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

Шаблони «Composite», «Flyweight», «Interpreter», «Visitor»

Виконав
студент групи ІА–22:
Щаблевський Д. Е.
Перевірив:
Мягкий Михайло Юрійович

Київ 2024

Зміст

Тема: шаблони «Composite», «**Flyweight**», «Interpreter», «Visitor.

Мета: ознайомитися з шаблонами проектування «Composite», «**Flyweight**», «Interpreter», «Visitor, та набути практичних навичок їх застосування. Реалізувати частину функціоналу програми за допомогою одного з розглянутих шаблонів для досягнення конкретних функціональних можливостей та забезпечення ефективної взаємодії між класами.

Хід роботи

..28 JSON Tool (ENG) (strategy, command, observer, template method, flyweight)

Display JSON schema with syntax highlight. Validate JSON schema and display errors. Create user friendly table\list box\other for read and update JSON schema properties metadata (description, example, data type, format, etc.). Auto save\restore when edit, maybe history. Can check JSON value by schema (Put schema and JSON = valid\invalid, display errors). Export schema as markdown table. JSON to "flat" view.

Короткі теоретичні відомості

Шаблони проєктування представляють собою стандартизовані підходи до вирішення типових задач, які часто зустрічаються при створенні інформаційних систем. Вони включають в себе опис проблеми, ефективне рішення та рекомендації щодо застосування в різних контекстах. Завдяки загальновизнаним назвам, шаблони легко розпізнаються та багаторазово використовуються.

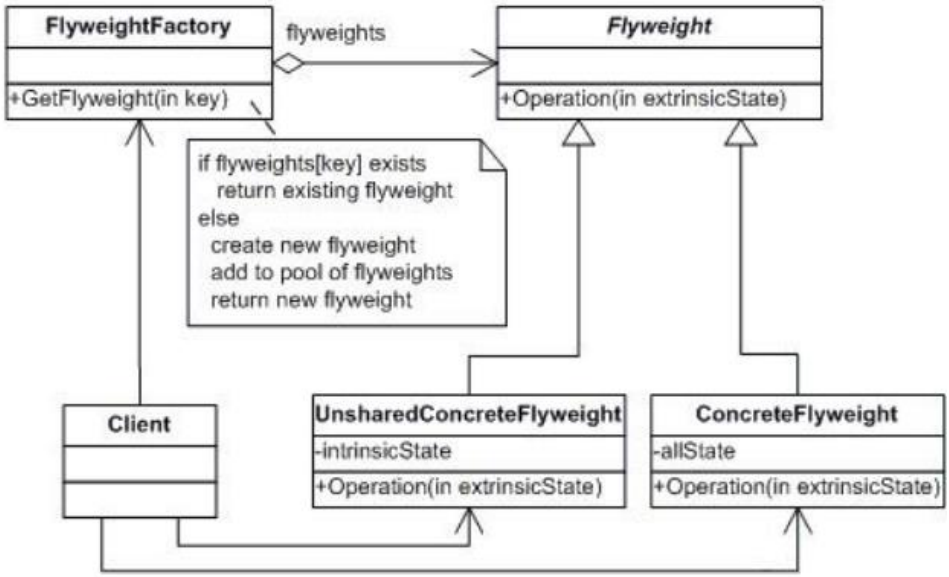
Ключовою особливістю роботи з шаблонами є ретельне моделювання предметної області, що дозволяє точно сформулювати проблему і підібрати найкращий шаблон. Використання таких рішень дає розробнику ряд переваг: вони допомагають зробити систему більш організованою, зрозумілою для вивчення та легкою у розширенні. Крім того, це підвищує гнучкість системи до змін і сприяє її простішій інтеграції з іншими рішеннями.

Таким чином, шаблони проєктування є перевіреними часом інструментами, які широко застосовуються в різних організаціях і проєктах для побудови ефективної архітектури в заданих умовах.

Навіщо використовувати шаблони ?

Шаблони проєктування сприяють зменшенню часу і зусиль на створення архітектури, надаючи системі важливі властивості, як-от гнучкість і адаптованість, а також спрощують її підтримку. Їх застосування полегшує комунікацію в команді, оскільки спільне розуміння шаблонів робить архітектуру зрозумілою для всіх учасників проєкту. Шаблони дозволяють уникнути "винаходу велосипеда", використовуючи перевірені практики, визнані спільнотою розробників, що економічно вигідно для бізнесу. Водночас їх використання повинно бути обґрунтованим, адже вони не є універсальним рішенням для всіх ситуацій.

Шаблон “Flyweight”



Призначення патерну Flyweight

Шаблон «Flyweight» (Легковаговик) дозволяє оптимізувати використання пам'яті та ресурсів шляхом створення централізованого пулу об'єктів, які можуть бути повторно використані, замість створення нових об'єктів для кожного запиту. Цей підхід корисний, коли потрібно обробляти велику кількість схожих об'єктів, що мають спільний стан. Легковаговики зберігають лише ті дані, які є унікальними для конкретного екземпляра, тоді як спільний стан зберігається в загальному пулі. Таким чином, Flyweight дозволяє зменшити витрати пам'яті, покращуючи ефективність системи при роботі з великими обсягами схожих даних.

Проблема

Шаблон «**Flyweight**» потрібен, якщо:

- Необхідно визначити загальну структуру алгоритму, залишаючи конкретні деталі реалізації для підкласів.
- Потрібно уникнути дублювання коду для алгоритмів зі схожою послідовністю дій.
- Важливо забезпечити контроль за порядком виконання операцій у спадкоємцях.

Рішення

Легковаговик оптимізує використання пам'яті шляхом створення централізованого пулу об'єктів, що можуть бути спільно використані без дублювання даних. Це ідеальний підхід для роботи з великими обсягами схожих об'єктів, зокрема коли об'єкти мають спільні характеристики, але існують різні екземпляри, що можуть змінюватися. Flyweight дозволяє зберігати спільний стан у загальному пулі, а індивідуальний стан — окремо для кожного екземпляра.

Застосування цього патерну дозволяє зменшити витрати на пам'ять, забезпечуючи ефективне управління великою кількістю об'єктів, зберігаючи при цьому узгодженість та зручність доступу до спільних властивостей. Наприклад, при обробці великої кількості даних, таких як налаштування або ресурси редактора, Flyweight допомагає централізовано керувати ресурсами, оптимізуючи використання пам'яті та покращуючи загальну ефективність програми.

Реалізувати не менше 3-х класів відповідно до обраної теми

Структура проекту з реалізованими класами зображена на рисунку 1.

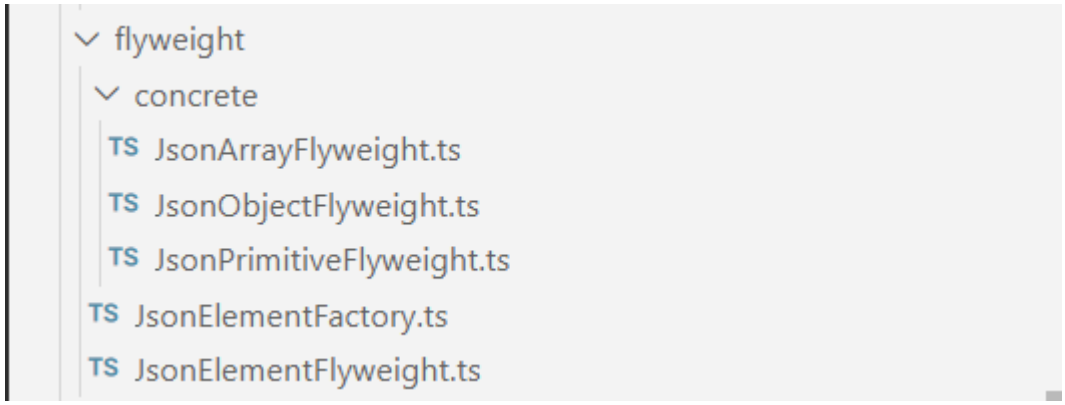


Рисунок 1 – структура класів

Опис класів

В ході виконання лабораторної роботи були реалізовані наступні класи:

- **JsonElementFlyweight -** Інтерфейс, що визначає базовий контракт для всіх типів JSON-елементів. Містить методи для отримання типу (**getType()**), значення (**getValue()**) та форматованого представлення елемента (format(indent: number)).
- **JsonElementFactory -** Фабрика, що забезпечує створення та управління екземплярами **JsonElementFlyweight**. Використовує кеш для збереження створених елементів, уникаючи дублювання. Має методи **getJsonElement(value)** для отримання елемента та **clearCache()** для очищення кешу.
- **JsonPrimitiveFlyweight -** Клас, що реалізує **JsonElementFlyweight** для примітивних типів JSON (рядків, чисел, булевих значень та null). Зберігає значення, визначає його тип через **getType()** та надає форматоване представлення через **format(indent: number)**.
- **JsonObjectFlyweight -** Клас, що реалізує **JsonElementFlyweight** для JSON-об'єктів. Зберігає об'єкт як структуру ключ-значення. Повертає "object" як тип через **getType(),** надає доступ до значення через **getValue()** та форматує об'єкт у вигляді структурованого JSON-рядка через **format(indent: number).**
- **JsonArrayFlyweight -** Клас, що реалізує **JsonElementFlyweight** для JSON-масивів. Зберігає масив як список елементів. Повертає "array" як тип через **getType(),** надає доступ до значення через **getValue()** та форматує масив у вигляді структурованого JSON-рядка через **format(indent: number).**

Реалізувати один з розглянутих шаблонів за обраною темою.

```
export class JsonElementFactory {
    private static instance: JsonElementFactory;
    private elementCache: Map<string, JsonElementFlyweight>;

    private constructor() {
        this.elementCache = new Map();
    }

    static getInstance(): JsonElementFactory {
        if (!JsonElementFactory.instance) {
            JsonElementFactory.instance = new JsonElementFactory();
        }
        return JsonElementFactory.instance;
    }

    getJsonElement(value: any): JsonElementFlyweight {
        const key = JSON.stringify(value);

        if (this.elementCache.has(key)) {
            return this.elementCache.get(key)!;
        }

        let element: JsonElementFlyweight;

        if (Array.isArray(value)) {
            element = new JsonArrayFlyweight(value);
        } else if (value !== null && typeof value === 'object') {
            element = new JsonObjectFlyweight(value);
        } else {
            element = new JsonPrimitiveFlyweight(value);
        }

        this.elementCache.set(key, element);
        return element;
    }

    clearCache(): void {
        this.elementCache.clear();
    }
}
```

Рисунок 2 – клас JsonElementFactory

```
export class JsonPrimitiveFlyweight implements JsonElementFlyweight {
    private readonly value: string | number | boolean | null;

    constructor(value: string | number | boolean | null) {
        this.value = value;
    }

    getType(): string {
        return typeof this.value;
    }

    getValue(): string | number | boolean | null {
        return this.value;
    }

    format(indent: number): string {
        return JSON.stringify(this.value);
    }
```

Рисунок 3 – клас JsonPrimitiveFlyweight

```
export class JsonObjectFlyweight implements JsonElementFlyweight {
    private readonly value: Record<string, any>;

    constructor(value: Record<string, any>) {
        this.value = value;
    }

    getType(): string {
        return 'object';
    }

    getValue(): Record<string, any> {
        return this.value;
    }

    format(indent: number): string {
        const spacing = ' '.repeat(indent);
        const entries = Object.entries(this.value)
            .map(([key, value]) => `${spacing}  "${key}": ${JSON.stringify(value)}`)
            .join(',\n');
        return `{\n${entries}\n${spacing}}`;
    }
}
```

Рисунок 4 – клас JsonObjectFlyweight

```
export class JsonArrayFlyweight implements JsonElementFlyweight {
    private readonly value: any[];

    constructor(value: any[]) {
        this.value = value;
    }

    getType(): string {
        return 'array';
    }

    getValue(): any[] {
        return this.value;
    }

    format(indent: number): string {
        const spacing = ' '.repeat(indent);
        const elements = this.value
            .map(item => `${spacing}  ${JSON.stringify(item)}`)
            .join(',\n');
        return `[\n${elements}\n${spacing}]`;
    }
}
```

Рисунок 5 – клас JsonArrayFlyweight

У класах **JsonElementFactory**, **JsonPrimitiveFlyweight**, **JsonObjectFlyweight**, та **JsonArrayFlyweight** використання конкретного екземпляра визначається динамічно під час виконання програми. Завдяки цьому, система працює з легковаговиками, які зберігають спільний стан у централізованому пулі об'єктів. Це забезпечує оптимізацію пам'яті та повторне використання однакових даних.

Проблема, яку допоміг вирішити шаблон «Flyweight»

Патерн «Flyweight» вирішує проблему надмірного споживання пам’яті при роботі з великою кількістю схожих об'єктів, забезпечуючи зберігання спільного стану у централізованому пулі. Це дозволяє зменшити витрати на створення нових екземплярів і гарантує узгоджений доступ до даних.

Переваги застосування патерну «Flyweight»

1. **Оптимізація пам'яті**: Спільне використання екземплярів для схожих об'єктів мінімізує використання пам'яті.

2. **Повторне використання**: Забезпечується легке повторне використання вже створених об'єктів, уникаючи дублювання.
3. **Гнучкість і адаптивність**: Легко додаються нові типи JSON-елементів (наприклад, нові об'єкти, масиви чи примітиви).

Цей підхід дозволяє ефективно обробляти складні JSON-структури, зберігаючи при цьому ресурси системи.

Діаграма класів для паттерну «Flyweight»

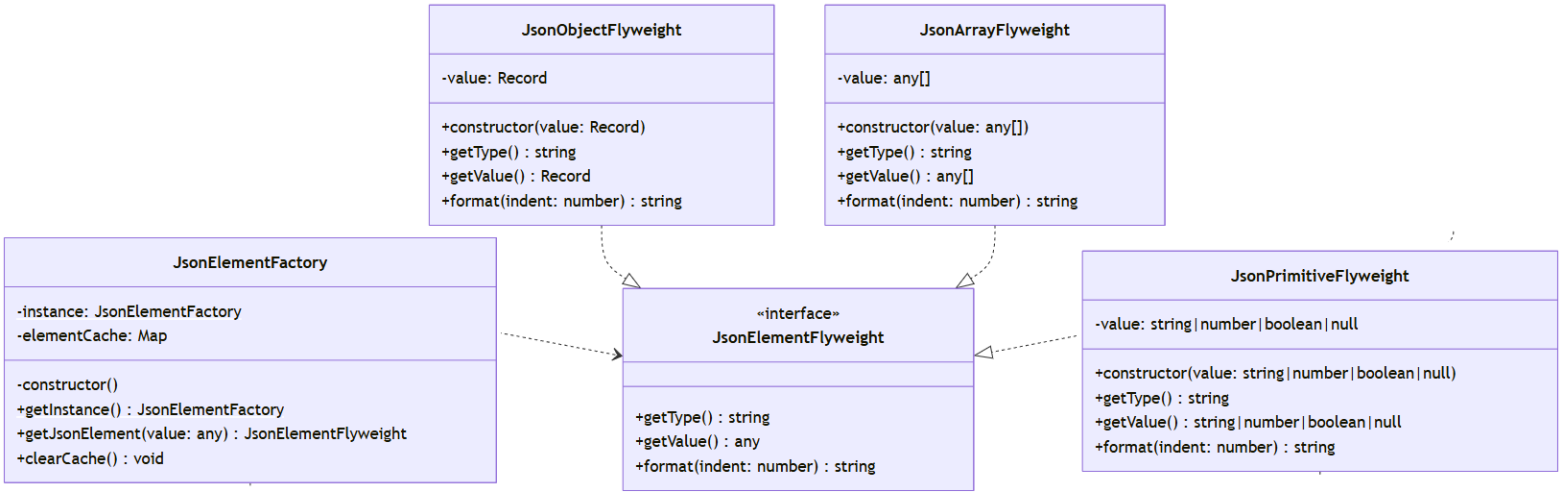


Рисунок 3 – діаграма класів

Посилання на репозиторій з кодом:

https://github.com/neodavis/json-tool

Висновок:

У ході виконання лабораторної роботи було реалізовано патерн проєктування Flyweight для оптимізації управління спільними ресурсами, зокрема для роботи з великими обсягами даних, що мають спільні характеристики. Патерн «Легковаговик» дозволив створити пул об'єктів для зберігання спільного стану, що забезпечило ефективне використання пам'яті та централізоване керування спільними ресурсами.