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

Шаблони «Mediator», «Facade», «Bridge», «Template method»

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

Київ 2024

Зміст

Тема: шаблони «mediator», «facade», «bridge», «**template method**».

Мета: ознайомитися з шаблонами проектування ««mediator», «facade», «bridge», «**template method**», 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.

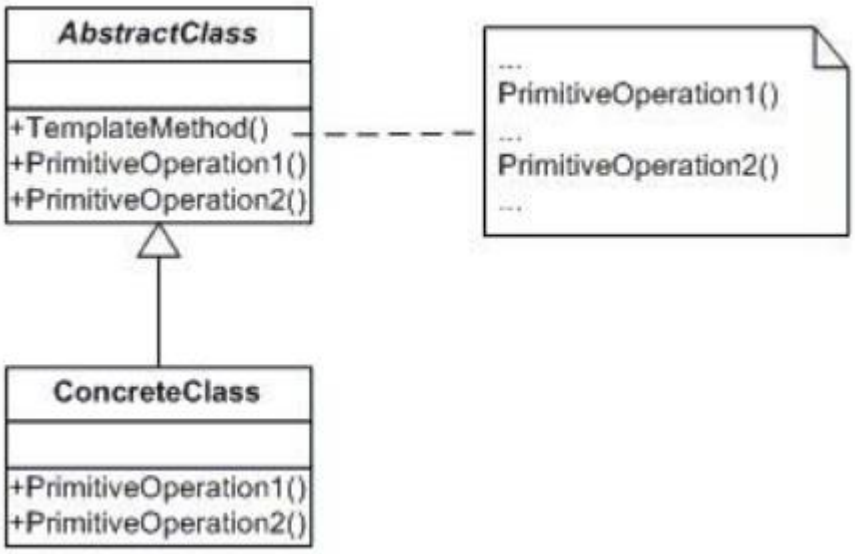
Ключовою особливістю роботи з шаблонами є ретельне моделювання предметної області, що дозволяє точно сформулювати проблему і підібрати найкращий шаблон. Використання таких рішень дає розробнику ряд переваг: вони допомагають зробити систему більш організованою, зрозумілою для вивчення та легкою у розширенні. 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 “Template method”



Призначення патерну Template method

Шаблон «Template Method» (Шаблонний метод) визначає структуру алгоритму у вигляді скелета, де конкретні кроки можуть бути реалізовані в підкласах. Цей шаблон дозволяє визначити основну послідовність дій, залишаючи деталі реалізації конкретних кроків для підкласів. Наприклад, алгоритм обробки даних може мати фіксовану структуру, але різні способи виконання окремих операцій, що визначаються у відповідних підкласах. Такий підхід зручний для забезпечення гнучкості й повторного використання коду,

оскільки зміни в конкретних реалізаціях не впливають на загальну послідовність алгоритму.

Проблема

Шаблон «**Template Method**» 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

Шаблонний метод дозволяє визначити «скелет» алгоритму в базовому класі, при цьому окремі етапи можуть бути реалізовані або перевизначені в підкласах. Це підходить для задач, де потрібно забезпечити узгодженість виконання основних операцій, але дозволити їхню варіативність. Застосування цього шаблону зручно, наприклад, для налаштування шаблонних процесів генерації звітів, перевірки даних, парсингу чи інших сценаріїв із фіксованою послідовністю, але змінними кроками.

Реалізувати не менше 3-х класів відповідно до обраної теми

Структура проекту з реалізованими класами зображена на рисунку 1.

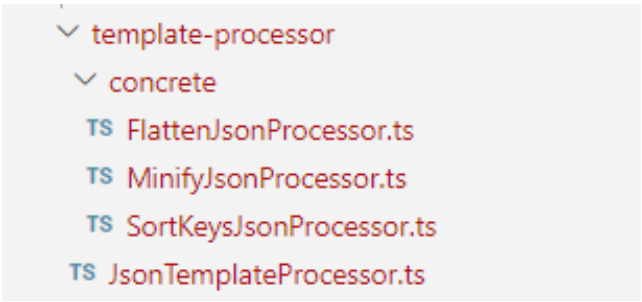


Рисунок 1 – структура класів

Опис класів

В ході виконання лабораторної роботи були реалізовані наступні класи:

- **JsonTemplateProcessor -** Абстрактний клас, який визначає шаблонний метод processJson(data: JsonObject). Забезпечує структуру обробки JSON: перевірка вхідних даних (validateInput), трансформація даних (transform), перевірка результату (validateOutput). validateInput та validateOutput мають базову реалізацію, яку можна перевизначити в підкласах. Метод transform є абстрактним, і його реалізація покладається на конкретні підкласи.

- **FlattenJsonProcessor -** Конкретна реалізація класу JsonTemplateProcessor. Реалізує метод transform, який виконує згортання вкладених об'єктів JSON у формат "ключ.ключ.значення". Використовує рекурсивну функцію flatten для обходу вкладених об'єктів.

- **MinifyJsonProcessor -** Конкретний клас, що успадковує JsonTemplateProcessor. Реалізує метод transform, що видаляє зайві структури або дані, які не потрібні, використовуючи методи JSON.stringify і JSON.parse. Перевизначає метод validateOutput, додаючи специфічну перевірку, щоб переконатися, що результуючий JSON не містить зайвих пробілів.

- **SortKeysJsonProcessor -** Конкретний клас, що успадковує JsonTemplateProcessor. Реалізує метод transform, який сортує ключі об'єкта JSON у алфавітному порядку. Використовує рекурсивну функцію sortObject для сортування вкладених об'єктів.

Реалізувати один з розглянутих шаблонів за обраною темою.

```
export abstract class JsonTemplateProcessor {
    public processJson(data: JsonObject): JsonObject {
        this.validateInput(data);
        const processed = this.transform(data);
        this.validateOutput(processed);
        return processed;
    }

    protected validateInput(data: JsonObject): void {
        if (!data || typeof data !== 'object') {
            throw new Error('Invalid input: JSON object required');
        }
    }

    protected validateOutput(data: JsonObject): void {
        if (!data || typeof data !== 'object') {
            throw new Error('Invalid output: JSON object required');
        }
    }

    protected abstract transform(data: JsonObject): JsonObject;
}
```

Рисунок 2 – клас JsonTemplateProcessor

```
export class SortKeysJsonProcessor extends JsonTemplateProcessor {
    protected transform(data: JsonObject): JsonObject {
        const sortObject = (obj: JsonObject): JsonObject => {
            const sorted: JsonObject = {};

            Object.keys(obj)
                .sort()
                .forEach(key => {
                    const value = obj[key];
                    sorted[key] = value && typeof value === 'object' && !Array.isArray(value)
                        ? sortObject(value as JsonObject)
                        : value;
                });

            return sorted;
        };

        return sortObject(data);
    }
}
```

Рисунок 3 – клас SortKeysJsonProcessor

```
export class MinifyJsonProcessor extends JsonTemplateProcessor {
    protected transform(data: JsonObject): JsonObject {
        const minified = JSON.parse(JSON.stringify(data));
        return minified;
    }

    protected override validateOutput(data: JsonObject): void {
        super.validateOutput(data);

        const stringified = JSON.stringify(data);
        if (stringified.includes('  ')) {
            throw new Error('Minification failed: contains extra whitespace');
        }
    }
}
```

Рисунок 4 – клас SortKeysJsonProcessor

```
export class MinifyJsonProcessor extends JsonTemplateProcessor {
    protected transform(data: JsonObject): JsonObject {
        const minified = JSON.parse(JSON.stringify(data));
        return minified;
    }

    protected override validateOutput(data: JsonObject): void {
        super.validateOutput(data);

        const stringified = JSON.stringify(data);
        if (stringified.includes('  ')) {
            throw new Error('Minification failed: contains extra whitespace');
        }
    }
}
```

Рисунок 5 – клас MinifyJsonProcessor

```
export class FlattenJsonProcessor extends JsonTemplateProcessor {
    protected transform(data: JsonObject): JsonObject {
        const result: JsonObject = {};

        const flatten = (obj: JsonObject, prefix = ''): void => {
            for (const [key, value] of Object.entries(obj)) {
                const newKey = prefix ? `${prefix}.${key}` : key;
                if (value && typeof value === 'object' && !Array.isArray(value)) {
                    flatten(value as JsonObject, newKey);
                } else {
                    result[newKey] = value;
                }
            }
        };

        flatten(data);
        return result;
    }
}
```

Рисунок 6 – клас FlattenJsonProcessor

Проблема, яку допоміг вирішити шаблон «Template Method»

Патерн «Template Method» вирішує проблему дублювання коду та забезпечує єдиний підхід до визначення загальної структури алгоритму, залишаючи можливість змінювати або доповнювати окремі його кроки. Це зручно, коли різні способи обробки даних мають однакову послідовність дій, але відрізняються в деталях.

Переваги застосування патерну «Template Method»

1. **Гнучкість і стандартизація**
   Основна структура алгоритму визначається в базовому класі, а підкласи реалізують варіативні кроки. Це дозволяє дотримуватись єдиного стандарту виконання операцій і зменшує ризик помилок.
2. **Спрощення підтримки коду**
   Завдяки виділенню загального алгоритму в базовий клас, код стає чистішим і легшим для підтримки. Спільна логіка знаходиться в одному місці, а деталі реалізації — в підкласах.
3. **Полегшення розширення функціональності**

Додавання нових варіантів реалізації не вимагає змін у базовому класі. Достатньо створити новий підклас із власною реалізацією специфічних кроків.

4. **Уникнення дублювання коду**
   Замість повторного написання однакової послідовності дій для кожного способу обробки даних, ці дії визначаються в одному місці, що спрощує модифікацію та зменшує ризик помилок.

Діаграма класів для паттерну «Template Method»

Рисунок 3 – діаграма класів

Посилання на репозиторій з кодом: https://github.com/neodavis/json-tool

Висновок:

У процесі виконання лабораторної роботи було успішно реалізовано патерн проектування «Template Method», що дозволяє визначити основну структуру алгоритму в базовому класі, залишаючи варіативні кроки для підкласів. Це дозволило значно спростити код, зменшити дублювання та підвищити гнучкість при розширенні функціональності. На прикладі обробки JSON-даних, таких як згортання, мінімізація та сортування ключів, було показано, як застосування цього патерну забезпечує організовану і зрозумілу архітектуру.

Переваги використання шаблону «Template Method» включають стандартизацію алгоритмів, полегшення їх підтримки, гнучкість при додаванні нових варіантів реалізації, а також усунення дублювання коду. Це дозволяє легко масштабувати та адаптувати систему під нові вимоги без значних змін у базовій структурі.

Загалом, шаблон «Template Method» є потужним інструментом для розробки програм, що потребують узгодженої послідовності дій з варіативними кроками, що робить його ефективним для багатьох сценаріїв в обробці даних та інших областях програмування.