Дисципліна: Об’єктно-орієнтоване конструювання програм
Лабораторна робота № 2 (4 год.)
Тема: **Патерни проєктування.**

Теоретичні відомості
План

1. Стратегія
2. Спостерігач
3. Декоратор

1. Стратегія

Патерн **Стратегія** (Strategy) являє собою шаблон проектування, який визначає набір алгоритмів, інкапсулює кожен з них і забезпечує їх взаємозамінність. Залежно від ситуації ми можемо легко замінити один використовуваний алгоритм іншим. При цьому заміна алгоритму відбувається незалежно від об'єкта, який використовує даний алгоритм.

Стратегію використовують у випадках, коли:

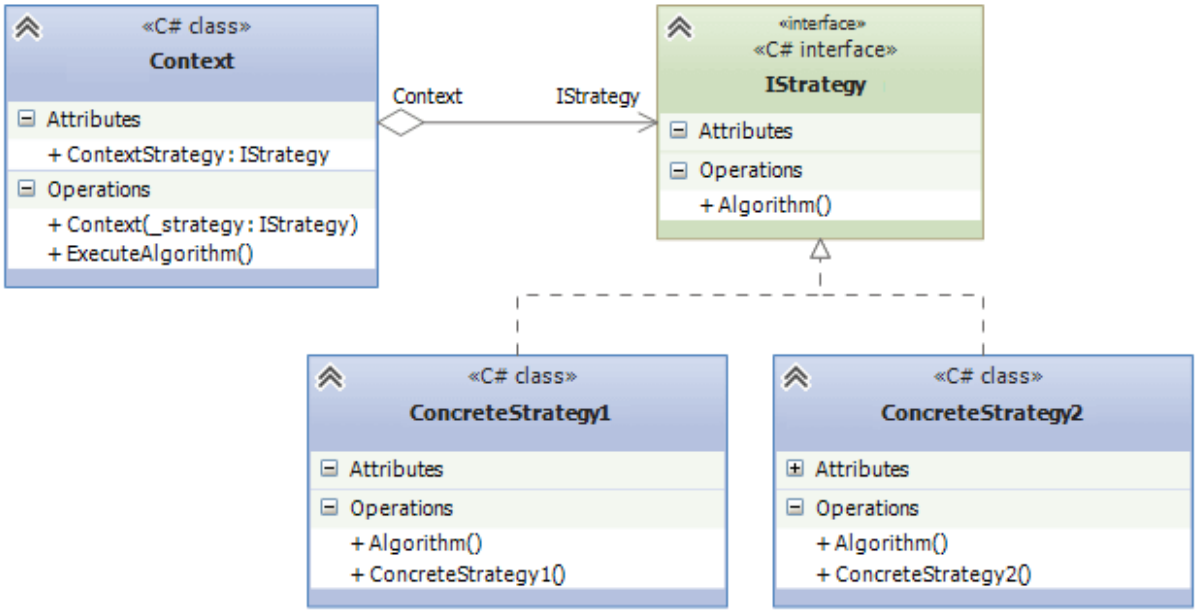
- є кілька споріднених класів, які відрізняються поведінкою. Можна задати один основний клас, а різні варіанти поведінки винести в окремі класи і при необхідності їх застосовувати

- необхідно забезпечити вибір з кількох варіантів алгоритмів, які можна легко змінювати в залежності від умов

- необхідно змінювати поведінку об'єктів на стадії виконання програми

- клас, який застосовує певну функціональність, нічого не повинен знати про її реалізацію

Формально патерн **Стратегія** можна виразити наступною схемою UML:



Формальне визначення патерну на мові C # може виглядати наступним чином:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

public interface IStrategy
{
   void Algorithm();
```

```
}

public class ConcreteStrategy1 : IStrategy
{
   public void Algorithm()
   { }
}

public class ConcreteStrategy2 : IStrategy
{
   public void Algorithm()
   { }
}

public class Context
{
   public IStrategy ContextStrategy { get; set; }

   public Context(IStrategy _strategy)
   {
      ContextStrategy = _strategy;
   }

   public void ExecuteAlgorithm()
   {
      ContextStrategy.Algorithm();
   }
}
```

Як видно з діаграми, тут є такі учасники:

Інтерфейс **IStrategy**, який визначає метод Algorithm(). Це загальний інтерфейс для всіх алгоритмів, що реалізуються. Замість інтерфейсу тут також можна було б використовувати абстрактний клас.

Класи **ConcreteStrategy1** і **ConcreteStrategy2**, які реалізують інтерфейс **IStrategy**, надаючи свою версію методу Algorithm(). Подібних класів-реалізацій може бути безліч.

Клас **Context** зберігає посилання на об'єкт **IStrategy** і пов'язаний з інтерфейсом **IStrategy** відношенням агрегації.

В даному випадку об'єкт **IStrategy** прописаний у властивості **ContextStrategy**, хоча для неї також можна було б визначити приватну змінну, а для динамічної установки використовувати спеціальний метод.

Розглянемо *приклад*. Існують різні легкові машини, які використовують різні джерела енергії: електрика, бензин, газ тощо. Є гібридні автомобілі. В цілому вони схожі й відрізняються переважно видом джерела енергії. У будь-який момент часу ми можемо змінити застосовуване джерело енергії, модифікувавши автомобіль. І в даному випадку цілком можна застосувати шаблон стратегії.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
class Program
{
    static void Main(string[] args)
    {
        Car auto = new Car(4, "Volvo", new PetrolMove());
        auto.Move();
        auto.Movable = new ElectricMove();
        auto.Move();

        Console.ReadLine();
    }
}
interface IMovable
{
    void Move();
}

class PetrolMove : IMovable
{
    public void Move()
    {
        Console.WriteLine("Переміщення на бензині");
    }
}



class ElectricMove : IMovable
{
    public void Move()
    {
        Console.WriteLine("Переміщення на електриці");
    }
}
class Car
{
    protected int passengers; // кіл-ть пасажирів
    protected string model; // модель автомобіля

    public Car(int num, string model, IMovable mov)
    {
        this.passengers = num;
        this.model = model;
        Movable = mov;
    }
    public IMovable Movable { private get; set; }
    public void Move()
    {
        Movable.Move();
    }
}
```

2. Спостерігач

Патерн **Спостерігач** (Observer) представляє собою поведінковий шаблон проектування, який використовує відношення "один до багатьох". В цьому відношенні є один спостережуваний об'єкт і множина спостерігачів. При зміні об'єкта, який спостерігається, автоматично відбувається інформування всіх спостерігачів.

Даний патерн ще називають **Publisher-Subscriber** (видавець-передплатник), оскільки відносини видавця і передплатників характеризують дію даного патерну: передплатники підписуються на email-розсилку з певного сайту. Сайт-видавець за допомогою email-розсилки повідомляє всіх передплатників про зміни. А передплатники отримують зміни і виконують певні дії: можуть зайти на сайт, можуть проігнорувати повідомлення тощо.

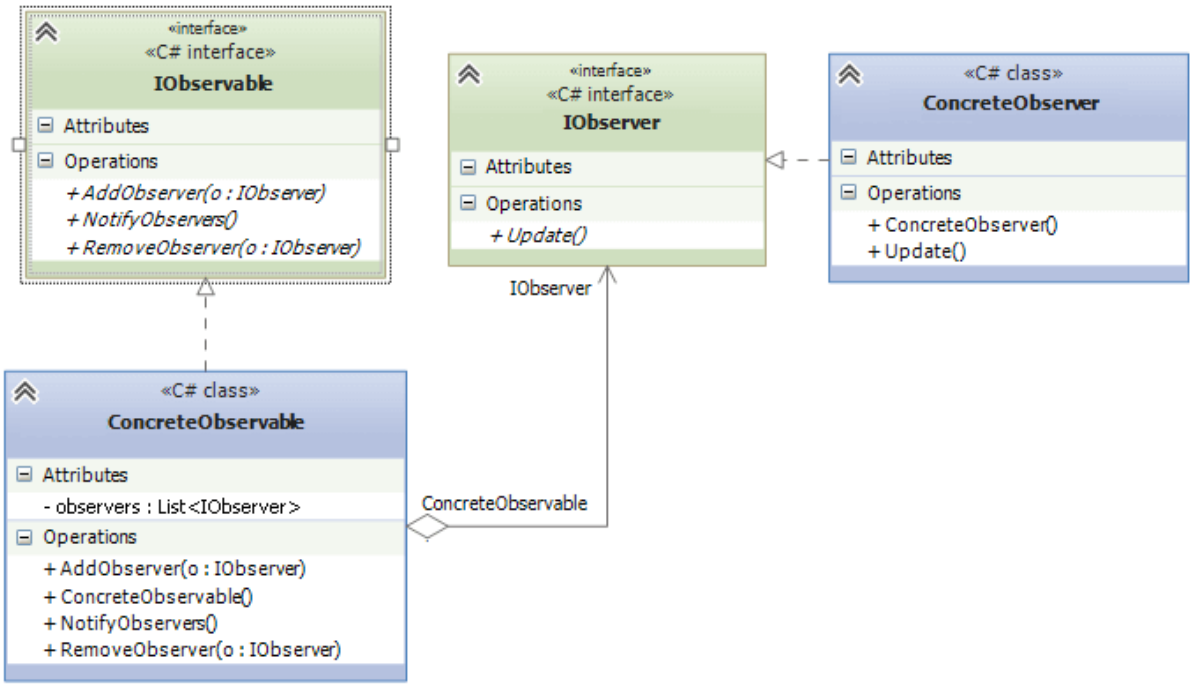
Патерн **Спостерігач** використовують у випадках, коли:

- система складається з безлічі класів, об'єкти яких повинні перебувати в узгоджених станах;

- загальна схема взаємодії об'єктів передбачає дві сторони: **головна** - розсилає повідомлення, **підпорядкована** - отримує повідомлення і реагує на них. Розділення логіки двох сторін дозволяє розглядати їх незалежно і використовувати окремо одну від одної;

- існує один об'єкт, що розсилає повідомлення, і безліч передплатників, які отримують повідомлення. При цьому точне число передплатників заздалегідь невідоме і в процесі роботи програми може змінюватися.

За допомогою діаграм UML даний шаблон можна виразити таким чином:



Формальне визначення патерну на мові C # може виглядати наступним чином:

```
interface IObservable
    {
       void AddObserver(IObserver o);
       void RemoveObserver(IObserver o);
       void NotifyObservers();
    }
    class ConcreteObservable : IObservable
    {
       private List<IObserver> observers;
       public ConcreteObservable()
       {
```

```
            observers = new List<IObserver>();
        }
        public void AddObserver(IObserver o)
        {
            observers.Add(o);
        }

        public void RemoveObserver(IObserver o)
        {
            observers.Remove(o);
        }

        public void NotifyObservers()
        {
            foreach (IObserver observer in observers)
                observer.Update();
        }
    }

    interface IObserver
    {
        void Update();
    }
    class ConcreteObserver : IObserver
    {
        public void Update()
        {
            } }
```

Учасники

– **IObservable**: представляє спостережуваний об'єкт. Визначає три методи: AddObserver () (для додавання спостерігача), RemoveObserver () (видалення спостерігача) і NotifyObservers () (повідомлення спостерігачам)

– **ConcreteObservable**: конкретна реалізація інтерфейсу **IObservable**. Визначає колекцію об'єктів спостерігачів.

– **IObserver**: представляє спостерігача, який підписується на все повідомлення, що спостерігається. Визначає метод Update (), який викликається спостережуваним об'єктом для повідомлення спостерігача.

– **ConcreteObserver**: конкретна реалізація інтерфейсу IObserver.

При цьому спостереженому об'єкту не потрібно нічого знати про спостерігачів крім того, що той реалізує метод Update(). За допомогою відношення агрегації реалізується слабкий зв’язок обох компонент.

У певний момент спостерігач може припинити спостереження. І після цього обидва об'єкти - спостерігач і спостерігається можуть продовжувати існувати в системі незалежно один від одного.

Розглянемо *приклад* застосування шаблону. Припустимо, у нас є біржа, де проходять торги, і є брокери і банки, які стежать за інформацією, що надходить і в залежності від інформації, що надійшла , виконують певні дії:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;

namespace ConsoleApp
{

    class Program
    {
        static void Main(string[] args)
        {
            Stock stock = new Stock();
            Bank bank = new Bank("ЮнитБанк", stock);
            Broker broker = new Broker("Петя", stock);
            // імітація торгів
            stock.Market();
            // брокер припиняє спостерігати за торгами
            broker.StopTrade();
            // імітація торгів
        stock.Market();

            Console.Read();
        }
    }

    interface IObserver
    {
        void Update(Object ob);
    }

    interface IObservable
    {
        void RegisterObserver(IObserver o);
        void RemoveObserver(IObserver o);
        void NotifyObservers();
    }

    class Stock : IObservable
    {
        StockInfo sInfo; // інформація про торгі

        List<IObserver> observers;
        public Stock()
        {
            observers = new List<IObserver>();
            sInfo = new StockInfo();
        }
        public void RegisterObserver(IObserver o)
        {
            observers.Add(o);
        }

        public void RemoveObserver(IObserver o)
        {
```

```
            observers.Remove(o);
        }

        public void NotifyObservers()
        {
            foreach (IObserver o in observers)
            {
                o.Update(sInfo);
            }
        }

        public void Market()
        {
            Random rnd = new Random();
            sInfo.USD = rnd.Next(20, 40);
            sInfo.Euro = rnd.Next(30, 50);
            NotifyObservers();
        }
    }

    class StockInfo
    {
        public int USD { get; set; }
        public int Euro { get; set; }
    }

    class Broker : IObserver
    {
        public string Name { get; set; }
        IObservable stock;
        public Broker(string name, IObservable obs)
        {
            this.Name = name;
            stock = obs;
            stock.RegisterObserver(this);
        }
        public void Update(object ob)
        {
            StockInfo sInfo = (StockInfo)ob;

            if (sInfo.USD > 30)
                Console.WriteLine("Брокер {0} продє долари;  Курс долара: {1}", this.Name,
sInfo.USD);
            else
                Console.WriteLine("Брокер {0} купляє долари;  Курс долара: {1}", this.Name,
sInfo.USD);
        }
        public void StopTrade()
        {
            stock.RemoveObserver(this);
            stock = null;
        }
```

```
    }

    class Bank : IObserver
    {
        public string Name { get; set; }
        IObservable stock;
        public Bank(string name, IObservable obs)
        {
            this.Name = name;
            stock = obs;
            stock.RegisterObserver(this);
        }
        public void Update(object ob)
        {
            StockInfo sInfo = (StockInfo)ob;

            if (sInfo.Euro > 40)
                Console.WriteLine("Банк {0} продає євро;  Курс євро: {1}", this.Name,
sInfo.Euro);
            else
                Console.WriteLine("Банк {0} купляє євро;  Курс євро: {1}", this.Name,
sInfo.Euro);
        } } }
```

3. Декоратор (Decorator) представляє собою структурний шаблон проектування, який дозволяє динамічно підключати до об'єкта додаткову функціональність.

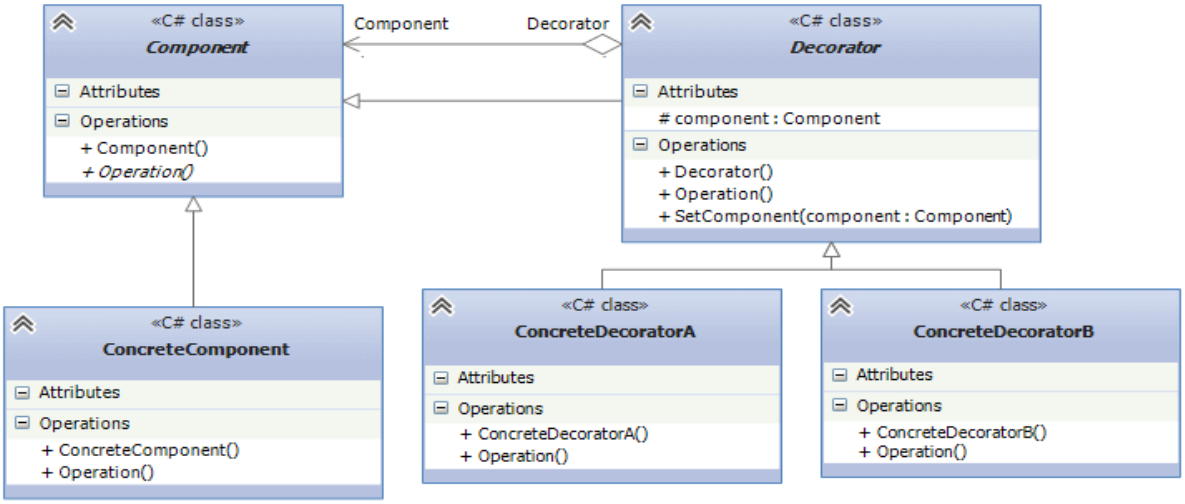
Для визначення нового функціоналу в класах зазвичай використовується спадкування. Декоратори надають спадкуванню більш гнучку альтернативу, оскільки дозволяють динамічно визначати нові можливості об'єктів.

Декоратори використовують у випадках, коли:

- потрібно динамічно додавати до об'єкта нові функціональні можливості. При цьому такі можливості можуть бути зняті з об'єкта

- застосування успадкування неприйнятно. *Наприклад*, якщо нам потрібно визначити множину різних функціональностей і для кожної функціональності успадкувати окремий клас, то структура класів може значно розростатися. Ще більше вона може розростатися, якщо нам необхідно створити класи, що реалізують можливі поєднування функціональностей.

Схематично шаблон "Декоратор" можна виразити таким чином:



Формальна організація патерну в C # може виглядати наступним чином:

```
abstract class Component
   {
      public abstract void Operation();
   }
   class ConcreteComponent : Component
   {
      public override void Operation()
      { }
   }
   abstract class Decorator : Component
   {
      protected Component component;
      public void SetComponent(Component component)
      {
         this.component = component;
      }

      public override void Operation()
      {
         if (component != null)
            component.Operation();
      }
   }
   class ConcreteDecoratorA : Decorator
   {
      public override void Operation()
      {
         base.Operation();
               }
```

Учасники

- **Component**: абстрактний клас, який визначає інтерфейс для успадкованих об'єктів

- **ConcreteComponent**: конкретна реалізація компонента, в яку за допомогою декоратора додається нова функціональність

- **Decorator**: реалізується у вигляді абстрактного класу і має той же базовий клас, що і об'єкти, що декоруються. Тому базовий клас **Component** повинен бути по можливості легким і визначати тільки базовий інтерфейс. Клас декоратора також зберігає посилання на об'єкт, що декорується у вигляді об'єкта базового класу **Component** і реалізує зв'язок з базовим класом як через успадкування, так і через відношення агрегації.

- Класи **ConcreteDecoratorA** і **ConcreteDecoratorB** представляють додаткові функціональності, якими повинен бути розширений об'єкт **ConcreteComponent**.

Розглянемо *приклад*. Припустимо, у нас є піцерія, де готуються різні типи піц з різними складовими. Є італійська, болгарська піци тощо. До кожної з них можуть додаватися помідори, сир та інші складові. У залежності від типу піци і комбінації добавок піца має певну вартість.

Тепер подивимося, як це можна відобразити у програмі на C#:

```
using System;

namespace ConsoleApp
{
   class Program
```

```
{
    static void Main(string[] args)
    {
        Pizza pizza1 = new ItalianPizza();
        pizza1 = new TomatoPizza(pizza1); // італійська піца з томатами
        Console.WriteLine("Назва: {0}", pizza1.Name);
        Console.WriteLine("Ціна: {0}", pizza1.GetCost());

        Pizza pizza2 = new ItalianPizza();
        pizza2 = new CheesePizza(pizza2);// італійська піца з сиром
        Console.WriteLine("Назва: {0}", pizza2.Name);
        Console.WriteLine("Ціна: {0}", pizza2.GetCost());

        Pizza pizza3 = new BulgerianPizza();
        pizza3 = new TomatoPizza(pizza3);
        pizza3 = new CheesePizza(pizza3);// болгарська піца з томатами і сиром
        Console.WriteLine("Назва: {0}", pizza3.Name);
        Console.WriteLine("Ціна: {0}", pizza3.GetCost());

        Console.ReadLine();
    }

    abstract class Pizza
    {
        public Pizza(string n)
        {
            this.Name = n;
        }
        public string Name { get; protected set; }
        public abstract int GetCost();
    }

    class ItalianPizza : Pizza
    {
        public ItalianPizza() : base("Італійська піцца")
        { }
        public override int GetCost()
        {
            return 10;
        }
    }
    class BulgerianPizza : Pizza
    {
        public BulgerianPizza()
            : base("Болгарська піца")
        { }
        public override int GetCost()
        {
            return 8;
        }
    }
```

```
    abstract class PizzaDecorator : Pizza
    {
        protected Pizza pizza;
        public PizzaDecorator(string n, Pizza pizza) : base(n)
        {
            this.pizza = pizza;
        }
    }

    class TomatoPizza : PizzaDecorator
    {
        public TomatoPizza(Pizza p)
            : base(p.Name + ", з томатами", p)
        { }

        public override int GetCost()
        {
            return pizza.GetCost() + 3;
        }
    }

    class CheesePizza : PizzaDecorator
    {
        public CheesePizza(Pizza p)
            : base(p.Name + ", з сиром", p)
        { }

        public override int GetCost()
        {
            return pizza.GetCost() + 5;
        }
    }

    }

}
```

Як компонент тут виступає абстрактний клас Pizza, який визначає базову функціональність у вигляді властивості Name і метод GetCost(). Ця функціональність реалізується двома підкласами ItalianPizza і BulgerianPizza, в яких жорстко закодовані назва піци і її ціна.

Декоратором є абстрактний клас PizzaDecorator, що успадкований від класу Pizza і містить посилання на об'єкт Pizza, який декорується. На відміну від формальної схеми тут установка на об'єкт відбувається не в методі SetComponent, а в конструкторі.

Окремі функціональності - додавання томатів і сиру до піци реалізовані через класи TomatoPizza і CheesePizza, які обгортають об'єкт Pizza і додають до його імені назви добавки, а до ціни - вартість добавки, перевизначаючи метод GetCost і змінюючи значення властивості Name.

Спочатку об'єкт BulgerianPizza обертається декоратором TomatoPizza, а потім CheesePizza. І таких обгортань ми можемо зробити безліч. Достатньо успадкувати від декоратора клас, який буде визначати додатковий функціонал.

А якби ми використовували спадкування, то в даному випадку тільки для двох видів піц з двома добавками нам би довелося створити вісім різних класів, які б описували всі можливі комбінації. Тому декоратори є кращим методом у даному випадку.

ЗАВДАННЯ

1. Продумати тему проекту (3-й семестр) так, щоб в коді можливо було застосувати один із розглянутих вище патернів (Спостерігач або Декоратор). Реалізуйте патерн за відповідною тематикою в новому проекті. Розробити для застосунку графічний інтерфейс (довільним чином).

2. Окремо розробити проект і застосувати в коді патерн Стратегія для наведених нижче алгоритмів. Розробити для застосунку графічний інтерфейс (довільним чином).

Варіанти:

1) сортування (символи, рядки, числа тощо);
2) обчислення площі правильних багатокутників;
3) обчислення периметра довільних багатокутників;
4) обчислення об'єму правильних тіл (наприклад, кулі, призми і куба);
5) обчислення траєкторії для різних видів руху (рівномірний, рівноприскорений);
6) обчислення суми (дійсних чисел, звичайних дробів, комплексних чисел);
7) обчислення різниці (дійсних чисел, звичайних дробів, комплексних чисел);
8) обчислення частки (дійсних чисел, звичайних дробів, комплексних чисел);
9) обчислення найбільшого добутку двох чисел з масиву (дійсних чисел, звичайних дробів, комплексних чисел);
10) обчислення найменшого добутку двох чисел з масиву (дійсних чисел, звичайних дробів, комплексних чисел);
11) обчислення найбільшої суми двох чисел з масиву (дійсних чисел, звичайних дробів, комплексних чисел);
12) обчислення найменшої суми двох чисел з масиву (дійсних чисел, звичайних дробів, комплексних чисел);
13) пошуку значення, яке найчастіше зустрічається в масиві (дійсних чисел, звичайних дробів, комплексних чисел);
14) пошуку всіх значень, які зустрічаються в масиві лише один раз (дійсних чисел, звичайних дробів, комплексних чисел);
15) з масиву вибрати елементи, які зустрічаються більше ніж один раз (дійсних чисел, рядків символів, комплексних чисел).