МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний аерокосмічний університет
«Харківський авіаційний інститут»

факультет програмної інженерії та бізнесу

кафедра інженерії програмного забезпечення

Розрахунково-графічна робота

з дисципліни «Архітектура та проектування програмного забезпечення .Net»
назва дисципліни
на тему: «Шаблони проєктування»

Виконав: студент 2 курсу групи № 621п
Освітньої програми (спеціальності)
121 інженерія програмного забезпечення
(шифр і назва ОП)
Литвинов А. О.
(прізвище й ініціали студента)
Прийняв: доцент Лучшев П. О.
(посада, науковий ступінь, прізвище й ініціали)
Кількість балів: _____

Харків – 2025

ЗМІСТ

МЕТА РОБОТИ

Вивчення стандартних ситуацій у процесі розробки складних програмних проектів та застосування шаблонів проектування (Design patterns) для їх вирішення.

ПОСТАНОВКА ЗАДАЧІ

Самостійно знайти в мережі Інтернет опис шаблонів проектування (Design patterns) наступних типів:

– що породжує (Creational patterns);
– структурного (Structural patterns);
– поведінкового (Behavioral pattern);
– паралельних обчислень (Concurrency pattern).

У репозиторії GitHub створити файл ReadMe.md і на підставі зібраного матеріалу сформувати текстовий опис шаблону та його графічне подання у вигляді відповідних UML-діаграм:

– статичної моделі (діаграма класів та/або діаграма модулів);
– динамічної моделі (діаграма взаємодії та/або стану);

Для побудови діаграм використовувати інструмент візуалізації Mermaid, який формує зображення з текстового опису на основі мови Markdown. На практичному етапі для кожного шаблону проектування розробити програмний проект, який демонструє особливості застосування заданих шаблонів проектування практично.

Додатково кожний шаблон проектування (design pattern) за варіантом створити у вигляді zip-файлу, який є шаблоном проекту або елемента (Project / Item Template) для середовища розробки Visual Studio.

Варіант 12:

Creational pattern: Prototype
Structural pattern: Composite
Behavioral pattern: Null object
Concurrency pattern: Monitor object

ТЕОРЕТИЧНІ ВІДОМОСТІ

Prototype

Тип шаблону: Creational pattern

Призначення: Створює нові об'єкти шляхом копіювання вже існуючих (прототипів), замість використання new.

Джерело: Прототип(шаблон проєктування)

Опис:

Prototype — це компонент архітектурного рівня, що реалізує механізм створення нових об’єктів шляхом копіювання вже існуючих. Замість того, щоб створювати об’єкти через new, клієнт отримує клон вже існуючого зразка.

Шаблон Prototype доцільно застосовувати у наступних випадках:

- Коли створення об’єкта є ресурсоємним або потребує складної конфігурації (наприклад, при читанні з бази даних, генерації на основі великої кількості параметрів).
- Коли необхідно уникнути залежності від конкретного класу під час створення нових об'єктів.
- Коли потрібно динамічно створювати об'єкти, типи яких визначаються під час виконання програми.
- У випадках, коли існує встановлений набір стандартних об'єктів-прототипів, які можуть слугувати основою для створення нових екземплярів.
- У редакторах, конструкторах інтерфейсів, ігрових рушіях, де користувач або система створює нові елементи на основі заздалегідь визначених шаблонів.

Застосування шаблону Prototype дозволяє зменшити кількість дубльованого коду, спростити структуру створення об’єктів та підвищити масштабованість програмного продукту.

Основні структурні елементи:

1. IPrototype - Інтерфейс, що оголошує метод Clone(), який повинен реалізовуватись усіма конкретними прототипами. Метод Clone() відповідає за створення копії об’єкта.
2. ConcretePrototype - Конкретна реалізація інтерфейсу IPrototype, що містить поля даних (наприклад, field1, field2) та реалізує метод Clone(). Метод повертає новий об’єкт з такими ж значеннями полів.
3. Client - Клас, який використовує прототипи для створення нових об’єктів. Він не створює об'єкти напряму через new, а викликає метод Clone() на існуючих екземплярах. Це дозволяє зменшити залежність від конкретних класів.

Статичну діаграму класів для шаблону Prototype зображено на рисунку 1.

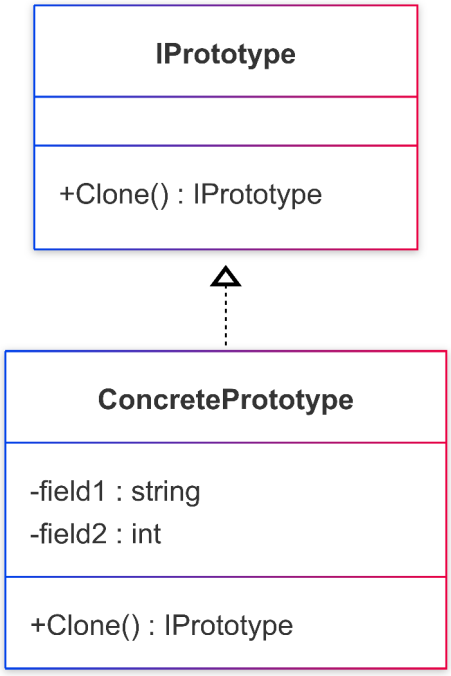

Рисунок 1 - Статична діаграма класів для шаблону Prototype.

Діаграму взаємодії у шаблоні Prototype зображено на рисунку 2.

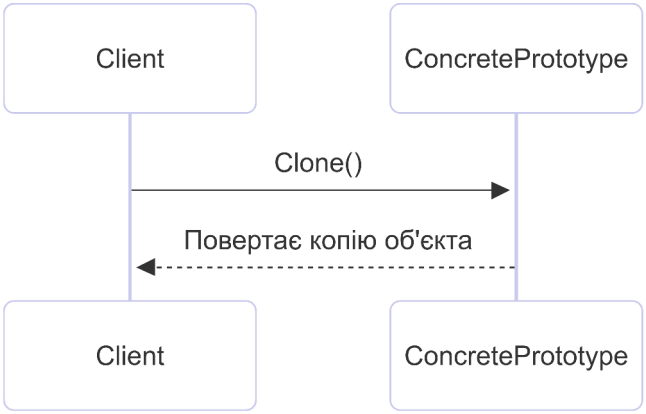

Рисунок 2 - Діаграма взаємодії у шаблоні Prototype.

Composite

Тип шаблону: Structural pattern

Призначення: Дозволяє об'єднувати об'єкти у деревоподібну структуру та працювати з окремими об’єктами та їх групами однаково.

Джерело: Компонувальник(Шаблон проєктування)

Опис:

Composite — це структурний шаблон проєктування, який дозволяє створювати ієрархічні структури об'єктів, де одиничні об’єкти і групи об’єктів обробляються однаковим чином. Завдяки цьому шаблону клієнтський код може працювати з усіма об'єктами через один інтерфейс, не знаючи, з чим саме він має справу.

Шаблон Composite доцільно застосовувати у наступних випадках:

- Коли потрібно представляти частково-цілісні ієрархії (наприклад, дерево UI-елементів, файлову систему, організаційну структуру).
- Коли клієнтам необхідно однаково взаємодіяти з простими та складеними об'єктами.
- Коли важливо мати можливість рекурсивно обробляти структуру без дублювання коду.
- У випадках, коли система має рекурсивну природу, тобто коли компоненти можуть містити інші компоненти того ж типу.

Застосування шаблону Composite дозволяє зменшити залежність клієнтського коду від конкретних типів об'єктів, уніфікує взаємодію з елементами і дає змогу гнучко модифікувати структуру без зміни логіки її обробки.

Основні структурні елементи:

1. Component - Абстрактний клас або інтерфейс, який описує загальні операції як для простих, так і для складених об’єктів. Наприклад, методи Add(), Remove(), Display().
2. Leaf - Конкретна реалізація Component, яка представляє окремі (атомарні) об'єкти. Вони не мають дочірніх елементів і реалізують логіку базових операцій.
3. Composite - Клас, який також реалізує Component, але містить колекцію інших об'єктів-компонентів. Цей клас реалізує логіку керування колекцією дочірніх елементів.
4. Client - Клас або модуль, який працює з усіма об’єктами через інтерфейс Component, не роблячи відмінностей між Leaf і Composite.

Статичну діаграму класів для шаблону Composite зображено на рисунку 3.

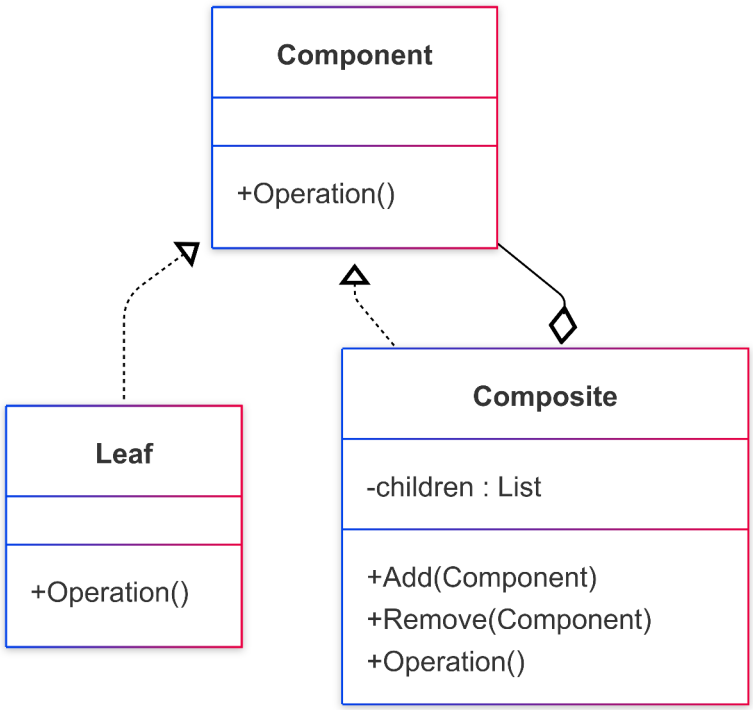

Рисунок 3 - Статична діаграма класів для шаблону Composite.

Діаграму взаємодії у шаблоні Composite зображено на рисунку 4.

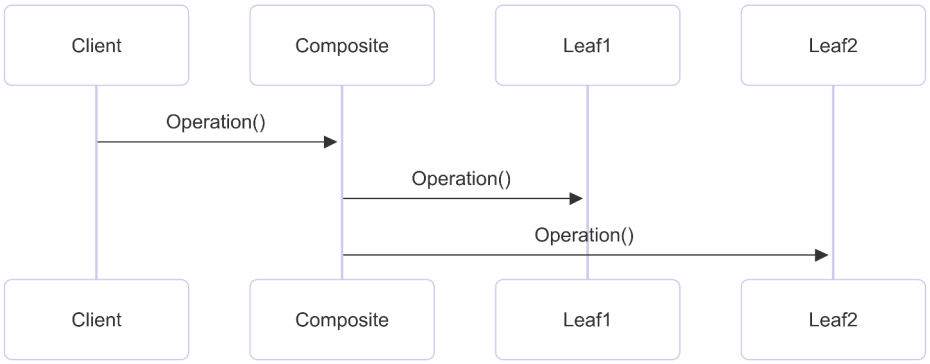

Рисунок 4 - Діаграма взаємодії у шаблоні Composite.

Null Object

Тип шаблону: Behavioral pattern

Призначення: Уникає перевірок на null, надаючи об'єкт, який поводиться як "порожній", але реалізує очікувану поведінку.

Джерело: https://www.geeksforgeeks.org/null-object-design-pattern/

Опис:

Null Object — це поведінковий шаблон проєктування, який пропонує використовувати спеціальний об’єкт-пустушку замість null. Такий об’єкт

реалізує інтерфейс або базовий клас, але не виконує жодних дій або забезпечує нейтральну поведінку, яка не впливає на логіку програми.

Завдяки цьому шаблону усувається необхідність у перевірках на null, а також знижується ризик помилок, пов'язаних із відсутністю об'єкта (наприклад, NullReferenceException).

Шаблон Null Object доцільно застосовувати у наступних випадках:

- Коли у програмі часто трапляється ситуація, коли об'єкт може бути відсутнім (null).
- Коли бажано, щоб у відсутності конкретного об'єкта все одно була доступна деяка безпечна за замовчуванням поведінка.
- Коли потрібно спростити клієнтський код, усунувши перевірки на null перед кожним викликом методу.
- У системах логування, безпечного виводу, шаблонах команд, коли "нічого не робити" — це теж поведінка.

Застосування шаблону Null Object дозволяє створювати гнучкіші, чистіші та більш надійні системи, де поведінка об'єктів є уніфікованою незалежно від їх реальної "наявності".

Основні структурні елементи:

1. AbstractObject (інтерфейс або абстрактний клас) - Визначає загальний інтерфейс для реального об'єкта та об’єкта-пустушки. Наприклад, метод Execute() або Log().
2. RealObject - Конкретна реалізація AbstractObject, яка виконує реальну корисну дію — наприклад, обробляє запит, виконує команду, або записує у лог.
3. NullObject - Конкретна реалізація AbstractObject, яка не робить нічого або виконує нейтральну дію. Вона замінює null, дозволяючи клієнтському коду викликати методи без перевірок.
4. Client - Клас, який використовує об’єкти через інтерфейс AbstractObject, не знаючи, чи працює з реальним об’єктом, чи з об’єктом-пустушкою. Це дозволяє однакове використання і реальних об'єктів, і NullObject, без додаткових умов.

Статичну діаграму класів для шаблону Null Object зображено на рисунку 5.

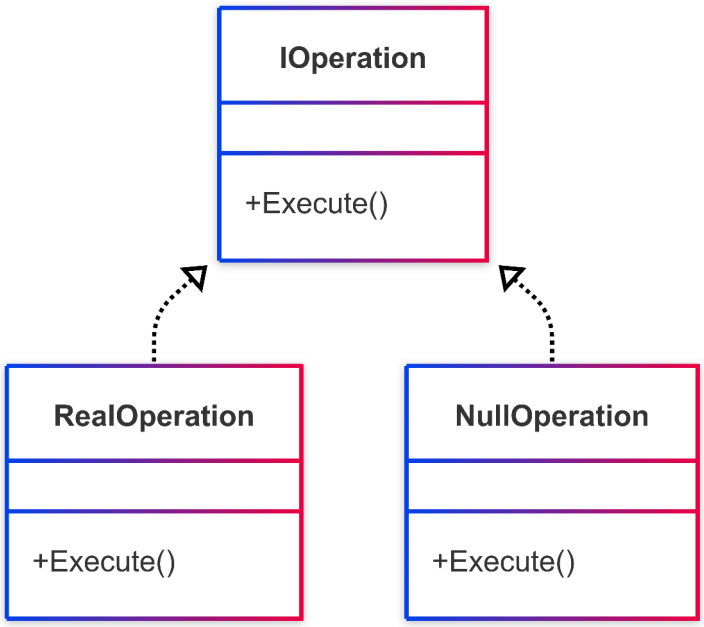

Рисунок 5 - Статична діаграма класів для шаблону Null Object.

Діаграму взаємодії у шаблоні Null Object зображено на рисунку 6.

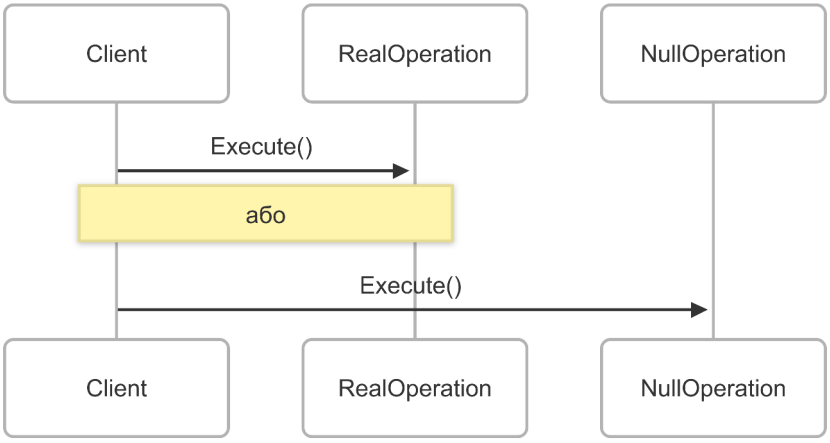

Рисунок 6 - Діаграма взаємодії у шаблоні Null Object.

Monitor Object

Тип шаблону: Concurrency pattern

Призначення: Забезпечує синхронізований доступ до об’єкта з кількох потоків, інкапсулюючи механізм блокування всередині самого об’єкта.

Джерело: https://softwarepatternslexicon.com/mastering-design-patterns/6/4/

Опис:

Monitor Object — це шаблон паралельного програмування, який дозволяє безпечно виконувати одночасний доступ до об’єкта з кількох потоків, забезпечуючи автоматичне управління синхронізацією.

Ідея полягає в тому, що всі методи об’єкта є критичними секціями, тобто їх виконання дозволено лише одному потоку одночасно. Синхронізація виконується всередині самого об'єкта, і клієнтський код не повинен явно керувати блокуваннями.

Завдяки шаблону Monitor Object забезпечується інкапсуляція синхронізації, що зменшує ризик помилок при роботі з потоками, таких як deadlock, race condition або неконсистентність даних.

Шаблон Monitor Object доцільно застосовувати у наступних випадках:

- Коли об’єкт використовується одночасно з кількох потоків і вимагає захисту спільних ресурсів.
- Коли потрібно ізолювати логіку синхронізації всередині самого класу.
- Коли бажано, щоб багатопотокова взаємодія залишалась прозорою для клієнта.

Застосування шаблону Monitor Object дозволяє створювати надійні та безпечні багатопотокові компоненти, де синхронізація відбувається автоматично, без участі клієнтського коду.

Основні структурні елементи:

1. MonitorObject - Клас, який містить критичну логіку або ресурси, доступ до яких потребує синхронізації. Всі методи, що змінюють або читають ці ресурси, заблоковані синхронізуючими механізмами.
2. Lock/Mutex - Об'єкт або конструкція, яка забезпечує виконання лише одного потоку всередині методів MonitorObject одночасно. Цей механізм не видно клієнту, оскільки він повністю інкапсульований.
3. Client - Потоки або зовнішній код, які звертаються до методів MonitorObject без потреби явно управляти синхронізацією. Monitor Object гарантує безпечне виконання навіть при одночасному доступі.

Статичну діаграму класів для шаблону Monitor Object зображено на рисунку 7.

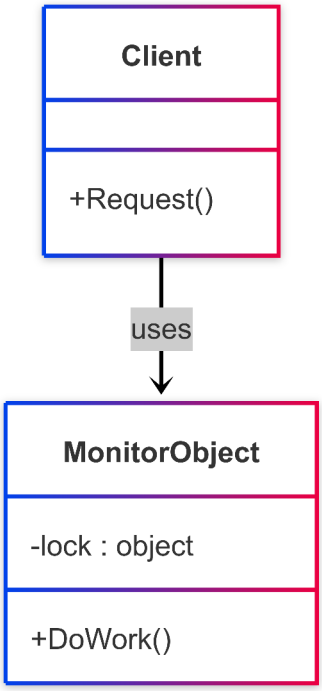

Рисунок 7 - Статична діаграма класів для шаблону Monitor Object.

Діаграму взаємодії у шаблоні Monitor Object зображено на рисунку 8.

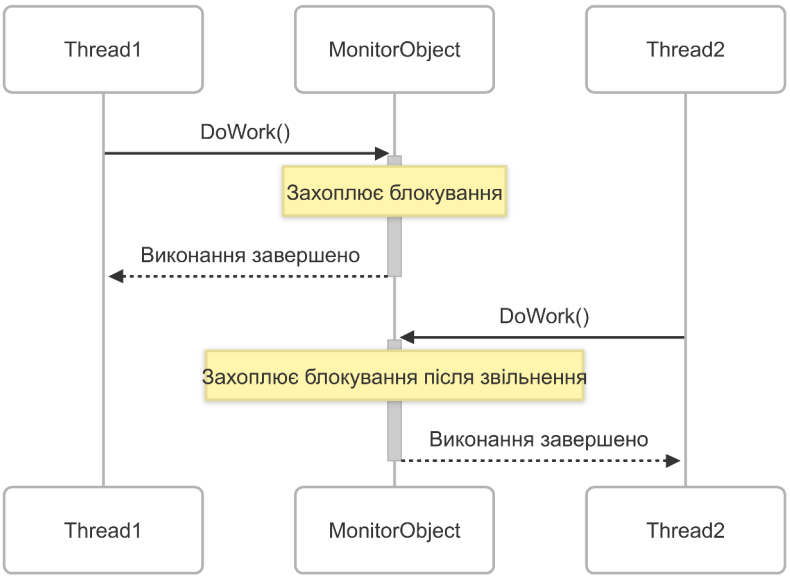

Рисунок 8 - Діаграма взаємодії у шаблоні Monitor Object.

ПРАКТИЧНА ЧАСТИНА

Лістинг програми PrototypeDemo

```
using System;
Console.OutputEncoding = System.Text.Encoding.UTF8;

Console.WriteLine("Демонстрація шаблону Prototype:\n");

// Створюємо оригінальний об'єкт
User originalUser = new User { Name = "Олена", Age = 30 };
Console.WriteLine("Оригінал:");
originalUser.Display();

// Клонуємо об'єкт
User clonedUser = (User)originalUser.Clone();
clonedUser.Name = "Марія"; // Змінюємо ім'я в копії

Console.WriteLine("\nКопія:");
clonedUser.Display();

Console.WriteLine("\nПеревірка, оригінального об'єкту:");
originalUser.Display();

public interface IPrototype
{
    IPrototype Clone();
}

// Конкретна реалізація прототипу — клас користувача
public class User : IPrototype
{
    public string Name { get; set; }
    public int Age { get; set; }

    // Метод, який створює копію об'єкта
    public IPrototype Clone()
    {
        return new User
        {
            Name = this.Name,
            Age = this.Age
        };
    }

    public void Display()
    {
        Console.WriteLine($"Ім'я: {Name}, Вік: {Age}");
    }
}
```

Лістинг програми CompositeDemo

```
Console.OutputEncoding = System.Text.Encoding.UTF8;
Console.WriteLine("Демонстрація шаблону Composite:\n");

var circle1 = new Circle("Червоне коло");
var square1 = new Square("Синій квадрат");

var group1 = new GraphicGroup("Група 1");
group1.Add(circle1);
group1.Add(square1);

var group2 = new GraphicGroup("Група 2");
group2.Add(new Circle("Зелене коло"));
group2.Add(group1);

group2.Draw();

// Абстрактний компонент
public abstract class Graphic
{
    public string Name { get; set; }

    public Graphic(string name)
    {
        Name = name;
    }

    // Метод, який мають реалізувати всі компоненти
    public abstract void Draw(int indent = 0);
}

// Лист — простий об'єкт (не має підоб'єктів)
public class Circle : Graphic
{
    public Circle(string name) : base(name) { }

    public override void Draw(int indent = 0)
    {
            Console.WriteLine(new string(' ', indent) + $"Коло:
{Name}");
    }
}

public class Square : Graphic
{
    public Square(string name) : base(name) { }

    public override void Draw(int indent = 0)
    {
```

```
            Console.WriteLine(new string(' ', indent) + $"Квадрат:
{Name}");
    }
}

// Composite — контейнер для графічних об'єктів
public class GraphicGroup : Graphic
{
    private List<Graphic> _children = new();

    public GraphicGroup(string name) : base(name) { }

    public void Add(Graphic graphic)
    {
        _children.Add(graphic);
    }

    public void Remove(Graphic graphic)
    {
        _children.Remove(graphic);
    }

    public override void Draw(int indent = 0)
    {
            Console.WriteLine(new string(' ', indent) + $"Група:
{Name}");
        foreach (var child in _children)
        {
            child.Draw(indent + 2); // малюємо дочірні з відступом
        }
    }
}
```

Лістинг програми NullObjectDemo

```
Console.OutputEncoding = System.Text.Encoding.UTF8;
Console.WriteLine("Демонстрація шаблону Null Object:\n");

// Реальний логер
ILogger realLogger = new ConsoleLogger();
var serviceWithLogger = new UserService(realLogger);
serviceWithLogger.CreateUser("Олексій");

Console.WriteLine();

// Null-об'єкт замість null
ILogger nullLogger = new NullLogger();
var serviceWithoutLogging = new UserService(nullLogger);
serviceWithoutLogging.CreateUser("Гість");

// Абстрактний інтерфейс або базовий клас
public interface ILogger
```

```
{
    void Log(string message);
}

// Реальна реалізація логера
public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine($"[LOG] {message}");
    }
}

// Null Object — нічого не робить, але реалізує інтерфейс
public class NullLogger : ILogger
{
    public void Log(string message)
    {
        // Нічого не робить
    }
}

// Клас, який залежить від логера, але не перевіряє на null
public class UserService
{
    private readonly ILogger _logger;

    public UserService(ILogger logger)
    {
        _logger = logger;
    }

    public void CreateUser(string name)
    {
        Console.WriteLine($"Створено користувача: {name}");
        _logger.Log($"Користувача '{name}' створено.");
    }
}
```

Лістинг програми MonitorObjectDemo

```
Console.OutputEncoding = System.Text.Encoding.UTF8;
Console.WriteLine("Демонстрація шаблону Monitor Object:\n");

Counter counter = new Counter();

// Створюємо 3 потоки, які одночасно інкрементують лічильник
Thread thread1 = new Thread(() => counter.Increment("Потік 1"));
Thread thread2 = new Thread(() => counter.Increment("Потік 2"));
Thread thread3 = new Thread(() => counter.Increment("Потік 3"));

thread1.Start();
```

```
thread2.Start();
thread3.Start();

thread1.Join();
thread2.Join();
thread3.Join();

// Monitor Object — інкапсулює синхронізований доступ
public class Counter
{
    private int _count = 0;
    private readonly object _lock = new();

    public void Increment(string threadName)
    {
        lock (_lock) // Синхронізований доступ
        {
                        Console.WriteLine($"{threadName} → Збільшує
лічильник...");
            int temp = _count;
            Thread.Sleep(100); // Емуляція навантаження
            _count = temp + 1;
              Console.WriteLine($"{threadName} → Поточне значення:
{_count}");
        }
    }
}
```

Лістинг програми MonitorObjectDemoExtra

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
#include <windows.h>

class Counter {
private:
    int count = 0;
    std::mutex mtx;

public:
    void Increment(const std::string& threadName) {
        std::lock_guard<std::mutex> lock(mtx);
         std::cout << threadName << " → Збільшує лічильник..." <<
std::endl;
        int temp = count;
         std::this_thread::sleep_for(std::chrono::milliseconds(100)
);
        count = temp + 1;
           std::cout << threadName << " → Поточне значення: " <<
count << std::endl;
```

```
    }
};

void threadFunction(Counter& counter, const std::string& name) {
    counter.Increment(name);
}

int main() {
    SetConsoleOutputCP(CP_UTF8);
    SetConsoleCP(CP_UTF8);

    std::cout << "Демонстрація шаблону Monitor Object (C++):\n" <<
std::endl;

    Counter counter;

    std::thread t1(threadFunction, std::ref(counter), "Потік 1");
    std::thread t2(threadFunction, std::ref(counter), "Потік 2");
    std::thread t3(threadFunction, std::ref(counter), "Потік 3");

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

Самооцінка виконання вимог роботи

Таблиця 1 — Виконання вимог

№ п/п	Вимоги до роботи		Бали	Відмітка про виконання
1.	Базові	Текстовий опис шаблону, його основних складових частин, їх призначення (*з посиланням на першоджерело!*)	4	+
2.		***UML*** модель шаблону у вигляді ***Mermaid*** діаграми	4	+
3.		Програмний проект на *C#* , що демонструє специфіку шаблону проектування (кожен шаблон окремо)	4	+
4.		Створення шаблону проектування (***design pattern***) у вигляді шаблону проекту або елемента (***project / item template***) для ***Visual Studio*** (***zip*** -файл)	4	+
5.	Додаткові	Реалізація шаблону проектування додатковою (будь-якою за власним бажанням) мовою програмування	4	+
6.	Часу	Своєчасне виконання базових вимог та звітності	4	+