C# 版本

1. The Strategy Pattern :

使用時機 : 在 Runtime 時候 才確定使用 方法

範例 : (電玩設計) 主角 選者 武器 攻擊敵人

優點 : 容易 維護, 如果有多一個 武器, 基本上只要 新增一個 武器 class 和 追加 switch 選項, 其他的部分, 幾乎不用修改

```
IWeapon weapon = null;
Character ch = new Character();

for (int i = 0; i < 5; i++)
{
    Console.WriteLine("Choose Weapon 1: Sword 2: Club 3: Axe");
    String item = Console.ReadLine();

    // 此為StrategyPattern 核心 技術
    switch (item)
    {
        case "1":
            weapon = new Sword();
            break;
        case "2":
            weapon = new Club();
            break;
        case "3":
            weapon = new Axe();
            break;
        // default 設定 寫法, 為  Null Object Pattern
        default :
            weapon = new Null();
            break;

    }

    ch.SetWeapon(weapon);
    ch.Attck();
}
```

2. Null Object Pattern :並非 Gof 提出的 23 Design Pattern 的其中一個

使用時機 : 當在 動態生成的時候, 如果有發生意外 非指定的 class 去實作, 就把此實作 生成 NULL, 也就是說一個 Null class 去實作但是 只要顯示 警告 or do nothing !!

優點 : 避免 Runtime exception, 一般用在 Switch 的 default 使用

(實際程式碼, 請參考上面Strategy 圖片)

3. Singleton Pattern

使用時機 ： 希望確保一個類別只能有一個 實體 時使用。

範例: log system 只要寫入一個檔案就可以，不需要分散到各處

核心概念 : (1)建構子 要是 private,

(2) 改用 Get static Instance 的方法取代 建構子,在此 method 裡面判斷，有資料 就回傳現有，沒有，產生一個新的，就是要確保 只有一個 instance !!

```
class Logger
{
    private static Logger logger = null; // 改成 static 的 instance, 確保只有一個 instance

    private Logger() { } // 基本上 建構子 要是 private, 且 設定為 空, 因為不會使用此 建構子

    // 要設定為 static 的 method
    public static Logger getInstance()
    {
        // 只有 Logger 回 Null 才要生成, 不然就回傳現有的
        if (logger == null)
        {
            logger = new Logger();
        }

        return logger;
    }

    public void WriteToFile(String msg)
    {
        Console.WriteLine(msg);
    }
}
```

4. Observer Pattern (或稱 publish/subscribe pattern)

使用時機 ： 個人感覺就是有 Event 的概念/前身

範例: (1) 當 FB 某人更新資料，請立即通知我

(2) GUI 元件設計使用最多，例如 當滑鼠按下去，請執行特定功能

```
class YouTubeChannel
{
    private List<ISubscriber> Subscribers = new List<ISubscriber>();// Genertic 和 interface 的搭配

    public void Subscribe(ISubscriber s)
    {
        Subscribers.Add(s);
    }

    public void UnSubscribe(ISubscriber s)
    {
        Subscribers.Remove(s);
    }

    // 通知所有訂閱的人
    public void NotifyObervers()
    {
        foreach (ISubscriber s in Subscribers)
        {
            s.Notify();
        }
    }
}
```

5. Iterator Pattern

使用時機 ： 可以做不同型態資料的資料查詢

範例 :當兩家公司資料整合時候(使用的資料存取方法不同, 一邊用 Sting[] 另一邊用 ArrayList), 用最少修改程式的方法做 資料查詢

兩個關鍵地方 : (1) Iterator 介面

```
/**
 * Iterator 的核心 : 要有四個基本 method, 作相關查詢所需要的功能
 */
public interface IIterator
{
    void First();// Reset back to the 1st element
    string Next();// Get next element :  回傳目前資料, 並且把 position 指定到下一個
    bool isDone();// End of Collection check : 確認是否為最後一個 element
    string CurrentItem();// Get current item : 回傳目前位置, 但是 position 不需要改變
}
```

(2) 還需要另一個 介面, 負責 整合 公司資料 產生一個 Iterator, 基本上只會有一個方法 : public IIterator CreateIterator()

```
/**
 * 此為 合併兩家公司 的介面, 透過 IIterator 來整合, 這是 IIterator 重要的其中一個介面
 */
public interface ISocialNetworking
{
    IIterator CreateIterator();// 把目前資料 轉成 Iterator
}
```

應用部分 :

```
// 分別顯是所有 IIterator 的資料
public static void PrintUsers(IIterator iterator)
{
    iterator.First();// 先重設定

    while ( !iterator.isDone() )// 判決是否為最後一個, 要有 ! 否定
    {
        Console.WriteLine(iterator.Next());// 取出目前所有資料, 並且把 position 改到下一個
    }
}
```

6. Decorator Pattern

使用時機 : 在執行過程中, 可以 [選者性] 加入功能 到物件中

範例 : 求一個冰淇淋價錢

1. 有基本款的價錢 : 草莓冰淇淋, 藍莓冰淇淋, 百香果冰淇淋 (ConcreteComponent)

2. 每一個 配料 都有 自己的 價錢, 不含 任何 種類的 冰淇淋 (ConcreteDecorator)

3. 最後 價錢 : 就看 客戶 是否有需要 加上 配料

3.1 如果沒有任何配料, 就是 基本款 價錢

3.2 如果有配料, 且 配料 可以同時使用, 就 基本款 價錢 + 每一個 配料 的 價錢

關鍵技術 : 使用 Interface, abstract class, 多型, 建構子 和 recursion 的方式

(1) 價錢功能抽出來變成 abstract class

```
public abstract class IceCream
{
    public abstract double Cost();// 因為每一項的價錢都不同
}
```

(2) 基本款 class

```
public class Chocolate : IceCream
{
    // Chocolate 的價錢 已經包含 IceCream
    // 基本上 Chocolate 可以看成 最根本 price
    public override double Cost()
    {
        return 1;
    }
}
```

(3) 配料 class(重點在於 recursion 寫法)

```
public class Sprinkle : Topping
{
    // 使用父類別的 建構子
    public Sprinkle(IceCream s) : base(s) { }

    // 0.25 只有算 Sprinkle 本身價錢, 所以還要補上 IceCream 的相關 class 的價錢
    public override double Cost()
    {
        // base 這關鍵字 可以省略
        // 類似 Recursion 遞回的概念
        // 為 關鍵技巧
        return 0.25 + base.IceCream.Cost();
    }
}
```

(4) 應用部分 ：

```
static void Main(string[] args)
{
    IceCream iceCream = new Chocolate();

    // 要在 Chocolate 上 追加 Sprinkle 和 Fudge
    // Chocolate 原本就是 1 元;  Sprinkle 要 0.25 元; Fudge 要 0.5 元
    iceCream = new Sprinkle(iceCream); //這行就是把 Chocolate + Sprinkle 整合, 在存回 iceCream; 記住 建構子中用 base(s)
    iceCream = new Fudge(iceCream); // 這行再把  Chocolate + Sprinkle 在追加 Fudge, 在存回 iceCream

    //   iceCream.Cost() 這邊是 recursion 的方式
    // 第一先 呼叫 Fudge class 的 Cost() :  0.5 + base.IceCream.Cost();
    // 第二步 呼叫 Sprinkle class 的 Cost()  : 0.5 + 0.25  + base.IceCream.Cost();
    // 第三步 呼叫 Chocolate 的 Cost() :  0.5 + 0.25  + 1;
    // 此時再把 所有金額合併 起來 !!
    Console.WriteLine("the cost of Chocolate with Sprinkle and Fudge : " + iceCream.Cost());

    IceCream iceCream2 = new Vanilla();
    iceCream2 = new Sprinkle(iceCream2);

    // 第一步 呼叫 Sprinkle class 的 Cost() :  0.25  + base.IceCream.Cost();
    // 第二步 呼叫 Vanilla class 的 Cost() : 0.25 + 1.5
    // 此時再把 所有金額合併 起來 !!
    Console.WriteLine("the cost of Vanilla with Sprinkle : " + iceCream2.Cost());
    Console.ReadKey();
}
```

7. Facade Pattern (基本 reuse 概念, 放在 方法中)

使用時機 : 在不同的 class 發現有相同的重複 部分, 改成 method 取代, 達到 reuse, 一般可以用在做 Library 部分

(1) Façade class

```
public class PizzaFacade
{
    private Dough dough;
    private Sauce saurce;
    private Topping topping;
    private Cheese cheese;
    private Oven oven;

    public PizzaFacade(Dough dough, Sauce saurce, Topping topping, Cheese cheese, Oven oven)
    {
        this.dough = dough;
        this.saurce = saurce;
        this.topping = topping;
        this.cheese = cheese;
        this.oven = oven;
    }

    public void MakePizza()
    {
        dough.AddSauce(saurce);
        dough.AddCheese(cheese);
        dough.AddTopping(topping);

        oven.SetTemperature(425);
        oven.SetTimer(20);
        oven.Cook(dough);
    }
}
```

(2) 應用部分

```
// 使用 Facade Pattern
public void Client1Facade()
{
    Dough dough = new Dough();
    Sauce saurce = new Sauce("Tomatoe");
    Topping mushroom = new Topping("Mushroom");
    Cheese mozzarella = new Cheese("Mozzarella");
    Oven oven = new Oven();

    // 基本上對 Client 來說, 重點是 取的 Pizza, 至於要如何實作 : 對 Client 來說  根本不重要 ( 資料封裝 ) !!
    PizzaFacade pf = new PizzaFacade(dough, saurce, mushroom, mozzarella, oven);
    pf.MakePizza();
}

// 使用 Facade Pattern
public void Client2Facade()
{
    Dough dough = new Dough();
    Sauce saurce = new Sauce("Tomatoe");
    Topping greenPepper = new Topping("GreenPepper");// 另一種  Topping
    Cheese whileCheese = new Cheese("whileCheese");// 另一種 Cheese
    Oven oven = new Oven();

    PizzaFacade pf = new PizzaFacade(dough, saurce, greenPepper, whileCheese, oven);
    pf.MakePizza();
}
```

8. Template Method Pattern (基本 reuse 概念, 封裝成 父類別)

使用時機 : (1) 將不變的部分移到父類別，去除 子類別 重覆的程式碼

(2) 制訂一些規格讓子類別遵守，減少程式碼重複，而子類別可用不同方式去實作方法

關鍵技術 :

(1) 定義一個 abstract 父類別，把 子類別 都要用到的功能 放入這裡面

(2) 子類別 繼承 並實做 父類別 的方法

(3) 用戶端用 父類別型別變數 存放 子類別實體，再呼叫其方法

Facade Pattern V.S. Template Method Pattern

基本 reuse 概念, 放在 方法中 V.S. 基本 reuse 概念, 封裝成 父類別

9. Adapter Pattern

使用時機 : 當兩個 class 的介面呼叫方法不同, 寫一個 class 當作中間的轉換

範例: (1) 一個舊型的 滑鼠 只有支援 PS/2 的 port , 目前 新電腦 只有 USB port, 所以 我們需要一個轉換器, 可以同時連接 PS/2 的 port 和 USB port

(2) 語言翻譯, 一方只有懂 中文, 另一方只懂 英文, 所以還有一個人 可以懂 雙方的語言

```
public class USBAdapter
{
    Mouse mouse = new Mouse();

    // 在此地方做一個 bridge 去聯結舊的程式碼
    public void NewConnect()
    {
        mouse.OldConnect();
        Console.WriteLine("Convert signal to USB");
        Console.WriteLine("Sending new convered signals to the computer");
    }
}
```

10. Command Pattern

使用時機 : 將一個請求封裝為一個物件，讓你可用不同的請求對客戶進行參數化

範例 :

關鍵技術 : 三個重要 class :

(1) Command class

```
public interface ICommand
{
    void Execute();
}
```

(2) Invoker class

```
// Invoker 基本上  就是負責發出  命令
public class RemoteController
{
    private List<ICommand> turnOnCommands = new List<ICommand>();// AC on and Light on
    private List<ICommand> turnOffCommands = new List<ICommand>();// AC off and Light off

    // 把所有  屬於 turnOnCommand 集中起來
    public void InsertNewOnCommand(ICommand command)
    {
        turnOnCommands.Add(command);
    }

    //  把所有  屬於 turnOffCommand 集中起來
    public void InsertNewOffCommand(ICommand command)
    {
        turnOffCommands.Add(command);
    }

    public void PressButtonOn(int buttonNumber)
    {
        turnOnCommands[buttonNumber].Execute();
    }

    public void PressButtonOff(int buttonNumber)
    {
        turnOffCommands[buttonNumber].Execute();
    }
}
```

(3) Receiver class

```
// receiver 執行命令的物件
// 提供 AirConditioner class 相關的功能
public class AirConditioner
{
    public void TurnOn()
    {
        Console.WriteLine("Air Conditioner turns on");
    }

    public void TurnOff()
    {
        Console.WriteLine("Air Conditioner turns off");
    }

    public void InreaseTemp()
    {
        Console.WriteLine("Increasing temperature");
    }

    public void DecreaseTemp()
    {
        Console.WriteLine("Decreasing temperature");
    }

}
```

11. Template Method Pattern (基本 reuse 概念, 封裝成 父類別)

使用時機 : (1) 將不變的部分移到父類別，去除 子類別 重覆的程式碼

(2) 制訂一些規格讓子類別遵守，減少程式碼重複，而子類別可用不同方式去實作方法

關鍵技術 :

(1) 定義一個 abstract 父類別，把 子類別 都要用到的功能 放入這裡面

(2) 子類別 繼承 並實做 父類別 的方法

(3) 用戶端用 父類別型別變數 存放 子類別實體，再呼叫其方法

Facade Pattern V.S. Template Method Pattern

基本 reuse 概念, 放在 方法中 V.S. 基本 reuse 概念, 封裝成 父類別

12. Composite Pattern

使用時機 : 使用 Tree structure 去存取具有 [層次性] 或 [組合性] 的物件

範例 : (1) 播放清單 -> 子播放清單 ->清單內歌曲

(2) 常用於 UI 視窗介面設計 (UI內有其他UI)

關鍵技術 : 共三個重要 class : Component, Composite 和 leaf node

(1) Component : 就是 tree 的 root (EmployeeComponent)

```
// 設定為 abstract class 的 Component
abstract class EmployeeComponent
{
    public string Name { private set; get; }

    public EmployeeComponent(String name)
    {
        this.Name = name;
    }

    // 這邊可以任何 method 去繼承
    public virtual void PrintSupervisorOf(int spaceing)
    {
        for (int i = 0; i < spaceing; i++)
        {
            Console.Write("     ");
        }
        Console.WriteLine("Name : " + this.Name);
    }
}
```

(2) Composite : 就是 tree 的 node (EmployeeComposite)

```
// 在 Composite 部分要提供 add (Component), remove (Component) and getChild(int) 等方法
class EmployeeComposite : EmployeeComponent
{
    private IList<EmployeeComponent> employees;

    public EmployeeComposite(string name) : base(name)
    {
        employees = new List<EmployeeComponent>();
    }

    public void AddEmployee(EmployeeComponent e)
    {
        employees.Add(e);
    }

    public void RemoveEmployee(EmployeeComponent e)
    {
        employees.Remove(e);
    }

    public override void PrintSupervisorOf(int spaceing)
    {
        base.PrintSupervisorOf(spaceing);

        foreach (EmployeeComponent e in employees)
        {
            e.PrintSupervisorOf(spaceing + 1);
        }
    }
}
```

(3) leaf node : 就是沒有 children 的 component (EmployeeLeaf)

```
// 基本上 Leaf node 只要實作 Component 即可
// 不需要有 add/remorve 等方法, 因為已經是最後一個 node, 所以不需要做額外 add/remorve
class EmployeeLeaf : EmployeeComponent
{
    public EmployeeLeaf(string name) : base(name)
    {

    }

    public override void PrintSupervisorOf(int spaceing)
    {
        base.PrintSupervisorOf(spaceing);
    }
}
```

13. Memento Pattern

使用時機：提供 undo/rollback 功能

例如：(1) DB 的資料要 rollback

(2) word 檔案 編輯 要做 undo 功能

關鍵技術：兩個重要 class： Memento 和 Originator

(1) Memento

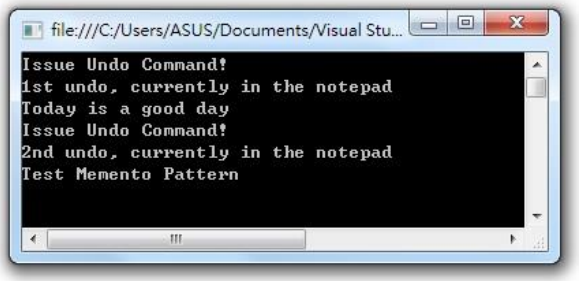
```
// Memento 就是 記錄 任何資料 改變原有的步驟 !
public class Memento
{
    // 記錄原本的資料, 所以資料格式 因該要和 Origintor 的資料格式 相同 !
    private string text;

    public Memento(string text)
    {
        this.text = text;
    }

    public string GetText()
    {
        return text;
    }
}
```

(2) Originator

```
class NotepadOriginator
{
    private string text;//目前的資料

    // 紀錄目前資料 傳給 Memento
    public Memento SetText(string text)
    {
        Memento me = new Memento(text);
        this.text = text;

        return me;
    }

    // 取的目前的資料
    public string GetText()
    {
        return this.text;
    }

    // 執行 undo 功能
    public void Undo(Memento previousText)
    {
        this.text = previousText.GetText();
    }
}
```

(3) 應用部分

```
// Main 部分 就是 caretaker !
static void Main(string[] args)
{
    // undo 清單, 一般來說 第一個 undo 可以為 空物件!! 表示回到未建立任何資料狀態, 此需求看情況是否要加入
    IList<Memento> undoList = new List<Memento>();
    NotepadOriginator notepad = new NotepadOriginator();
    Memento undo;

    // 建立  第一個 undo 可以為 空物件 (沒必要, 看需求)
    undo = notepad.SetText("");
    undoList.Add(undo);

    // first version
    undo = notepad.SetText("Test Memento Pattern");// 輸入一些資料做測試
    undoList.Add(undo);// 加入 undo 清單

    // 2nd version
    undo = notepad.SetText("Today is a good day");// 輸入一些資料做測試
    undoList.Add(undo);// 加入 undo 清單

    // 1st 執行 undo
    Console.WriteLine("Issue Undo Command!");
    notepad.Undo(undoList[2]);

    Console.WriteLine("1st undo, currently in the notepad");
    Console.WriteLine(notepad.GetText());

    // 2nd 執行 undo
    Console.WriteLine("Issue Undo Command!");
    notepad.Undo(undoList[1]);

    Console.WriteLine("2nd undo, currently in the notepad");
    Console.WriteLine(notepad.GetText());

    Console.ReadKey();
}
```



14. Prototype Pattern

使用時機: 有複製物件的功能需求時 (但是只有小部分要修改)

理由 : 之所以不要都用 new instance 的方式, 因為在建構子 參數可能會很多, 導致每次都要輸入大量相同的資料, 以浪類資源, Clone : is much easier and lots of less expensive than creating a new instance

關鍵技術

(1) 建立 abstract class，內有一個 return 此父類別的 clone 方

```
// 要宣告成 abstract class
public abstract class PS4Game
{
    public string Title { get; set; }
    public string ProductKey { get; set; }

    public abstract PS4Game Clone();// 最重要的 method


    // 產生 大寫的 AQEFF-DRGJA-KIJDF-IQDFY 類似的 key
    public static string ProductKeyGeneration()
    {
        Random r = new Random();
        StringBuilder productKey = new StringBuilder();

        for (int i = 0; i < 20; i++)
        {
            if (i % 5 == 0 && i != 0)
            {
                productKey.Append("-");
            }

            // 產生 ASCII code
            productKey.Append((char)(r.Next(26) + 65));
        }

        return productKey.ToString();
    }
}
```

(2) 子類別實作 clone 方法(在 .NET 裡只要實做 Icloneable 就可以，第一步可省略)

```
public class FinalFantasy : PS4Game
{
    public FinalFantasy(string title)
    {
        Title = title;
    }

    public override PS4Game Clone()
    {
        /*
         * The MemberwiseClone method creates a shallow copy by creating a new object,
         * and then copying the nonstatic fields of the current object to the new object.
         *
         * If a field is a value type, a bit-by-bit copy of the field is performed.
         *
         * If a field is a reference type, the reference is copied but the referred object is not;
         * therefore, the original object and its clone refer to the same object.
         */
        return (PS4Game)this.MemberwiseClone();
    }
}
```

3. 用戶端可呼叫子類別的 clone 方法複製出一個相同的子類別

Factory Pattern V.S. Prototype Pattern

Factory Pattern : create a new instance !

Prototype Pattern : create a Prototype or essentially a new clone, not a new object