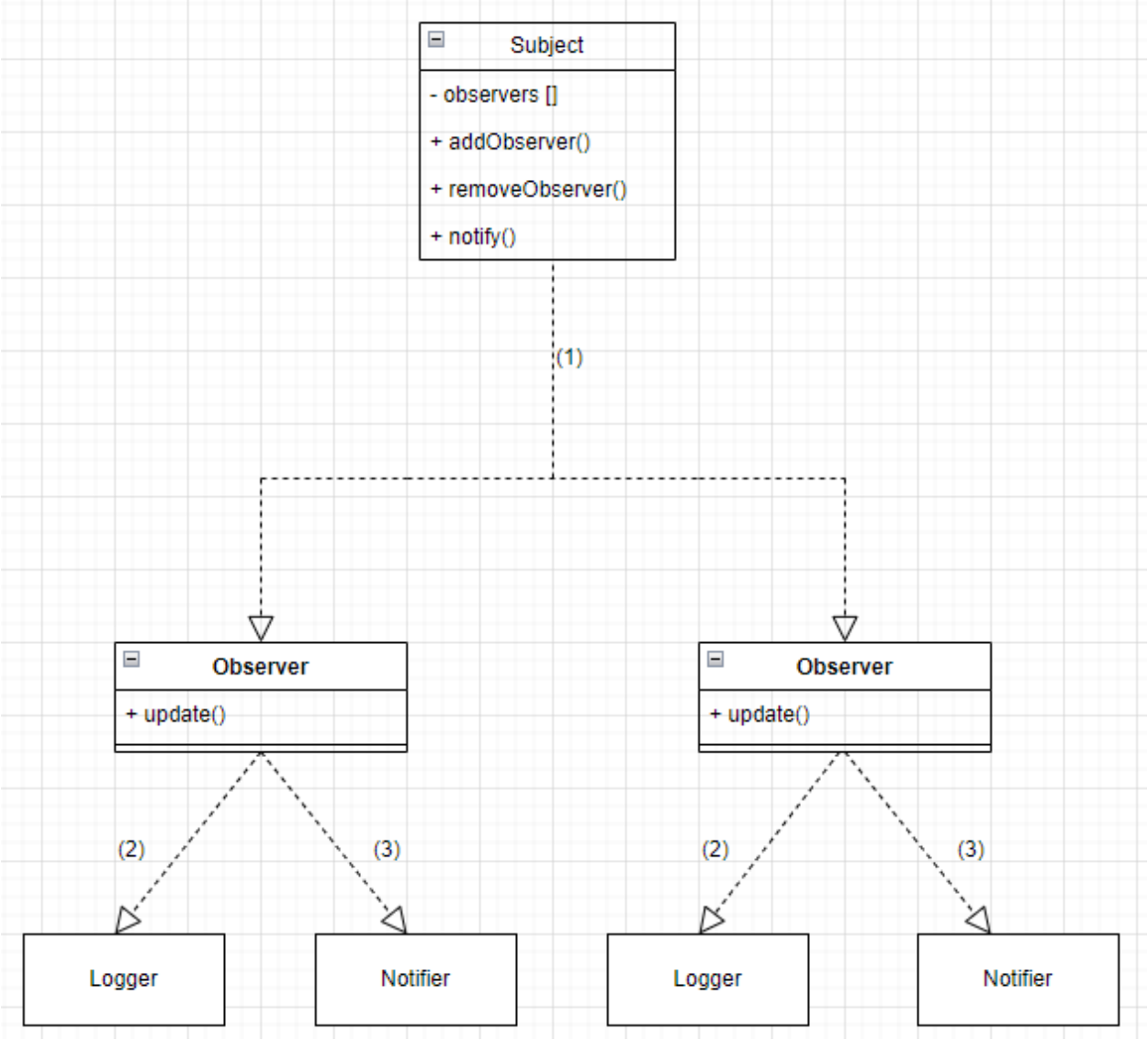


Рис 1- реалізації паттерну Observer

Опис компонентів діаграми:

1. Subject (Суб'єкт)

- **Опис**: Це клас, який є спостережуваним об'єктом. Він містить логіку для управління списком спостерігачів і сповіщення їх про зміни.
- **Компоненти**:
  - **observers**: Це масив або колекція, що зберігає всі об'єкти-спостерігачі, які підписані на отримання сповіщень.
  - **addObserver()**: Метод для додавання спостерігача до списку спостерігачів. Якщо об'єкт хоче отримувати повідомлення про зміни в суб'єкті, він викликає цей метод.
  - **removeObserver()**: Метод для видалення спостерігача зі списку спостерігачів. Це дозволяє припинити отримання оновлень.
  - **notify()**: Метод, який викликається, коли суб'єкт змінює свій стан. Цей метод проходиться по всіх спостерігачах і викликає їх метод update(), сповіщаючи їх про зміни.

2. Observer (Спостерігач)

- **Опис**: Це інтерфейс або абстрактний клас, який реалізує метод update(). Спостерігачі підписуються на зміни суб'єкта і реагують на них, коли отримують сповіщення.
- **Компоненти**:
  - **update()**: Метод, який викликається суб'єктом, коли відбувається зміна стану. Кожен спостерігач реалізує цей метод для виконання власних дій при отриманні сповіщення.

3. Logger (Логер)

- **Опис**: Один з конкретних спостерігачів, який реалізує метод update(), щоб логувати події або зміни, що відбулися у суб'єкті.
- **Дія**: Коли суб'єкт викликає метод notify(), логер може, наприклад, записати інформацію про подію в файл або вивести повідомлення в консоль. Це корисно для ведення журналу подій, моніторингу чи налагодження.

4. Notifier (Сповіщувач)

- **Опис**: Інший конкретний спостерігач, який реалізує метод update() для виконання дій, пов'язаних з надсиланням сповіщень, наприклад, відправлення електронної пошти або повідомлення користувачеві.
- **Дія**: Коли спостерігач отримує сповіщення від суб'єкта, він може, наприклад, надіслати повідомлення на вказану адресу або через інший канал зв'язку. Це корисно для автоматичних сповіщень користувачів або системи.

Зв'язки між компонентами:

- **Subject (Суб'єкт) → Observer (Спостерігач)**:
  Суб'єкт зберігає список спостерігачів і повідомляє їх про зміни через метод notify(). Кожен спостерігач може бути різним, в залежності від того, яку поведінку вони повинні реалізувати при отриманні сповіщення. У нашому випадку, спостерігачами можуть бути **Logger** або **Notifier**.
- **Observer (Спостерігач) → update()**:
  Кожен спостерігач реалізує метод update(), в якому визначено, як він повинен реагувати на повідомлення від суб'єкта. Це забезпечує гнучкість, оскільки кожен спостерігач може мати різну реалізацію цього методу (наприклад, один може лише логувати події, а інший — відправляти сповіщення).
- **Subject (Суб'єкт) → notify() → Observer (Спостерігач)**:
  Коли стан суб'єкта змінюється, він викликає метод notify(), сповіщаючи усіх своїх спостерігачів про зміни. Спостерігачі обробляють ці сповіщення по-різному в залежності від того, що вони повинні робити, але всі вони викликають свій метод update().

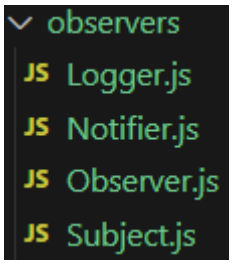
Загальний процес роботи:

1. **Subject** змінює свій стан.
2. **Subject** викликає метод notify(), щоб повідомити усіх спостерігачів.
3. Спостерігачі (як **Logger** або **Notifier**) отримують це повідомлення через метод update().
4. Кожен спостерігач реалізує свою власну логіку на основі отриманого повідомлення (наприклад, логування події або відправлення сповіщення).

Ця архітектура дозволяє легко масштабувати систему, додаючи нові типи спостерігачів без необхідності змінювати код суб'єкта, що робить систему більш гнучкою і менш залежною.

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

Створення файлів для паттерну:



a) Файл Observer.js

```
class Observer {
    update(data) {
      throw new Error("Метод update повинен бути реалізований");
    }
  }

  module.exports = Observer;
```

b) Файл Subject.js

```
class Subject {
    constructor() {
      this.observers = [];
    }

    addObserver(observer) {
      this.observers.push(observer);
    }

    removeObserver(observer) {
      this.observers = this.observers.filter(obs => obs !== observer);
    }

    notify(data) {
      this.observers.forEach(observer => observer.update(data));
    }
  }

  module.exports = Subject;
```

c) Файл Logger.js

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

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

module.exports = Logger;
```

d) Файл Notifier.js

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

class Notifier extends Observer {
  update(data) {
    console.log(`[Notifier] Сповіщення: ${data.event}`);
    // Наприклад, логіка надсилання повідомлення.
  }
}

module.exports = Notifier;
```

Відповідь від сервера:

```
[Logger] Подія: Новий запит, Деталі: {"method":"GET","url":"/vue/scripts/compare-chunk-d6fa90e81733824549021.js"}
[Notifier] Сповіщення: Новий запит
[Logger] Подія: Новий запит, Деталі: {"method":"GET","url":"/vue/scripts/congratulations-chunk-342e52051733824549021.js"}
[Notifier] Сповіщення: Новий запит
```

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

Висновок

У процесі виконання роботи було реалізовано паттерн проектування **Observer** для спостереження та реагування на зміни стану об'єкта (Суб'єкта). Цей паттерн дозволяє ефективно організувати комунікацію між об'єктами, де один об'єкт (Суб'єкт) може сповіщати кілька інших об'єктів (Спостерігачів) про свої зміни, не маючи з ними жорсткої залежності.

У рамках реалізації було створено три основні компоненти:

1. **Subject** (Суб'єкт), який змінює свій стан та сповіщає про це зареєстрованих спостерігачів.

2. **Observer** (Спостерігач), який реагує на зміни у Суб'єкті, викликаючи певні дії у відповідь на отримані оновлення.
3. Конкретні реалізації спостерігачів, такі як **Logger** і **Notifier**, які виконують конкретні дії: записують зміни в журнал чи надсилають сповіщення.

Реалізація цього паттерну продемонструвала високий рівень гнучкості та масштабованості системи, адже для додавання нових типів спостерігачів не потрібно змінювати логіку суб'єкта, що дозволяє мінімізувати зв'язки між компонентами та підвищує підтримуваність коду.

Використання паттерну **Observer** є особливо корисним у випадках, коли об'єкти повинні повідомляти інші об'єкти про зміни свого стану без необхідності знати деталі їхньої реалізації. Це дозволяє ефективно масштабувати систему, додаючи нові функціональні можливості без необхідності значних змін в основному коді.