Дисципліна: Об’єктно-орієнтоване конструювання програм

Лабораторна робота № 3 (4 год.)

Тема: Структурні патерни проєктування

Теоретичні відомості

Структурні патерни спрямовані на компоновку класів і об'єктів в більш складні структури. Патерни рівня класів використовують успадкування для комбінування інтерфейсів і реалізацій у системі. Простим прикладом може бути множинне наслідування інтерфейсів, яке дає змогу організовувати композиції інтерфейсів, або множинне наслідування класів, яке уможливлює створення класів-наслідників, які об’єднують властивості всіх своїх батьківських класів. Структурні патерни особливо корисні у ситуаціях, коли потрібно змусити працювати в одній системі незалежно розроблені один від одного модулі.

Існує 7 структурних патернів:

- Адаптер (Adapter)
- Міст (Bridge)
- Компонувальник (Composite)
- Декоратор (Decorator)
- Фасад (Facade)
- Легковаговик (Flyweight)
- Замісник (Proxy)

Адаптер (Adapter)

Перетворює інтерфейс (як набір властивостей та методів) одного класу в інтерфейс другого класу, який буде очікуватись клієнтами. Цей патерн забезпечує сумісну роботу класів з несумісними інтерфейсами. Адаптер є патерном рівня об’єктів та класів.

Патерн використовується, коли:

- потрібно використати вже існуючий у системі клас, проте його інтерфейс не відповідає очікуванням клієнта;
- потрібно створити клас, що буде повторно використовуватись з класами, що мають несумісні інтерфейси.

Приклад реалізації на C#:

```
public class Compound
{
   protected float boilingPoint;
```

```
    protected float meltingPoint;
    protected double molecularWeight;
    protected string molecularFormula;
    public virtual void Display()
    {
        Console.WriteLine("\nCompound: Unknown ------ ");
    }
}

// The 'Adapter' class
public class RichCompound : Compound
{
    private string chemical;
    private ChemicalDatabank bank;
    public RichCompound(string chemical)
    {
        this.chemical = chemical;
    }

    public override void Display()
    {
        bank = new ChemicalDatabank();

        boilingPoint = bank.GetCriticalPoint(chemical, "B");
        meltingPoint = bank.GetCriticalPoint(chemical, "M");
        molecularWeight = bank.GetMolecularWeight(chemical);
        molecularFormula = bank.GetMolecularStructure(chemical);

        Console.WriteLine("\nCompound: {0} ------ ", chemical);
        Console.WriteLine(" Formula: {0}", molecularFormula);
        Console.WriteLine(" Weight : {0}", molecularWeight);
        Console.WriteLine(" Melting Pt: {0}", meltingPoint);
        Console.WriteLine(" Boiling Pt: {0}", boilingPoint);
    }
}

// The 'Adaptee' class
public class ChemicalDatabank
{
    public float GetCriticalPoint(string compound, string point)
    {
        if (point == "M")
        {
            switch (compound.ToLower())
            {
                case "water": return 0.0f;
                case "benzene": return 5.5f;
                case "ethanol": return -114.1f;
                default: return 0f;
            }
        }
        else
```

```
        {
            switch (compound.ToLower())
            {
                case "water": return 100.0f;
                case "benzene": return 80.1f;
                case "ethanol": return 78.3f;
                default: return 0f;
            }
        }
    }

    public string GetMolecularStructure(string compound)
    {
        switch (compound.ToLower())
        {
            case "water": return "H20";
            case "benzene": return "C6H6";
            case "ethanol": return "C2H5OH";
            default: return "";
        }
    }

    public double GetMolecularWeight(string compound)
    {
        switch (compound.ToLower())
        {
            case "water": return 18.015;
            case "benzene": return 78.1134;
            case "ethanol": return 46.0688;
            default: return 0d;
        }
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Compound unknown = new Compound(); // Non-adapted chemical compound
        unknown.Display();
        // Adapted chemical compounds
        Compound water = new RichCompound("Water");
        water.Display();
        Compound benzene = new RichCompound("Benzene");
        benzene.Display();
        Compound ethanol = new RichCompound("Ethanol");
        ethanol.Display();
        Console.ReadKey();
    }
}
```

Міст (Bridge)

Забезпечує незалежність рівня абстракції класу від елементів його реалізації, так, щоб абстракцію і реалізацію можна було змінювати незалежно одне від одного. Міст є патерном рівня об'єктів.

Патерн використовується, коли:

- потрібно уникнути постійної прив'язки між абстракцією та імплементацією. Наприклад, коли потрібно підмінити імплементацію під час виконання;
- одночасно абстракція та імплементація повинні бути доступними для розширення незалежно один від одного.

Приклад реалізації на C#:

```
public abstract class ReadingApp
{
   protected IDisplayFormatter displayformatter;
   public ReadingApp(IDisplayFormatter displayformatter)
   {
      this.displayformatter = displayformatter;
   }

   public string Text { get; set; }
   public abstract void Display();
}

public class Windows8App : ReadingApp
{
   public Windows8App(IDisplayFormatter displayFormatter) : base(displayFormatter)
   {
   }

   public override void Display()
   {
      displayformatter.Display("This is for Windows8\n" + Text);
   }
}

public class Windows10App : ReadingApp
{
   public Windows10App(IDisplayFormatter displayFormatter) : base(displayFormatter)
   {
   }

   public override void Display()
   {
      displayformatter.Display("This is for Windows10\n" + Text);
   }
}

public interface IDisplayFormatter
```

```
{
    void Display(string text);
}

public class NormalDisplay : IDisplayFormatter
{
    public void Display(string text)
    {
        Console.WriteLine(text);
    }
}

public class ReverseDisplay : IDisplayFormatter
{
    public void Display(string text)
    {
        Console.WriteLine(new String(text.Reverse().ToArray()));
    }
}

class Client
{
    static void Main(string[] args)
    {
        ReadingApp readingApp = new Windows10App(new NormalDisplay()) { Text = "Read this
text" };
        readingApp.Display();

        ReadingApp readingAppon8 = new Windows8App(new NormalDisplay()) { Text = "Read
this text" };
        readingAppon8.Display();

        ReadingApp readingAppR = new Windows10App(new ReverseDisplay()) { Text = "Read
this text" };
        readingAppR.Display();

        ReadingApp readingAppon8R = new Windows8App(new ReverseDisplay()) { Text =
"Read this text" };
        readingAppon8R.Display();

        Console.Read();
    }
}
```

Компонувальник (Composite)

Дозволяє будувати ієрархії із об’єктів у вигляді дерев. Надаючи можливість клієнтському коду однаково трактувати окремі об’єкти (листя) й складені об’єкти (гілки). Компонувальник є патерном рівня об’єктів.

Патерн використовується, коли:

- потрібна можливість за потреби ігнорувати те, що композиції із об’єктів та окремі об’єкти є різними речами та проводити над ними однакові операції;
- потрібно представити ієрархії об’єктів у вигляді «частина-ціле».

Приклад реалізації на C#:

```
public abstract class DrawingElement
{
   protected string name;
   public DrawingElement(string name)
   {
      this.name = name;
   }
   public abstract void Add(DrawingElement d);
   public abstract void Remove(DrawingElement d);
   public abstract void Display(int indent);
}

public class PrimitiveElement : DrawingElement
{
   public PrimitiveElement(string name) : base(name)
   {
   }

   public override void Add(DrawingElement c)
   {
      Console.WriteLine("Cannot add to a PrimitiveElement");
   }

   public override void Remove(DrawingElement c)
   {
      Console.WriteLine("Cannot remove from a PrimitiveElement");
   }

   public override void Display(int indent)
   {
      Console.WriteLine(new String('-', indent) + " " + name);
   }
}

public class CompositeElement : DrawingElement
{
   List<DrawingElement> elements = new List<DrawingElement>();

   public CompositeElement(string name) : base(name)
   {
   }

   public override void Add(DrawingElement d)
   {
      elements.Add(d);
   }
```

```
    public override void Remove(DrawingElement d)
    {
        elements.Remove(d);
    }

    public override void Display(int indent)
    {
        Console.WriteLine(new String('-', indent) + "+ " + name);
        foreach (DrawingElement d in elements)
        {
            d.Display(indent + 2);
        }
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        CompositeElement root = new CompositeElement("Picture");
        root.Add(new PrimitiveElement("Red Line"));
        root.Add(new PrimitiveElement("Blue Circle"));
        root.Add(new PrimitiveElement("Green Box"));
        CompositeElement comp = new CompositeElement("Two Circles");
        comp.Add(new PrimitiveElement("Black Circle"));
        comp.Add(new PrimitiveElement("White Circle"));
        root.Add(comp);
        PrimitiveElement pe = new PrimitiveElement("Yellow Line");
        root.Add(pe);
        root.Remove(pe);
        root.Display(1);
        Console.ReadKey();
    }
}
```

Декоратор (Decorator)

Надає можливість динамічно додавати об’єкту нові обов’язки, доповнюючи новим станом чи поведінкою, через загортання їх у певні обгортки. Декоратор є патерном рівня об’єктів.

Патерн використовується, коли:

- потрібно динамічно додавати та забирати певні обов’язки у об’єктів;
- потрібно передбачити багато комбінацій можливих станів системи, і розширення через наслідування стає непрактичним через надто громіздку ієрархію та часте дублювання коду.

Приклад реалізації на C#:

```
public abstract class LibraryItem
{
   private int numCopies;
   public int NumCopies
   {
      get { return numCopies; }
      set { numCopies = value; }
   }
   public abstract void Display();
}

public class Book : LibraryItem
{
   private string author;
   private string title;
   public Book(string author, string title, int numCopies)
   {
      this.author = author;
      this.title = title;
      this.NumCopies = numCopies;
   }

   public override void Display()
   {
      Console.WriteLine("\nBook ------ ");
      Console.WriteLine(" Author: {0}", author);
      Console.WriteLine(" Title: {0}", title);
      Console.WriteLine(" # Copies: {0}", NumCopies);
   }
}

public class Video : LibraryItem
{
   private string director;
   private string title;
   private int playTime;
   public Video(string director, string title, int numCopies, int playTime)
   {
      this.director = director;
      this.title = title;
      this.NumCopies = numCopies;
      this.playTime = playTime;
   }

   public override void Display()
   {
      Console.WriteLine("\nVideo ----- ");
      Console.WriteLine(" Director: {0}", director);
      Console.WriteLine(" Title: {0}", title);
      Console.WriteLine(" # Copies: {0}", NumCopies);
      Console.WriteLine(" Playtime: {0}\n", playTime);
   }
```

```
}

public abstract class Decorator : LibraryItem
{
    protected LibraryItem libraryItem;
    public Decorator(LibraryItem libraryItem)
    {
        this.libraryItem = libraryItem;
    }

    public override void Display()
    {
        libraryItem.Display();
    }
}

public class Borrowable : Decorator
{
    protected readonly List<string> borrowers = new List<string>();

    public Borrowable(LibraryItem libraryItem)
        : base(libraryItem)
    {
    }

    public void BorrowItem(string name)
    {
        borrowers.Add(name);
        libraryItem.NumCopies--;
    }

    public void ReturnItem(string name)
    {
        borrowers.Remove(name);
        libraryItem.NumCopies++;
    }

    public override void Display()
    {
        base.Display();

        foreach (string borrower in borrowers)
        {
            Console.WriteLine(" borrower: " + borrower);
        }
    }
}

public class Program
{
    public static void Main(string[] args)
    {
```

```
        // Create book
        Book book = new Book("Worley", "Inside ASP.NET", 10);
        book.Display();

        // Create video
        Video video = new Video("Spielberg", "Jaws", 23, 92);
        video.Display();

        // Make video borrowable, then borrow and display
        Console.WriteLine("\nMaking video borrowable:");
        Borrowable borrowvideo = new Borrowable(video);
        borrowvideo.BorrowItem("Customer #1");
        borrowvideo.BorrowItem("Customer #2");
        borrowvideo.Display();
        Console.ReadKey();
    }
}
```

Фасад (Facade)

Забезпечує наявність одного уніфікованого інтерфейсу високого рівня до певної кількості інших інтерфейсів у підсистемі. Фасад є патерном рівня об’єктів.

Патерн використовується, коли:

- потрібно полегшити використання складної підсистеми, надаючи клієнтам простий інтерфейс для часто повторюваних операцій, що рідко потребують кастомізації;
- потрібно надати один високорівневий інтерфейс, щоб сховати або замінити багато низькорівневих інтерфейсів;
- потрібно розбити систему на рівні та надати одну уніфіковану точкувходу на рівень.

Приклад реалізації на C#:

```
public class SubSystemOne
{
    public void MethodOne()
    {
        Console.WriteLine(" SubSystemOne Method");
    }
}

public class SubSystemTwo
{
    public void MethodTwo()
    {
        Console.WriteLine(" SubSystemTwo Method");
    }
}
```

```
public class SubSystemThree
{
    public void MethodThree()
    {
        Console.WriteLine(" SubSystemThree Method");
    }
}

public class SubSystemFour
{
    public void MethodFour()
    {
        Console.WriteLine(" SubSystemFour Method");
    }
}

public class Facade
{
    SubSystemOne one;
    SubSystemTwo two;
    SubSystemThree three;
    SubSystemFour four;
    public Facade()
    {
        one = new SubSystemOne();
        two = new SubSystemTwo();
        three = new SubSystemThree();
        four = new SubSystemFour();
    }
    public void MethodA()
    {
        Console.WriteLine("\nMethodA() ---- ");
        one.MethodOne();
        two.MethodTwo();
        four.MethodFour();
    }
    public void MethodB()
    {
        Console.WriteLine("\nMethodB() ---- ");
        two.MethodTwo();
        three.MethodThree();
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Facade facade = new Facade();
        facade.MethodA();
        facade.MethodB();
```

```
        Console.ReadKey();
    }
}
```

Легковаговик (Flyweight)

Дозволяє більш ефективно зберігати об’єкти, заощаджуючи ресурс оперативної пам’яті. Об’єкти, що мають однаковий стан, не зберігають у собі зайві дані. Легковаговик є патерном рівня об’єктів.

Патерн використовується, коли:

- система оперує великою кількістю об’єктів;
- потрібно особливо дбайливо ставитись до ресурсу фізичної пам’яті та одночасно унікальність та точна ідентифікація об’єктів є неважливими.

Приклад реалізації на C#:

```
public class CharacterFactory
{
    private Dictionary<char, Character> characters = new Dictionary<char, Character>();

    public Character GetCharacter(char key)
    {
        Character character = null;
        if (characters.ContainsKey(key))
        {
            character = characters[key];
        }
        else
        {
            switch (key)
            {
                case 'A': character = new CharacterA(); break;
                case 'B': character = new CharacterB(); break;
                //...
                case 'Z': character = new CharacterZ(); break;
            }
            characters.Add(key, character);
        }
        return character;
    }
}

// The 'Flyweight' abstract class
public abstract class Character
{
    protected char symbol;
    protected int width;
    protected int height;
    protected int ascent;
    protected int descent;
```

```
    protected int pointSize;
    public abstract void Display(int pointSize);
}

// A) ConcreteFlyweight class
public class CharacterA : Character
{
    public CharacterA()
    {
        symbol = 'A';
        height = 100;
        width = 120;
        ascent = 70;
        descent = 0;
    }
    public override void Display(int pointSize)
    {
        this.pointSize = pointSize;
        Console.WriteLine(symbol + " (pointsize " + this.pointSize + ")");
    }
}

// B) ConcreteFlyweight class
public class CharacterB : Character
{
    public CharacterB()
    {
        symbol = 'B';
        height = 100;
        width = 140;
        ascent = 72;
        descent = 0;
    }

    public override void Display(int pointSize)
    {
        this.pointSize = pointSize;
        Console.WriteLine(this.symbol + " (pointsize " + this.pointSize + ")");
    }
}

// ... C, D, E, etc.
// Z) ConcreteFlyweight class
public class CharacterZ : Character
{
    public CharacterZ()
    {
        symbol = 'Z';
        height = 100;
        width = 100;
        ascent = 68;
        descent = 0;
```

```
    }

    public override void Display(int pointSize)
    {
        this.pointSize = pointSize;
        Console.WriteLine(this.symbol + " (pointsize " + this.pointSize + ")");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        // Build a document with text
        string document = "AAZZBBZB";
        char[] chars = document.ToCharArray();
        CharacterFactory factory = new CharacterFactory();
        int pointSize = 10;   // extrinsic state
        // For each character use a flyweight object
        foreach (char c in chars)
        {
            pointSize++;
            Character character = factory.GetCharacter(c);
            character.Display(pointSize);
        }
        Console.ReadKey();
    }
}
```

Замісник (Proxy)

Забезпечує наявність об’єкта-замісника для контролю доступу до іншого об’єкта-оригіналу. Замісник є патерном рівня об’єктів.

Патерн використовується, коли:

- потрібно скрити реальне розташування об’єкта-оригіналу в пам’яті від клієнта (віддалений замісник);
- потрібно створювати ресурсоємні об’єкти за запитом, реалізуючи відкладену ініціалізацію (віртуальний замісник);
- потрібно проводити додаткові операції до чи після передачі виклика об’єкту-оригіналу, наприклад, верифікацію або валідацію (захисний замісник).

Приклад реалізації на C#:

```
// Клас об'єкта-оригіналу
public class ForexProvider
{
    private bool isAutheticated;
    public string Request()
    {
```

```
        return "Request completed";
    }

    public string GetPrice(string currencyName)
    {
        return string.Format("The forex rate is {0} for {1}", 23, currencyName);
    }

    public string SetPrice(string currencyName, int value)
    {
        if (isAutheticated)
            return string.Format("The forex rate of {0} is set to {1}", currencyName, value);
        else
            return string.Format("Please autheticate yourself");
    }

    public bool Authenticate(string pwd)
    {
        if (pwd == "pwd")
        {
            isAutheticated = true;
            return true;
        }
        else
            return false;
    }
}

// Інтерфейс для доступу до об'єкта-оригіналу
public interface IForexProvider
{
    string Request();
    string GetPrice(string currencyName);
    string SetPrice(string currencyName, int value);
    bool Authenticate(string pwd);
}

// Замісник
public class ForexProviderProxy : IForexProvider
{
    private ForexProvider forexProvider;

    public bool Authenticate(string pwd)
    {
        if (forexProvider == null)
            forexProvider = new ForexProvider();
        return forexProvider.Authenticate(pwd);
    }

    public string GetPrice(string currencyName)
    {
        if (forexProvider == null)
```

```
            forexProvider = new ForexProvider();
        return forexProvider.GetPrice(currencyName);
    }

    public string Request()
    {
        if (forexProvider == null)
            forexProvider = new ForexProvider();
        return forexProvider.Request();
    }

    public string SetPrice(string currencyName, int value)
    {
        if (forexProvider == null)
            forexProvider = new ForexProvider();
        return forexProvider.SetPrice(currencyName, value);
    }
}

// Клієнт нічого не знає про об'єкт-оригінал і має справу лише з замісником
class Client
{
    static void Main(string[] args)
    {
        IForexProvider forex = new ForexProviderProxy();
        forex.Request();
        forex.SetPrice("Indian Rupee", 65);

        Console.Read();
    }
}
```

ЗАВДАННЯ

Розробити класи (згідно варіанту) з використанням структурного патерну проєктування. Реалізувати програму на C# з простим графічним інтерфейсом. У висновках обґрунтувати вибір патерну і пояснити, яким чином він дав змогу спростити виконання вашого варіанту завдання.

Варіанти:

1	Зберігання інформації про електронний документ: назва, автор, розмір файлу, тип файлу, шлях до файлу. Методи отримання інформації про документ: виведення у вигляді таблиці на екран, копіювання в буфер обміну, зберігання в текстовий файл. Пошук документів за автором, за повною назвою або за її частиною. Передбачити можливість додавання нового виду отримання інформації про документ (наприклад, зберігання в форматі html).
2	Програма обліку персоналу в організації. Організація має директора, заступника директора і складається з відділів (підрозділів). Кожен відділ має керівника, заступників, спеціалістів, службовців і робітників. Передбачити ієрархію відділів та ієрархію працівників у відділі. Методи отримання інформації про

	ієрархію організації і про будь-якого її відділу. Методи прийняття на роботу та звільнення працівників, переведення з одного відділу до іншого, зміни посади. Передбачити можливість створення або ліквідації відділу.
3	Програма обліку товарів на складі. Характеристики товарів: назва, вартість, дата надходження, вага, габарити (ШхГхВ). Методи прийому товару на склад та відвантаження зі складу. Методи отримання інформації про всі товари на складі. Пошук товарів за вибраною характеристикою. Підрахунок кількості товару з однаковою назвою, не враховуючи інші характеристики. Передбачити можливість додавання характеристик товарів (наприклад, температура зберігання).
4	Програма для каво-машини, яка може з певного набору інгредієнтів готувати: еспресо, американо, латте, капучіно, какао, гарячий шоколад. Методи вибору виду напою, кількості цукру, розрахунку ціни залежно від інгредієнтів. Передбачити можливість додавання нового виду напою.
5	Робота поштового відділення. Характеристики посилок: відправник, отримувач, відстань, дата відправлення, дата доставки. Характеристики відправника або отримувача: ім'я, адреса, номер телефону. Методи відправлення і отримання посилки. Метод повернення посилки відправнику. Розрахунок вартості доставки від ваги та відстані. Передбачити можливість відправлення або отримання декількох посилок.
6	Он-лайн магазин автомобільних ламп. Типи ламп: світло-діодні, галогенні, ксенонові. Популярні цоколі ламп: D1R, D1S, D2R, D2S, D4R, D4S, H1, H10, H11, H13, H7, H8, H9, HB3, HB4. Лампи різняться за потужністю і температурою світіння. Передбачити надходження до магазину ламп з іншим цоколем. Передбачити можливість розширення асортименту, наприклад, лампи для мотоциклів.
7	Програма вибору вантажного автомобіля для перевезення вантажів. Врахувати різні типи кузова вантажних автомобілів і різні типи причепів. Вантажопідйомність транспортного засобу (зазвичай від 1 до 25 тонн). Метод розрахунку кількості автомобілів/причепів для перевезення заданої ваги вантажу. Врахувати можливість залучення іншого виду транспорту, наприклад легкових автомобілів з типом кузова "універсал".
8	Онлайн-калькулятор покрівлі приватного будинку. Тип даху: односхилий, двосхилий, чотирисхилий. Види покрівлі: рулонні, штучні, листові, плівкові, мастикові. Врахувати що різні види покрівлі виготовляють з різних матеріалів. Розрахувати необхідну кількість упаковок/рулонів/відер/штук вибраного матеріалу для покрівлі залежно від розмірів даху.
9	Проектування системи крапельного поливу теплиць. Залежно від розмірів теплиці підібрати: діаметри і довжини магістральних трубопроводів, довжини трубок і стрічок, кількість крапельниць, робочий тиск і продуктивність насосу, кількість сполучної і запірної фурнітури, необхідної для створення розгалуженої системи крапельного зрошення.

Додаткова література

1. Електронний ресурс. Джерело доступу:

https://learn.ztu.edu.ua/pluginfile.php/632/mod_resource/content/1/DesignPatterns_AndriyBuday.pdf

2. Електронний ресурс. Джерело доступу:

https://refactoring.guru/uk/design-patterns/structural-patterns