

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І. І. МЕЧНИКОВА
ФАКУЛЬТЕТ МАТЕМАТИКИ, ФІЗИКИ ТА ІНФОРМАЦІЙНИЙ ТЕХНОЛОГІЙ
КАФЕДРА МАТЕМАТИЧНОГО ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНИХ СИСТЕМ

Лабораторна робота №3
з дисципліни: «Інженерія програмного забезпечення»
за темою: «Патерн проектування Decorator»

Виконав:
студент 3 курсу
Владислав Краковський

Перевірив:
викладач
Пенко В.Г.

Зміст

1 ЗАВДАННЯ РОБОТИ.....	3
2 ХІД РОБОТИ.....	4
Висновок.....	11

1 ЗАВДАННЯ РОБОТИ

Мета роботи: Ознайомитися з патерном проєктування “Декоратор” (Decorator), навчитись реалізовувати його на мові програмування C++, зрозуміти основні переваги, недоліки та сфери застосування цього патерну.

Умова роботи:

Приклад – обчислення у колекціях

Реалізувати кілька різновидів класів, здатних обчислювати суму та добуток елементів числової колекції. Ці різновиди можуть відрізнятися способом представлення колекції. Далі розширити архітектуру класів реалізації додаткової функціональності. Наприклад:

- округлення суми чи добутку елементів колекції;
- суми елементів колекції, або деякого стандартного значення, якщо обчислена сума занадто велика.

Рисунок 1.1 - Умова

Варіанти самостійних завдань

1. Додати до попереднього проєкту деяку додаткову функціональність декораторів.

Рисунок 1.2 - Варіант задачі

Система контролю версій Git та рефакторинг

Варіанти самостійних завдань

У наведеному вище викладі кожного патерну наводяться структурний та реальний приклад. Реалізувати структурний приклад та зафіксувати його в системі контролю версій. Далі здійснити серію фіксацій, яка трансформує структурний приклад у реальний. Деякі з проміжних фіксацій повинні полягати у застосуванні того чи іншого прийому рефакторингу.

Рисунок 1.3 - Завдання з GitHub

2 ХІД РОБОТИ

Опис патерну Decorator:

Патерн Decorator (Декоратор) — це структурний патерн проєктування, який дозволяє динамічно додавати об'єктам нову функціональність, не змінюючи їхньої початкової структури.

Decorator обгортає оригінальний об'єкт у спеціальний “декораторський” клас, який реалізує той самий інтерфейс і додає нову поведінку до основної.

Як реалізується патерн Decorator?

1. Створюється загальний інтерфейс (абстрактний клас) для базового об'єкта.
2. Створюються конкретні класи, які реалізують цей інтерфейс (основна функціональність).
3. Створюється абстрактний клас-декоратор, який також реалізує цей інтерфейс і містить посилання на об'єкт того ж типу.
4. Конкретні декоратори наслідують абстрактний декоратор і додають нову поведінку до викликів основного об'єкта.
5. Декоратори можуть вкладатися один в одного, що дозволяє створювати складні ланцюги розширень.

На C++ це означає створення базового класу, класу-декоратора з посиланням на базовий об'єкт, і класів-нащадків-декораторів, які додають поведінку.

Для чого використовується патерн Decorator?

- Коли треба динамічно додавати об'єктам нові можливості або поведінку.
- Коли неможливо (або недоцільно) використовувати спадкування для розширення функціональності.
- Для створення "гнучких" конфігурацій об'єктів (наприклад, у графічних редакторах, при додаванні ефектів, у потоках вводу-виводу).

Типові приклади:

- Оформлення GUI-компонентів (наприклад, додати бордер, тінь, прокрутку).
- Розширення можливостей класу (наприклад, файл ввід/вивід з архівацією чи шифруванням).

Переваги патерну Decorator

- Гнучкість — нова поведінка додається динамічно без зміни існуючих класів.
- Можна комбінувати декоратори в будь-якому порядку для створення нових комбінацій функціональності.
- Дотримання принципу відкритості/закритості (Open/Closed Principle).

Недоліки патерну Decorator

- Ускладнення структури програми — велика кількість малих класів-декораторів.
- Діагностика і налагодження може бути складнішим, бо поведінка розподілена по багатьох об'єктах.
- Створення “ланцюжків” декораторів — іноді важко зрозуміти загальний порядок виконання.

Запишемо розв'язок поставленої задачі з використання патерну Decorator:

```
#include <iostream>
#include <vector>
#include <memory>
#include <algorithm>
using namespace std;

// === Абстрактный компонент ===
class WiseCollection {
public:
    virtual double Sum() const = 0;
    virtual double Product() const = 0;
    virtual ~WiseCollection() = default;
};

// === Конкретный компонент ===
class SimpleWiseCollection : public WiseCollection {
private:
    vector<double> data;

public:
    SimpleWiseCollection(const vector<double>& values) :
    data(values) {}

    double Sum() const override {
        double sum = 0;
        for (double val : data) sum += val;
        return sum;
    }

    double Product() const override {
        double prod = 1;
```

```

        for (double val : data) prod *= val;
        return prod;
    }
};

// === Базовый декоратор ===
class WiseDecorator : public WiseCollection {
protected:
    shared_ptr<WiseCollection> collection;

public:
    WiseDecorator(shared_ptr<WiseCollection> coll) :
        collection(coll) {}

    double Sum() const override {
        return collection->Sum();
    }

    double Product() const override {
        return collection->Product();
    }
};

// === Декоратор ограничения результата ===
class BoundedCollection : public WiseDecorator {
private:
    double bound;

public:
    BoundedCollection(shared_ptr<WiseCollection> coll, double b)
        : WiseDecorator(coll), bound(b) {}

    double Sum() const override {

```

```

        return min(WiseDecorator::Sum(), bound);
    }

    double Product() const override {
        return min(WiseDecorator::Product(), bound);
    }
};

// === Декоратор логирования ===
class LoggingDecorator : public WiseDecorator {
public:
    LoggingDecorator(shared_ptr<WiseCollection> coll)
        : WiseDecorator(coll) {}

    double Sum() const override {
        double result = WiseDecorator::Sum();
        cout << "[Log] Sum = " << result << endl;
        return result;
    }

    double Product() const override {
        double result = WiseDecorator::Product();
        cout << "[Log] Product = " << result << endl;
        return result;
    }
};

// === Декоратор фильтрации по порогу ===
class ThresholdFilterDecorator : public WiseDecorator {
private:
    double threshold;

```



```

public:
    ThresholdFilterDecorator(shared_ptr<WiseCollection> coll,
double t)
        : WiseDecorator(coll), threshold(t) {}

    double Sum() const override {
        double rawSum = collection->Sum();
        cout << "[Filter] Applied threshold = " << threshold <<
endl;
        return (rawSum >= threshold) ? rawSum : 0.0;
    }

    double Product() const override {
        double rawProd = collection->Product();
        cout << "[Filter] Applied threshold = " << threshold <<
endl;
        return (rawProd >= threshold) ? rawProd : 1.0;
    }
};

// === main() ===
int main() {
    vector<double> values = { 1.5, 2.0, 3.0 }; // sum = 6.5,
product = 9.0
    auto simple = make_shared<SimpleWiseCollection>(values);

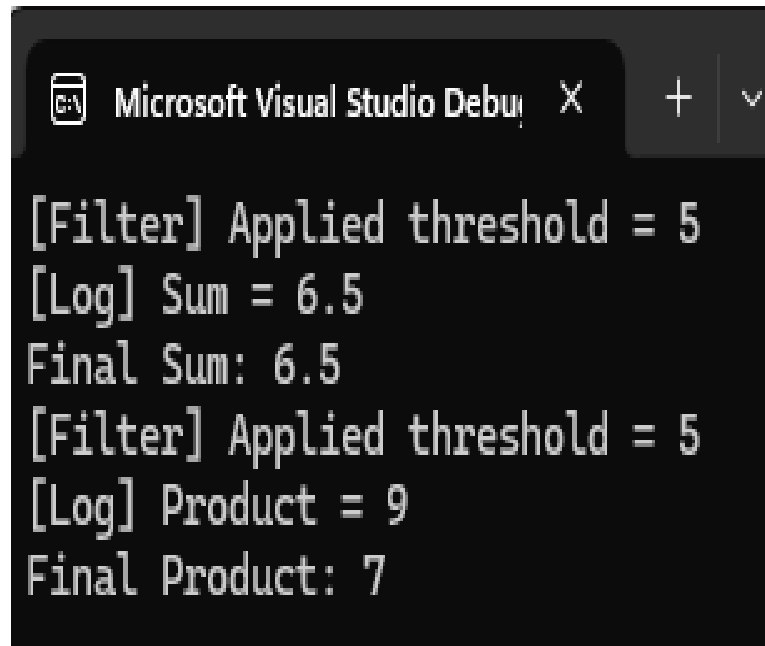
    // Цепочка декораторов: фильтрация -> логирование ->
ограничение
    auto filtered = make_shared<ThresholdFilterDecorator>(simple,
5.0);
    auto logged = make_shared<LoggingDecorator>(filtered);
    auto bounded = make_shared<BoundedCollection>(logged, 7.0);

    cout << "Final Sum: " << bounded->Sum() << endl;
    cout << "Final Product: " << bounded->Product() << endl;

```

```
return 0;  
}
```

Подивимось скріншот з результатом на ілюстрації 2.1:



```
Microsoft Visual Studio Debug Console  
[Filter] Applied threshold = 5  
[Log] Sum = 6.5  
Final Sum: 6.5  
[Filter] Applied threshold = 5  
[Log] Product = 9  
Final Product: 7
```

Рисунок 2.1 — Результат роботи програми

Програма була завантажена на GitHub згідно з завданням, ілюстрація 2.2:

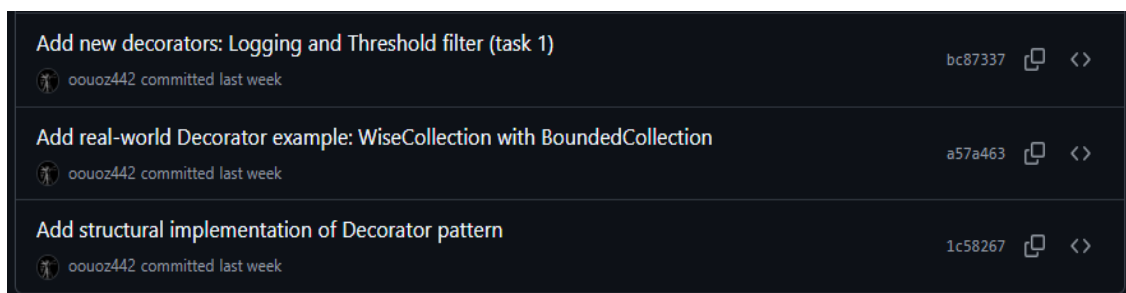


Рисунок 2.2 - GitHub

Висновок

Патерн “Decorator” — це потужний інструмент для динамічного розширення функціональності об’єктів у програмуванні. Він дозволяє створювати гнучкі та масштабовані системи, де нову поведінку можна додати без зміни існуючого коду. Разом з тим, використання декораторів може ускладнити структуру програми, тому варто застосовувати цей підхід там, де дійсно потрібна динамічна розширюваність.