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 №8

Технології розроблення програмного забезпечення

Тема роботи: « ШАБЛОНИ «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR»*»*

Виконав студент групи ІА-23:

Каширов Д. О.

Перевірив:

Мягкий Михайло Юрійович

Київ 2024

Мета роботи: Метою цієї лабораторної роботи є застосування патернів проектування в розробці програмного забезпечення для покращення гнучкості та підтримки розширюваності системи. У рамках роботи були реалізовані патерни **Observer**, **Mediator** та **Interpreter**, що дозволяють ефективно управляти взаємодією між компонентами системи, обробляти події та інтерпретувати команди. Таке впровадження дозволяє створити більш масштабовану та підтримувану архітектуру для веб-додатку, що забезпечує зручність для подальших модифікацій і розширень.

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 функціоналу робочої програми у вигляді класів і їх взаємодій для досягнення конкретних функціональних можливостей.

3. Застосування одного з розглянутих шаблонів при реалізації програми.

1. **Підготовка середовища:**
    - Спочатку було налаштовано сервер за допомогою фреймворку **Express**, який забезпечує обробку запитів та відповідів у веб-додатку.
    - Було створено структуру папок і файлів для організації проекту, включаючи каталоги для моделей, контролерів, маршрутизаторів і спостерігачів.
2. **Імплементація патерна Observer:**
    - Для початку було розроблено клас **Subject**, який відповідає за збереження списку спостерігачів та їх оновлення, а також створено інтерфейс для спостерігачів.
    - Було створено два спостерігачі: **Logger** та **Notifier**, які реагують на події, що відбуваються в системі.
    - Кожен запит до серверу та інші події (наприклад, запуск сервера, помилки) сповіщають спостерігачів через метод **notify()**.
3. **Імплементація патерна Mediator:**
    - Було додано патерн **Mediator**, що обробляє комунікацію між компонентами системи, замінюючи прямі зв'язки між об'єктами.
    - **Mediator** отримує сповіщення від **Subject** і передає їх відповідним спостерігачам. Це дозволяє централізовано керувати всіма подіями в системі.
4. **Імплементація патерна Interpreter:**
    - Додано патерн **Interpreter**, який дозволяє інтерпретувати команди, отримані з запитів. Для цього було створено клас **CommandExpression**, який відповідає за виконання команд у певному контексті.
    - Інтерпретатор отримує та виконує команди типу HTTP-запитів, наприклад, DELETE /article, і повертає результат виконання.
5. **Інтеграція патернів у веб-додаток:**
    - У додатку реалізовано маршрути для обробки запитів від клієнта. Для кожного запиту сповіщаються спостерігачі, а також обробляється інтерпретація команд через **Interpreter**.

- Запити можуть бути оброблені з повідомленнями, логами та сповіщеннями для користувача.

6. **Тестування та перевірка:**
    - Після реалізації патернів було перевірено, як система реагує на запити, чи коректно працюють спостерігачі та **Mediator**.
    - Проведено тестування серверу, щоб перевірити, чи правильно обробляються події, зокрема запуск сервера, помилки та запити до різних маршрутів.
7. **Запуск сервера:**
    - Після успішного впровадження всіх патернів та тестування коду було запущено сервер. На консоль виводяться сповіщення про запуск сервера, а також події, що відбуваються при обробці запитів.

Робота продемонструвала ефективне використання патернів проектування для управління складними взаємодіями між компонентами системи та забезпечення зручної архітектури для подальших розширень.

Короткі теоретичні відомості:

1. Шаблон «Composite» (Компонування)

Шаблон проектування **Composite** дозволяє об'єднати об'єкти в структури типу «дерево» для того, щоб клієнти могли працювати з ними як з єдиним об'єктом. Він використовується для представлення частин об'єкта, що можуть бути об'єднані в ієрархічні структури, де кожен компонент (як «лист» дерева, так і «композит») має однаковий інтерфейс.

- **Переваги**: Полегшує роботу з деревоподібними структурами, дозволяючи користуватися однаковими методами для окремих об'єктів і для їх груп.
- **Приклад використання**: Ієрархії елементів інтерфейсу користувача, файлові системи, структури даних.

2. Шаблон «Flyweight» (Летючий вагон)

Шаблон проектування **Flyweight** дозволяє зберігати великі об'єкти, використовуючи мінімальну кількість пам'яті. Цей шаблон дозволяє об'єднувати спільні характеристики об'єктів та зберігати лише унікальні

атрибути. Він зазвичай використовується в ситуаціях, коли потрібно створити велику кількість схожих об'єктів, що займають багато пам'яті.

- **Переваги**: Зменшує використання пам'яті, оскільки спільні характеристики об'єктів зберігаються в єдиному місці, і тільки унікальні дані займають додаткову пам'ять.
- **Приклад використання**: Графічні об'єкти, ігрові світи, де потрібно зберігати багато об'єктів одного типу з мінімальними відмінностями.

3. Шаблон «Interpreter» (Інтерпретатор)

Шаблон проектування **Interpreter** використовується для опису граматики певної мови, а також для створення механізму для інтерпретації виразів цієї мови. Цей шаблон визначає граматику мови у вигляді об'єктної моделі і дозволяє інтерпретувати вирази з цієї мови. Використовується для обробки мов специфікацій або мови запитів.

- **Переваги**: Зручний для реалізації мов програмування, калькуляторів, лексичних і синтаксичних аналізаторів.
- **Приклад використання**: Мови запитів (наприклад, SQL), мовні процесори, калькулятори.

4. Шаблон «Visitor» (Візитор)

Шаблон проектування **Visitor** дозволяє додавати нові операції до об'єктів, не змінюючи самі об'єкти. Це досягається за рахунок створення спеціальних класів-візиторів, які можуть обходити структуру об'єктів та виконувати операції, не змінюючи їх структуру. Шаблон дозволяє додавати нові операції до існуючих об'єктів без зміни їх класів.

- **Переваги**: Спрощує додавання нових операцій до класів без зміни їх коду. Це корисно для операцій, які повинні виконуватися над різними об'єктами структури.
- **Приклад використання**: Операції над складними структурами даних (дерева, графи), наприклад, у обробці елементів HTML-документу або аналізі математичних виразів.

Ці шаблони проектування забезпечують гнучкість, масштабованість і простоту в управлінні складними системами, що потребують обробки даних або взаємодії об'єктів у різних контекстах.

Діаграма реалізації паттерну INTERPRETER:

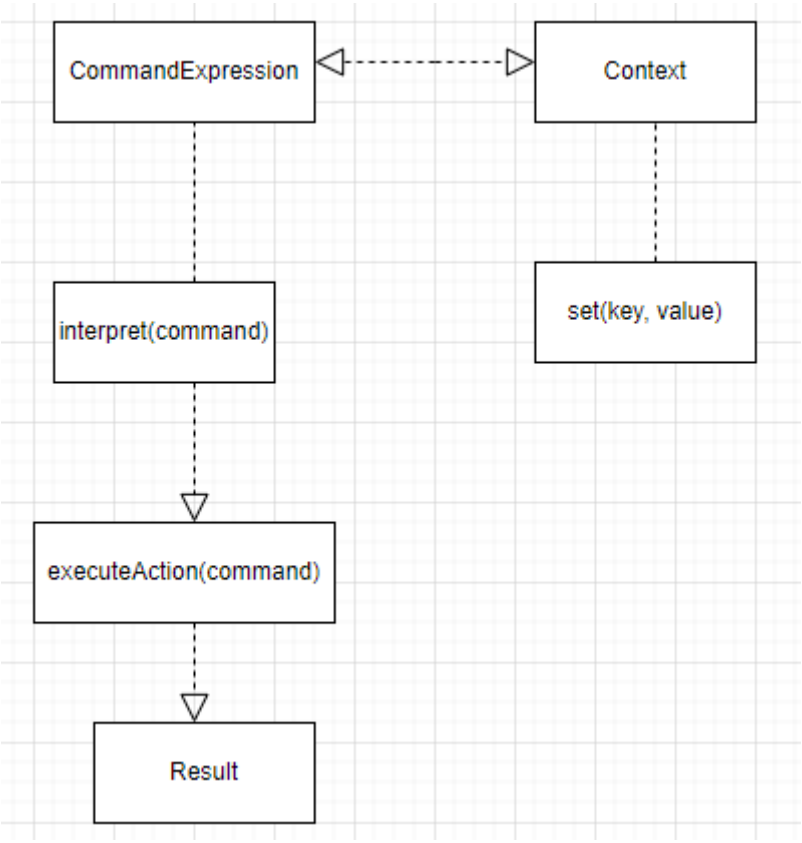


Рис 1- реалізації паттерну INTERPRETER

Опис компонентів діаграми паттерну INTERPRETER:

1. **CommandExpression**:
    - **Опис**: Це абстрактний клас або інтерфейс, який визначає методи для інтерпретації команд. Ваша реалізація використовує цей клас для створення конкретних виразів, які можуть бути інтерпретовані.
    - **Функція**: Інтерпретує введену команду на основі заданого контексту. У вашій реалізації це командна стрічка або HTTP запит, наприклад, DELETE /article.
    - **Методи**: Основний метод — interpret(), який виконує інтерпретацію і повертає результат.
2. **Context**:
    - **Опис**: Клас, що зберігає поточний стан та всі необхідні дані для інтерпретації команд.

- **Функція**: Контекст надає значення для змінних та параметрів, необхідних для виконання команд. Це дозволяє CommandExpression працювати з різними умовами або параметрами без необхідності змінювати сам вираз.
- **Методи**:
  - set(key, value) — зберігає дані в контексті.
  - get(key) — отримує значення, яке зберігається в контексті.

3. **Interpreter**:
   - **Опис**: Інтерпретатор, який використовує CommandExpression та Context для виконання команд.
   - **Функція**: Цей компонент бере команду у вигляді виразу, інтерпретує її на основі поточного контексту і виконує відповідну дію. Інтерпретатор також відповідає за визначення, чи потрібно виконувати певну операцію (наприклад, на основі HTTP методів).
   - **Методи**: Викликає методи interpret() для виконання команд.
4. **executeAction**:
   - **Опис**: Функція або метод, який виконує конкретну дію на основі інтерпретованої команди.
   - **Функція**: Наприклад, якщо команда містить HTTP запит, цей метод буде здійснювати запит за допомогою таких бібліотек, як axios, і повертати результат виконання запиту.
5. **Result**:
   - **Опис**: Об'єкт або структура даних, яка містить результат виконання команди.
   - **Функція**: Після виконання команди інтерпретатор повертає результат, який може бути використаний для подальших дій. Це може бути результат HTTP запиту або інший тип виходу, в залежності від контексту.

Ці компоненти працюють разом для створення потужної та гнучкої архітектури, що дозволяє легко додавати нові команди та змінювати поведінку системи без значних змін у коді.

Вхідні коди з проекту LiteBrowse з паттерном INTERPRETER:

Створення файле для паттерну:

Файл Interpreter.js

```
// observers/Interpreter.js

// Абстрактний клас для виразів
class Expression {
    interpret(context) {
      throw new Error('Метод interpret не реалізований');
    }
  }

  // Простий вираз для команд
  class CommandExpression extends Expression {
    constructor(command) {
      super();
      this.command = command;
    }

    interpret(context) {
      // Логіка для виконання команд
      console.log(`Виконання команди: ${this.command}`);
      return this.command;
    }
  }

  // Складений вираз для комбінування команд
  class AndExpression extends Expression {
    constructor(expression1, expression2) {
      super();
      this.expression1 = expression1;
      this.expression2 = expression2;
    }

    interpret(context) {
      const result1 = this.expression1.interpret(context);
      const result2 = this.expression2.interpret(context);
      return result1 && result2;
    }
  }

  // Контекст для зберігання інформації для інтерпретації
  class Context {
    constructor() {
```

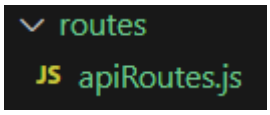
```
    this.data = {};
  }

  set(key, value) {
    this.data[key] = value;
  }

  get(key) {
    return this.data[key];
  }
}

module.exports = { CommandExpression, AndExpression, Context };
```

Файл apiRoutes.js



Для API маршрутів, тестовий файл

```
const express = require('express');
const router = express.Router();

// Приклад API маршруту
router.get('/example', (req, res) => {
  res.json({ message: 'Це приклад API маршруту' });
});

module.exports = router;
```

Діаграма реалізації паттерну Visitor:

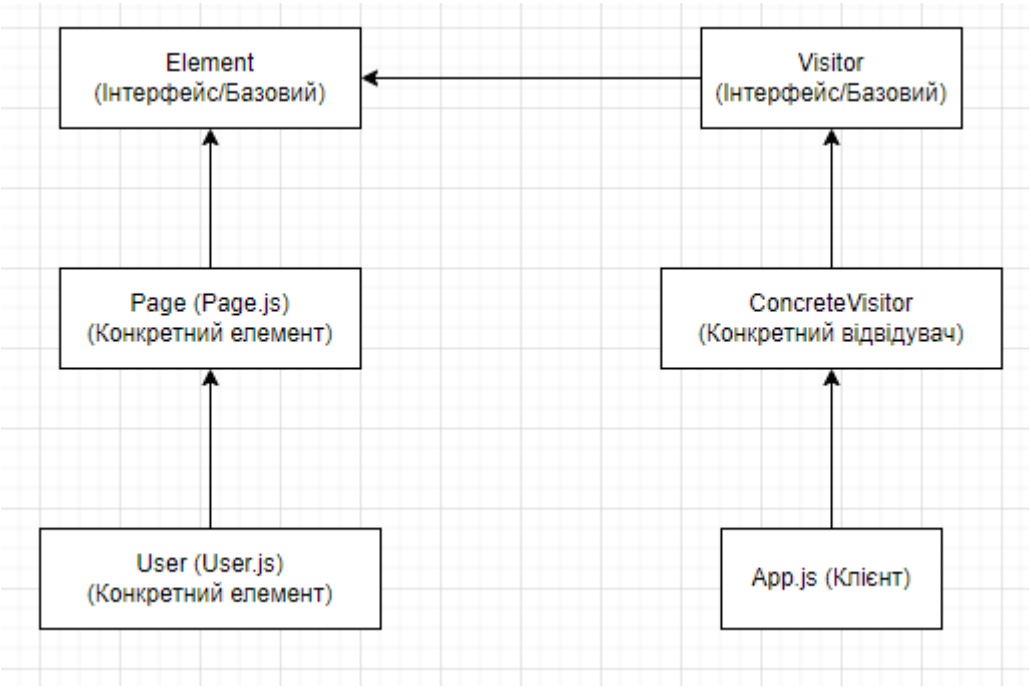


Рис 1- реалізації паттерну Visitor

Опис компонентів діаграми паттерну Visitor:

1. **Element (Інтерфейс/Базовий клас)**
    - Це базовий інтерфейс або клас для елементів, які мають бути оброблені відвідувачем.
    - Включає метод accept(visitor), який отримує екземпляр відвідувача.
2. **Page (Конкретний елемент)**
    - Клас, який представляє веб-сторінку в проєкті.
    - Реалізує метод accept(visitor), який викликає метод visitPage у переданого відвідувача.
3. **User (Конкретний елемент)**
    - Клас, що представляє користувача в проєкті.
    - Реалізує метод accept(visitor), який викликає метод visitUser у переданого відвідувача.
4. **Visitor (Інтерфейс/Базовий клас)**

- Інтерфейс або базовий клас для всіх відвідувачів.
- Декларує методи для обробки кожного типу елементів, наприклад, visitPage(page) і visitUser(user).

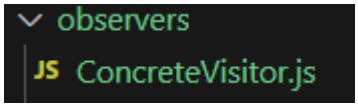
5. **ConcreteVisitor (Конкретний відвідувач)**
   - Клас, що реалізує логіку обробки для кожного типу елемента.
   - Має методи visitPage(page) і visitUser(user), де реалізується конкретна логіка обробки для Page і User.
6. **App.js (Клієнт)**
   - Компонент, який ініціює процес, створює екземпляри елементів (Page і User) та передає їм відвідувача (ConcreteVisitor).
   - Забезпечує інтеграцію між елементами та відвідувачем, дозволяючи відвідувачу обробляти дані елементів.

Взаємодія:

- Клієнт створює екземпляри елементів (Page і User) та передає їм відвідувача (ConcreteVisitor) через метод accept(visitor).
- Кожен елемент викликає відповідний метод у відвідувачі (visitPage або visitUser), надаючи себе як параметр для обробки.
- Відвідувач реалізує логіку обробки для кожного типу елементів у своїх методах.

Вхідні коди з проекту LiteBrowse з паттерном Visitor:

Створення файле для паттерну:



Файл Visitor.js

```
// src/observers/Visitor.js
class Visitor {
    visitPage(page) {}
    visitUser(user) {}
}
```

```
module.exports = Visitor;
```

Файл ConcreteVisitor.js

```
// src/observers/ConcreteVisitor.js
const Visitor = require('./Visitor');

class ConcreteVisitor extends Visitor {
    visitPage(page) {
        console.log(`Обробка сторінки: ${page.title}`);
    }

    visitUser(user) {
        console.log(`Обробка користувача: ${user.name}`);
    }
}

module.exports = ConcreteVisitor;
```

Діаграма реалізації паттерну p2p:

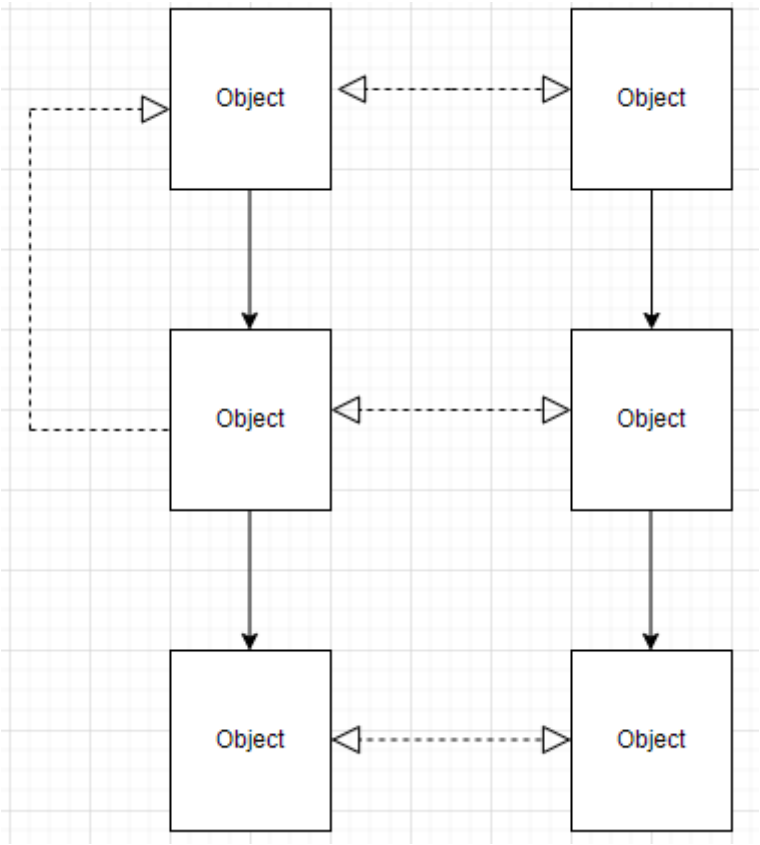


Рис 1- реалізації паттерну p2p

Опис компонентів діаграми паттерну p2p:

Peer (Пір):

- **Опис**: Це основний компонент у патерні P2P (peer-to-peer). Пір — це будь-який вузол або учасник мережі, який може діяти як сервер або клієнт одночасно. Всі піри мають однакові функції і можуть взаємодіяти безпосередньо один з одним.
- **Функціональність**:
    - Взаємодія з іншими пірами для обміну даними (наприклад, передача файлів, обмін повідомленнями).
    - Обробка запитів від інших пірів.
    - Може бути як джерелом, так і отримувачем даних.
    - Може підключатися до іншого піра або відключатися від нього.
    - Забезпечує безпечну передачу даних через мережу.

З'єднання між пірами:

- **Опис**: З'єднання між піром можуть бути різного типу: пряме з'єднання або за допомогою централізованих або дистрибутивних мереж.
- **Функціональність**:
    - Створення каналу комунікації між пірами.
    - Обмін інформацією, даними або запитами.
    - Забезпечує взаємодію між пірами без необхідності в центральному сервері.
    - Важливим аспектом є те, що з'єднання може бути тимчасовим і створюватися тільки для необхідної взаємодії.

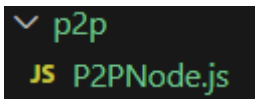
Ключові характеристики патерну P2P:

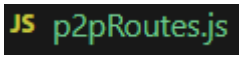
- **Розподіленість**: У мережах P2P кожен пір може бути частиною більшої дистрибутивної мережі, без потреби в централізованому сервері для комунікації.
- **Динамічність**: Пір може приєднуватися або відключатися від мережі в будь-який час. Інші піри можуть автоматично виявляти та адаптуватися до змін.

- **Незалежність**: Усі піри є рівноправними, і жоден не має більшої влади чи контролю над іншими.

Вхідні коди з проекту LiteBrowse з паттерном p2p:

Створення файле для паттерну:





Файл P2PNode.js

```
// src/p2p/P2PNode.js
const Peer = require('simple-peer');
const EventEmitter = require('events');

class P2PNode extends EventEmitter {
    constructor() {
        super();
        this.peers = [];
    }

    createPeer(isInitiator) {
        const peer = new Peer({ initiator: isInitiator });
        this._setupPeerEvents(peer);
        this.peers.push(peer);
        return peer;
    }

    _setupPeerEvents(peer) {
        peer.on('signal', (data) => {
            this.emit('signal', data); // Відправляємо сигнал
        });

        peer.on('connect', () => {
            console.log('P2P connection established');
            this.emit('connect');
        });

        peer.on('data', (data) => {
            console.log('Received data:', data.toString());
            this.emit('data', data.toString());
        });

        peer.on('error', (err) => {
            console.error('P2P error:', err);
            this.emit('error', err);
```

```
        });
    }

    connectToPeer(signal) {
        this.peers.forEach((peer) => peer.signal(signal));
    }

    sendMessage(message) {
        this.peers.forEach((peer) => peer.send(message));
    }
}

module.exports = P2PNode;
```

Файл P2PRoutes.js

```
// src/routes/p2pRoutes.js
const express = require('express');
const router = express.Router();

const signals = [];

router.post('/signal', (req, res) => {
    const { signal } = req.body;
    signals.push(signal);
    res.json({ success: true });
});

router.get('/signal', (req, res) => {
    res.json(signals);
});

module.exports = router;
```

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

Висновок

У ході виконання восьмої роботи було реалізовано три патерни проектування: **Interpreter**, **Visitor** та **P2P**, що дозволило глибше зрозуміти принципи розробки гнучких та масштабованих програмних рішень.

1. **Interpreter** дозволяє ефективно обробляти вирази та команди, що дозволяє зручно реалізувати лінгвістичні системи, мови програмування або навіть алгоритми для розпізнавання та виконання різних інструкцій у програмі. Патерн спрощує обробку складних структур даних через поділ на окремі інтерпретовані елементи, що полегшує розширення та підтримку таких систем.
2. **Visitor** забезпечує гнучке додавання нових операцій до об'єктів, не змінюючи їх внутрішню структуру. Цей патерн дозволяє централізувати обробку об'єктів різних типів в одному місці, що робить код більш масштабованим і зручним для підтримки. Вибір Visitor для цього проекту дозволив реалізувати процеси обробки сторінок та користувачів без необхідності змінювати ці класи.
3. **P2P (Peer-to-Peer)** дозволяє створювати дистрибутивні мережі без необхідності централізованих серверів, що особливо важливо для розподілених систем, де кожен учасник може взаємодіяти з іншими в реальному часі. Патерн дозволяє організувати ефективну комунікацію між учасниками, що робить систему більш надійною та стійкою до відмов.

Завдяки використанню цих трьох патернів вдалося побудувати модульну, гнучку та масштабовану архітектуру для роботи з різними типами даних та взаємодією між учасниками системи. Це дозволяє легко додавати нові можливості та адаптувати систему до змінних вимог. Кожен з патернів підвищує ефективність реалізації відповідних функцій, оптимізуючи процес розробки та полегшуючи підтримку програми в майбутньому.