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

Лабораторна робота №9

з дисципліни: «Інженерія програмного забезпечення»

за темою: «Патерн проектування Composite»

Виконав:
студент 3 курсу
Владислав Краковський

Перевірив:
викладач
Пенко В.Г.

Одеса 2025

Зміст

1 ЗАВДАННЯ РОБОТИ

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

Умова роботи:

Приклад – вивід багаторівневого тексту.

Текст має ієрархічну структуру. На першому рівні глави. Розділ складається з абзаців. Абзаци – із пропозицій. Пропозиції – зі слів. На основі патерну Composite реалізувати акуратне виведення тексту.

Рисунок 1.1 - Умова

Варіанти самостійних завдань

1. Модифікувати попереднє застосування так, щоб перше слово в абзаці починалося з великої літери. Чому абзаци не притримуються ширини?

Рисунок 1.2 - Самостійне завдання

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 Composite:

Composite (Компоновщик, Композит) — це структурний патерн проєктування, який дозволяє об’єднувати об’єкти у деревоподібні структури та працювати з ними як з єдиним цілим.
Composite дає змогу клієнту однаково взаємодіяти як з простими об’єктами (“листям”), так і зі складними структурами (“гілками”, “композитами”).

Як реалізується патерн Composite?

1. Створюється спільний інтерфейс (абстрактний клас) для всіх елементів структури (наприклад, Component).
2. Реалізуються два типи класів:
    - Leaf (лист) — простий об’єкт, який не містить підлеглих.
    - Composite (композит) — складний об’єкт, який може містити інші об’єкти, як листя, так і інші композити.
3. Composite містить колекцію об’єктів-елементів (дітей) та реалізує операції для додавання, видалення й обходу цих елементів.
4. Клієнт працює з усією структурою через спільний інтерфейс, не розрізняючи листя і композити.

У C++ це означає створення ієрархії з базовим класом та двома типами нащадків: Leaf та Composite.

Для чого використовується патерн Composite?

- Коли потрібно організувати ієрархічні структури (дерева, меню, документи, файлові системи).
- Для створення складних об’єктів із вкладеними підструктурами.
- Для уніфікації роботи з окремими й складеними об’єктами.

Типові приклади:

- Меню та підменю у графічних інтерфейсах.
- Документ із главами, параграфами, словами.
- Файлова система з файлами і папками.

Переваги патерну Composite

- Єдиний інтерфейс для роботи з усією ієрархією (простих і складних елементів).
- Гнучкість — можна динамічно додавати нові типи компонентів і структури.
- Спрощення коду клієнта — не потрібно перевіряти тип елемента під час обходу структури.

Недоліки патерну Composite

− Може ускладнювати реалізацію операцій, якщо листя і композити суттєво відрізняються за поведінкою.
− Іноді порушується принцип суворої типізації — не всі методи мають сенс для всіх елементів.
− Може бути “надмірним” для простих структур.

Запишемо розв’язок поставленої задачі з використання патерну Composite:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <cctype>

// Forward declaration WordElement
class WordElement;

// Абстрактный компонент
class TextElement {
protected:
    TextElement* parent = nullptr;
public:
    virtual ~TextElement() {}
    void SetParent(TextElement* p) { parent = p; }
    virtual void Add(TextElement* element) {}
    virtual void Remove(TextElement* element) {}
    virtual void Display() const = 0;
};

// "Целый текст" (корень)
class WholeText : public TextElement {
    std::string title;
    std::vector<TextElement*> chapters;
public:
    WholeText(const std::string& t) : title(t) {}

    void Add(TextElement* element) override {
        chapters.push_back(element);
```

```
        element->SetParent(this);
    }
    void Remove(TextElement* element) override {
        auto it = std::remove(chapters.begin(), chapters.end(),
element);
        if (it != chapters.end()) chapters.erase(it,
chapters.end());
    }
    void Display() const override {
        for (char ch : title)
            std::cout << (char)std::toupper(ch) << ' ';
        std::cout << "\n\n";
        for (auto chapter : chapters) chapter->Display();
    }
};

// Глава
class ChapterElement : public TextElement {
    std::string title;
    std::vector<TextElement*> paragraphs;
public:
    ChapterElement(const std::string& t) : title(t) {}
    void Add(TextElement* element) override {
        paragraphs.push_back(element);
        element->SetParent(this);
    }
    void Remove(TextElement* element) override {
        auto it = std::remove(paragraphs.begin(),
paragraphs.end(), element);
        if (it != paragraphs.end()) paragraphs.erase(it,
paragraphs.end());
    }
    void Display() const override {
        std::cout << title << std::endl << std::endl;
```

```
        for (auto par : paragraphs) par->Display();
    }
};

// Объявляем ParagraphElement заранее, метод Display реализуем
позже
class ParagraphElement : public TextElement {
    std::vector<TextElement*> words;
    int width;
public:
    ParagraphElement(int w) : width(w) {}
    void Add(TextElement* element) override {
        words.push_back(element);
        element->SetParent(this);
    }
    void Remove(TextElement* element) override {
        auto it = std::remove(words.begin(), words.end(),
element);
        if (it != words.end()) words.erase(it, words.end());
    }
    void Display() const override; // только объявление!
};

// Слово (лист)
class WordElement : public TextElement {
    std::string word;
public:
    WordElement(const std::string& w) : word(w) {}
    int GetLength() const { return
static_cast<int>(word.length()); }
    void Add(TextElement*) override {
        throw std::runtime_error("Cannot add to a WordElement");
    }
```

```
    void Remove(TextElement*) override {
        throw std::runtime_error("Cannot remove from a
WordElement");
    }
    void Display(bool capitalize = false) const {
        if (capitalize && !word.empty()) {
            std::string copy = word;
            copy[0] = std::toupper(copy[0]);
            std::cout << copy << " ";
        }
        else {
            std::cout << word << " ";
        }
    }
    void Display() const override { Display(false); }
};

// Теперь определяем метод ParagraphElement::Display
void ParagraphElement::Display() const {
    int horPosition = 0;
    bool first = true;
    for (const auto& word : words) {
        const WordElement* we = dynamic_cast<const
WordElement*>(word);
        int len = we ? we->GetLength() : 0;
        if (horPosition + len > width && !first) {
            std::cout << "\n";
            horPosition = 0;
        }
        if (we) we->Display(first);
        else word->Display();
        horPosition += len + 1;
        first = false;
```

```
    }
    std::cout << "\n\n";
}

int main() {
    WholeText* book = new WholeText("My great book");
    ChapterElement* chapter1 = new ChapterElement("Introduction");
    ChapterElement* chapter2 = new ChapterElement("Conclusion");
    ParagraphElement* p1 = new ParagraphElement(15);
    ParagraphElement* p2 = new ParagraphElement(25);
    ParagraphElement* p3 = new ParagraphElement(25);
    WordElement* w1 = new WordElement("sunday");
    WordElement* w2 = new WordElement("monday");
    WordElement* w3 = new WordElement("tuesday");
    WordElement* w4 = new WordElement("wednesday");
    WordElement* w5 = new WordElement("thursday");
    WordElement* w6 = new WordElement("friday");
    WordElement* w7 = new WordElement("saturday");

    book->Add(chapter1);
    book->Add(chapter2);
    chapter1->Add(p1);
    chapter1->Add(p2);
    chapter2->Add(p3);
    chapter2->Add(p1);

    p1->Add(w1); p1->Add(w2); p1->Add(w3);
    p2->Add(w4); p2->Add(w5); p2->Add(w6);
    p3->Add(w7); p3->Add(w1); p3->Add(w2);

    book->Display();
```

```
    // Очистка памяти
    delete w1; delete w2; delete w3; delete w4; delete w5; delete
w6; delete w7;
    delete p1; delete p2; delete p3;
    delete chapter1; delete chapter2; delete book;


    return 0;
}
```

Подивимось скріншот з результатом на ілюстрації 2.1:

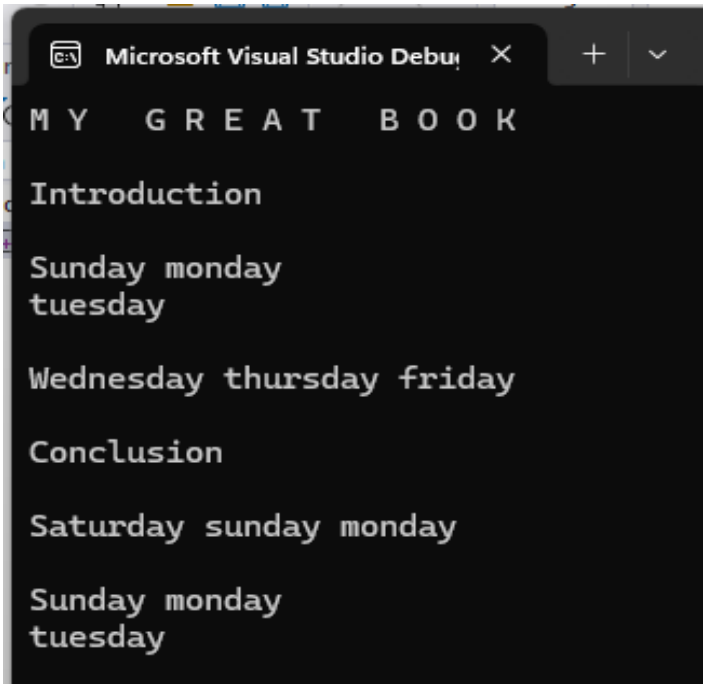

Рисунок 2.1 — Результат роботи програми

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

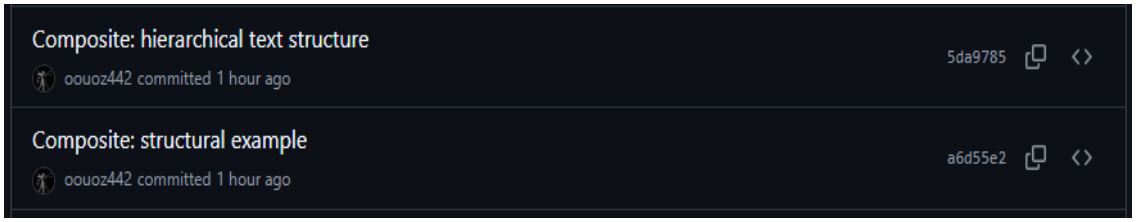

Рисунок 2.2 - GitHub

Висновок

Патерн “Composite” дозволяє ефективно моделювати складні ієрархічні структури в програмному забезпеченні. Він спрощує клієнтський код і забезпечує гнучкість розширення, але може призвести до деякого ускладнення архітектури та збільшення кількості класів. Composite доцільно застосовувати, коли потрібно оперувати складними деревоподібними об’єктами, де важлива уніфікованість операцій над елементами різного типу.