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))

Житник О.М.

(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 11**

|  |  |  |  |
| --- | --- | --- | --- |
| ***Creational pattern*** | ***Structural pattern*** | ***Behavioral pattern*** | ***Concurrency pattern*** |
| Multiton | Adapter or Wrapper | Observer | Double checked locking |

**Мультитон –** це шаблон проєктування, який дозволяє створювати лише один екземпляр об’єкта, але розширює його можливістю впорядковувати декілька екземплярів у вигляді іменованих пар ім'я-значення.

Замість того, щоб забезпечувати наявність лише одного об'єкта в програмі, мультитон гарантує існування лише одного входження з відповідним ключем. [6].

**UML-діаграми (рисунок 1 – 2):**

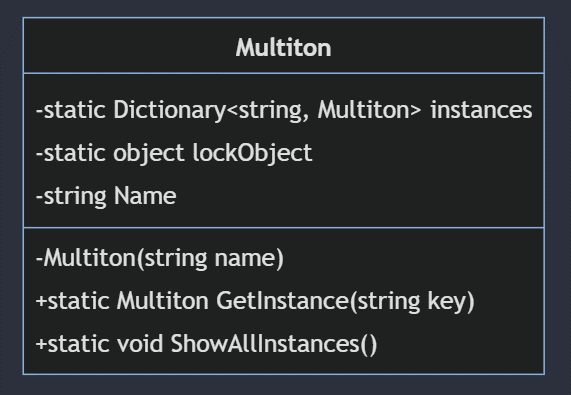


Рисунок 1. Static model

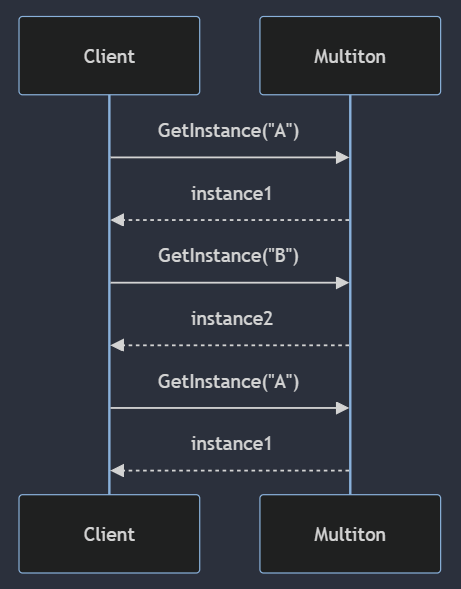


Рисунок 2. Dynamic model

**Основні структурні елементи:**

Приватний конструктор: Запобігає створенню екземплярів класу ззовні.

Словник (Dictionary) або подібна структура даних: Зберігає створені екземпляри за їхніми ключами.

Статичний метод GetInstance: Використовується для отримання екземпляра за ключем. Створює новий екземпляр, якщо він ще не існує.

Статичний метод для виведення всіх екземплярів (опціонально): Метод, який показує всі створені екземпляри.

**Адаптер —** це структурний патерн, який дозволяє подружити несумісні об’єкти[7].

Адаптер виступає прошарком між двома об’єктами, перетворюючи виклики одного у виклики, що зрозумілі іншому.

Патерн можна часто зустріти в C#-коді, особливо там, де потрібна конвертація різних типів даних або спільна робота класів з різними інтерфейсами.

**UML-діаграми (рисунок 3 – 4):**

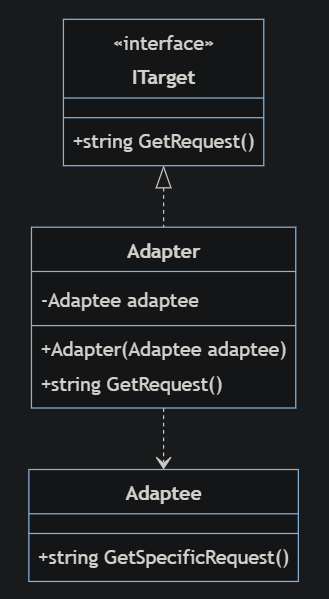


Рисунок 3. Static model

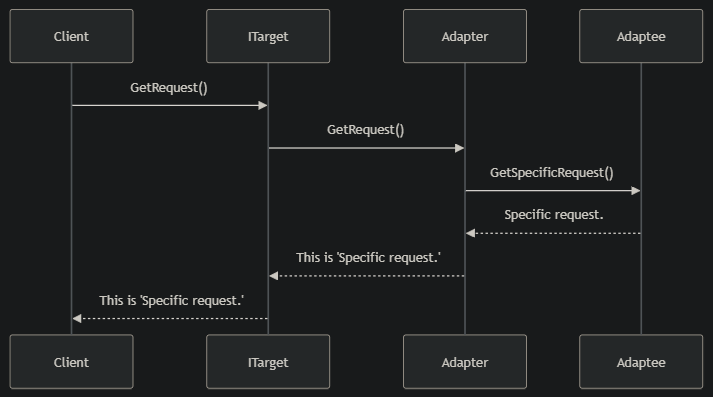


Рисунок 4. Dynamic model

**Основні структурні елементи:**

Клієнт (Client): Клас або об'єкт, який використовує інтерфейс Target для виконання своїх завдань.

Цільовий інтерфейс (Target): Інтерфейс, якого потребує клієнт для роботи. Це інтерфейс, який адаптер реалізує для забезпечення сумісності з клієнтом.

Адаптований клас (Adaptee): Існуючий клас з несумісним інтерфейсом, який потрібно адаптувати до інтерфейсу Target. Це клас, який містить корисні методи, але його інтерфейс не підходить клієнту.

Адаптер (Adapter): Клас, який реалізує інтерфейс Target та обгортає (компонує) клас Adaptee. Він перетворює інтерфейс Adaptee у відповідний інтерфейс Target, щоб клієнт міг його використовувати.

**Спостерігач —** це поведінковий патерн, який дозволяє об’єктам повідомляти інші об’єкти про зміни свого стану.[8].

При цьому спостерігачі можуть вільно підписуватися і відписуватись від цих повідомлень.

Спостерігач часто зустрічається в коді C#, особливо там, де до відносин між компонентами застосовується модель подій. Спостерігач дозволяє окремим компонентам реагувати на події, які відбуваються в інших компонентах.

**UML-діаграма (рисунок 5 – 6):**

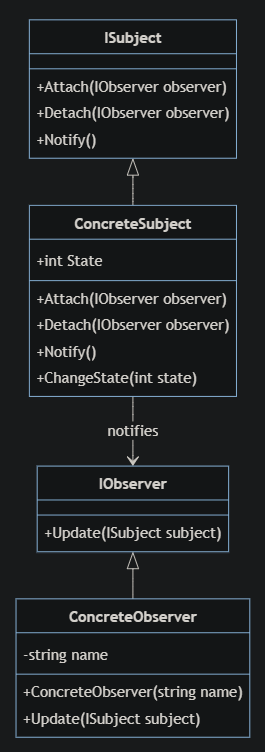


Рисунок 5. State model

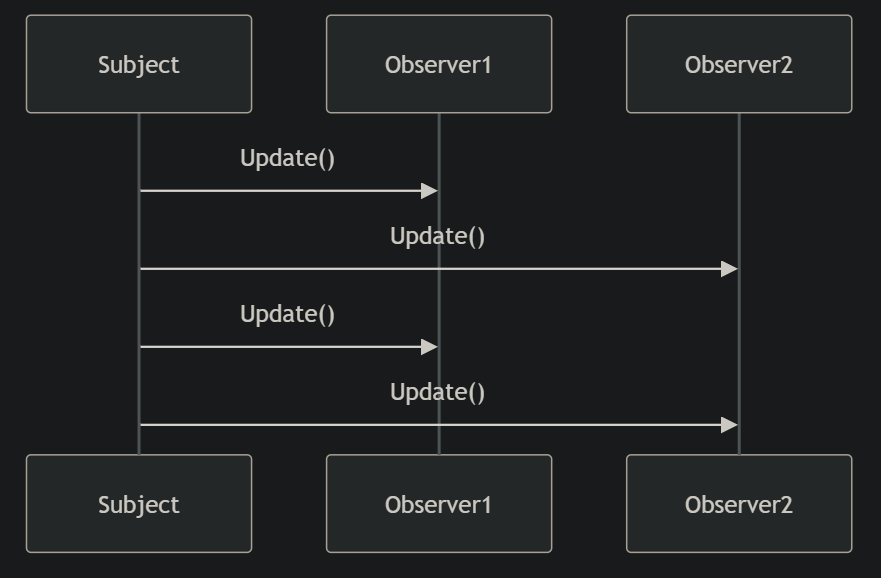


Рисунок 6. Dynamic model

**Основні структурні елементи:**

Subject (Суб'єкт): Інтерфейс або абстрактний клас: Визначає методи для додавання, видалення та повідомлення спостерігачів.

Attach(IObserver observer): Додає спостерігача до списку.

Detach(IObserver observer): Видаляє спостерігача зі списку.

Notify(): Повідомляє всіх спостерігачів про зміну стану.

Observer (Спостерігач):

Інтерфейс або абстрактний клас: Визначає метод оновлення, який буде викликаний суб'єктом для повідомлення про зміни.

Update(ISubject subject): Оновлює стан спостерігача відповідно до стану суб'єкта.

ConcreteSubject (Конкретний Суб'єкт):

Реалізує інтерфейс Subject і зберігає стан, який цікавить спостерігачів.

Викликає метод Notify при зміні стану для повідомлення всіх спостерігачів.

ConcreteObserver (Конкретний Спостерігач): Реалізує інтерфейс Observer і оновлює свій стан відповідно до змін у суб'єкті.

**Подвійне замикання** - це шаблон проектування програмного забезпечення, який використовується для зменшення накладних витрат на придбання замка шляхом тестування критерію замикання («підказка щодо замка») перед придбанням замка. Блокування відбувається лише в тому випадку, якщо перевірка критерію блокування вказує на те, що потрібне блокування. [9].

Патерн зазвичай використовується для зменшення накладних витрат на блокування при реалізації "лінивої ініціалізації" в багатопотоковому середовищі, особливо як частина патерну Singleton. Лінива ініціалізація дозволяє уникнути ініціалізації значення до першого звернення.

**UML-діаграми (рисунок 7 – 8):**

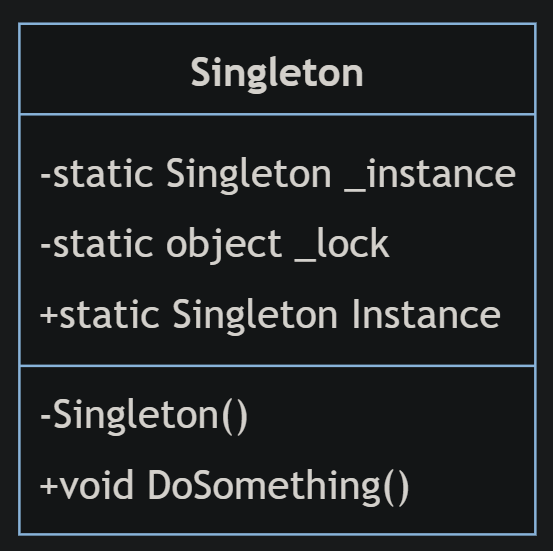
****

Рисунок 7. Static model

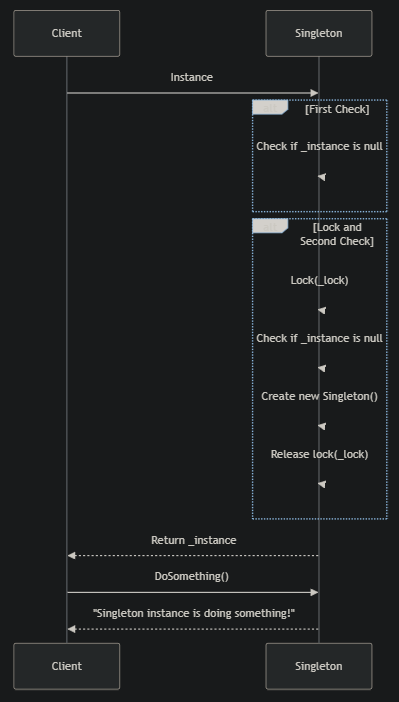
****

Рисунок 8. Dynamic model

**Основні структурні елементи:**

Singleton Instance (Єдиний екземпляр): Це статичний член класу, який зберігає єдиний екземпляр об'єкта, що створюється.

Lock Object (Об'єкт блокування): Це об'єкт, який використовується для синхронізації доступу до блоку коду, що ініціалізує єдиний екземпляр. Він запобігає одночасному доступу до цього блоку з різних потоків.

Double-Checked Locking (Подвійна перевірка блокування): Перша перевірка: Здійснюється поза блокуванням для швидкої перевірки наявності ініціалізованого екземпляра.

# **Лістинг програм**

## Шаблон «Мультитон»:

***Мовою С#:***

public class Multiton

{

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

private static readonly object lockObject = new object();

public string Name { get; private set; }

private Multiton(string name)

{

Name = name;

}

public static Multiton GetInstance(string key)

{

lock (lockObject)

{

if (!instances.ContainsKey(key))

{

instances[key] = new Multiton(key);

}

return instances[key];

}

}

public static void ShowAllInstances()

{

foreach (var instance in instances)

{

Console.WriteLine($"Key: {instance.Key}, Instance Name: {instance.Value.Name}");

}

}

}

class Program

{

static void Main()

{

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

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

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

Console.WriteLine(instance1.Name);

Console.WriteLine(instance2.Name);

Console.WriteLine(instance3.Name);

Multiton.ShowAllInstances();

}

}

***Мовою Python:***

import threading

class Multiton:

\_instances = {}

\_lock = threading.Lock()

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

self.name = name

@classmethod

def get\_instance(cls, key):

with cls.\_lock:

if key not in cls.\_instances:

cls.\_instances[key] = cls(key)

return cls.\_instances[key]

@classmethod

def show\_all\_instances(cls):

for key, instance in cls.\_instances.items():

print(f"Key: {key}, Instance Name: {instance.name}")

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

instance1 = Multiton.get\_instance("A")

instance2 = Multiton.get\_instance("B")

instance3 = Multiton.get\_instance("A")

print(instance1.name)

print(instance2.name)

print(instance3.name)

Multiton.show\_all\_instances()

## Шаблон «Адаптер»

***Мовою C#:***

namespace AdapterPattern

{

public interface ITarget

{

string GetRequest();

}

public class Adaptee

{

public string GetSpecificRequest()

{

return "Specific request.";

}

}

public class Adapter : ITarget

{

private readonly Adaptee \_adaptee;

public Adapter(Adaptee adaptee)

{

this.\_adaptee = adaptee;

}

public string GetRequest()

{

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

}

}

class Program

{

static void Main(string[] args)

{

Adaptee adaptee = new Adaptee();

ITarget target = new Adapter(adaptee);

Console.WriteLine("Adaptee interface is incompatible with the client.");

Console.WriteLine("But with adapter, the client can call its method.");

Console.WriteLine(target.GetRequest());

}

}

}***Мовою Python:***

class ITarget:

def get\_request(self):

pass

class Adaptee:

def get\_specific\_request(self):

return "Specific request."

class Adapter(ITarget):

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

self.\_adaptee = adaptee

def get\_request(self):

return f"This is '{self.\_adaptee.get\_specific\_request()}'"

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

adaptee = Adaptee()

target = Adapter(adaptee)

print("Adaptee interface is incompatible with the client.")

print("But with adapter, the client can call its method.")

print(target.get\_request())

## Шаблон Спостерігач:

***Мовою C#:***

namespace ObserverPattern

{

public interface IObserver

{

void Update(ISubject subject);

}

public interface ISubject

{

void Attach(IObserver observer);

void Detach(IObserver observer);

void Notify();

}

public class ConcreteSubject : ISubject

{

public int State { get; private set; } = 0;

private List<IObserver> \_observers = new List<IObserver>();

public void Attach(IObserver observer)

{

\_observers.Add(observer);

}

public void Detach(IObserver observer)

{

\_observers.Remove(observer);

}

public void Notify()

{

foreach (var observer in \_observers)

{

observer.Update(this);

}

}

public void ChangeState(int state)

{

State = state;

Notify();

}

}

public class ConcreteObserver : IObserver

{

private string \_name;

public ConcreteObserver(string name)

{

\_name = name;

}

public void Update(ISubject subject)

{

if (subject is ConcreteSubject concreteSubject)

{

Console.WriteLine($"Observer {\_name}: Reacted to the state change to {concreteSubject.State}");

}

}

}

class Program

{

static void Main(string[] args)

{

ConcreteSubject subject = new ConcreteSubject();

ConcreteObserver observer1 = new ConcreteObserver("A");

ConcreteObserver observer2 = new ConcreteObserver("B");

subject.Attach(observer1);

subject.Attach(observer2);

subject.ChangeState(1);

subject.ChangeState(2);

subject.Detach(observer1);

subject.ChangeState(3);

}

}

}

***Мовою Python:***

from abc import ABC, abstractmethod

class IObserver(ABC):

@abstractmethod

def update(self, subject):

pass

class ISubject(ABC):

@abstractmethod

def attach(self, observer):

pass

@abstractmethod

def detach(self, observer):

pass

@abstractmethod

def notify(self):

pass

class ConcreteSubject(ISubject):

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

self.\_state = 0

self.\_observers = []

def attach(self, observer):

self.\_observers.append(observer)

def detach(self, observer):

self.\_observers.remove(observer)

def notify(self):

for observer in self.\_observers:

observer.update(self)

def change\_state(self, state):

self.\_state = state

self.notify()

class ConcreteObserver(IObserver):

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

self.\_name = name

def update(self, subject):

if isinstance(subject, ConcreteSubject):

print(f"Observer {self.\_name}: Reacted to the state change to {subject.\_state}")

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

subject = ConcreteSubject()

observer1 = ConcreteObserver("A")

observer2 = ConcreteObserver("B")

subject.attach(observer1)

subject.attach(observer2)

subject.change\_state(1)

subject.change\_state(2)

subject.detach(observer1)

subject.change\_state(3)

## Шаблон Подвійне замикання:

***Мовою С#:***

public sealed class Singleton

{

private static Singleton \_instance = null;

private static readonly object \_lock = new object();

private Singleton()

{

}

public static Singleton Instance

{

get

{

if (\_instance == null)

{

lock (\_lock)

{

if (\_instance == null)

{

\_instance = new Singleton();

}

}

}

return \_instance;

}

}

public void DoSomething()

{

Console.WriteLine("Singleton instance is doing something!");

}

}

class Program

{

static void Main()

{

Singleton singleton1 = Singleton.Instance;

Singleton singleton2 = Singleton.Instance;

if (singleton1 == singleton2)

{

Console.WriteLine("Both instances are the same.");

}

singleton1.DoSomething();

}

}

***Мовою Python:***

from threading import Lock

class Singleton:

\_instance = None

\_lock = Lock()

def \_\_new\_\_(cls):

with cls.\_lock:

if not cls.\_instance:

cls.\_instance = super().\_\_new\_\_(cls)

return cls.\_instance

def do\_something(self):

print("Singleton instance is doing something!")

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

singleton1 = Singleton()

singleton2 = Singleton()

if singleton1 is singleton2:

print("Both instances are the same.")

singleton1.do\_something()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 проаналізовано стандартні ситуації, які виникають при створенні складних програмних проектів, а також застосування шаблонів проектування для їх вирішення. У нашому випадку були розглянуті патерни Мультитон, Адаптер, Спостерігач і Подвійне замикання. Проведено аналіз цих патернів програмування, у тому числі створення UML-діаграм, прикладів програмного забезпечення, шаблонів тощо.

# **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. Мультитон (шаблон проєктування) [Електронний ресурс] – Режим доступу до ресурсу: <https://uk.wikipedia.org/wiki/%D0%9C%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D1%82%D0%BE%D0%BD_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D1%94%D0%BA%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)>.
7. Адаптер [Електронний ресурс] – Режим доступу до ресурсу: <https://refactoring.guru/uk/design-patterns/adapter>.
8. Спостерігач [Електронний ресурс] – Режим доступу до ресурсу: <https://refactoring.guru/uk/design-patterns/observer>.
9. Double-checked locking [Електронний ресурс] – Режим доступу до ресурсу: <https://en.wikipedia.org/wiki/Double-checked_locking>.