МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний аерокосмічний університет ім. М. Є. Жуковського

«Харківський авіаційний інститут»

факультет програмної інженерії та бізнесу

кафедра інженерії програмного забезпечення

**Розрахункова робота**

з дисципліни « Архітектура та проектування ПЗ »

*назва дисципліни*

на тему: ШАБЛОНИ ПРОЕКТУВАННЯ

Виконав: студент 2 курсу групи № 621п

напряму підготовки (спеціальності)

121 інженерія программного забезпечення

(шифр і назва напряму підготовки (спеціальності))

Гордеюк Р.О.

(прізвище й ініціали студента)

Прийняв: ст. викладач каф. 603

Сьомочкін М. О.

(посада, науковий ступінь, прізвище й ініціали)

Національна шкала:

Кількість балів:

Харків – 2024

**Розрахунково-графічна робота   
«Шаблони проектування»**

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

**Завдання**. Самостійно знайти в мережі Інтернет (відповідно до варіанта завдання в табл. 2) 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***[1], 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.

Для завдання підвищеного рівня складності (див. табл. 1) необхідний шаблон проектування (***design******pattern***) оформити у вигляді zip-файлу, який є шаблоном проекту або елемента ( ***Project / Item Template*** ) для середовища розробки ***Visual******Studio*** [2-5].

# **ВИКОНАННЯ РОБОТИ**

**Варіант 7**

|  |  |  |  |
| --- | --- | --- | --- |
| ***Creational pattern*** | ***Structural pattern*** | ***Behavioral pattern*** | ***Concurrency pattern*** |
| Abstract factory | Facade | State | Active object |

**Абстрактна фабрика (abstract factory)** — це породжуючий патерн проектування, який вирішує проблему створення цілих сімейств пов’язаних продуктів, без прив’язки коду до конкретних класів продуктів [6].

Абстрактна фабрика задає інтерфейс створення всіх доступних типів продуктів, а кожна конкретна реалізація фабрики породжує продукти однієї з варіацій. Клієнтський код викликає методи фабрики для отримання продуктів, замість самостійного створювання їх за допомогою оператора new. При цьому, фабрика сама стежить за тим, щоб створюваний продукт був потрібної варіації.

**UML-діаграми (рисунок 1 – 2):**

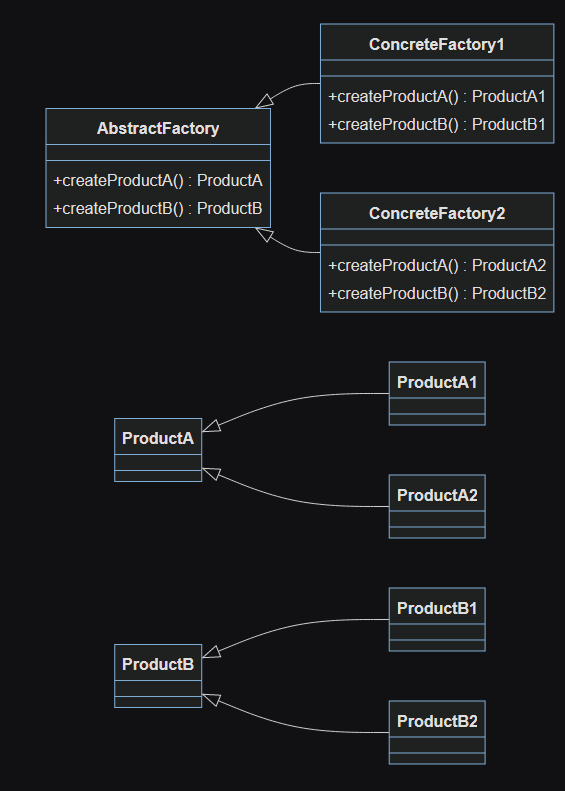


Рисунок 1. Static model

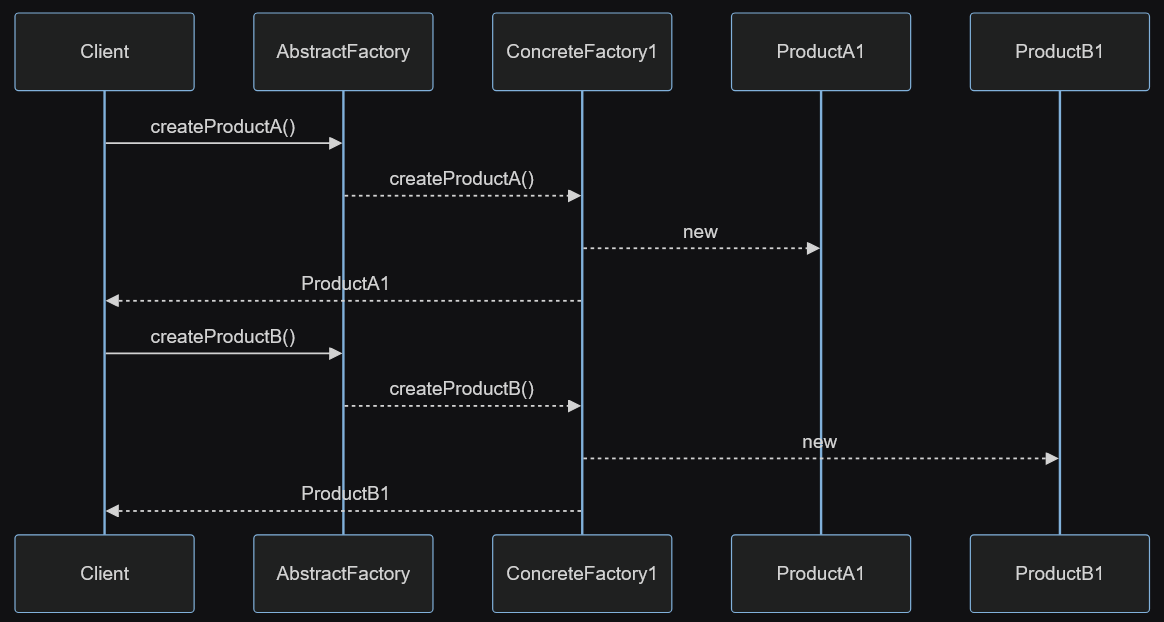


Рисунок 2. Dynamic model

**Основні структурні елементи:**

Абстрактна фабрика (IAbstractFactory): Інтерфейс, що оголошує методи для створення абстрактних продуктів.

Конкретні фабрики (ConcreteFactory1, ConcreteFactory2): Реалізують методи абстрактної фабрики для створення конкретних продуктів.

Абстрактні продукти (IAbstractProductA, IAbstractProductB): Інтерфейси для продуктів, які створюються фабриками.

Конкретні продукти (ConcreteProductA1, ConcreteProductA2, ConcreteProductB1, ConcreteProductB2): Реалізації абстрактних продуктів.

Клієнт (Client): Використовує інтерфейс абстрактної фабрики для створення об'єктів продуктів, не знаючи їх конкретних класів.

**Фасад (facade)** — це структурний патерн проектування, який надає простий інтерфейс до складної системи класів, бібліотеки або фреймворку [7].

Крім того, що Фасад дозволяє знизити загальну складність програми, він також допомагає винести код, який залежить від зовнішньої системи, в одне місце.

Патерн часто зустрічається в клієнтських додатках, написаних на C#, які використовують складні бібліотеки або API.

**UML-діаграми (рисунок 3 – 4):**

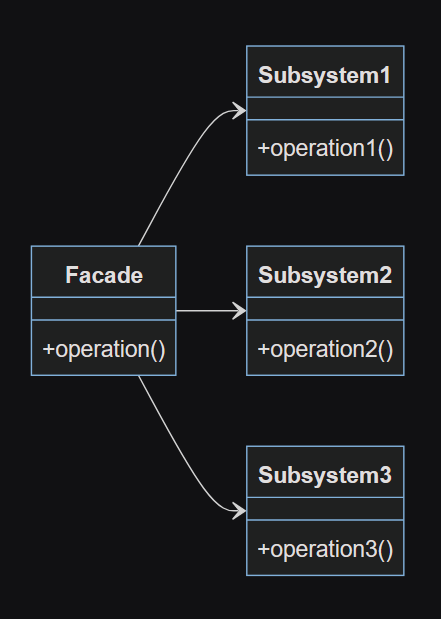


Рисунок 3. Static model

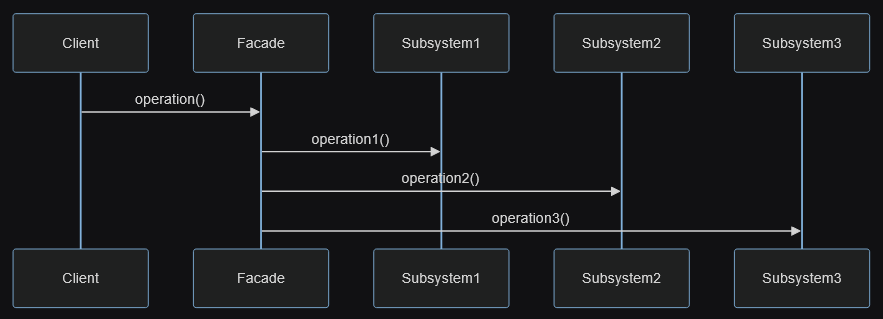


Рисунок 4. Dynamic model

**Основні структурні елементи:**

Підсистеми (Subsystem1, Subsystem2, Subsystem3): Це класи, що виконують окремі операції.

* Subsystem1: Містить метод Operation1(), який виконує "Підсистема 1 Операція 1".
* Subsystem2: Містить метод Operation2(), який виконує "Підсистема 2 Операція 2".
* Subsystem3: Містить метод Operation3(), який виконує "Підсистема 3 Операція 3".

Фасад (Facade): Клас, що спрощує інтерфейс для роботи з підсистемами.

* Містить приватні поля для кожної підсистеми: Subsystem1 subsystem1, Subsystem2 subsystem2, Subsystem3 subsystem3.
* Конструктор: Ініціалізує підсистеми.
* Метод Operation(): Викликає методи підсистем у певному порядку для виконання складної операції.

**Стан (state)** — це поведінковий патерн, що дозволяє динамічно змінювати поведінку об’єкта при зміні його стану [8].

Поведінки, які залежать від стану, переїзджають в окремі класи. Початковий клас зберігає посилання на один з таких об’єктів-станів та делегує йому роботу.

Патерн Стан часто використовують в C# для перетворення в об’єкти величезних стейт-машин, побудованих на операторах switch.

**UML-діаграма (рисунок 5 – 6):**

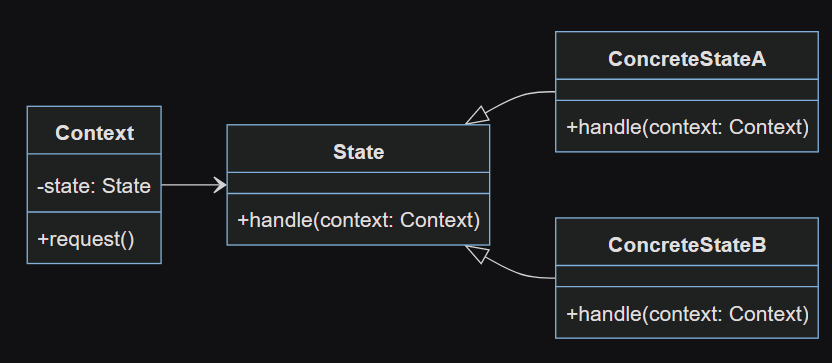


Рисунок 5. State model

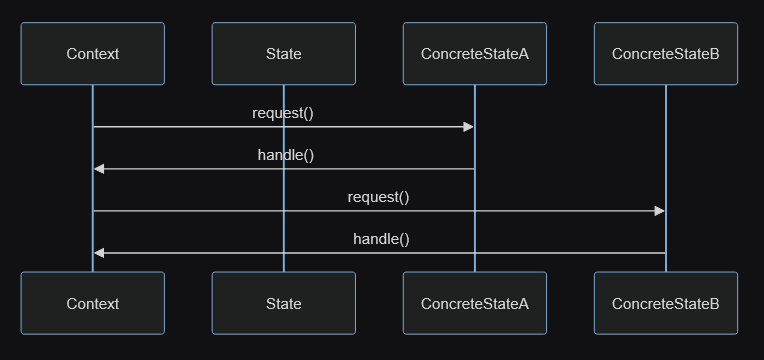


Рисунок 6. Dynamic model

**Основні структурні елементи:**

Контекст (Context):

* Зберігає посилання на об'єкт стану, що відображає поточний стан контексту.
* Викликає методи об'єкта стану для обробки запитів.

Стан (State):

* Абстрактний клас, що визначає інтерфейс для обробки запитів.
* Містить посилання на контекст для можливості зміни свого стану.
* Абстрактні методи Handle1() та Handle2(): Визначають дії, які виконуються в різних станах.

Конкретні стани (ConcreteStateA, ConcreteStateB):

* Реалізують поведінку, специфічну для певного стану контексту.

**Активний об’єкт (active object)** - це шаблон проектування, який дозволяє розділити виконання запитів від їх обробки. Основна ідея полягає в тому, що запити представляються як об’єкти, які можуть бути виконані асинхронно [9].

Active Object дозволяє відокремити виконання запитів від їх обробки. Це особливо корисно в асинхронних системах, де обробка запитів може займати час, і ви хочете, щоб ваша програма продовжувала працювати без блокування.

Він часто використовується в системах, де є багато асинхронних запитів, таких як мережеві додатки, графічні інтерфейси, аудіо/відео обробка тощо.

**UML-діаграми (рисунок 7 – 8):**

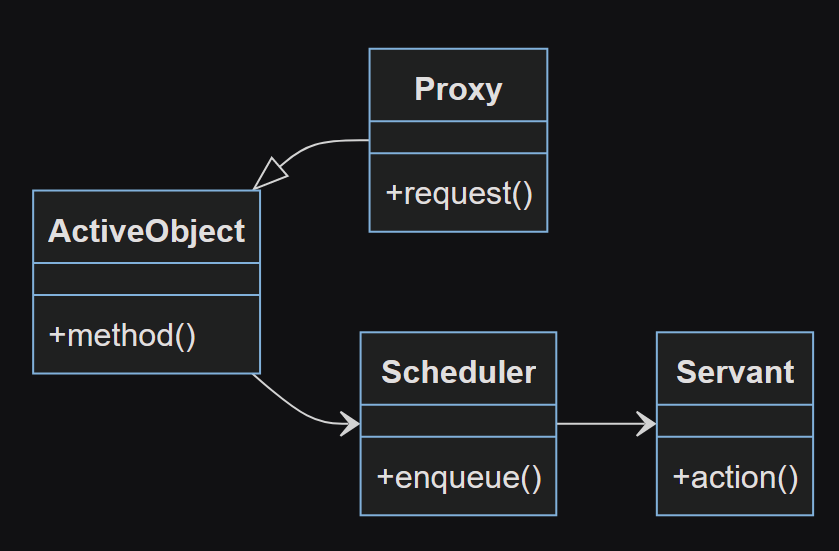
****

Рисунок 7. Static model

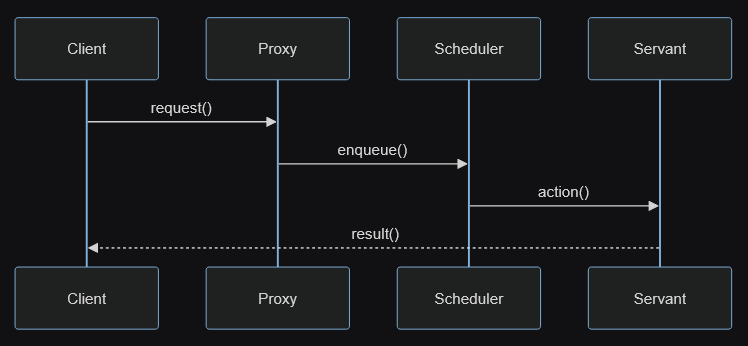
****

Рисунок 8. Dynamic model

**Основні структурні елементи:**

Слуга (Servant):

* Клас, що виконує основну дію.

Розкладальник (Scheduler):

* Клас, що відповідає за планування та виконання задач асинхронно.

Проксі (Proxy):

* Клас, що виступає посередником між клієнтом та слугою, передаючи запити розкладальнику.

# **ЛІСТИНГ ПРОГРАМ**

## Шаблон «Абстрактна фабрика»:

***Мовою С#:***

namespace AbstractFactory

{

public interface IAbstractFactory

{

IAbstractProductA CreateProductA();

IAbstractProductB CreateProductB();

}

class ConcreteFactory1 : IAbstractFactory

{

public IAbstractProductA CreateProductA()

{

return new ConcreteProductA1();

}

public IAbstractProductB CreateProductB()

{

return new ConcreteProductB1();

}

}

class ConcreteFactory2 : IAbstractFactory

{

public IAbstractProductA CreateProductA()

{

return new ConcreteProductA2();

}

public IAbstractProductB CreateProductB()

{

return new ConcreteProductB2();

}

}

public interface IAbstractProductA

{

string UsefulFunctionA();

}

class ConcreteProductA1 : IAbstractProductA

{

public string UsefulFunctionA()

{

return "Результат вiд продукту A1.";

}

}

class ConcreteProductA2 : IAbstractProductA

{

public string UsefulFunctionA()

{

return "Результат вiд продукту A2.";

}

}

public interface IAbstractProductB

{

string UsefulFunctionB();

string AnotherUsefulFunctionB(IAbstractProductA collaborator);

}

class ConcreteProductB1 : IAbstractProductB

{

public string UsefulFunctionB()

{

return "Результат вiд продукту B1.";

}

public string AnotherUsefulFunctionB(IAbstractProductA collaborator)

{

var result = collaborator.UsefulFunctionA();

return $"Результат вiд B1, спiвпрацюючи з ({result})";

}

}

class ConcreteProductB2 : IAbstractProductB

{

public string UsefulFunctionB()

{

return "Результат вiд продукту B2.";

}

public string AnotherUsefulFunctionB(IAbstractProductA collaborator)

{

var result = collaborator.UsefulFunctionA();

return $"Результат вiд B2, спiвпрацюючи з ({result})";

}

}

class Client

{

public void Main()

{

Console.WriteLine("Клiєнт: Тестуємо клiєнтський код з першим типом фабрики.");

ClientMethod(new ConcreteFactory1());

Console.WriteLine();

Console.WriteLine("Клiєнт: Тестуємо той самий клiєнтський код з другим типом фабрики.");

ClientMethod(new ConcreteFactory2());

}

public void ClientMethod(IAbstractFactory factory)

{

var productA = factory.CreateProductA();

var productB = factory.CreateProductB();

Console.WriteLine(productB.UsefulFunctionB());

Console.WriteLine(productB.AnotherUsefulFunctionB(productA));

}

}

class AbstractFactory

{

static void Main(string[] args)

{

new Client().Main();

}

}

}

***Мовою Python:***

from abc import ABC, abstractmethod

class IAbstractFactory(ABC):

@abstractmethod

def create\_product\_a(self):

pass

@abstractmethod

def create\_product\_b(self):

pass

class ConcreteFactory1(IAbstractFactory):

def create\_product\_a(self):

return ConcreteProductA1()

def create\_product\_b(self):

return ConcreteProductB1()

class ConcreteFactory2(IAbstractFactory):

def create\_product\_a(self):

return ConcreteProductA2()

def create\_product\_b(self):

return ConcreteProductB2()

class IAbstractProductA(ABC):

@abstractmethod

def useful\_function\_a(self):

pass

class ConcreteProductA1(IAbstractProductA):

def useful\_function\_a(self):

return "Результат від продукту A1."

class ConcreteProductA2(IAbstractProductA):

def useful\_function\_a(self):

return "Результат від продукту A2."

class IAbstractProductB(ABC):

@abstractmethod

def useful\_function\_b(self):

pass

@abstractmethod

def another\_useful\_function\_b(self, collaborator):

pass

class ConcreteProductB1(IAbstractProductB):

def useful\_function\_b(self):

return "Результат від продукту B1."

def another\_useful\_function\_b(self, collaborator):

result = collaborator.useful\_function\_a()

return f"Результат від B1, співпрацюючи з ({result})"

class ConcreteProductB2(IAbstractProductB):

def useful\_function\_b(self):

return "Результат від продукту B2."

def another\_useful\_function\_b(self, collaborator):

result = collaborator.useful\_function\_a()

return f"Результат від B2, співпрацюючи з ({result})"

class Client:

def main(self):

print("Клієнт: Тестуємо клієнтський код з першим типом фабрики")

self.client\_method(ConcreteFactory1())

print()

print("Клієнт: Тестуємо той самий клієнтський код з другим типом фабрики")

self.client\_method(ConcreteFactory2())

def client\_method(self, factory):

product\_a = factory.create\_product\_a()

product\_b = factory.create\_product\_b()

print(product\_b.useful\_function\_b())

print(product\_b.another\_useful\_function\_b(product\_a))

if \_\_name\_\_ == "\_\_main\_\_":

client = Client()

client.main()

## Шаблон Facade:

***Мовою C#:***

namespace FacadePattern

{

class Subsystem1

{

public void Operation1()

{

Console.WriteLine("Пiдсистема 1 Операцiя 1");

}

}

class Subsystem2

{

public void Operation2()

{

Console.WriteLine("Пiдсистема 2 Операцiя 2");

}

}

class Subsystem3

{

public void Operation3()

{

Console.WriteLine("Пiдсистема 3 Операцiя 3");

}

}

class Facade

{

private Subsystem1 subsystem1;

private Subsystem2 subsystem2;

private Subsystem3 subsystem3;

public Facade()

{

subsystem1 = new Subsystem1();

subsystem2 = new Subsystem2();

subsystem3 = new Subsystem3();

}

public void Operation()

{

subsystem1.Operation1();

subsystem2.Operation2();

subsystem3.Operation3();

}

}

class Program

{

static void Main(string[] args)

{

Facade facade = new Facade();

facade.Operation();

}

}

}

***Мовою Python:***

class Subsystem1:

def operation1(self):

print("Пiдсистема 1 Операцiя 1")

class Subsystem2:

def operation2(self):

print("Пiдсистема 2 Операцiя 2")

class Subsystem3:

def operation3(self):

print("Пiдсистема 3 Операцiя 3")

class Facade:

def \_\_init\_\_(self):

self.subsystem1 = Subsystem1()

self.subsystem2 = Subsystem2()

self.subsystem3 = Subsystem3()

def operation(self):

self.subsystem1.operation1()

self.subsystem2.operation2()

self.subsystem3.operation3()

if \_\_name\_\_ == "\_\_main\_\_":

facade = Facade()

facade.operation()

## Шаблон State:

***Мовою C#:***

namespace State

{

class Context

{

private State \_state = null;

public Context(State state)

{

this.TransitionTo(state);

}

public void TransitionTo(State state)

{

Console.WriteLine($"Контекст: Перехiд до {state.GetType().Name}.");

this.\_state = state;

this.\_state.SetContext(this);

}

public void Request1()

{

this.\_state.Handle1();

}

public void Request2()

{

this.\_state.Handle2();

}

}

abstract class State

{

protected Context \_context;

public void SetContext(Context context)

{

this.\_context = context;

}

public abstract void Handle1();

public abstract void Handle2();

}

class ConcreteStateA : State

{

public override void Handle1()

{

Console.WriteLine("ConcreteStateA обробляє запит1.");

Console.WriteLine("ConcreteStateA хоче змiнити стан контексту.");

this.\_context.TransitionTo(new ConcreteStateB());

}

public override void Handle2()

{

Console.WriteLine("ConcreteStateA обробляє запит2.");

}

}

class ConcreteStateB : State

{

public override void Handle1()

{

Console.WriteLine("ConcreteStateB обробляє запит1.");

}

public override void Handle2()

{

Console.WriteLine("ConcreteStateB обробляє запит2.");

Console.WriteLine("ConcreteStateB хоче змiнити стан контексту.");

this.\_context.TransitionTo(new ConcreteStateA());

}

}

class Program

{

static void Main(string[] args)

{

var context = new Context(new ConcreteStateA());

context.Request1();

context.Request2();

}

}

}

***Мовою Python:***

class Context:

def \_\_init\_\_(self, state):

self.transition\_to(state)

def transition\_to(self, state):

print(f"Контекст: Перехід до {state.\_\_class\_\_.\_\_name\_\_}.")

self.\_state = state

self.\_state.set\_context(self)

def request1(self):

self.\_state.handle1()

def request2(self):

self.\_state.handle2()

class State:

def set\_context(self, context):

self.\_context = context

def handle1(self):

raise NotImplementedError()

def handle2(self):

raise NotImplementedError()

class ConcreteStateA(State):

def handle1(self):

print("ConcreteStateA обробляє запит1.")

print("ConcreteStateA хоче змінити стан контексту.")

self.\_context.transition\_to(ConcreteStateB())

def handle2(self):

print("ConcreteStateA обробляє запит2.")

class ConcreteStateB(State):

def handle1(self):

print("ConcreteStateB обробляє запит1.")

def handle2(self):

print("ConcreteStateB обробляє запит2.")

print("ConcreteStateB хоче змінити стан контексту.")

self.\_context.transition\_to(ConcreteStateA())

if \_\_name\_\_ == "\_\_main\_\_":

context = Context(ConcreteStateA())

context.request1()

context.request2()

## Шаблон Active object:

***Мовою С#:***

using System;

using System.Threading.Tasks;

namespace ActiveObject

{

class Servant

{

public void Action()

{

Console.WriteLine("Слуга виконує дiю");

}

}

class Scheduler

{

public void Enqueue(Action action)

{

Task.Run(action);

}

}

class Proxy

{

private Scheduler scheduler;

private Servant servant;

public Proxy()

{

scheduler = new Scheduler();

servant = new Servant();

}

public void Request()

{

scheduler.Enqueue(servant.Action);

}

}

class Program

{

static void Main(string[] args)

{

Proxy proxy = new Proxy();

proxy.Request();

Console.WriteLine("Клiєнт продовжує виконання");

}

}

}

***Мовою Python:***

import asyncio

class Servant:

async def action(self):

print("Слуга виконує дiю")

class Scheduler:

async def enqueue(self, action):

await action()

class Proxy:

def \_\_init\_\_(self):

self.scheduler = Scheduler()

self.servant = Servant()

async def request(self):

await self.scheduler.enqueue(self.servant.action)

async def main():

proxy = Proxy()

await proxy.request()

print("Клiєнт продовжує виконання")

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(main())

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Таблиця з позначкою вимог до роботи (табл. 1):  Таблиця 1 | | | | |
| № | Складність | Вимоги до роботи | % | Оцінка |
|  | Базовий рівень | Текстовий опис шаблону, його основних складових частин, їх призначення (*з посиланням на першоджерело!* ) | 5 |  |
|  | ***UML*** модель шаблону у вигляді ***Mermaid*** діаграми | 5 |  |
|  | Програмний проект на *С#* , що ­демонструє ­специфіку шаблону проектування (кожен шаблон окремо) | 5 |  |
|  | Реалізація шаблону проектування додатковою (будь-якою) мовою програмування | 5 |  |
|  | Підвищений ­рівень | Створення шаблону проектування (***design******pattern***) у вигляді шаблону проекту або елемента (***project*** */* ***item******template***) для ***Visual******Studio*** (***zip*** -файл) | 5 |  |

# **ВИСНОВОК**

Під час виконання розрахункової роботи було вивчено стандартні ситуацій у процесі розробки складних програмних проектів та застосування шаблонів проектування (Design patterns) для їх вирішення. У нашому випадку булі розглянуті таки паттерни як Абстрактна фабрика, Фасад, Стан та Активний об’єкт. Був проведений аналіз заданих паттернів у вигляді створення UML-діаграм, програмних прикладів, шаблонів тощо. Оформлено звіт.

# **ДЖЕРЕЛА**

1. Mermaid Diagramming and charting tool [Електронний ресурс] – Режим доступу: [https://mermaid.js.org/intro/](https://mermaid.js.org/intro/%20) .
2. Практичний посібник. Створення шаблонів проектів [Електронний ресурс] – Режим доступу: [https://docs.microsoft.com/ru-ru/visualstudio/ide/how-to-create-project-templates?view=vs-2017](https://docs.microsoft.com/ru-ru/visualstudio/ide/how-to-create-project-templates?view=vs-2017%20)
3. How to create custom Visual Studio file templates. [Електронний ресурс] – Режим доступу: <https://youtube.com/watch?v=AWf8NNWJ5BI>
4. Create custom project templates in Visual Studio [Електронний ресурс] – Режим доступу: <https://www.ecanarys.com/Blogs/ArticleID/180/Create-custom-project-templates-in-Visual-Studio>
5. Custom Item Templates in Visual Studio [Електронний ресурс] – Режим доступу: [https://www.youtube.com/watch?v=3uYN3mDFP-o](https://www.youtube.com/watch?v=3uYN3mDFP-o%20)
6. Абстрактна фабрика [Електронний ресурс] – Режим доступу до ресурсу: <https://refactoring.guru/uk/design-patterns/abstract-factory.>
7. Фасад [Електронний ресурс] – Режим доступу до ресурсу: <https://refactoring.guru/uk/design-patterns/facade>.
8. Стан [Електронний ресурс] – Режим доступу до ресурсу: <https://refactoring.guru/uk/design-patterns/state>.
9. Шаблони проєктування: їх види, особливості й переваги [Електронний ресурс] – Режим доступу до ресурсу: <https://highload.today/uk/shabloni-proyektuvannya-yih-vidi-osoblivosti-j-perevagi/>.