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 № 1**

з дисципліни « Архітектура програмного забезпечення .Net »

*назва дисципліни*

на тему: «ШАБЛОНИ ПРОЕКТУВАННЯ»

Виконав: студент 2 курсу групи № 622п

освітньої програми

121 інженерія програмного забезпечення

(шифр і назва ОП)

Горбунов Іван Сергійович

(прізвище й ініціали студента)

Прийняв: к. т. н., доцент

Лучшев Павло Олександрович

(посада, науковий ступінь, прізвище й ініціали)

Кількість балів:

Харків – 2023

**ЗМІСТ**

[Постановка завдання 3](#_Toc159492133)

[Порядок виконання роботи 3](#_Toc159492134)

[Виконання роботи 5](#_Toc159492135)

[Висновок 11](#_Toc159492136)

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

Таблиця 1 – виконання вимог до роботи

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

Постановка завдання

Самостійно знайти в мережі Інтернет (відповідно до варіанта) 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.

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Група . № | Creational  pattern | Structural pattern | Behavioral pattern | Concurrency  pattern |
| 622п.07 | Message Pattern | Servant Pattern | Multiton | Adapter |

Склад розрахунково-графічної роботи:

1. ***GitHub*** репозиторій:

* файл ***ReadMe.md*** з ***UML*** моделями та текстовим описом;
* програмні проекти із демонстрацією шаблонів;
* проєкт (Project / Item Template) для завдання підвищеної складності.

1. Звіт, що здається без архівації, наприклад: ***РГР 621 П'яточкінПП.docx***

* Титульна сторінка.
* Варіант та постановка задачі.
* Теоретичні відомості (з файлу ReadMe.md, окремо по кожному шаблону) з посиланням на джерело нформації:
* призначення та застосування шаблону проектування.
* опис шаблону проектування ( UML -діаграма).
* опис основних структурних елементів.
* Лістинг програми (кожний шаблон).
* Таблиця з позначкою виконання вимог на роботу (табл.1).

Виконання роботи

Посилання на *GitHub* репозиторій:

<https://github.com/HorbunovIvan/RGR>

Вміст файлу *ReadMe.md* з *UML* моделями та текстовим описом

**1.** **Message Pattern**

Шаблон Message Pattern використовується для організації обміну повідомленнями між різними компонентами системи. Це дозволяє компонентам взаємодіяти асинхронно, що підвищує гнучкість і масштабованість системи.

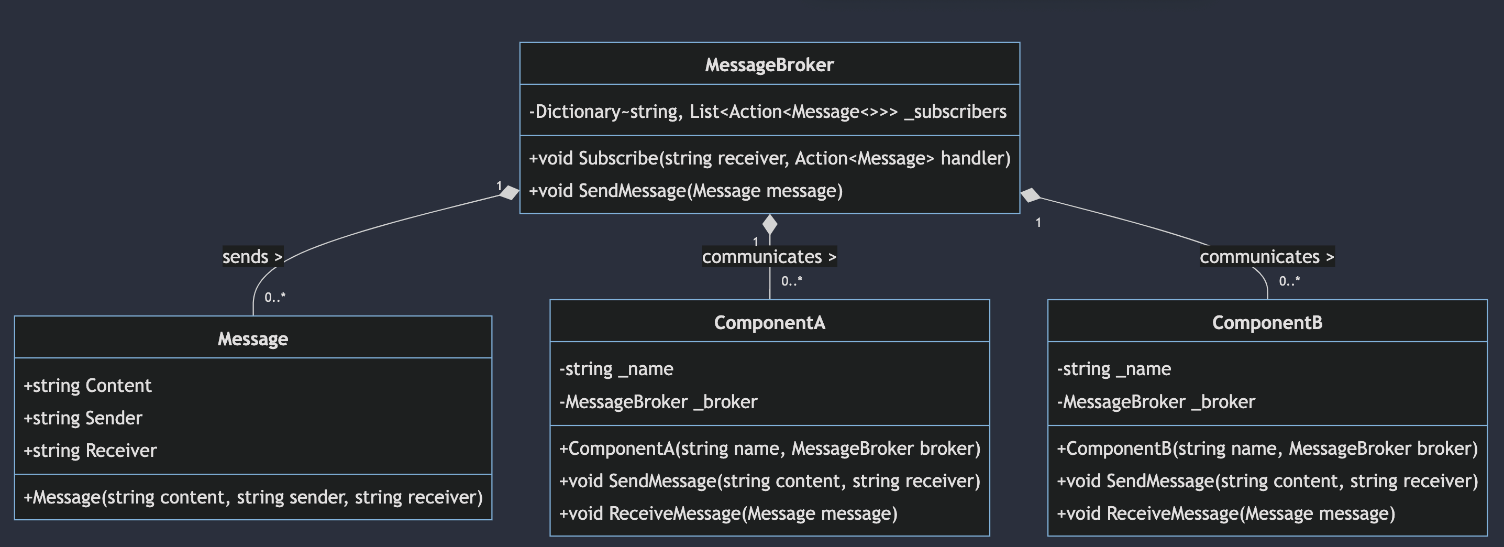


Рисунок 1 – UML діаграма шаблону дизайну Message Pattern

**Основні складові**

* Відправник (Sender): Компонент, що відправляє повідомлення.
* Отримувач (Receiver): Компонент, що отримує повідомлення.
* Повідомлення (Message): Структура даних, що передає інформацію між відправником і отримувачем.
* Черга повідомлень (Message Queue): Буфер, де зберігаються повідомлення до їх обробки отримувачем.

**Переваги**

* Асинхронність: Відправник і отримувач можуть працювати незалежно один від одного.
* Масштабованість: Система легко розширюється за рахунок додавання нових відправників або отримувачів.
* Гнучкість: Легко змінювати логіку обробки повідомлень без змін у відправниках.

**Недоліки**

* Складність відлагодження: Відстеження асинхронних повідомлень може бути складним.
* Затримки: Можливі затримки в обробці повідомлень через використання черги.

**Приклад використання**

public class Message {

public string Content { get; set; }

}

public class MessageQueue {

private Queue<Message> \_queue = new Queue<Message>();

public void Enqueue(Message message) {

\_queue.Enqueue(message);

}

public Message Dequeue() {

return \_queue.Dequeue();

}

}

public class Sender {

private MessageQueue \_queue;

public Sender(MessageQueue queue) {

\_queue = queue;

}

public void SendMessage(string content) {

\_queue.Enqueue(new Message { Content = content });

}

}

public class Receiver {

private MessageQueue \_queue;

public Receiver(MessageQueue queue) {

\_queue = queue;

}

public void ReceiveMessages() {

while (\_queue.Count > 0) {

Message message = \_queue.Dequeue();

Console.WriteLine("Received message: " + message.Content);

}

}

}

**2. Servant Pattern**

Шаблон Servant Pattern дозволяє реалізувати загальний функціонал для групи класів, централізуючи його в одному об'єкті-сервантові.

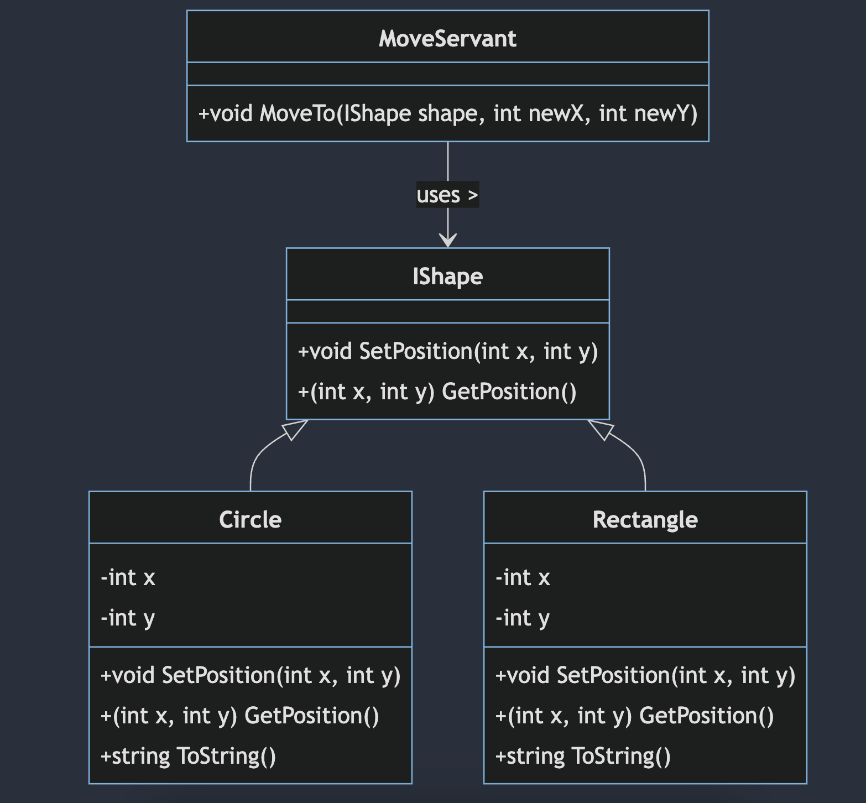


Рисунок 2 – UML діаграма шаблону дизайну Servant Pattern

**Основні складові**

* Клієнти (Clients): Класи, що використовують функціонал серванта.
* Сервант (Servant): Об'єкт, що надає загальний функціонал для клієнтів.
* Інтерфейс (Interface): Загальний інтерфейс для взаємодії з сервантовим функціоналом.

**Переваги**

* Повторне використання коду: Загальний функціонал централізовано і легко повторно використовується.
* Зниження складності: Логіка розподіляється між клієнтами і сервантом, зменшуючи складність класів-клієнтів.

**Недоліки**

* Залежність: Клієнти стають залежними від серванта.
* Продуктивність: Використання серванта може впливати на продуктивність через додатковий рівень викликів.

**Приклад використання**

public interface IMovable {

void Move(int x, int y);

}

public class Servant {

public void MoveTo(IMovable movable, int x, int y) {

movable.Move(x, y);

}

}

public class ClientA : IMovable {

public void Move(int x, int y) {

Console.WriteLine($"ClientA moved to {x}, {y}");

}

}

public class ClientB : IMovable {

public void Move(int x, int y) {

Console.WriteLine($"ClientB moved to {x}, {y}");

}

}

// Використання

Servant servant = new Servant();

ClientA clientA = new ClientA();

ClientB clientB = new ClientB();

servant.MoveTo(clientA, 10, 20);

servant.MoveTo(clientB, 30, 40);

**3. Multiton**

Шаблон Multiton гарантує, що для кожного унікального ключа створюється тільки один екземпляр класу, який зберігається і надається за запитом.

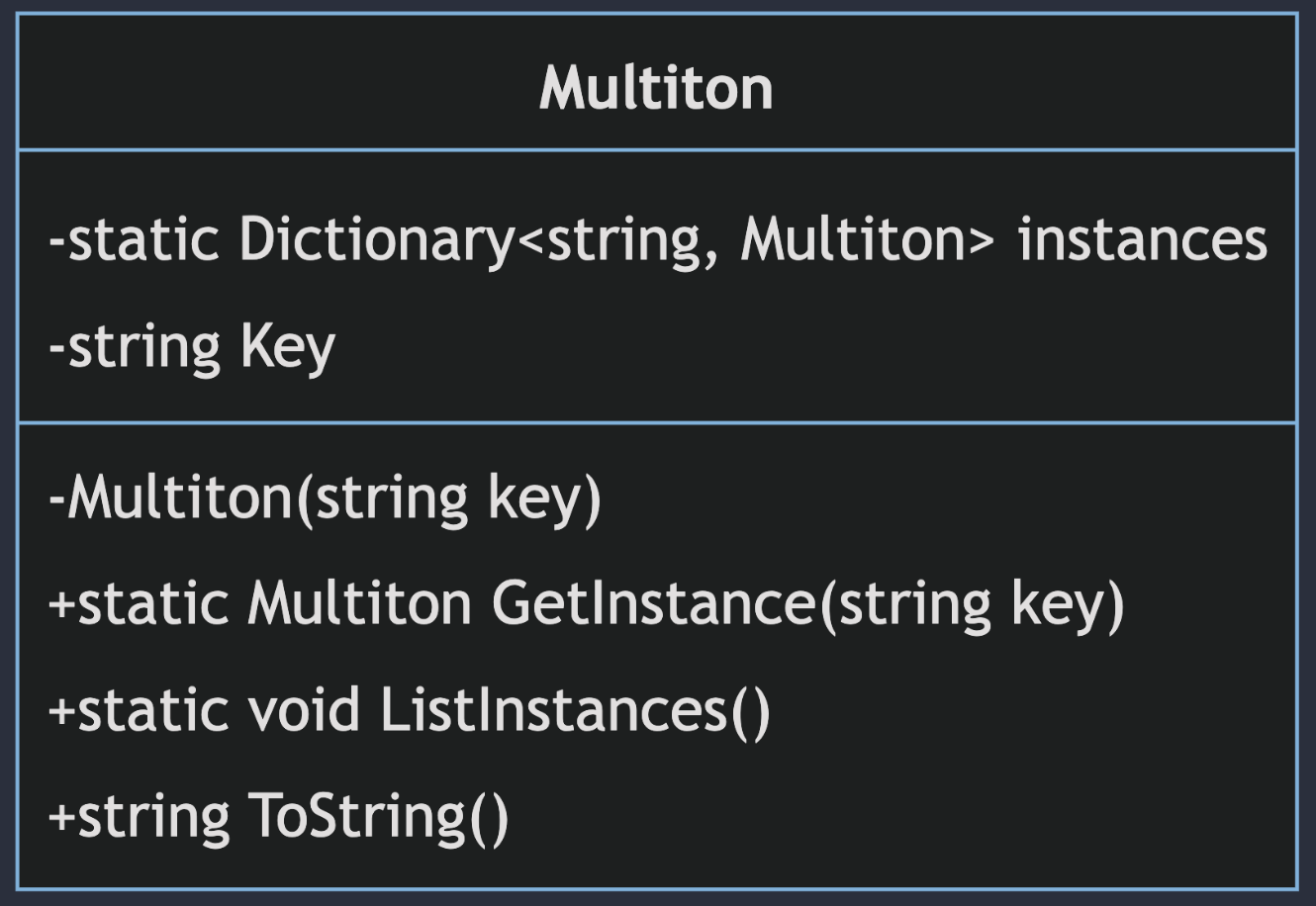


Рисунок 3 – UML діаграма шаблону дизайну Multiton

**Основні складові**

* Клас Multiton: Клас, що зберігає екземпляри об'єктів, асоційованих з унікальними ключами.
* Ключі: Унікальні ідентифікатори, що використовуються для отримання відповідних екземплярів.

**Переваги**

* Контроль екземплярів: Гарантується наявність одного екземпляра для кожного ключа.
* Ефективність: Знижується кількість створюваних об'єктів, що зменшує навантаження на пам'ять.

**Недоліки**

* Ускладнення коду: Логіка управління екземплярами може ускладнити код.
* Залежність від ключів: Необхідно чітко визначати та контролювати ключі.

**Приклад використання**

public class Multiton {

private static Dictionary<string, Multiton> \_instances = new Dictionary<string, Multiton>();

public string Name { get; private set; }

private Multiton(string name) {

Name = name;

}

public static Multiton GetInstance(string key) {

if (!\_instances.ContainsKey(key)) {

\_instances[key] = new Multiton(key);

}

return \_instances[key];

}

}

// Використання

var instance1 = Multiton.GetInstance("Instance1");

var instance2 = Multiton.GetInstance("Instance2");

var sameInstance1 = Multiton.GetInstance("Instance1");

Console.WriteLine(instance1 == sameInstance1); // True

Console.WriteLine(instance1 == instance2); // False

**4. Adapter**

Шаблон Adapter дозволяє об'єктам з несумісними інтерфейсами працювати разом шляхом перетворення інтерфейсу одного об'єкта до форми, сумісної з інтерфейсом іншого.

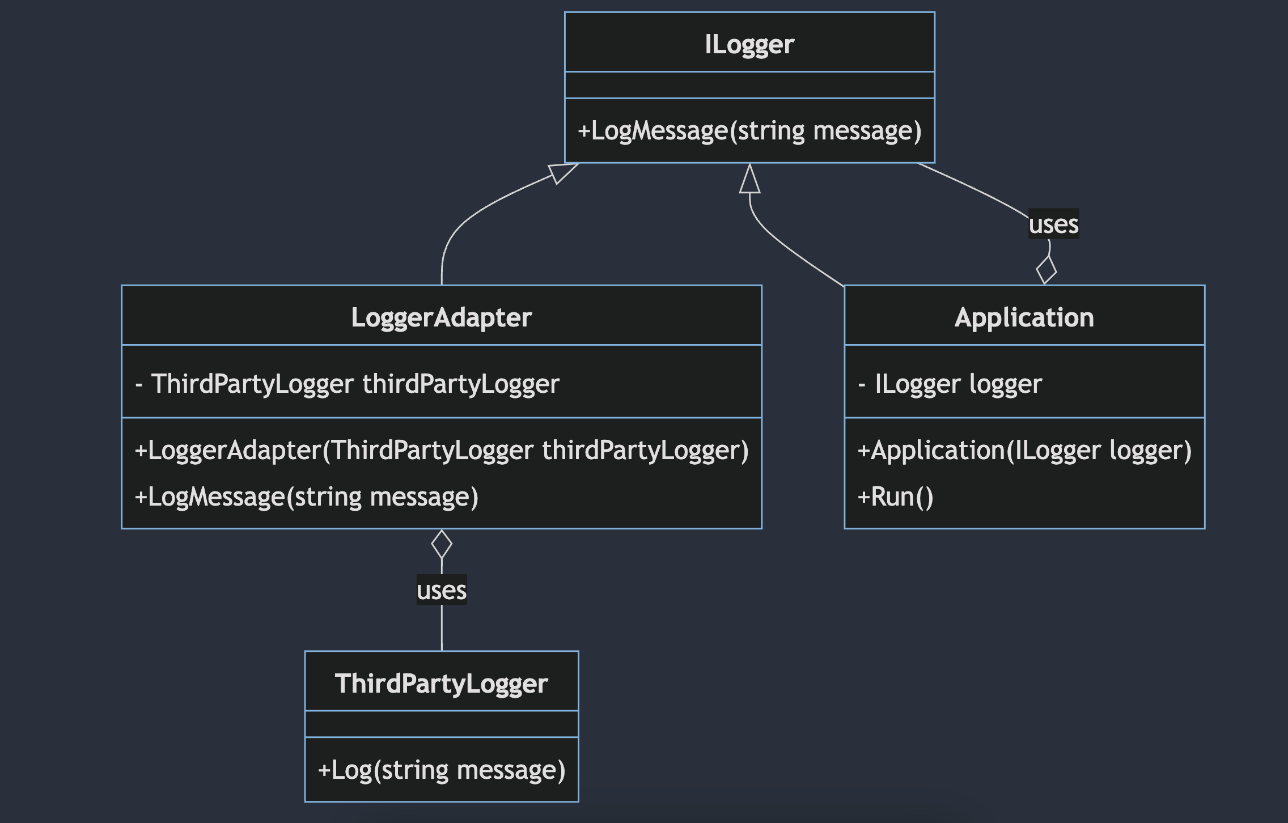


Рисунок 1 – UML діаграма шаблону дизайну Adapter

**Основні складові**

* Клієнт (Client): Клас, що використовує цільовий інтерфейс.
* Цільовий інтерфейс (Target Interface): Інтерфейс, необхідний клієнту.
* Адаптер (Adapter): Клас, що перетворює інтерфейс адаптованого класу до цільового інтерфейсу.
* Адаптований клас (Adaptee): Клас з несумісним інтерфейсом, що підлягає адаптації.

**Переваги**

* Гнучкість: Дозволяє інтегрувати класи з несумісними інтерфейсами без зміни їх коду.
* Повторне використання: Можна використовувати існуючі класи в нових контекстах.

**Недоліки**

* Складність: Додає додатковий рівень абстракції, що може ускладнити розуміння коду.
* Надмірність: Інколи може бути занадто складним рішенням для простих проблем сумісності.

**Приклад використання**

// Цільовий інтерфейс

public interface ITarget {

string GetRequest();

}

// Адаптований клас

public class Adaptee {

public string GetSpecificRequest() {

return "Specific request.";

}

}

// Адаптер

public class Adapter : ITarget {

private readonly Adaptee \_adaptee;

public Adapter(Adaptee adaptee) {

\_adaptee = adaptee;

}

public string GetRequest() {

return $"This is '{\_adaptee.GetSpecificRequest()}'";

}

}

// Використання

Adaptee adaptee = new Adaptee();

ITarget target = new Adapter(adaptee);

Console.WriteLine(target.GetRequest());

**Посилання на першоджерела**

[Medium](https://justgokus.medium.com/what-is-the-multiton-design-pattern-eeeb5dd8bc7d), [Wikipedia](https://en.wikipedia.org/wiki/Multiton_pattern" \t "_blank), [Stack Overflow](https://stackoverflow.com/questions/9631087/multiton-real-world-example" \t "_blank), [Code Chuckle](https://medium.com/@codechuckle/demystifying-design-patterns-multiton-pattern-859b50638a9f" \t "_blank), [Code Project](https://www.codeproject.com/Articles/1178694/Singleton-and-Multiton-Pattern" \t "_blank), [DZone](https://dzone.com/articles/adapter-design-pattern-in-java), [Pentalog](https://www.pentalog.com/blog/design-patterns/adapter-design-pattern/" \t "_blank), [Stackify](https://stackify.com/design-patterns-explained-adapter-pattern-with-code-examples/" \t "_blank), [GeeksforGeeks](https://www.geeksforgeeks.org/adapter-pattern/" \t "_blank), [OODesign](https://www.oodesign.com/adapter-pattern" \t "_blank), [The Coded Self](https://www.thecodedself.com/The-Difference-Between-an-Adapter-and-a-Wrapper/" \t "_blank), [Stack Interface](https://stackinterface.com/adapter-design-pattern/" \t "_blank), ChatGPT, [FreeCodeCamp](https://www.freecodecamp.org/news/design-pattern-for-modern-backend-development-and-use-cases/), [Visual Paradigm](https://tutorials.visual-paradigm.com/interpreter-pattern-tutorial/), [Simple Tech Talks](https://simpletechtalks.com/interpreter-design-pattern-explained-with-simple-example/" \t "_blank), [GeeksforGeeks](https://www.geeksforgeeks.org/interpreter-design-pattern/" \t "_blank), [Dot Net Tutorials](https://dotnettutorials.net/lesson/interpreter-design-pattern/" \t "_blank), [InfoWorld](https://www.infoworld.com/article/3243279/how-to-work-with-the-interpreter-design-pattern.html" \t "_blank), [TutorialsPoint](https://www.tutorialspoint.com/design_pattern/interpreter_pattern.htm), [Educative](https://www.educative.io/courses/deep-dive-into-the-internals-of-the-database/optimistic-vs-pessimistic-concurrency-control), [IBM](https://www.ibm.com/docs/en/db2/10.5?topic=management-locks-concurrency-control), [OSTEP](https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks-usage.pdf), [GeeksforGeeks](https://www.geeksforgeeks.org/two-phase-locking-protocol/" \t "_blank), [Wikipedia](https://en.wikipedia.org/wiki/Lock_%28computer_science%29), [Hacking with Rust](https://www.hackingwithrust.net/2024/01/22/simplifying-concurrency-easy-implementation-of-double-checked-locking-pattern-in-rust/), [Oracle Blogs](https://blogs.oracle.com/maa/post/from-chaos-to-order-the-importance-of-concurrency-control-within-the-database-2-of-6" \t "_blank),

Висновок

У ході виконання завдання було знайдено та описано шаблони проектування кожного типу: породжуючі (Creational), структурні (Structural), поведінкові (Behavioral) та паралельні обчислення (Concurrency). Для кожного з цих шаблонів був створений відповідний текстовий опис та наведено графічне подання у вигляді UML-діаграм, використовуючи інструмент візуалізації Mermaid.

Також було розроблено програмний проект для кожного шаблону проектування з метою продемонструвати його особливості та застосування на практиці. Кожен з цих проектів ілюструє використання відповідного шаблону проектування в реальних ситуаціях розробки програмного забезпечення.

Для завдання підвищеного рівня складності було створено шаблон проектування у вигляді zip-файлу, який може бути використаний як шаблон проекту або елемент (Project / Item Template) для середовища розробки Visual Studio.

*Додаток А*Машинний лістинг файлу **Messaging**

Реалізацція C#:

using System;

using System.Collections.Generic;

// Клас повідомлень

public class Message

{

public string Content { get; }

public string Sender { get; }

public string Receiver { get; }

public Message(string content, string sender, string receiver)

{

Content = content;

Sender = sender;

Receiver = receiver;

}

}

// Посередник повідомлень

public class MessageBroker

{

private readonly Dictionary<string, List<Action<Message>>> \_subscribers = new Dictionary<string, List<Action<Message>>>();

public void Subscribe(string receiver, Action<Message> handler)

{

if (!\_subscribers.ContainsKey(receiver))

{

\_subscribers[receiver] = new List<Action<Message>>();

}

\_subscribers[receiver].Add(handler);

}

public void SendMessage(Message message)

{

if (\_subscribers.ContainsKey(message.Receiver))

{

foreach (var handler in \_subscribers[message.Receiver])

{

handler(message);

}

}

}

}

// Компоненти

public class ComponentA

{

private readonly string \_name;

private readonly MessageBroker \_broker;

public ComponentA(string name, MessageBroker broker)

{

\_name = name;

\_broker = broker;

\_broker.Subscribe(\_name, ReceiveMessage);

}

public void SendMessage(string content, string receiver)

{

var message = new Message(content, \_name, receiver);

\_broker.SendMessage(message);

}

private void ReceiveMessage(Message message)

{

Console.WriteLine($"{\_name} отримано повідомлення від {message.Sender}: {message.Content}");

}

}

public class ComponentB

{

private readonly string \_name;

private readonly MessageBroker \_broker;

public ComponentB(string name, MessageBroker broker)

{

\_name = name;

\_broker = broker;

\_broker.Subscribe(\_name, ReceiveMessage);

}

public void SendMessage(string content, string receiver)

{

var message = new Message(content, \_name, receiver);

\_broker.SendMessage(message);

}

private void ReceiveMessage(Message message)

{

Console.WriteLine($"{\_name} отримано повідомлення від {message.Sender}: {message.Content}");

}

}

// Використання

public class Program

{

public static void Main()

{

var broker = new MessageBroker();

var componentA = new ComponentA("Manager", broker);

var componentB = new ComponentB("Client", broker);

componentA.SendMessage("Вітаю, Client!", "Client");

componentB.SendMessage("Привіт, Manager!", "Manager");

}

}

*Додаток Б*Машинний лістинг файлу **Disign Servant**

Реалізацція C#:

public interface IShape

{

void SetPosition(int x, int y);

(int x, int y) GetPosition();

}

// Клас створення форм

public class Circle : IShape

{

private int x, y;

public void SetPosition(int x, int y)

{

this.x = x;

this.y = y;

}

public (int x, int y) GetPosition()

{

return (x, y);

}

public override string ToString()

{

return $"Коло в координтах ({x}, {y})";

}

}

public class Rectangle : IShape

{

private int x, y;

public void SetPosition(int x, int y)

{

this.x = x;

this.y = y;

}

public (int x, int y) GetPosition()

{

return (x, y);

}

public override string ToString()

{

return $"Прямокутник в координтах ({x}, {y})";

}

}

// Клас Servant

public class MoveServant

{

public void MoveTo(IShape shape, int newX, int newY)

{

shape.SetPosition(newX, newY);

}

}

// Використання

public class Program

{

public static void Main()

{

IShape circle = new Circle();

IShape rectangle = new Rectangle();

MoveServant moveServant = new MoveServant();

// Встановлення початкових позицій

circle.SetPosition(0, 0);

rectangle.SetPosition(10, 10);

// Переміщення фігур за допомогою servant

moveServant.MoveTo(circle, 5, 5);

moveServant.MoveTo(rectangle, 15, 15);

// Відображення нових позицій

Console.WriteLine(circle); // Output: Коло (5, 5)

Console.WriteLine(rectangle); // Output: Прямокутник(15, 15)

}

}

*Додаток В*Машинний лістинг файлу **Multiton**

Реалізацція C#:

using System;

using System.Collections.Generic;

public class Multiton

{

// Словник для зберігання екземплярів

private static readonly Dictionary<string, Multiton> instances = new Dictionary<string, Multiton>();

// Конструктор для запобігання інстанціювання

private Multiton(string key)

{

Key = key;

}

// Доступ до ключа

public string Key { get; }

// Метод для отримання екземпляру за ключем

public static Multiton GetInstance(string key)

{

if (!instances.ContainsKey(key))

{

instances[key] = new Multiton(key);

}

return instances[key];

}

public static void ListInstances()

{

foreach (var instance in instances)

{

Console.WriteLine($"Клавіша: {instance.Key}, Екземпляр: {instance.Value}");

}

}

}

public class Program

{

public static void Main()

{

// Доступ до екземплярів за допомогою ключів.

var instance1 = Multiton.GetInstance("Esc");

var instance2 = Multiton.GetInstance("Space");

var instance3 = Multiton.GetInstance("Ctr");

// Перевіряє, чи існують однакові екземпляри для одного і того ж ключа.

Console.WriteLine(ReferenceEquals(instance1, instance3)); // True

Console.WriteLine(ReferenceEquals(instance1, instance2)); // False

// Перебирає всі випадки

Multiton.ListInstances();

}

}

*Додаток Г*Машинний лістинг файлу **Adapter01**

Реалізацція C#:

public interface ILogger

{

void LogMessage(string message);

}

// Клас для логування повідомлень

public class MyLogger

{

public void Log(string message)

{

Console.WriteLine($"Адаптація: {message}");

}

}

// Адаптер, який реалізує інтерфейс ILogger та використовує клас MyLogger

public class LoggerAdapter : ILogger

{

private MyLogger myLogger;

public LoggerAdapter(MyLogger myLogger)

{

this.myLogger = myLogger;

}

public void LogMessage(string message)

{

myLogger.Log(message);

}

}

// Клас, який використовує інтерфейс ILogger

public class Application

{

private ILogger logger;

public Application(ILogger logger)

{

this.logger = logger;

}

public void Run()

{

logger.LogMessage("Адаптація завершена.");

}

}

// Використання

class Program

{

static void Main(string[] args)

{

// Створення екземпляра

MyLogger myLogger = new MyLogger();

// Створення адаптера

ILogger loggerAdapter = new LoggerAdapter(myLogger);

// Створення додатку

Application app = new Application(loggerAdapter);

// Запуск

app.Run();

}

}