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 проектування ПЗ .Net »

*назва дисципліни*

на тему: «ШАБЛОНИ ПРОЕКТУВАННЯ»

Виконав: студент 2 курсу групи № 621п

освітньої програми

121 інженерія програмного забезпечення

(шифр і назва ОП)

Чорний Р.С.

(прізвище й ініціали студента)

Прийняв: асистент каф. 603

Сьомочкін М.О.

(посада, науковий ступінь, прізвище й ініціали)

Кількість балів:

Харків – 2024

**ПОСТАНОВКА ЗАВДАННЯ**

Вивчити стандартні ситуації 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 ЗАВДАННЯ**

Варіант 621п 23

Завдання за варіантом:

1. Abstract Factory (Creational Pattern)

Опис:

Шаблон Abstract Factory дозволяє створювати сімейства взаємозалежних або пов'язаних об'єктів без зазначення їх конкретних класів. Він забезпечує інтерфейс для створення об'єктів, але дозволяє підкласам змінювати тип створюваних об'єктів.

Основні складові частини:

- AbstractFactory: оголошує інтерфейс для створення абстрактних продуктів.

- ConcreteFactory: реалізує операції створення конкретних продуктів.

- AbstractProduct: оголошує інтерфейс для типу продукту.

- Product: визначає конкретні реалізації абстрактних продуктів.

- Client: використовує тільки інтерфейси, оголошені в AbstractFactory та AbstractProduct.

UML діаграма статичної моделі (Рис. 1)

Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

Автоматически созданное описание

Рисунок 1 – Створена діаграма

UML діаграма динамічної моделі (Рис. 2)

Изображение выглядит как снимок экрана, текст, линия, диаграмма

Автоматически созданное описание

Рисунок 2 – Створена діаграма

Facade (Structural Pattern)

Шаблон Facade надає уніфікований інтерфейс до набору інтерфейсів у підсистемі. Facade визначає високорівневий інтерфейс, який полегшує використання підсистеми.

Основні складові частини:

- Facade: надає спрощений інтерфейс для клієнта.

- Subsystem Classes: реалізують функціональність підсистеми і не мають знання про Facade.

UML діаграма статичної моделі (Рис 3)

Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описание

Рисунок 3 – Створена діаграма

UML діаграма динамічної моделі (Рис 4)

Изображение выглядит как текст, снимок экрана, диаграмма

Автоматически созданное описание

Рисунок 4 – Створена діаграма

Visitor (Behavioral Pattern)

Шаблон Visitor дозволяє додавати нові операції до об'єктів без зміни їх класів. Він відокремлює алгоритм від структури даних, над якими цей алгоритм працює.

Основні складові частини:

- Visitor: оголошує інтерфейс для візитів.

- ConcreteVisitor: реалізує конкретні операції.

- Element: приймає візитора.

- ConcreteElement: реалізує прийняття візитора.

UML діаграма статичної моделі (Рис 5)

Изображение выглядит как снимок экрана, текст, Шрифт, линия

Автоматически созданное описание

Рисунок 5 – Створена діаграма

UML діаграма динамічної моделі (Рис 6)

Изображение выглядит как снимок экрана, текст, диаграмма

Автоматически созданное описание

Рисунок 6 – Створена діаграма

Balking (Concurrency Pattern)

Шаблон Balking використовується для запобігання виконанню певної дії, якщо об'єкт знаходиться у невідповідному стані.

Основні складові частини:

- Client: ініціює запит.

- BalkingObject: об'єкт, який відмовляється від дії, якщо він знаходиться у невідповідному стані.

- State: визначає стани об'єкта.

UML діаграма статичної моделі (Рис 7)

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рисунок 7 – Створена діаграма

UML діаграма динамічної моделі (Рис 8)

Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

Автоматически созданное описание

Рисунок 8 – Створена діаграма

Шаблони проектування

Abstract Factory

using System;

// Abstract Product A

interface IAbstractProductA {

string UsefulFunctionA();

}

// Concrete Product A1

class ConcreteProductA1 : IAbstractProductA {

public string UsefulFunctionA() {

return "The result of the product A1.";

}

}

// Concrete Product A2

class ConcreteProductA2 : IAbstractProductA {

public string UsefulFunctionA() {

return "The result of the product A2.";

}

}

// Abstract Product B

interface IAbstractProductB {

string UsefulFunctionB();

}

// Concrete Product B1

class ConcreteProductB1 : IAbstractProductB {

public string UsefulFunctionB() {

return "The result of the product B1.";

}

}

// Concrete Product B2

class ConcreteProductB2 : IAbstractProductB {

public string UsefulFunctionB() {

return "The result of the product B2.";

}

}

// Abstract Factory

interface IAbstractFactory {

IAbstractProductA CreateProductA();

IAbstractProductB CreateProductB();

}

// Concrete Factory 1

class ConcreteFactory1 : IAbstractFactory {

public IAbstractProductA CreateProductA() {

return new ConcreteProductA1();

}

public IAbstractProductB CreateProductB() {

return new ConcreteProductB1();

}

}

// Concrete Factory 2

class ConcreteFactory2 : IAbstractFactory {

public IAbstractProductA CreateProductA() {

return new ConcreteProductA2();

}

public IAbstractProductB CreateProductB() {

return new ConcreteProductB2();

}

}

// Client Code

class Client {

public void Main() {

Console.WriteLine("Client: Testing client code with the first factory type...");

ClientMethod(new ConcreteFactory1());

Console.WriteLine();

Console.WriteLine("Client: Testing the same client code with the second factory type...");

ClientMethod(new ConcreteFactory2());

}

public void ClientMethod(IAbstractFactory factory) {

var productA = factory.CreateProductA();

var productB = factory.CreateProductB();

Console.WriteLine(productB.UsefulFunctionB());

Console.WriteLine(productA.UsefulFunctionA());

}

}

// Run the program

class Program {

static void Main(string[] args) {

new Client().Main();

}

}

Facade

using System;

// Subsystem classes

class SubsystemA {

public string OperationA() {

return "Subsystem A, Method A\n";

}

}

class SubsystemB {

public string OperationB() {

return "Subsystem B, Method B\n";

}

}

class SubsystemC {

public string OperationC() {

return "Subsystem C, Method C\n";

}

}

// Facade

class Facade {

protected SubsystemA \_subsystemA;

protected SubsystemB \_subsystemB;

protected SubsystemC \_subsystemC;

public Facade(SubsystemA subsystemA, SubsystemB subsystemB, SubsystemC subsystemC) {

this.\_subsystemA = subsystemA;

this.\_subsystemB = subsystemB;

this.\_subsystemC = subsystemC;

}

public string Operation() {

string result = "Facade initializes subsystems:\n";

result += this.\_subsystemA.OperationA();

result += this.\_subsystemB.OperationB();

result += this.\_subsystemC.OperationC();

result += "Facade orders subsystems to perform the action:\n";

result += this.\_subsystemA.OperationA();

result += this.\_subsystemB.OperationB();

result += this.\_subsystemC.OperationC();

return result;

}

}

// Client Code

class Client {

public static void ClientCode(Facade facade) {

Console.Write(facade.Operation());

}

}

// Run the program

class Program {

static void Main(string[] args) {

SubsystemA subsystemA = new SubsystemA();

SubsystemB subsystemB = new SubsystemB();

SubsystemC subsystemC = new SubsystemC();

Facade facade = new Facade(subsystemA, subsystemB, subsystemC);

Client.ClientCode(facade);

}

}

Visitor

using System;

using System.Collections.Generic;

// Visitor Interface

interface IVisitor {

void VisitConcreteElementA(ConcreteElementA element);

void VisitConcreteElementB(ConcreteElementB element);

}

// Concrete Visitor

class ConcreteVisitor1 : IVisitor {

public void VisitConcreteElementA(ConcreteElementA element) {

Console.WriteLine(element.ExclusiveMethodOfConcreteElementA() + " + ConcreteVisitor1");

}

public void VisitConcreteElementB(ConcreteElementB element) {

Console.WriteLine(element.SpecialMethodOfConcreteElementB() + " + ConcreteVisitor1");

}

}

// Element Interface

interface IElement {

void Accept(IVisitor visitor);

}

// Concrete Elements

class ConcreteElementA : IElement {

public void Accept(IVisitor visitor) {

visitor.VisitConcreteElementA(this);

}

public string ExclusiveMethodOfConcreteElementA() {

return "A";

}

}

class ConcreteElementB : IElement {

public void Accept(IVisitor visitor) {

visitor.VisitConcreteElementB(this);

}

public string SpecialMethodOfConcreteElementB() {

return "B";

}

}

// Object Structure

class ObjectStructure {

private List<IElement> \_elements = new List<IElement>();

public void Attach(IElement element) {

\_elements.Add(element);

}

public void Detach(IElement element) {

\_elements.Remove(element);

}

public void Accept(IVisitor visitor) {

foreach (IElement element in \_elements) {

element.Accept(visitor);

}

}

}

// Client Code

class Program {

static void Main(string[] args) {

ObjectStructure structure = new ObjectStructure();

structure.Attach(new ConcreteElementA());

structure.Attach(new ConcreteElementB());

ConcreteVisitor1 visitor1 = new ConcreteVisitor1();

structure.Accept(visitor1);

}

}

Balking

using System;

using System.Threading;

using System.Threading.Tasks;

class BalkingObject {

private bool \_isInitialized = false;

private readonly object \_lock = new object();

public void Initialize() {

lock (\_lock) {

if (\_isInitialized) {

Console.WriteLine("Initialization already in progress. Balking.");

return;

}

\_isInitialized = true;

Console.WriteLine("Initialization started.");

// Simulating some work with a delay

Task.Delay(2000).ContinueWith(t => {

lock (\_lock) {

\_isInitialized = false;

Console.WriteLine("Initialization completed.");

}

});

}

}

}

// Client Code

class Program {

static void Main(string[] args) {

BalkingObject obj = new BalkingObject();

Parallel.Invoke(

() => obj.Initialize(),

() => obj.Initialize(),

() => obj.Initialize()

);

Console.ReadLine();

}

}

Реалізація шаблонів додатковою мовою

Abstract Factory (Python)

from abc import ABC, abstractmethod

# Abstract Product A

class AbstractProductA(ABC):

@abstractmethod

def useful\_function\_a(self) -> str:

pass

# Concrete Product A1

class ConcreteProductA1(AbstractProductA):

def useful\_function\_a(self) -> str:

return "The result of the product A1."

# Concrete Product A2

class ConcreteProductA2(AbstractProductA):

def useful\_function\_a(self) -> str:

return "The result of the product A2."

# Abstract Product B

class AbstractProductB(ABC):

@abstractmethod

def useful\_function\_b(self) -> str:

pass

# Concrete Product B1

class ConcreteProductB1(AbstractProductB):

def useful\_function\_b(self) -> str:

return "The result of the product B1."

# Concrete Product B2

class ConcreteProductB2(AbstractProductB):

def useful\_function\_b(self) -> str:

return "The result of the product B2."

# Abstract Factory

class AbstractFactory(ABC):

@abstractmethod

def create\_product\_a(self) -> AbstractProductA:

pass

@abstractmethod

def create\_product\_b(self) -> AbstractProductB:

pass

# Concrete Factory 1

class ConcreteFactory1(AbstractFactory):

def create\_product\_a(self) -> AbstractProductA:

return ConcreteProductA1()

def create\_product\_b(self) -> AbstractProductB:

return ConcreteProductB1()

# Concrete Factory 2

class ConcreteFactory2(AbstractFactory):

def create\_product\_a(self) -> AbstractProductA:

return ConcreteProductA2()

def create\_product\_b(self) -> AbstractProductB:

return ConcreteProductB2()

# Client Code

def client\_code(factory: AbstractFactory) -> None:

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

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

print(f"{product\_b.useful\_function\_b()}")

print(f"{product\_a.useful\_function\_a()}")

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

print("Client: Testing client code with the first factory type:")

client\_code(ConcreteFactory1())

print("\nClient: Testing the same client code with the second factory type:")

client\_code(ConcreteFactory2())

Facade (Python)

class SubsystemA:

def operation\_a(self) -> str:

return "Subsystem A, Method A\n"

class SubsystemB:

def operation\_b(self) -> str:

return "Subsystem B, Method B\n"

class SubsystemC:

def operation\_c(self) -> str:

return "Subsystem C, Method C\n"

class Facade:

def \_\_init\_\_(self, subsystem\_a: SubsystemA, subsystem\_b: SubsystemB, subsystem\_c: SubsystemC) -> None:

self.\_subsystem\_a = subsystem\_a

self.\_subsystem\_b = subsystem\_b

self.\_subsystem\_c = subsystem\_c

def operation(self) -> str:

result = "Facade initializes subsystems:\n"

result += self.\_subsystem\_a.operation\_a()

result += self.\_subsystem\_b.operation\_b()

result += self.\_subsystem\_c.operation\_c()

result += "Facade orders subsystems to perform the action:\n"

result += self.\_subsystem\_a.operation\_a()

result += self.\_subsystem\_b.operation\_b()

result += self.\_subsystem\_c.operation\_c()

return result

# Client Code

def client\_code(facade: Facade) -> None:

print(facade.operation())

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

subsystem\_a = SubsystemA()

subsystem\_b = SubsystemB()

subsystem\_c = SubsystemC()

facade = Facade(subsystem\_a, subsystem\_b, subsystem\_c)

client\_code(facade)

```

### Visitor (Python)

```python

# Visitor Pattern in Python

from abc import ABC, abstractmethod

class Visitor(ABC):

@abstractmethod

def visit\_concrete\_element\_a(self, element) -> None:

pass

@abstractmethod

def visit\_concrete\_element\_b(self, element) -> None:

pass

class ConcreteVisitor1(Visitor):

def visit\_concrete\_element\_a(self, element) -> None:

print(f"{element.exclusive\_method\_of\_concrete\_element\_a()} + ConcreteVisitor1")

def visit\_concrete\_element\_b(self, element) -> None:

print(f"{element.special\_method\_of\_concrete\_element\_b()} + ConcreteVisitor1")

class Element(ABC):

@abstractmethod

def accept(self, visitor: Visitor) -> None:

pass

class ConcreteElementA(Element):

def accept(self, visitor: Visitor) -> None:

visitor.visit\_concrete\_element\_a(self)

def exclusive\_method\_of\_concrete\_element\_a(self) -> str:

return "A"

class ConcreteElementB(Element):

def accept(self, visitor: Visitor) -> None:

visitor.visit\_concrete\_element\_b(self)

def special\_method\_of\_concrete\_element\_b(self) -> str:

return "B"

class ObjectStructure:

def \_\_init\_\_(self) -> None:

self.\_elements = []

def attach(self, element: Element) -> None:

self.\_elements.append(element)

def detach(self, element: Element) -> None:

self.\_elements.remove(element)

def accept(self, visitor: Visitor) -> None:

for element in self.\_elements:

element.accept(visitor)

# Client Code

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

structure = ObjectStructure()

structure.attach(ConcreteElementA())

structure.attach(ConcreteElementB())

visitor1 = ConcreteVisitor1()

structure.accept(visitor1)

Balking (Python)

import threading

import time

class BalkingObject:

def \_\_init\_\_(self) -> None:

self.\_is\_initialized = False

self.\_lock = threading.Lock()

def initialize(self) -> None:

with self.\_lock:

if self.\_is\_initialized:

print("Initialization already in progress. Balking.")

return

self.\_is\_initialized = True

print("Initialization started.")

# Simulating some work with a delay

time.sleep(2)

with self.\_lock:

self.\_is\_initialized = False

print("Initialization completed.")

# Client Code

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

obj = BalkingObject()

threads = []

for i in range(3):

thread = threading.Thread(target=obj.initialize)

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

Створюємо шаблони проектування (zip файл) (Рис 9)

Изображение выглядит как текст, программное обеспечение, Мультимедийное программное обеспечение, снимок экрана

Автоматически созданное описание

Рисунок 9 – Створені шаблони

**ВИСНОВОК**

Під час виконання лабораторної роботи було розглянуто стандартні ситуації у процесі розробки складних програмних проектів та застосування шаблонів проектування (Design patterns) для їх вирішення. Розроблено текстовий опис шаблонів, їх графічне подання у вигляді UML діаграм та розроблено програмний проект для кожного шаблону.