.NET 技術講座

認識 SOLID 物件導向程式設計原則

多奇數位創意有限公司

技術總監 黃保翕 (Will 保哥)

http://blog.miniasp.com/

在程式開發的過程會遇到各種情境

- 開發的時間較長？還是維護的時間較長？
- 有團隊一起進行開發？還是一個人寫 Code？
- 一個長期維護的專案，需求變更的頻繁程度？
- 你如何讓程式碼具備有**可讀性**與**擴充性**？
- 如何避免在修改程式的過程中引發連鎖反應？(改A壞B)

OOP 的四個特性

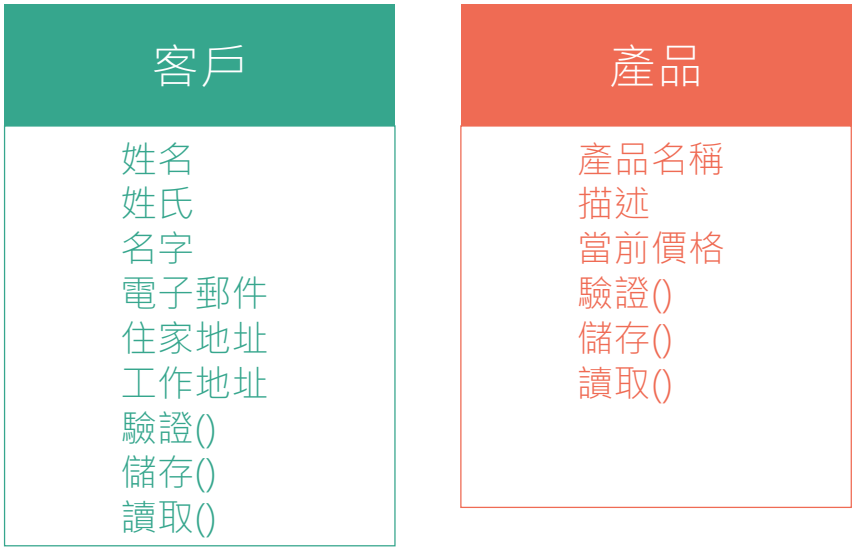
OOP =物件導向程式設計

物件導向程式設計的 4 個重要特性

- **抽象 (Abstraction)**
  - 將真實世界的需求轉換成為 OOP 中的類別
  - **類別**可以包含**狀態**(屬性)與**行為**(方法)。
- **封裝 (Encapsulation)**
  - 隱藏/保護內部實作的細節，並可以對屬性或方法設定存取層級 (public, private, protected)。
- **繼承 (Inheritance)**
  - 可讓您建立新類別以重複使用、擴充和修改其他類別中定義的行為。
- **多型 (Polymorphism)**
  - 在相同的介面下，可以用不同的型別來實現。
  - 多型有分成好幾種不同類型。

從需求或規格中的進行 "抽象化" 的過程

- 我們將建立新客戶管理系統
- 該系統必須管理商業，住宅，政府和教育類型的客戶
- 我們必須最小程度地紀錄客戶的姓名，姓氏，名字，電子郵件地址以及住家地址和工作地址
- 它必須管理產品，我們必須最小程度地記錄產品名稱、描述和當前價格



透過 "抽象化" 過程定義出類別

```
public class Customer
{
    public int CustomerId { get; private set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public string EmailAddress { get; set; }
    public string HomeAddress { get; set; }
    public string WorkAddress { get; set; }
    public bool Validate() { return true; }
    public void Save() { }
    public void Load() { }
}

public class Product
{
    public int ProductId { get; private set; }
    public string ProductName { get; set; }
    public string ProductDescription { get; set; }
    public decimal? CurrentPrice { get; set; }
}
```

對實作的細節進行 "封裝" (隱藏、保護)

```
public class Product
{
    public Product(int productId)
    {
        this.ProductId = productId;
    }
    public int ProductId { get; private set; }
    public string ProductName { get; set; }
    public string ProductDescription { get; set; }
    private decimal? currentPrice;
    public decimal? CurrentPrice
    {
        get { return currentPrice; }
        set
        {
            if (value == null || value >= 0)
            {
                currentPrice = value;
            }
            else
            {
                currentPrice = null;
            }
        }
    }
```

```
    public Product Retrieve(int productId)
    {
        // 在這裡撰寫相關程式碼，可以取回該 Product 物件
        return new Product();
    }

    protected bool Save()
    {
        // 在這裡撰寫相關程式碼，儲存這個 Product 物件
        return true;
    }

    protected bool Validate()
    {
        var isValid = true;

        if (string.IsNullOrWhiteSpace(ProductName))
            isValid = false;
        if (CurrentPrice == null)
            isValid = false;

        return isValid;
    }
}
```

透過 "繼承" 來重複利用、擴充和修改基底類別的定義

```
class BaseClass
{
    public string Name { get; set; }
    public int Age { get; set; }
    public virtual void Output() => Console.WriteLine("Hello");
    public BaseClass(string name) => Name = name;
    public BaseClass() => Name = "";
}

class DerivedClass : BaseClass
{
    public string Department { get; set; }
    public DerivedClass() :base("Default")
    {
        base.Age = 18;
        //base.Department = "IT";
        base.Output();
        this.Age = 19;
        this.Department = "IT";
        this.Output();
    }
    public override void Output() => Console.WriteLine("Hello !!");
}
```

基底類別

建構函式多載 (overloading)

預設(無參數)建構函式

衍生類別

繼承基底類別

呼叫基底類別建構函式

設定基底類別屬性

基底類別沒有該屬性

執行基底類別方法

設定衍生類別屬性

執行衍生類別方法

覆寫方法 (overriding)

在 C# 中所有類型都是 "多型"

- 在設計時期 (Design Time)
  - **基底類別**可以定義和實作「虛擬」屬性或方法 (`virtual`)
  - **衍生類別**可以「覆寫」這些虛擬的屬性或方法 (`override`)
- 在執行時期 (Runtime)
  - 當呼叫基底類別的虛擬方法時，會改呼叫子類別覆寫的方法
  - 常見的多型範例
- 在 C# 中，所有類型都是**多型**類型
  - 因為所有類型 (包括使用者定義的類型) 都是繼承自 Object
- 如果要在 C# 中設計防止衍生類別覆寫虛擬成員
  - `public sealed override void DoWork() { }`
  - 範例程式
- 多載 (Overloading) 比較有點爭議 (有些人認為這不算多型)
  - 範例程式

內聚力與耦合力

Cohesion & Coupling

何謂 "模組" (Module)

- 一種抽象的概念
- 以 C# 舉例
  - 可能是一個「類別」(class)
  - 可能是一個「方法」(method)
  - 可能是一個「組件」(assembly)

內聚力 Cohesion

- 在一個 "模組" 內完成 "一件工作" 的度量指標

- **高內聚力**
  - 在一個 "模組" 內只完成一件工作
  - 內聚力高，意味著該模組可以獨立運作，也意味著更容易重複利用
  - 範例：一個 class 只負責一件事情 (例如寄送郵件)
- **低內聚力**
  - 在一個 "模組" 內完成多份工作
  - 內聚力低，意味著這個模組會造成難以維護/測試/重用/理解
  - 範例：所有功能寫在一個 class 裡面或一個 method 有 5000 行程式碼

- **最佳實務**
  - 在設計模組的時候，要盡量設計出**高內聚力**的程式碼。
  - 若要在一個模組內完成多項工作，建議拆成多個不同的類別
  - 實現 SRP 就是實現「**提高內聚力**」的一種表現

耦合力 Coupling

- 模組與模組之間的關聯強度
  - 模組之間相互依賴的程度
  - 衡量兩個模組的緊密連接程度
  - 範例：在 ClassB 裡面，直接 "建立" 了 ClassA 的物件實體，就會建立 ClassA 與 ClassB 之間的 "耦合關係"。

- **高耦合力**
  - 意味著當改了 A 模組時，相關聯的 B 模組就會容易被影響 (改A壞B)
- **低耦合力**
  - 當在修改模組的時候，有越少的模組被影響，就意味著耦合力較低

- **最佳實務**
  - 在設計不同模組的時候，要盡量設計出**低耦合力**的程式碼。
  - 實現 **DIP** 就是實現「**降低耦合力**」的一個原則

隨堂測驗

```
public class InvitationService
{
    public void SendInvite(string email, string firstName, string lastName)
    {
        if (String.IsNullOrWhiteSpace(firstName) ||
            String.IsNullOrWhiteSpace(lastName))
        {
            throw new Exception("Name is not valid!");
        }

        if (!email.Contains("@") || !email.Contains("."))
        {
            throw new Exception("Email is not valid!!");
        }
        SmtpClient client = new SmtpClient();
        client.Send(new MailMessage("mysite@nowhere.com", email)
        { Subject = "Please join me at my party!" });
    }
}
```

隨堂測驗

```
public class Order
{
    private ShoppingCartContents cart;
    private float salesTax;

    public Order(ShoppingCartContents cart, float salesTax)
    {
        this.cart = cart;
        this.salesTax = salesTax;
    }

    public float OrderTotal()
    {
        float cartTotal = 0;
        for (int i = 0; i < cart.items.Length; i++)
        {
            cartTotal += cart.items[i].Price * cart.items[i].Quantity;
        }
        cartTotal += cartTotal * salesTax;
        return cartTotal;
    }
}
```

```
public class ShoppingCartContents
{
    public ShoppingCart[] items;
}
```

```
public class ShoppingCart
{
    public float Price;
    public int Quantity;
}
```

SOLID 物件導向設計原則

The S.O.L.I.D. Principles of Object Oriented Design

何謂 "原則" (Principle)

- A **principle** is a concept or value that is a guide for **behavior** or **evaluation**

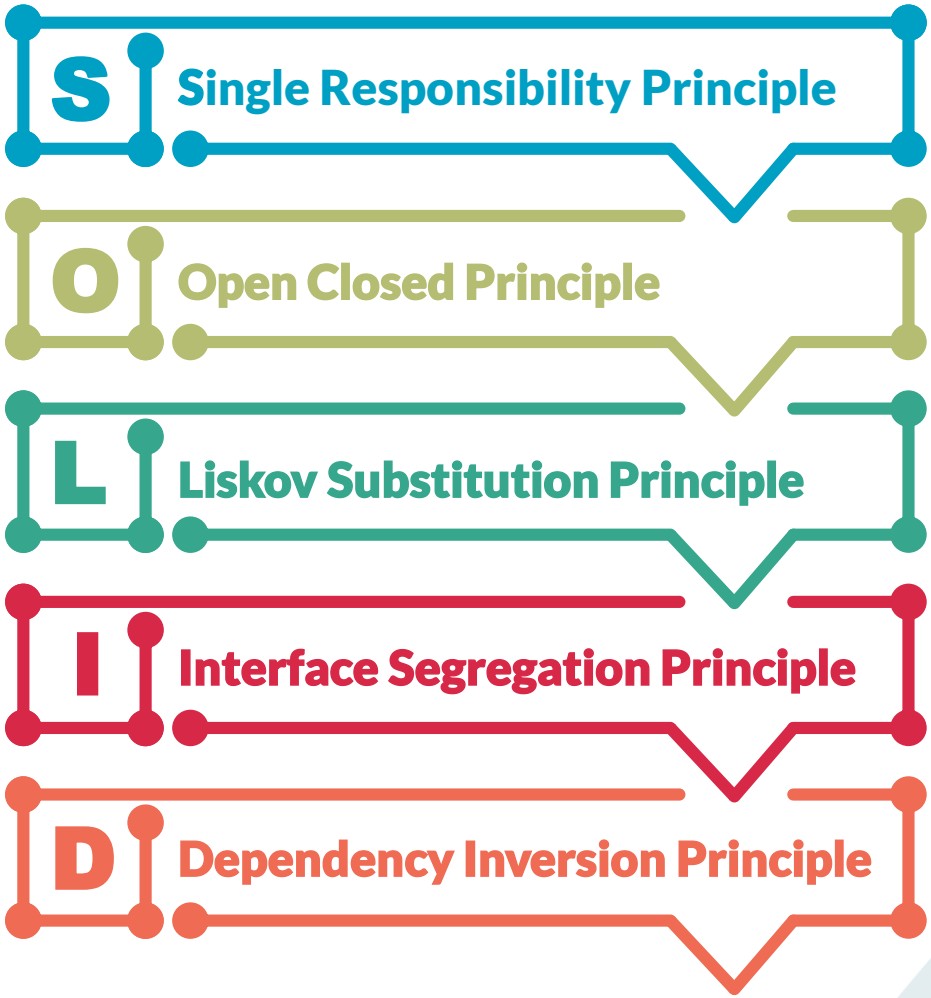
- 所謂「**原則**」(principle) 就是一種「概念」或「價值」，用來導引你產生**適切的行為**與**價值評量方法**。

- 白話文解釋
  - 依循 SOLID 原則，可以**寫出比較好的程式碼**
  - 依循 SOLID 原則，能夠**判斷程式碼的好壞**（Code Smell）

https://en.wikipedia.org/wiki/Principle

OOP 物件導向程式設計的 SOLID 設計原則

- 單一責任原則 SRP
- 開放封閉原則 OCP
- 里氏替換原則 LSP
- 介面隔離原則 ISP
- 相依反轉原則 DIP



Working Effectively with Legacy Code || ArticleS.UncleBob.PrinciplesOfOod

學習 SOLID 物件導向設計原則的好處

- 降低程式碼複雜程度
- 具有較佳程式碼可讀性
- 提升模組可重複利用性
- 讓模組具有高內聚力、低耦合力
- 面臨變更需求時可減少破壞現有模組的風險

單一責任原則 SRP

Single Responsibility Principle

單一責任原則 Single Responsibility Principle

- A **class** should have only one reason to change.

- 一個類別應該只有一個改變的理由！

https://en.wikipedia.org/wiki/Single_responsibility_principle

何謂 "責任" (Responsibility)

- 責任 = reason to change (改變的理由)

- 當一個類別擁有多個不同的責任
  意味著一個類別負責多項不同的工作
  當需求變更時，更動一個類別的理由也可能不只一個

請問以下類別有多少責任？

```
public class OrderManager
{
    public bool LoadOrder()
    {
        // 1. 建立資料庫連線 (包含寫死的連接字串)
        // 2. 執行 ADO.NET 資料存取 (包含資料篩選)
        // 3. 跑迴圈取得資料 (包含資料格式轉換)
        // 4. 回傳資料
    }
}
```

關於 SRP 的基本精神

- 一個 "類別" 負擔太多責任時，意味著該類別可以被切割
  - 可以透過定義一個全新的 "**類別**" 輕鬆做到
  - 對類別進行適度的切割，方便日後管理與維護！
- SRP 主要精神就是**提高內聚力**
  - **高內聚力**意味者你可以想到**一個清楚的理由**去改它！

低內聚力的示意圖

SRP 主要精神就是提高內聚力

圖片來源

常見的設計問題

- 將所有功能寫在一個類別中
  - 類別複雜度過高
  - 維護時經常找不到應該要改哪裡
  - 發生邏輯問題時找不到 Bug 在哪裡
  - 使用類別時不知道應該要呼叫哪個方法

關於 SRP 的使用時機

- **兩個責任**會在**不同時間點**產生變更需求
  - 當你**想改資料庫查詢語法**與**修改系統紀錄的邏輯**時，都會改到同一個類別，那就需要拆開！
  - 範例程式：修改前、修改後

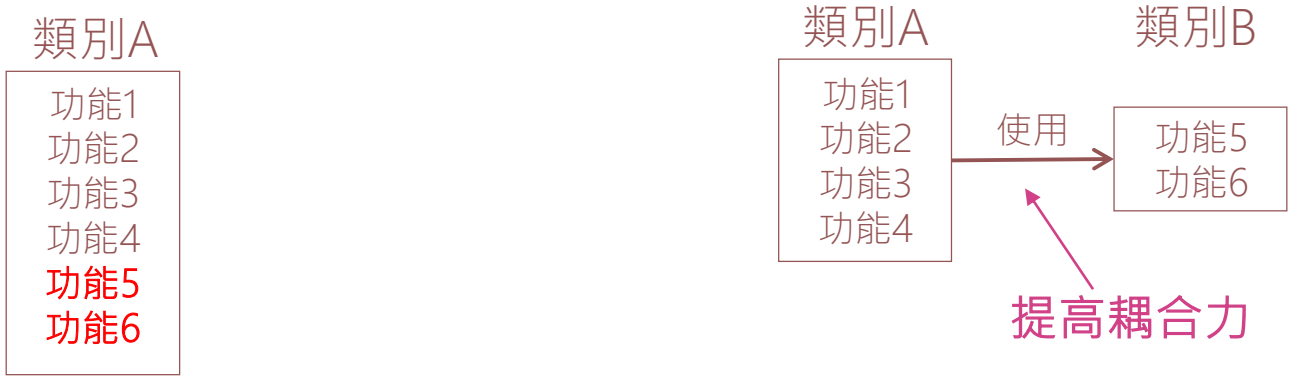
- 類別中有一段程式碼有**重複利用**的需求
  - 這段程式碼在其他類別也用的到

- 系統中有個**非必要**的功能 (**未來需求**)，老闆又逼你要實作時
  - 責任會直接依附在類別中，但對維護造成困擾

SRP 討論事項

- 你怎樣確認一個類別被賦予了過多的責任？
- 套用 SRP 可能會有副作用，因為類別變多導致耦合力增加

從類別A中，拆出不同責任方法到
類別B(新類別)中，使其關注點分離



類別 A 包山包海，
將會具有多個責任

提高耦合力，意味者 "改B壞A" 的機會大幅增加！

關於 SRP 還需要注意的事

- 參考 YAGNI（You Ain't Gonna Need It）原則
  - 不用急於在第一時間就專注於分離責任
  - 尚未出現的需求 (未來的需求) 不需要預先分離責任
  - 當需求變更的時候，再進行類別分割即可！

- SRP 是 SOLID 中最簡單的，但卻是最難做到的
  - 需要不斷提升你的**開發經驗**與**重構技術**
  - 如果你沒有足夠的經驗去定義一個物件的 Responsibility 那麼建議你不要過早進行 SRP 規劃！

SRP 練習

找出多個責任
並進行修正

- 請試著找出並說明 OrderManager 類別，不符合 SRP 的地方
- 並請嘗試套用 SRP 原則重構這個類別
- 提示：
  如何判斷有哪些責任存在

練習情境
請試著找出該 OrderManager 類別，不符合 單一責任原則 地方

- 請指出這個類別是否**違反**了 SRP 原則？
  請說明**理由**與**如何改善**

```
public class OrderManager
{
    public List<Product> products = new List<Product>();

    public void Processing()
    {
        // 1. 檢查商品庫存數量是否足夠
        // 2. 進行付款處理程序
        // 3. 進行送貨處理程序
    }
}
```

在這裡的詳細程式碼將予以省略
此 Processing 方法內，將主要會
處理這三件事情

練習情境
請試著修正該 OrderManager 類別，使其符合 單一責任原則

- 將多個責任使用新類別分離出來，但還有甚麼問題?

```
public class Product { }
public class Customer { }
public class Stock
{
    public void CheckAvailability(
        IEnumerable<Product> products) { }
}
public class Payment
{
    public void Processing(
        Customer customer,
        IEnumerable<Product> products) { }
}
public class Shipment
{
    public void SendProducts(
        Customer customer,
        IEnumerable<Product> products) { }
}
```

```
public class OrderManager
{
    public List<Product> Products =
            new List<Product>();
    public Customer Customer { get; set; }
    public OrderManager()
    {

    }
    public void Processing()
    {
        new Stock().CheckAvailability(
            Products);
        new Payment().Processing(
            Customer, Products);
        new Shipment().SendProducts(
            Customer, Products);
    }
}
```

若客戶想要增加 Line Pay 付款方法，要改多少 Code？

開放封閉原則 OCP

Open Closed Principle

開放封閉原則 Open Closed Principle

- Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

- 軟體實體 (類別、模組、函式等) 應能**開放擴充**但**封閉修改**

- 藉由**增加新的程式碼**來擴充系統的功能，
  而不是藉由**修改原本已經存在的程式碼**來擴充系統

https://en.wikipedia.org/wiki/Open–closed_principle

關於 OCP 的基本精神

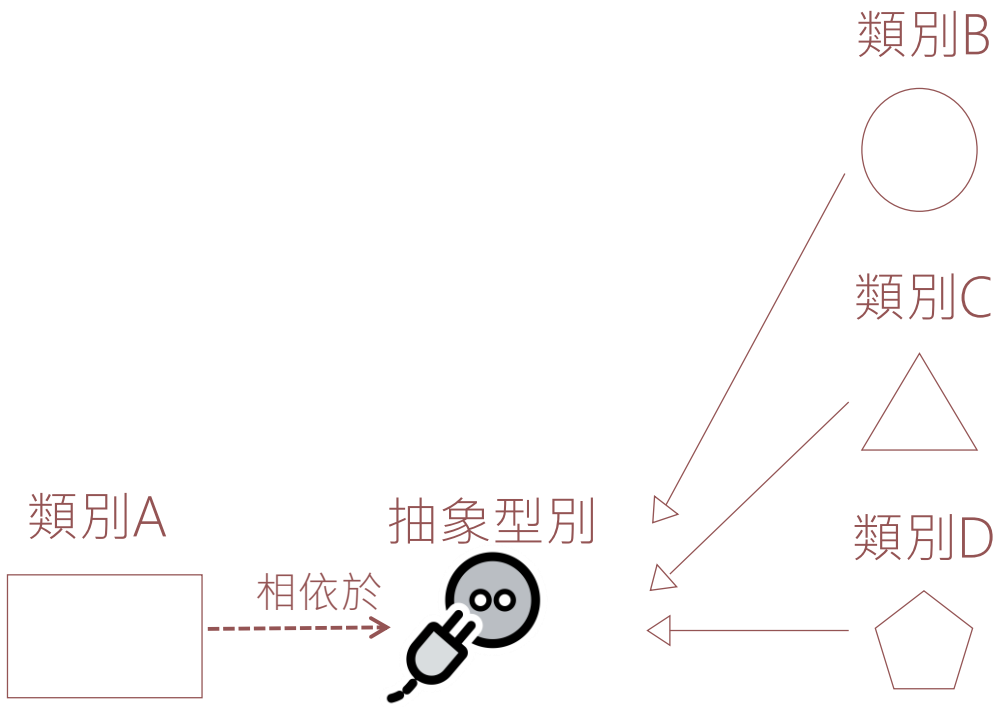
- 一個 "類別" 需要 "**開放**"，意味著該類別可以被擴充！
  - 可以透過 "**繼承**" 輕鬆做到
  - C# 還有 "**擴充方法**" 可以輕鬆擴充既有類別
- 一個 "類別" 需要 "**封閉**"，意味著有其他人正在使用這個類別！
  - 如果程式已經編譯，但又已經有人在使用原本的類別
  - 封閉修改可以有效避免未知的問題發生

常見的設計問題

- 耦合力過高、擴充不易

關於 OCP 的實作方式

- 採用分離與相依的技巧 (**相依於抽象**)
  - 缺點：**需要針對原有程式碼進行重構**



關於 OCP 的 C# 範例

- 透過**抽象類別**限制其修改，並透過繼承開放擴充不同實作

```
public abstract class DataProvider  {
    public abstract int OpenConnection();
    public abstract int CloseConnection();
    public abstract int ExecuteCommand();
}
```

https://code.msdn.microsoft.com/OOPS-Principles-SOLID-7a4e69bf

關於 OCP 的使用時機

- 你既有的類別**已經被清楚定義**，處於一個**強調穩定**的狀態
- 你**需要擴充**現有的類別，**加入新需求**的屬性或方法
- 你**擔心**修改現有程式碼會**破壞現有系統**的運作
- 系統剛開始設計時就決定要採用 OCP 模式
  - 可以透過 "**介面**" 或 "**抽象類別**" 進行實作

OCP 討論事項

- 當您剛接受維護一份 2 年前的程式碼，你會怎樣做？
  - 修改之前寫過的類別？
  - 擴充之前寫過的類別？
  - 直接修改舊有原始碼，會有哪些風險存在呢？
- 如何讓系統在**擴充需求**時更簡單、更容易、更安全？
- C# 可以透過 `interface` 實踐 OCP 原則嗎？如何做到？
- 如何進行**抽象化設計**？多少人用過 C# **抽象類別**？

OCP 練習

擴展Log有多種輸出功能

- 請試著透過 OCP 原則重構程式碼
- 體驗：OCP 的過程是昂貴的，但是日後容易進行變更與擴充

練習情境
試著透過 OCP 原則重構程式碼

```
public class AppEvent
{
    public void GenerateEvent(string message)
    {
        Logger fooLogger = new Logger();
        fooLogger.Log(message);
    }
}

public class Logger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

若客戶想要增加 Log 輸出到檔案的功能，
你會如何改寫程式碼？

練習情境
試著透過 OCP 原則重構程式碼

- 沒學過 SOLID 的開發者，可能會這樣寫：

```
public class AppEvent
{
    public void GenerateEvent(string message)
    {
        Logger fooLogger = new Logger("Console");
        fooLogger.Log(message);
    }
}
public class Logger
{
    private readonly string _Target;

    public Logger(string target) { _Target = target; }
    public void Log(string message)
    {
        if (_Target == "Console")
            Console.WriteLine(message);
        else if (_Target == "File")
            File.WriteAllText("MyLog", message);
        else
            throw new NotImplementedException();
    }
}
```

這裡採用的是修改原始程式碼的方式，進行變更需求之功能擴充！

此時，若又想要增加將訊息傳送到遠端 Web API 或 Storage 呢？

- 採用分離與相依的技巧

```
public interface ILogger
{
    void Log(string message);
}

public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
public class FileLogger : ILogger
{
    public void Log(string message)
    {
        File.WriteAllText(
          "MyLog", message);
    }
}
```

想要增加輸出到遠端 Web API，
我們該如何處理呢？

想要產生不同 Logger
傳入指定字串即可

```
public class AppEvent
{
    private readonly ILogger _Logger;

    public AppEvent(string loggerType)
    {
        this._Logger = LoggerFactory.CreateLogger(
                        loggerType);
    }
    public void GenerateEvent(string message) {
        _Logger.Log(message); }
}
```

工廠模式

```
public class LoggerFactory
{
    public static ILogger CreateLogger(
                string loggerType)
    {
        if (loggerType == "Console")
            return new ConsoleLogger();
        else if (loggerType == "File")
            return new FileLogger();
        else throw new NotImplementedException();
    }
}
```

里氏替換原則 LSP

Liskov Substitution Principle

里氏替換原則 Liskov Substitution Principle

- **Subtypes must be substitutable for their base types.**
  - **subtypes** (衍生型別) = 類別
  - **base types** (基底型別) = 介面、抽象類別、基底類別

- 子型別必需可替換為他的基底型別

- 如果你的程式有採用**繼承**或**介面**，然後建立出幾個不同的**衍生型別** (Subtypes)。在你的系統中只要是**基底型別**出現的地方，都可以用**子型別**來取代，而不會破壞程式原有的行為。

https://en.wikipedia.org/wiki/Liskov_substitution_principle

關於 LSP 的基本精神

- 當實作繼承時，必須確保型別轉換後還能得到正確的結果
  - 每個**衍生類別**都可以正確地替換為**基底類別**，且程式在執行時不會有異常的情況 (如發生執行時期例外)
  - 必須正確的實作 "**繼承**" 與 "**多型**"

常見的設計問題

- 不正確的實作 "**繼承**" 與 "**多型**"
  - 第一版：沒有繼承，單純計算矩形面積
  - 第二版：新增需求，增加 Square 類別 (套用 OCP 原則)
  - 第三版：重構程式，正確套用 LSP 原則

- 實作繼承時，在特定情況下發生執行時期錯誤 (Runtime Error)
  - 範例程式
  - 違反 LSP 原則有時候較難發現

關於 LSP 的實作方式

- 採用**類別繼承方式**來進行開發
  - 需注意繼承的實作方式
- 採用**合約設計方式**來進行開發
  - 利用 "**介面**" (`interface`) 來定義**基底型別** (base type)

關於 LSP 的使用時機

- 當你需要透過**基底型別**對**多型**物件進行操作時

LSP 討論事項

- 在教導新人時，如何有效的避免繼承的錯誤實作？
- 你會用 **抽象類別**、**類別** 或 **介面** 來實現 LSP 原則？為什麼？

介面隔離原則 ISP

Interface Segregation Principle

介面隔離原則 Interface Segregation Principle

- A: Many **client specific interfaces** are **better than** **one general purpose interface**.
- B: **Clients** **should not** be **forced** to depend upon **interfaces** that **they don't use**.

- A: 多個**用戶端專用的介面**優於一個**通用需求介面**
- B: **用戶端**不應該強迫相依於**沒用到的介面**

- 針對不同需求的**用戶端**，僅開放其對應需求的介面就好

https://en.wikipedia.org/wiki/Interface_segregation_principle

關於 ISP 的基本精神

- 把不同需求的屬性與方法，放在不同的介面中
  - 不要讓你的 `interface` 包山包海
  - 特定需求沒用到的方法，不要加入到介面中，另外建一個
  - 可以拿 `interface` 當成群組來用 (屬性與方法)
- 使得系統可以更容易的達成**鬆散耦合**、**安全重構**、**功能擴充**

常見的設計問題

- 將所有 API 需求都定義在一個超大介面中

- **用戶端**相依於一堆**用不到**的介面方法

  - 如果有多個類別已經實作同一個**胖介面**
  - 就會導致某些類別實作出**用戶端用不到**的方法
  - 這時應該可以拆分多的用戶端專用的介面來進行實作
  - 所以一個實作介面的類別，不應該強迫去實作出這個類別**不需要**的方法 (備註：這裡的**不需要**是指**用戶端不需要**)

關於 ISP 的實作方式

- 依據**用戶端**需求，將**介面**進行**分割**或**群組**

關於 ISP 的使用時機

- 當介面需要被分割的時候

- 類別的使用時機可以被切割的時候

  - 假設類別有 20 個方法，並實作一個有 15 個方法的界面
  - 有某個**用戶端**只會使用該類別中的 10 個方法
  - 你就可以為這類別的 10 個方法定義介面並設定實作介面
  - 你的**用戶端**就可以改用**介面**操作
  - 這個過程也可以用來降低主程式與這個類別的耦合力

ISP 討論事項

- 你工作中是否有設計過**超大介面**的經驗？
- 設計介面的時候，介面的大小該如何判斷？如何群組？
- 介面可以實作介面，使用的時機為何？

ISP 練習

讓介面進行瘦身

- 請試著找出該 IAllInOneCar 介面，不符合 介面隔離原則 地方
- 請試著修正該 IAllInOneCar 介面，使其符合 介面隔離原則
- 體驗：採用將介面分開成不同群組之技能

練習情境
請試著找出該 IAllInOneCar 介面，不符合 介面隔離原則 地方

```
public void Main()
{
    Driver o = new Driver();
    o.StartEngine();
    o.Drive();
    o.StopEngine();
}

public interface IAllInOneCar
{
    void StartEngine();
    void Drive();
    void StopEngine();
    void ChangeEngine();
}
```

```
public class Driver : IAllInOneCar
{
    public void ChangeEngine()
    { throw new NotImplementedException(); }

    public void Drive()
    {  }

    public void StartEngine()
    {  }

    public void StopEngine()
    {  }
}
```

練習情境
請試著修正該 IAllInOneCar 介面，使其符合 介面隔離原則

- 修正後結果

```
public interface IDriver
{
    void StartEngine();
    void Drive();
    void StopEngine();
}

public class Driver : IDriver
{
    public void Drive()
    {  }

    public void StartEngine()
    {  }

    public void StopEngine()
    {  }
}
```

```
public void Main()
{
    IDriver o = new Driver();
    o.StartEngine();
    o.Drive();
    o.StopEngine();
}
```

```
public interface IMechanic
{
    void ChangeEngine();
}

public class Mechanic : IMechanic
{
    public void ChangeEngine()
    { throw new NotImplementedException(); }
}
```

相依反轉原則 DIP

Dependency Inversion Principle

相依反轉原則 Dependency Inversion Principle

- A. **High-level modules** should not depend on **low-level modules**. Both should depend on abstractions.
- B. **Abstractions** should not depend on details. Details should depend on abstractions.

- A. **高階模組**不應該依賴於**低階模組**，兩者都應**相依於抽象**
  - 高階模組 ➔ Caller (呼叫端)
  - 低階模組 ➔ Callee (被呼叫端)
- B. **抽象**不應該相依於**細節**。而**細節**則應該**相依於抽象**

https://en.wikipedia.org/wiki/Dependency_inversion_principle

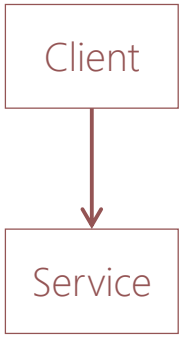
關於 DIP 的基本精神

- 所有類別都要相依於抽象，而不是具體實作
  - 可透過 **DI Container** 達到目的
- 為了要達到類別間鬆散耦合的目的
  - 開發過程中，所有**類別之間**的**耦合關係**一律透過**抽象介面**

常見的設計問題

- 類別與類別之間緊密耦合，改 A 壞 B 的狀況層出不窮

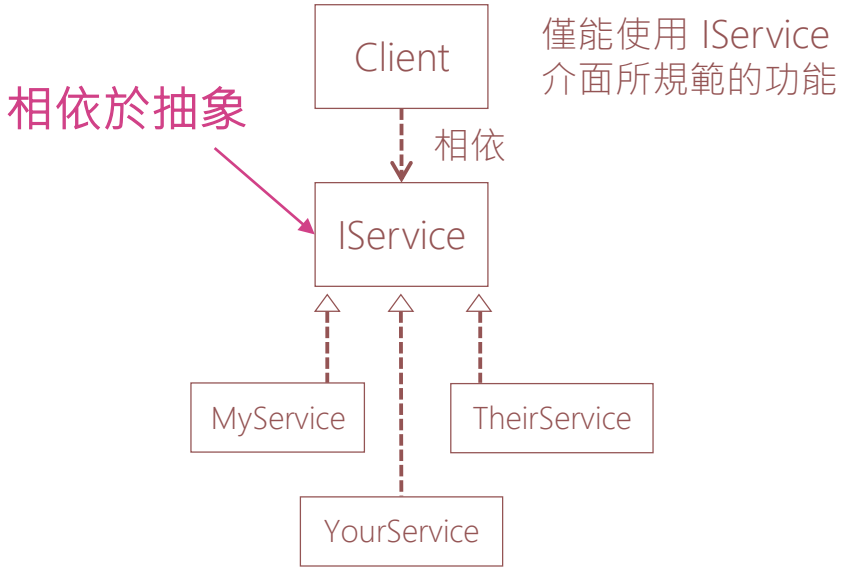
```
public class Client
{
    Service _Service;
    public void Client()
    {
        Service fooObj = new Service();
    }
}

public class Service {}
```



直接使用 Service 物件提供的功能

什麼是 "相依反轉" ?

```
public class Client
{
    IService _Service;

    public void Client(IService service)
    {
        _Service = service;
    }
}

public interface IService { }
public class MyService : IService { }
public class YourService : IService { }
public class TheirService : IService { }
```



關於 DIP 的實作方式

- 型別全部都相依於抽象，而不是具體實作

- 經過套用 DIP 之後，原來有相依於類別的程式碼
  - 都改成相依於**抽象型別**
  - 從**緊密耦合**關係變成**鬆散耦合**關係
  - 可以依據需求，隨時**抽換具體實作類別**

關於 DIP 的使用時機

- 想要降低耦合的時候
- 希望類別都相依於抽象，讓團隊可以**更有效率**的開發系統
- 想要可以替換具體實作，讓系統變得更有彈性
  - 符合 DIP 通常也意味著符合 OCP 與 LSP 原則
  - 只要再多考量 SRP 與 ISP 就很棒了！
- 想要導入 **TDD (測試驅動開發)** 或 **單元測試** 的時候

DIP 討論事項

- 你在工作中是否有遇過類似的設計方式？ (相依注入)
- 如果一個類別非常穩定，也沒有變更需求，需要套用 DIP 嗎？
- 大量套用 DIP 有缺點嗎？

DIP 練習

將原有相依於具體程式碼，修改改成相依於抽象

- 請試著找出底下程式碼 SecurityService，不符合 相依反轉原則地方
- 請試著修正程式碼 SecurityService，產生相依具體類別的抽象介面
- 體驗：採用採用都要相依於抽象，而不是具體實作之技能
- 完成後，您能體驗修改後帶來的好處嗎？

練習情境
請試著找出底下程式碼不符合 相依反轉原則 地方

- 請指出以下兩個類別哪些地方違反了 DIP 原則
  並請說明您的理由與說明如何改善的方法

```
public class SecurityService
{
    public bool LoginUser(string userName, string password)
    {
        LoginService service = new LoginService();
        return service.ValidateUser(userName, password);
    }
}

public class LoginService
{
    public bool ValidateUser(string userName, string password)
    { throw new NotImplementedException(); }
}
```

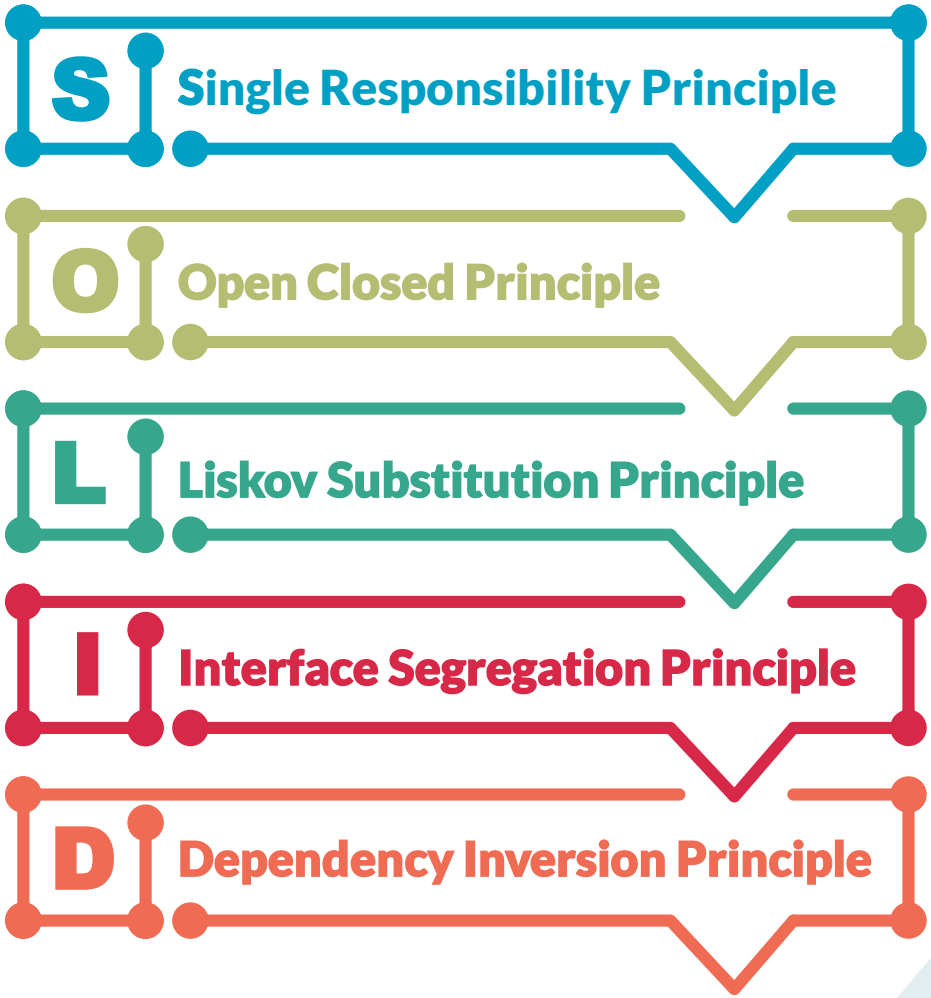
練習情境
請試著修正程式碼，使其符合 相依反轉原則

- 修正相依於抽象，使用建構式傳入具體實作物件

```
public class SecurityService
{
    private readonly ILoginService _LoginService;

    public SecurityService(ILoginService loginService)
    {
        this._LoginService = loginService;
    }

    public bool LoginUser(string userName, string password)
    {
        return _LoginService.ValidateUser(
                userName, password);
    }
}

public void Main()
{
    new SecurityService(new LoginService());
}
```

OOP 物件導向程式設計的 SOLID 設計原則

- 單一責任原則 SRP
- 開放封閉原則 OCP
- 里氏替換原則 LSP
- 介面隔離原則 ISP
- 相依反轉原則 DIP



Working Effectively with Legacy Code || ArticleS.UncleBob.PrinciplesOfOod

綜合討論

General Discussion

實體課程 http://coolrare.accupass.com

- .NET / C# 開發實戰：掌握相依性注入的觀念與開發技巧
  - 7/25 (三) 平日班 / 8/18 (六) 假日班
- Angular 練功坊：從入門到進階
  - 8/8 (三) 平日班 / 8/18 (六) 假日班
- Xamarin.Forms 跨平台行動開發一日實戰營 (表單 App / 派工 App)
- ASP.NET Web API 2 開發實戰 / Entity Framework 6 開發實戰
- 《台北》ASP.NET Core 2.1 開發實戰：快速上手篇 (平日班)

線上課程 https://www.udemy.com/user/coolrare/

- 精通 Git 版本控管：從入門到進階
- 給 C# 開發人員的第一堂 .NET Core 入門課
- Angular 開發實戰：從零開始
- Visual Studio 2017 開發環境全面解析
- Language-Integrated Query 快速上手 (C#) (LINQ)

企業內訓

- 來信至 training@miniasp.com