МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний аерокосмічний університет ім. М. Є. Жуковського

«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 проектування»

Виконав: студент 2 курсу групи № 621п

освітньої програми

121 інженерія програмного забезпечення

(шифр і назва ОП)

Головкіна О. А.

(прізвище й ініціали студента)

Прийняв: к.т.н., доцент каф. 603

Лучшев П. О.

(посада, науковий ступінь, прізвище й ініціали)

Кількість балів:

Харків – 2025

РОЗРАХУНКОВО-ГРАФІЧНА РОБОТА

Шаблони проектування

ЗМІСТ

[МЕТА РОБОТИ 3](#_Toc196493290)

[ПОСТАНОВКА ЗАДАЧІ 3](#_Toc196493291)

[Варіант 3 4](#_Toc196493292)

[ТЕОРЕТИЧНІ ВІДОМОСТІ 4](#_Toc196493293)

[Resource Acquisition Is Initialization (RAII) 4](#_Toc196493294)

[Decorator 6](#_Toc196493295)

[Protocol Stack 9](#_Toc196493296)

[Lock 11](#_Toc196493297)

[ПРАКТИЧНА ЧАСТИНА 14](#_Toc196493298)

[Лістинг програми RAII-Project 14](#_Toc196493299)

[Лістинг програми Decorator-Project 15](#_Toc196493300)

[Лістинг програми ProtocolStack-Project 17](#_Toc196493301)

[Лістинг програми Lock-Project 18](#_Toc196493302)

[Лістинг додаткової програми RAII-Cpipi 20](#_Toc196493303)

[ВИКОНАННЯ ВИМОГ НА РОБОТУ 21](#_Toc196493304)

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 (відповідно до варіанта завдання) 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 3

* Resource Acquisition Is (Creational patterns);
* Decorator (Structural patterns);
* Protocol stack (Behavioral pattern);
* Lock (Concurrency pattern).

ТЕОРЕТИЧНІ ВІДОМОСТІ

Resource Acquisition Is Initialization (RAII)

**Тип шаблону:** Породжуючий (Creational Pattern)

**Призначення:** Автоматичне управління ресурсами через життєвий цикл об’єкта

**Джерело:** [Концепція RAII](https://habr.com/ru/companies/otus/articles/778942/)

**Опис:**

RAII (Resource Acquisition Is Initialization) — це концепція, за якої ресурс прив’язується до об'єкта: він виділяється при створенні об'єкта та автоматично звільняється при його знищенні. Це забезпечує безпечне та надійне управління ресурсами без необхідності ручного контролю.

RAII допомагає уникати утечок пам’яті, неправильного закриття файлів і помилок у роботі з ресурсами. Особливо корисною ця концепція є в умовах виникнення виключень, адже ресурси гарантовано звільняються деструкторами, навіть якщо сталася помилка.

Код із використанням RAII стає чистішим і зрозумілішим: логіка управління ресурсами інкапсульована в об'єктах, що спрощує супровід і знижує ризик помилок. RAII підходить для роботи з пам’яттю, файлами, з’єднаннями та іншими ресурсами, зокрема й у багатопотокових середовищах.

Завдяки цій концепції створювати надійні програми простіше — ресурси використовуються ефективно, а звільнення відбувається автоматично й безпечно.

**Основні складові:**

Resource — ресурс, який потрібно отримати (наприклад, файл, з'єднання, м'ютекс).

RAIIWrapper — клас-обгортка, який:

* у конструкторі викликає acquire() — ініціалізує ресурс
* у деструкторі викликає release() — звільняє ресурс

Клієнтський код — створює об’єкт RAIIWrapper і більше не хвилюється про звільнення ресурсу.

**UML-діаграми:**

Діаграма класів на рисунку 1:

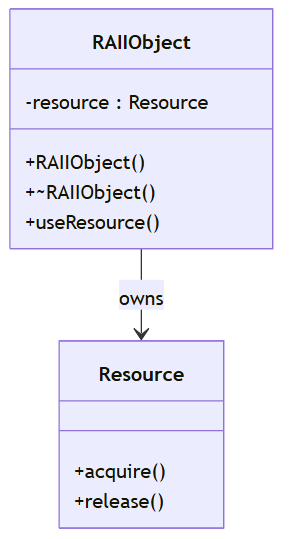


Рисунок 1 – Діаграма класів RAII

Діаграма взаємодії на рисунку 2:

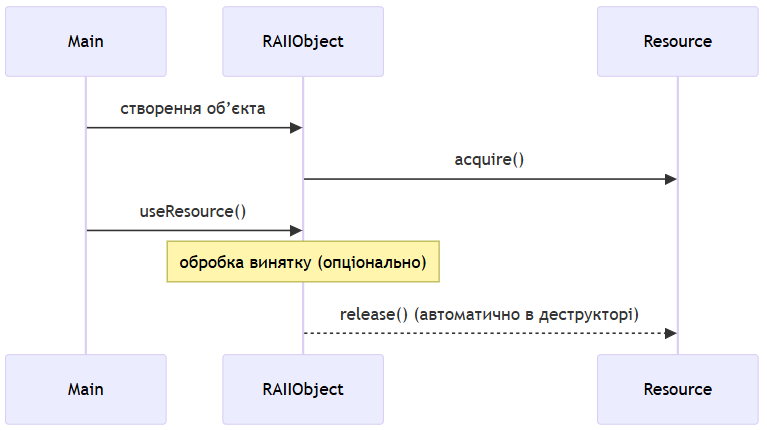


Рисунок 2 – Діаграма взаємодії RAII

Діаграма станів на рисунку 3:

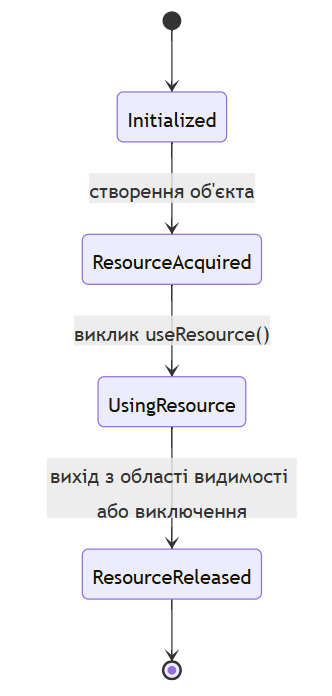


Рисунок 3 – Діаграма станів RAII

Decorator

**Тип шаблону:** Структурний (Structural Pattern)

**Призначення:** Дає змогу динамічно додавати об’єктам нову функціональність, загортаючи їх у корисні «обгортки»

**Джерело:** [Refactoring Guru – Декоратор](https://refactoring.guru/uk/design-patterns/decorator)

**Опис:**

Декоратор дозволяє динамічно додавати об’єктам нову поведінку, не змінюючи їхній код. Це досягається за рахунок обгортання об'єкта в інші об'єкти-декоратори, які реалізують той самий інтерфейс і розширюють або змінюють функціональність.

**Принцип роботи:**

1. Створюється базовий об'єкт (ConcreteComponent).
2. Він обгортається у декоратор (ConcreteDecorator).
3. Кожен декоратор реалізує ту саму інтерфейсну структуру та додає свою поведінку.
4. Можна вкладати декоратори один в одного, комбінуючи функціональність.

**Основні складові:**

* Component (Компонент) - абстрактний інтерфейс або базовий клас, який визначає стандартну поведінку об'єктів. Визначає контракт, спільний для всіх об'єктів (як основних, так і декораторів).
* ConcreteComponent (Конкретний компонент) - реалізація Component, до якої можна додавати нову функціональність. Основний об'єкт, який буде декоруватися.
* Decorator (Декоратор) - абстрактний клас, що реалізує інтерфейс Component і зберігає посилання на інший Component. Основа для створення декораторів, делегує виклики вкладеному об'єкту.
* ConcreteDecorator (Конкретний декоратор) - клас, що успадковується від Decorator і додає нову поведінку до компонента. Розширює або змінює функціональність, не змінюючи структуру основного класу.

**UML-діаграми:**

Діаграма класів на рисунку 4:

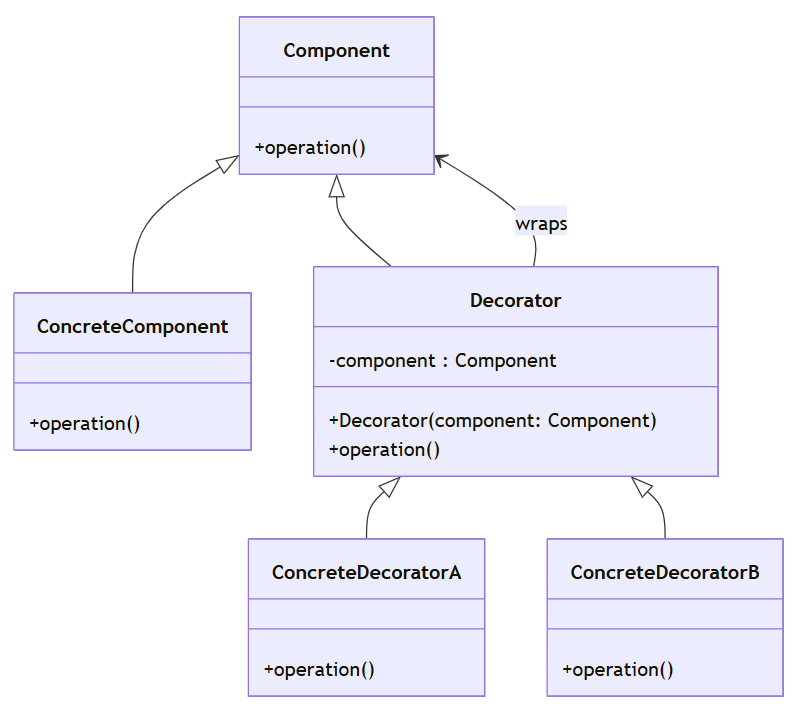


Рисунок 4 – Діаграма класів Decorator

Діаграма взаємодії на рисунку 5:

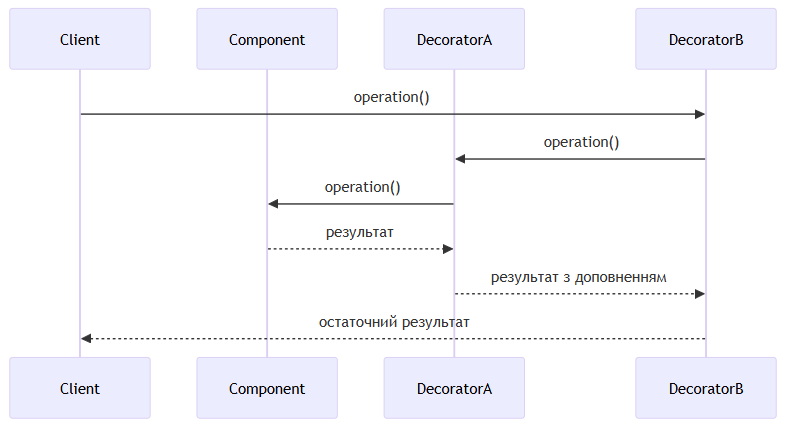


Рисунок 5 – Діаграма взаємодії Decorator

Діаграма станів на рисунку 6:

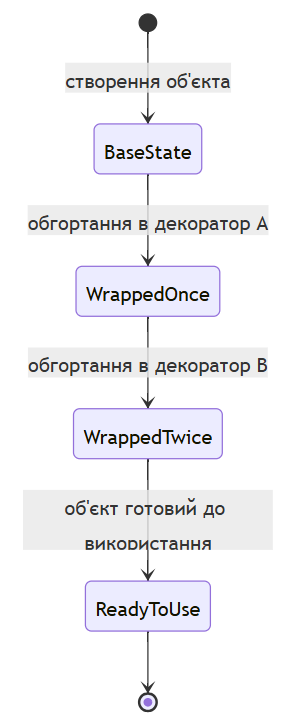


Рисунок 6 – Діаграма станів Decorator

Protocol Stack

**Тип шаблону:** Поведінковий (Behavioral pattern)

**Призначення:** Забезпечує гнучке управління шарами протоколу, дозволяючи динамічно додавати або видаляти шари без необхідності змінювати інші частини стеку

**Джерело:** [Eventhelix – Protocol Stack](https://www.eventhelix.com/design-patterns/protocol-stack/)

**Опис:**

У традиційних реалізаціях стеків протоколів шари жорстко пов'язані між собою, що ускладнює їх модифікацію. Шаблон Protocol Stack розв'язує цю проблему, впроваджуючи архітектуру, де шари можуть бути динамічно додані або видалені під час виконання програми.

**Принцип роботи:**

1. Protocol Stack реалізується як клас, який підтримує двозв'язний список активних шарів.
2. Кожен Protocol Layer реалізує стандартний інтерфейс для взаємодії з сусідніми шарами.
3. Методи Add\_Layer та Remove\_Layer дозволяють динамічно змінювати структуру стеку, додаючи або видаляючи шари в будь-якому місці.
4. Методи Handle\_Transmit та Handle\_Receive забезпечують передачу даних через стек, делегуючи обробку відповідним шарам.

**Основні складові:**

* Protocol Stack: Клас, який управляє стеком протоколів, підтримуючи динамічне додавання та видалення шарів.
* Protocol Layer: Базовий клас для всіх шарів протоколу, який визначає інтерфейси для взаємодії з іншими шарами.

**UML-діаграми:**

Діаграма класів на рисунку 7:

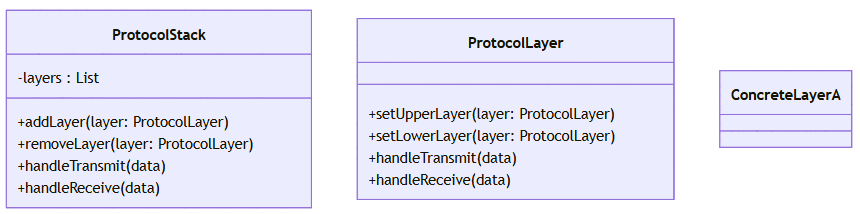


Рисунок 7 – Діаграма класів Protocol Stack

Діаграма взаємодії на рисунку 8:

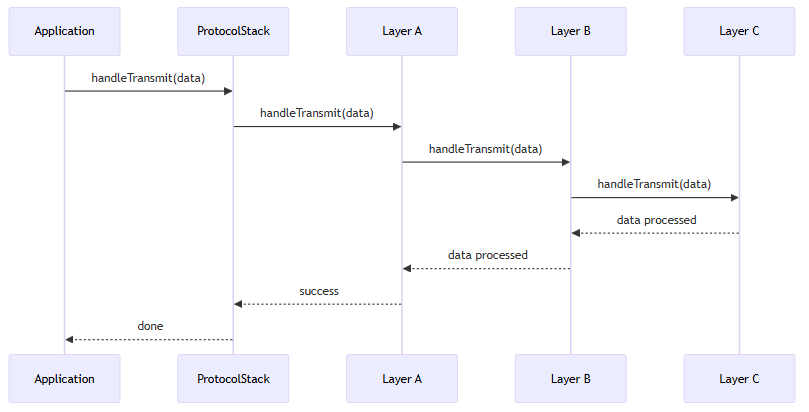


Рисунок 8 – Діаграма взаємодії Protocol Stack

Діаграма станів на рисунку 9:

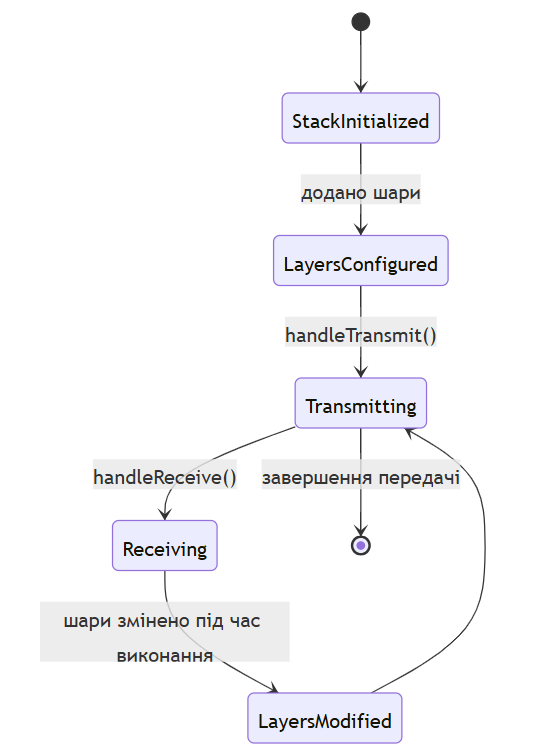


Рисунок 9 – Діаграма станів Protocol Stack

Lock

**Тип шаблону:** Паралельні обчислення (Concurrency pattern).

**Призначення:** Контроль одночасного доступу до спільного ресурсу, дозволяє гарантувати, що тільки один потік має доступ до ресурсу в конкретний момент часу

**Джерело:** [Wikipedia – Lock](http://en.wikipedia.org/wiki/Lock_(computer_science))

**Опис:**

Lock (Замикання) — це шаблон синхронізації, який забезпечує взаємне виключення при доступі до спільних ресурсів у багатопотоковому середовищі. Він гарантує, що тільки один потік може одночасно отримати доступ до критичної секції коду або даних, запобігаючи гонкам даних та пошкодженню стану.

Lock використовується для ізоляції критичних ділянок коду, які не повинні виконуватись одночасно кількома потоками. Це досягається шляхом "захоплення" замка перед виконанням і "звільнення" після завершення. Інші потоки чекають, поки замок стане доступним.

**Принцип роботи:**

1. Потік намагається захопити замок (lock).
2. Якщо замок вільний, потік отримує доступ до ресурсу.
3. Інші потоки блокуються, поки замок не буде звільнений.
4. Після завершення роботи потік звільняє замок (unlock), дозволяючи іншим потокам продовжити.

**Основні складові:**

* Lock / Mutex / Monitor: Примітиви синхронізації, які реалізують механізм замикання.
* Критична секція (Critical Section): Фрагмент коду, який виконується під захистом замка.
* Condition Variable (опціонально): Механізм, що дозволяє потокам чекати на певну умову всередині замкненої секції.

**UML-діаграми:**

Діаграма класів на рисунку 10:

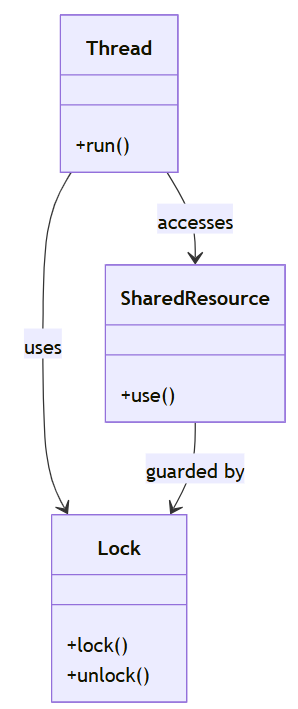


Рисунок 10 – Діаграма класів Protocol Stack

Діаграма взаємодії на рисунку 11:

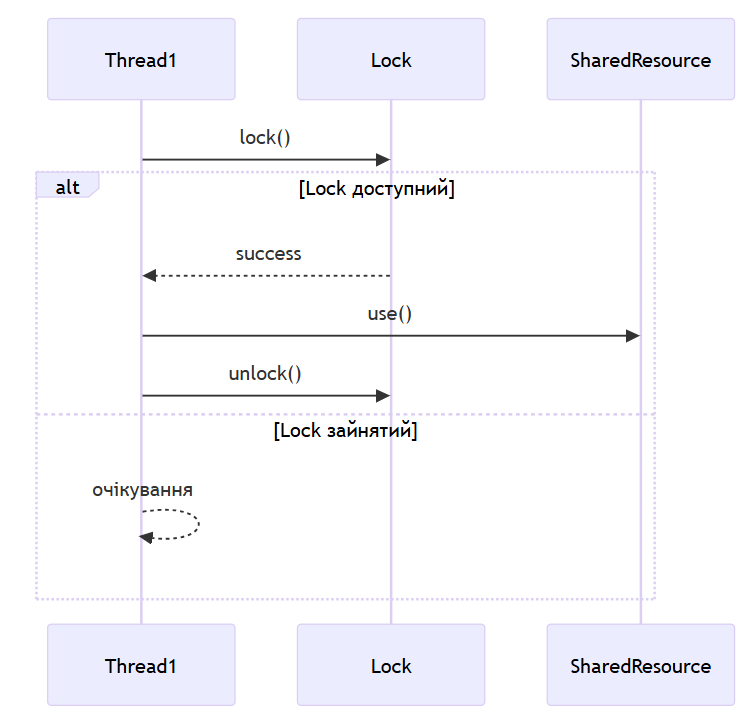


Рисунок 11 – Діаграма взаємодії Protocol Stack

Діаграма станів на рисунку 12:

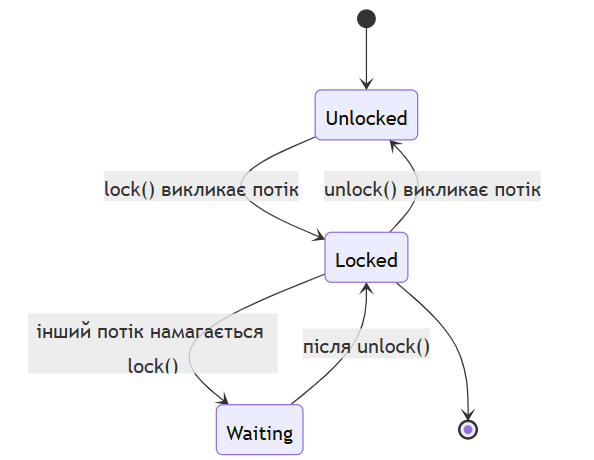


Рисунок 12 – Діаграма станів Protocol Stack

ПРАКТИЧНА ЧАСТИНА

Лістинг програми RAII-Project

using System;

using System.IO;

namespace RAII\_Project

{

public class FileManager : IDisposable

{

private readonly FileStream file = null!;

private bool disposed = false;

// Конструктор, який ініціалізує ресурс (відкриває файл)

public FileManager(string filePath)

{

try

{

file = new FileStream(filePath, FileMode.OpenOrCreate, FileAccess.ReadWrite);

Console.WriteLine("The file is opened.");

}

catch (Exception ex)

{

Console.WriteLine($"The file was not opened: {ex.Message}");

}

}

// Метод для запису в файл (демонстрація використання ресурсу)

public void WriteToFile(string text)

{

if (file != null)

{

using StreamWriter writer = new(file);

writer.WriteLine(text);

Console.WriteLine("Text has been written into the file!");

}

}

// Деструктор (або метод Dispose) для автоматичного закриття ресурсу

~FileManager()

{

Dispose(false);

}

// Реалізація IDisposable для коректного звільнення ресурсів

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

// Закриття файлу та очищення ресурсів

private void Dispose(bool disposing)

{

if (!disposed)

{

if (disposing)

{

file.Dispose();

}

Console.WriteLine("The file was closed and is cleaned.");

disposed = true;

}

}

}

class Program

{

static void Main()

{

string filePath = "test.txt";

// Використовуємо конструкцію using для автоматичного звільнення ресурсу

using (FileManager fileManager = new FileManager(filePath))

{

fileManager.WriteToFile("Hello world!\n" + DateTime.Now);

}

// У цей момент, коли блок using завершується, викликається Dispose і файл закривається

Console.WriteLine("Resourses is cleaned automatically");

}

}

}

Лістинг програми Decorator-Project

using System;

namespace Decorator\_Project

{

// Абстрактний клас напою

public abstract class Beverage

{

public string Description { get; set; } = "Unknown Beverage";

public virtual string GetDescription() => Description;

public abstract decimal Cost();

}

// Базовий клас декоратора

public abstract class BeverageDecorator : Beverage

{

protected Beverage beverage;

public BeverageDecorator(Beverage beverage)

{

this.beverage = beverage;

}

public override string GetDescription() => beverage.GetDescription();

}

// Конкретний напій - кава

public class Coffee : Beverage

{

public Coffee()

{

Description = "Coffee";

}

public override decimal Cost() => 1.99m;

}

// Конкретний напій - чай

public class Tea : Beverage

{

public Tea()

{

Description = "Tea";

}

public override decimal Cost() => 0.75m;

}

// Декоратор для молока

public class Milk : BeverageDecorator

{

public Milk(Beverage beverage) : base(beverage) { }

public override string GetDescription() => beverage.GetDescription() + ", Milk";

public override decimal Cost() => beverage.Cost() + 0.50m;

}

// Декоратор для цукру

public class Sugar : BeverageDecorator

{

public Sugar(Beverage beverage) : base(beverage) { }

public override string GetDescription() => beverage.GetDescription() + ", Sugar";

public override decimal Cost() => beverage.Cost() + 0.30m;

}

// Основна програма

class Program

{

static void Main()

{

Beverage beverage1 = new Coffee();

Console.WriteLine(beverage1.GetDescription() + " $" + beverage1.Cost());

// Додаємо молоко

beverage1 = new Milk(beverage1);

Console.WriteLine(beverage1.GetDescription() + " $" + beverage1.Cost());

// Додаємо цукор

beverage1 = new Sugar(beverage1);

Console.WriteLine(beverage1.GetDescription() + " $" + beverage1.Cost());

Console.WriteLine();

// Чай з молоком

Beverage beverage2 = new Tea();

beverage2 = new Milk(beverage2);

Console.WriteLine(beverage2.GetDescription() + " $" + beverage2.Cost());

}

}

}

Лістинг програми ProtocolStack-Project

using System;

namespace ProtocolStack\_Project

{

// Базовий клас протоколу

public abstract class Protocol

{

protected Protocol nextProtocol = null!;

public void SetNextProtocol(Protocol nextProtocol)

{

this.nextProtocol = nextProtocol;

}

public abstract string ProcessData(string data);

}

// Протокол для видалення зайвих пробілів

public class TrimProtocol : Protocol

{

public override string ProcessData(string data)

{

string trimmedData = data.Trim();

Console.WriteLine($"Trimmed data: {trimmedData}");

if (nextProtocol != null)

return nextProtocol.ProcessData(trimmedData);

return trimmedData;

}

}

// Протокол для перетворення всіх літер на великі

public class UppercaseProtocol : Protocol

{

public override string ProcessData(string data)

{

string upperCaseData = data.ToUpper();

Console.WriteLine($"Uppercased data: {upperCaseData}");

if (nextProtocol != null)

return nextProtocol.ProcessData(upperCaseData);

return upperCaseData;

}

}

// Протокол для додавання префікса

public class PrefixProtocol : Protocol

{

private readonly string \_prefix;

public PrefixProtocol(string prefix)

{

\_prefix = prefix;

}

public override string ProcessData(string data)

{

string prefixedData = \_prefix + data;

Console.WriteLine($"Prefixed data: {prefixedData}");

if (nextProtocol != null)

return nextProtocol.ProcessData(prefixedData);

return prefixedData;

}

}

// Стек протоколів

public class ProtocolStack

{

private readonly Protocol firstProtocol;

public ProtocolStack(Protocol firstProtocol)

{

this.firstProtocol = firstProtocol;

}

public void Process(string data)

{

firstProtocol.ProcessData(data);

}

}

// Основна програма

class Program

{

static void Main()

{

// Створюємо протоколи

Protocol uppercase = new UppercaseProtocol();

Protocol trim = new TrimProtocol();

Protocol prefix = new PrefixProtocol("HAVE ");

// Зв'язуємо їх у ланцюжок

uppercase.SetNextProtocol(trim);

trim.SetNextProtocol(prefix);

// Запускаємо стек

ProtocolStack protocolStack = new(uppercase);

string data = " a nice day! ";

Console.WriteLine("Original Data: " + data);

protocolStack.Process(data);

}

}

}

Лістинг програми Lock-Project

using System;

using System.Threading;

namespace Lock\_Project

{

public class BankAccount

{

private decimal balance;

private readonly object lockObject = new();

public BankAccount(decimal initialBalance)

{

balance = initialBalance;

}

// Метод для зняття грошей з рахунку

public void Withdraw(decimal amount)

{

lock (lockObject) // Забезпечуємо синхронізований доступ

{

if (balance >= amount)

{

Console.WriteLine($"Withdrawing {amount}...");

balance -= amount;

Console.WriteLine($"{amount} was withdrawn. Balance: {balance}");

}

else

{

Console.WriteLine($"Insufficient funds for withdrawal {amount}. Balance: {balance}");

}

}

}

// Метод для перевірки балансу

public decimal GetBalance()

{

lock (lockObject)

{

return balance;

}

}

}

class Program

{

static void Main()

{

// Створюємо банківський рахунок з початковим балансом

BankAccount account = new(1000);

// Створюємо кілька потоків, які намагаються зняти гроші

Thread thread1 = new(() => WithdrawMoney(account, 500));

Thread thread2 = new(() => WithdrawMoney(account, 700));

Thread thread3 = new(() => WithdrawMoney(account, 300));

// Запускаємо потоки

thread1.Start();

thread2.Start();

thread3.Start();

// Чекаємо, поки всі потоки завершаться

thread1.Join();

thread2.Join();

thread3.Join();

}

// Метод для зняття грошей в окремому потоці

static void WithdrawMoney(BankAccount account, decimal amount)

{

account.Withdraw(amount);

}

}

}

Лістинг додаткової програми RAII-Cpipi

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

class FileManager {

private:

ofstream file;

string filePath;

bool isOpen = false;

public:

// Constructor that opens the file

FileManager(const string& path)

: filePath(path)

{

file.open(filePath, ios::out | ios::app); // open for writing, append to end

if (file.is\_open()) {

isOpen = true;

cout << "File opened successfully." << endl;

}

else {

cout << "Failed to open the file." << endl;

}

}

// Method for writing text to the file

void WriteToFile(const string& text) {

if (isOpen) {

file << text << endl;

cout << "Text written to the file successfully." << endl;

}

}

// Destructor that automatically closes the file

~FileManager() {

if (isOpen) {

file.close();

cout << "File closed and resources cleaned up." << endl;

}

}

};

int main() {

string path = "test.txt";

{

FileManager fileManager(path);

fileManager.WriteToFile("Hello, world!");

// After leaving this block, the fileManager object will be destroyed, and the destructor will be called

}

cout << "Resources automatically released." << endl;

return 0;

}

ВИКОНАННЯ ВИМОГ НА РОБОТУ

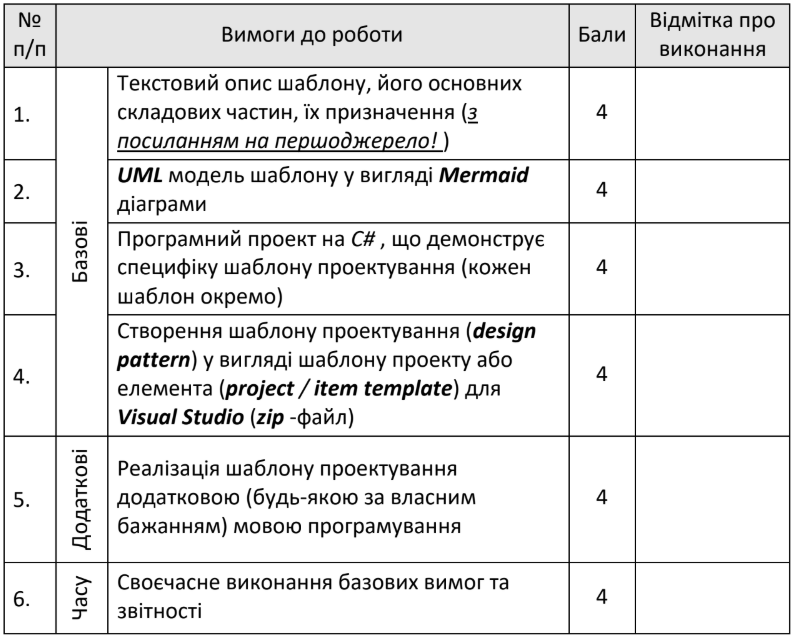


Рисунок 13 – Таблиця з позначкою виконання вимог на роботу