SOLID 之 LSP (里氏替換原則)

Overview

`繼承`、`封裝` 與 `多型` 是 OOP 的三大特性，其中 `繼承` 算是最早被人們大量使用的 OOP 特性，但也是最容易被 `誤用` 的特性，`里氏替換原則` 主要在定義 `怎樣才算是好的繼承?`。

Outline

OOP 進化史

OOP 1.0

在 90 年代初期，也就是 OOP 發展初期，人們大量使用 `繼承` 特性寫程式：

- 先寫一個 default 功能版的 class
- 繼承該 class，並將功能不一樣的地方加以 override

這種 `傳統繼承` 有個問題：

- 為了 code reuse 而繼承，但繼承鍊卻沒有家族關係
- Override 以後可能與原本 class 意義完全不同，`多型` 崩潰
- 繼承鍊容易超長超複雜

> OOP 1.0 基本上以 `繼承` 為重點，強調 `code reuse`

OOP 2.0

1994 年的 Design Pattern 提出 `多用組合，少用繼承`，建議以 interface 方式取代 `繼承`。

> OOP 2.0 的 `繼承` 不再用於 code reuse，而是退化成實現 `多型`，強調 `開放封閉`

OOP 3.0

2002 年 `APPP (Agile Principles, Patterns, and Practices)` 提出 SOLID 原則，`LSP` 建議以 `多型` 為前提使用 `繼承`。

> OOP 3.0 基本上也是以 `多型` 為重點，定義 `什麼是好的物件導向?`

里氏替換原則

> LSP : Liskov Substitution Principle
>
> 子類別必須能夠替換父類別

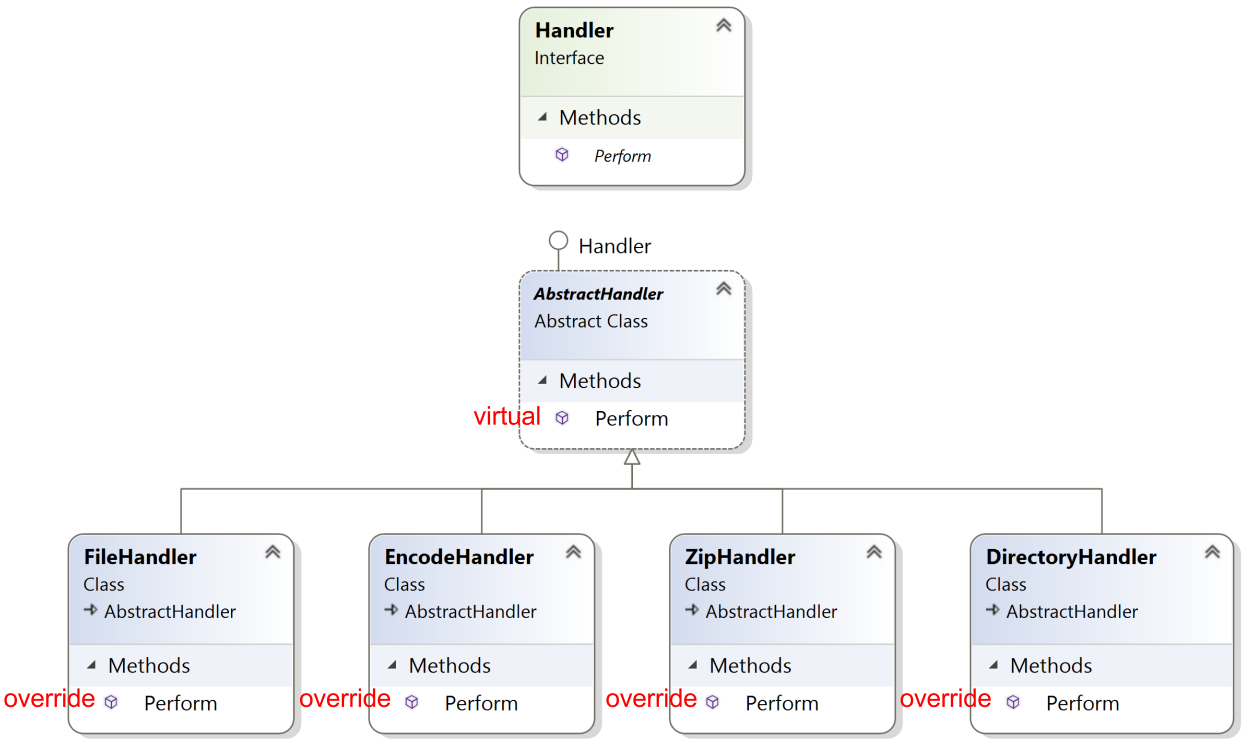
> 白話：凡宣告父類別之處，皆能在不修改程式碼前提下使用子類別，符合 `開放封閉` 原則

`里氏替換原則` 否定了 `傳統繼承`，認為子類別只能 `有限度` 改變父類別功能，且必須遵守父類別的架構，才能無痛以子類別替換父類別。

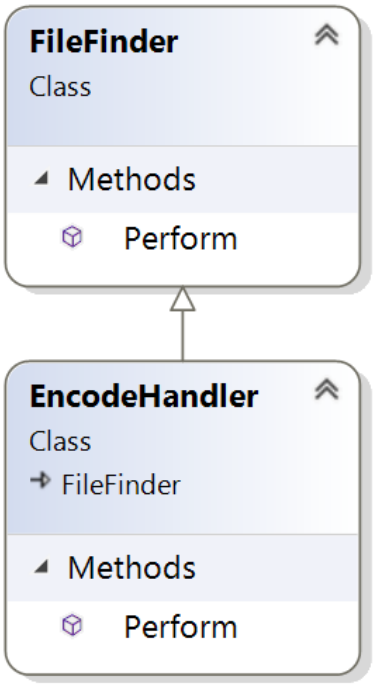
> LSP 是 `更嚴格` 的繼承，將繼承退化成只使用在 `多型`，而不用在 `code reuse`，也是目前公認較好的繼承使用方式

典型違反 LSP

功能與繼承鍊無關



在 homework 3，我們曾經實作各種 handler，若使用 `傳統繼承`，我們會這樣實現：



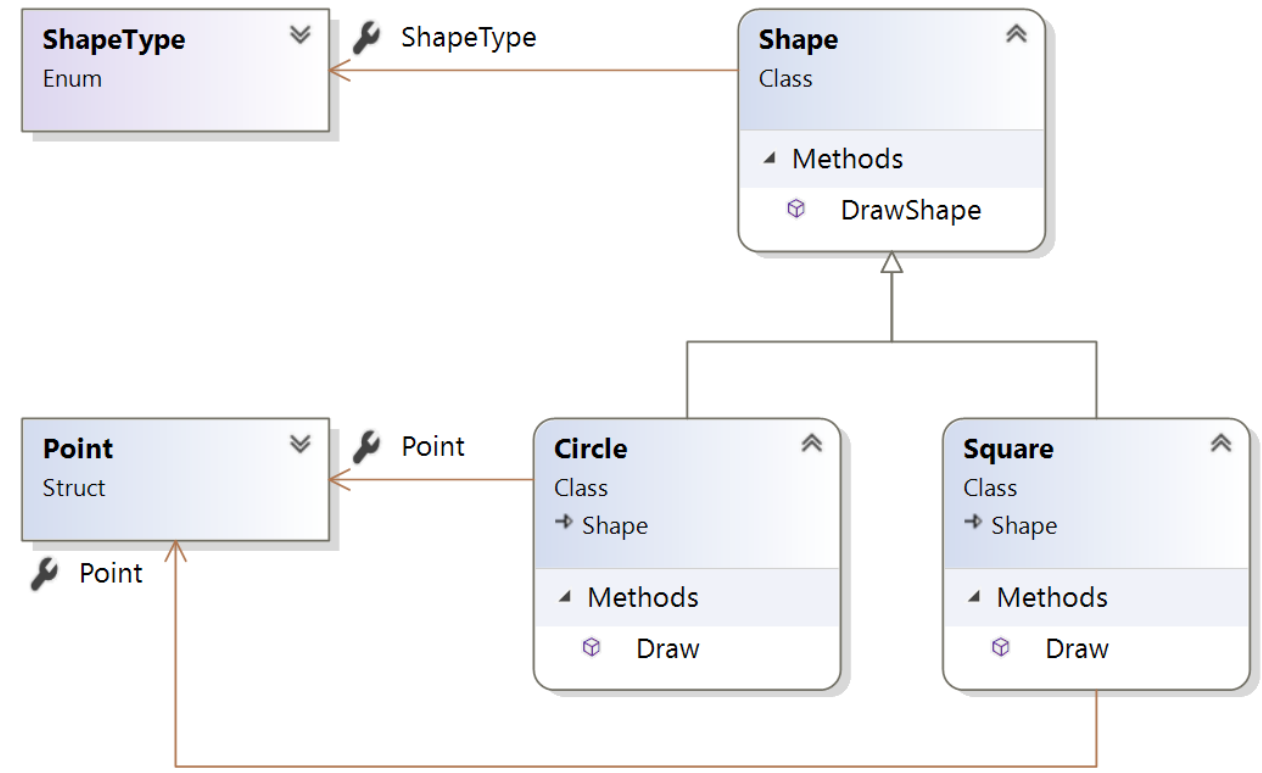
由 FileHandler 先實做一個版本，但由於 `EncodeHandler.Perform()` 與 `FileHandler.Perform()` 不一樣，因此 `EncodeHandler` 會繼承 `FileHandler`，並將 `Perform()` 加以 override。

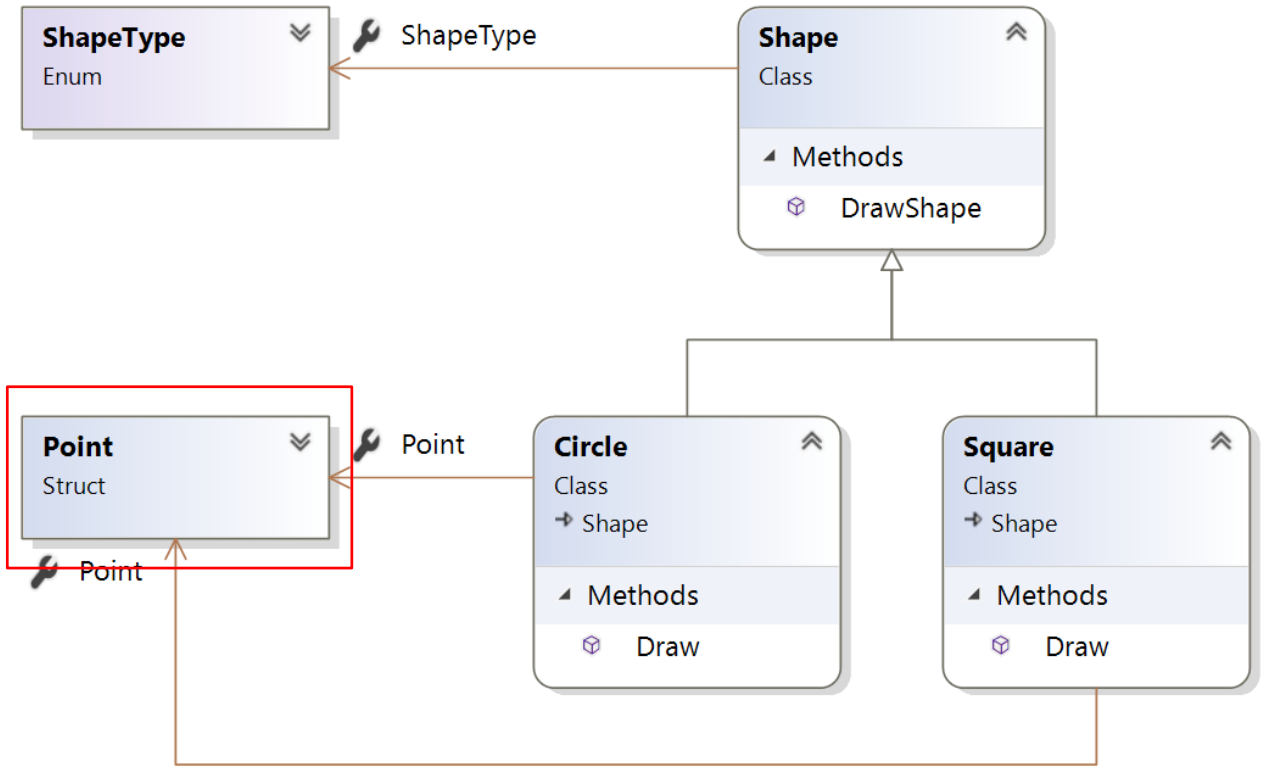
如此雖然 `FileHandler` 為 `EncodeHandler` 的父類別，但 `EncodeHandler` 卻無法取代 `FileHandler`，因為功能差異太大，完全違反 `里氏替換原則`。

> 解決方式就是新建立 `AbstractHandler`，作為 `FileHandler` 與 `EncodeHandler` 的父類別，讓 `FileHandler` 與 `EncodeHandler` 成為 `兄弟關係`，而非 `父子關係`。

自己判斷型別與轉型

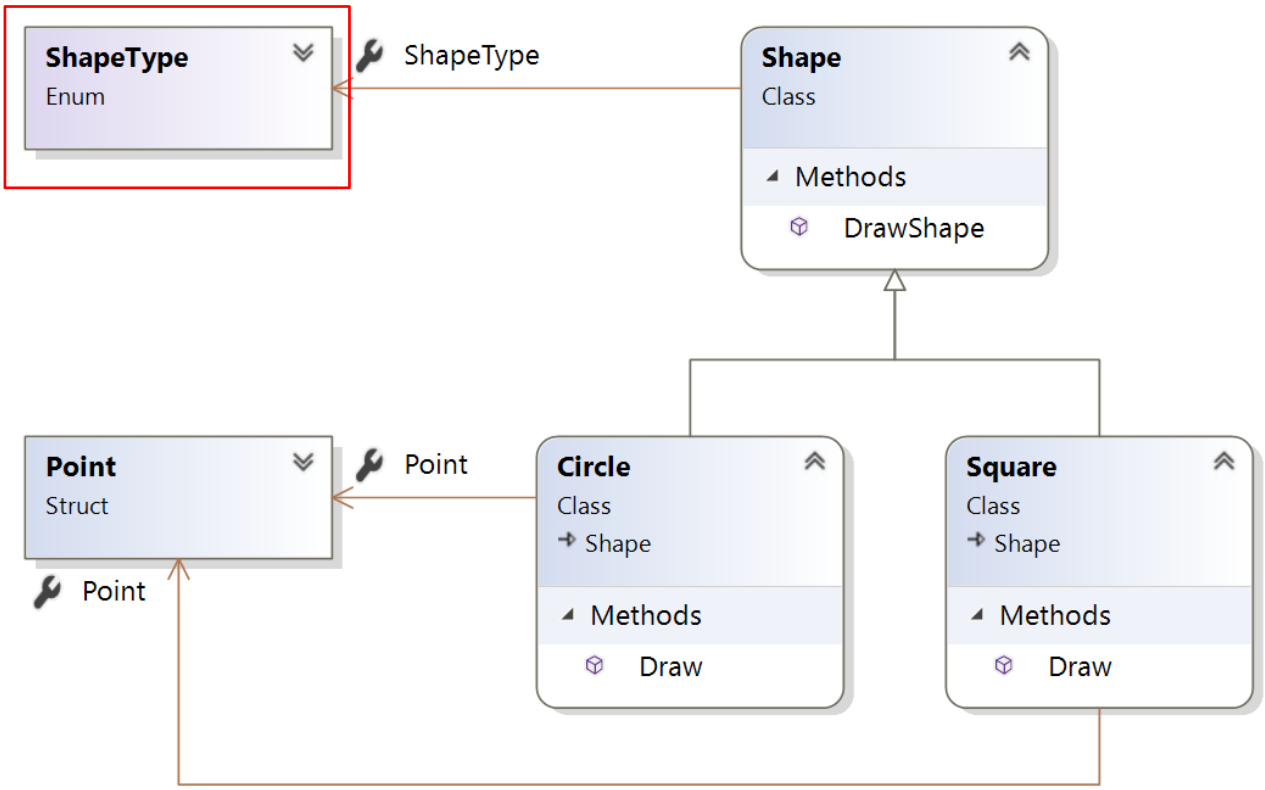
重構前





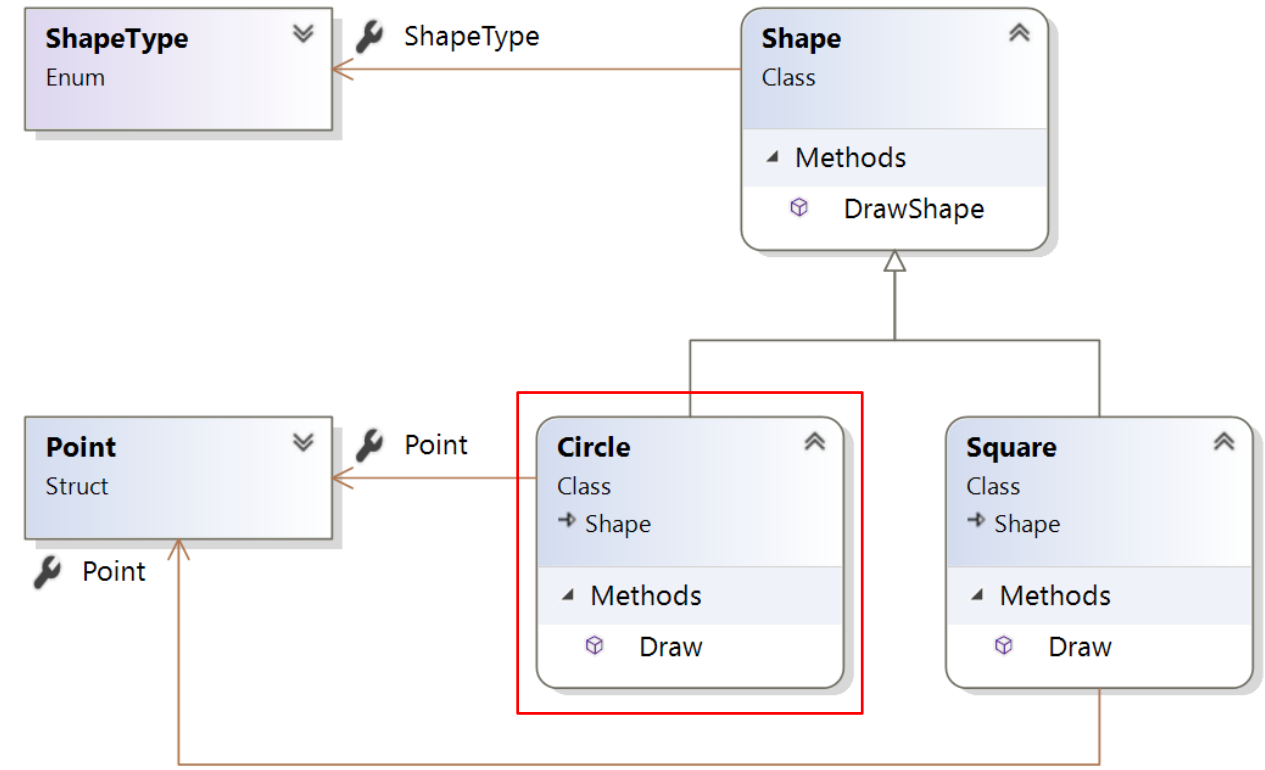
Point.cs

```
1 struct Point { double x, y; }
```



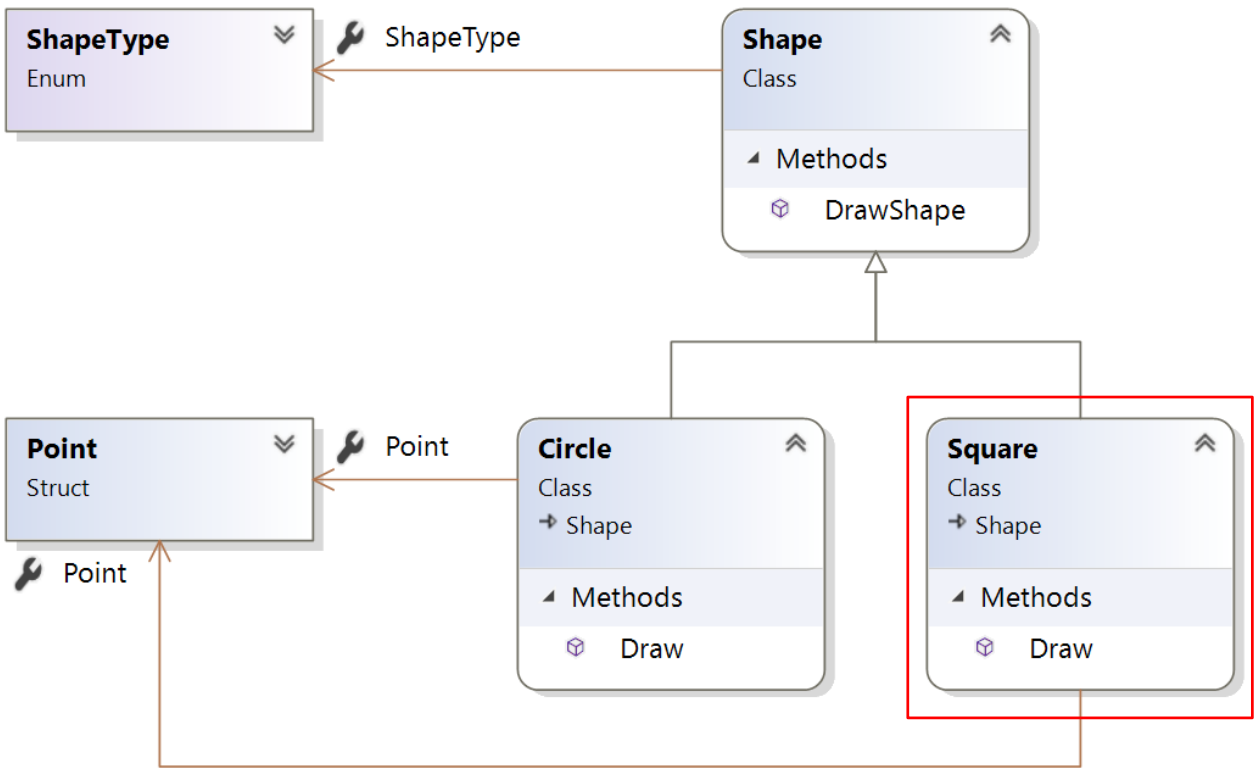
ShapeType.cs

```
1  public enum ShapeType {square, circle};
```



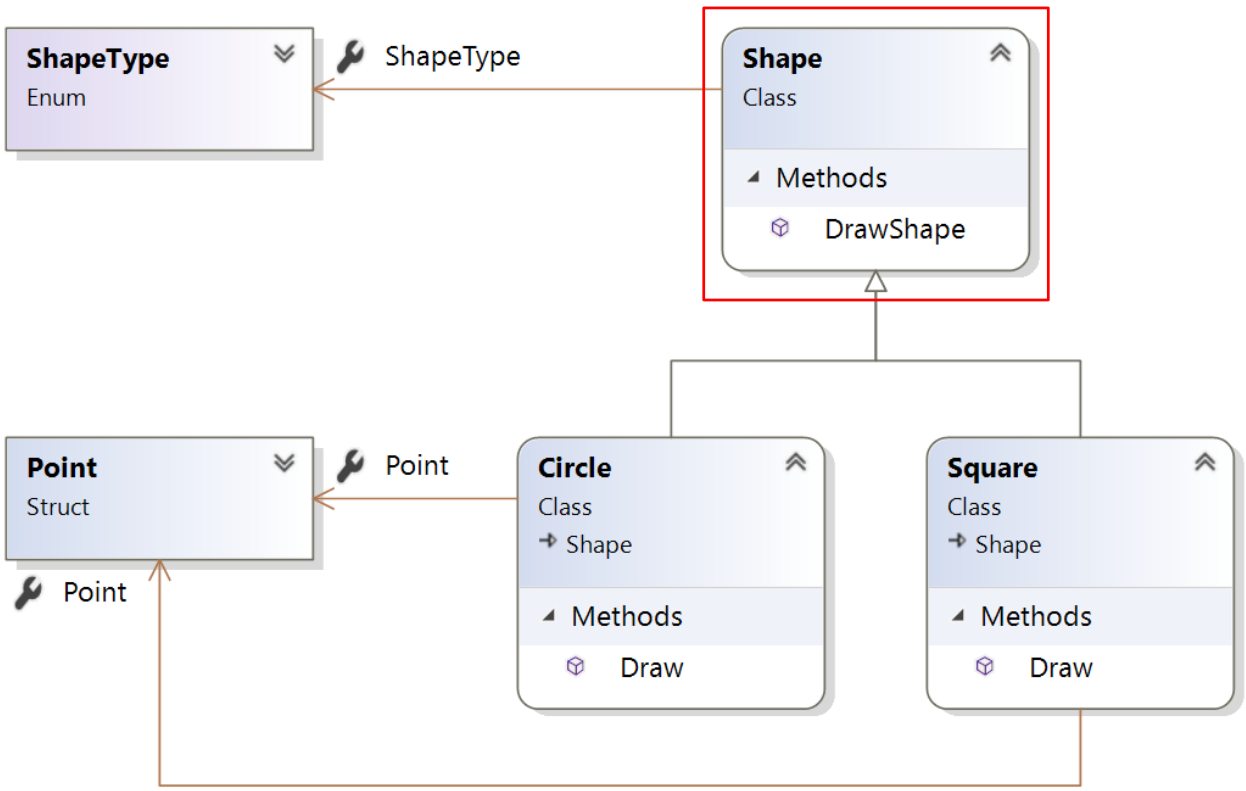
Circle.cs

```
public class Circle : Shape
{
    private Point center;
    private double radius;

    private Circle() : base(ShapeType.circle) {}

    public void Draw()
    {
        ...
    }
}
```



Square.cs

```
public class Square : Shape
{
    private Point topLeft;
    private double side;

    public Square() : base(ShapeType.square) {}

    public void Draw()
    {
        ...
    }
}
```



Shape.cs

```
1  public class Shape
2  {
3      private ShapeType type;
4
5      public Shape(ShapeType type)
6      {
7          this.type = type;
8      }
9
10     public static void DrawShape(Shape shape)
11     {
12         if (shape.type == ShapeType.square)
13         {
14             (shape as Square).Draw();
15         }
16         else if (shape.type == ShapeType.circle)
17         {
18             (shape as Circle).Draw();
19         }
20     }
21 }
```

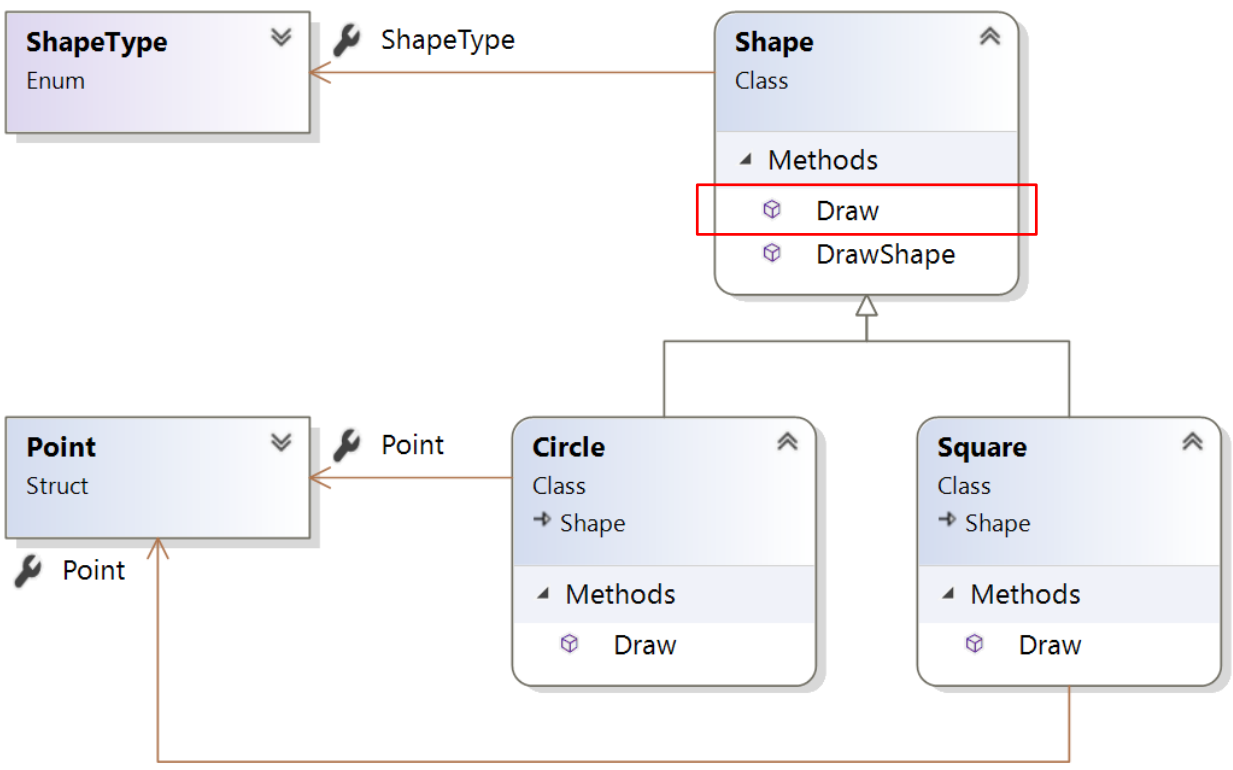
第 10 行

```
public static void DrawShape(Shape shape)
{
    if (shape.type == ShapeType.square)
    {
        (shape as Square).Draw();
    }
    else if (shape.type == ShapeType.circle)
    {
        (shape as Circle).Draw();
    }
}
```

雖然使用父類別的 `Shape` 為參數型別，卻要判斷 `Shape` 型別，然後過轉型才使用 `Draw()`，無法直接以子類別取代父類別，違反 里氏替換原則 。

若將來有新的繼承於 `Shape` 的型別，如 `Triangle`，則 `DrawShape()` 必須增加 `if else` 判斷型別，違反 開放封閉原則 。

重構後



Shape.cs

```
1  public class Shape
2  {
3      private ShapeType type;
4
5      public Shape(ShapeType type)
6      {
7          this.type = type;
8      }
9
10     public virtual void Draw() {}
11
12     public static void DrawShape(Shape shape)
13     {
14         shape.Draw();
15     }
16 }
```

10 行

```
1  public virtual void Draw() {}
```

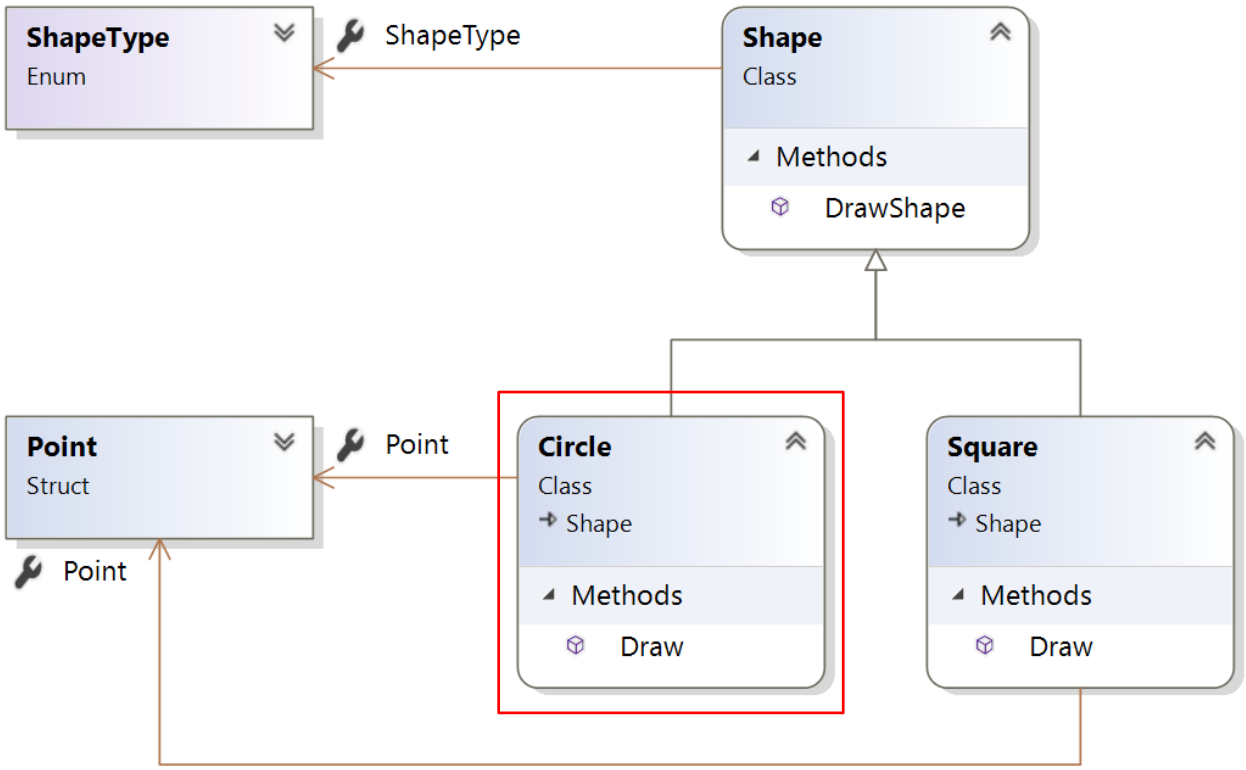
宣告 `Draw()` 為 `virtual`，準備使用 `多型`。

12 行

```
1  public static void DrawShape(Shape shape)
2  {
3      shape.Draw();
4  }
```

無論傳入 `Sqaure` 或 `Circle`，皆可取代父類別 `Shape`，符合 `里式替換原則`。

將來若有繼承 `Shape` 的新類別，也不用修改用戶端程式碼，符合 `開放封閉原則`。



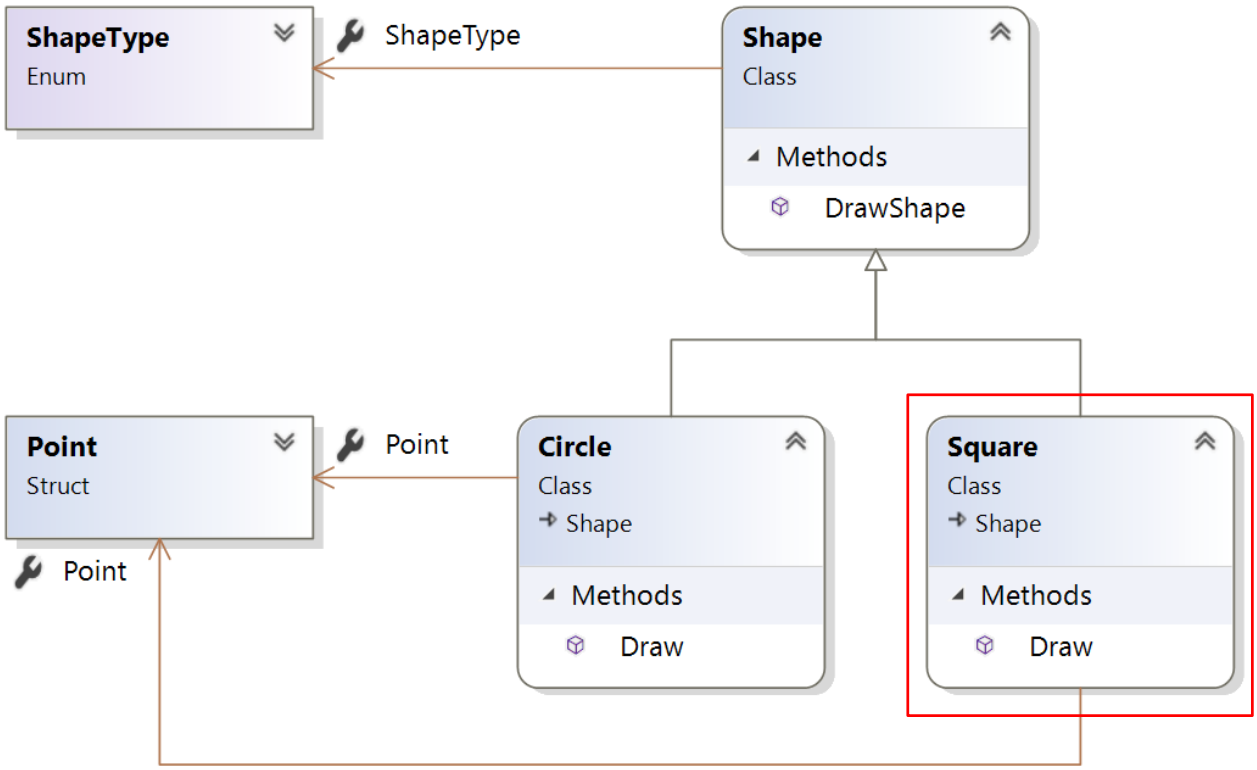
Circle.cs

```
1  public class Circle : Shape
2  {
3      private Point center;
4      private double radius;
5
6      private Circle() : base(ShapeType.circle) {}
7
8      public override void Draw()
9      {
10         ...
11     }
12 }
```

第 8 行

```
public override void Draw()
{
    ...
}
```

直接 `override` `Draw()`，使用 多型。



Square.cs

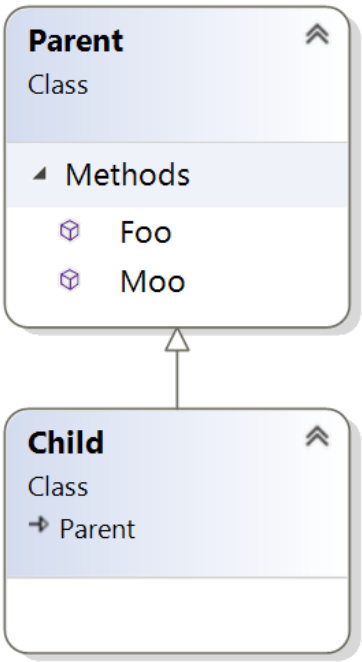
```
public class Square : Shape
{
    private Point topLeft;
    private double side;

    public Square() : base(ShapeType.square) {}

    public override void Draw()
    {
        ...
    }
}
```

直接 `override` `Draw()`，使用 多型。

> OOP 心法
>
> 實務上若會用到 判斷型別 與 轉型，大概都是 多型 沒用好
>
> 違反 里氏替換原則，也會一起違反 開放封閉原則

出現退化函式

重構前

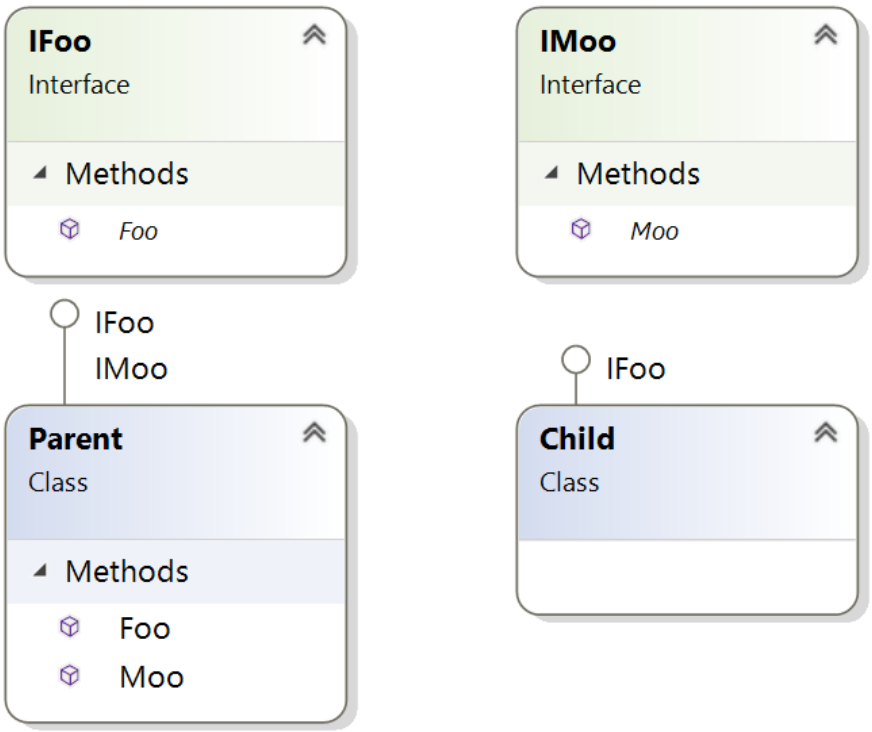



```
public class Parent
{
    public virtual void Foo() { ... }
    public virtual void Moo() { ... }
}

public class Child : Parent
{
    public override void Foo() { ... }
    public override void Moo() {}
}
```

`Moo()` 沒有使用 `base.Moo()`，因此 `Child.Moo()` 對於 `Parent.Moo()` 為空實作，也就是出現 `退化函式`，這使的子類別 `無法` 替換父類別，違反 `里氏替換原則`。

> 有空實作違反 `里氏替換原則`，也違反 `界面隔離原則`

重構後

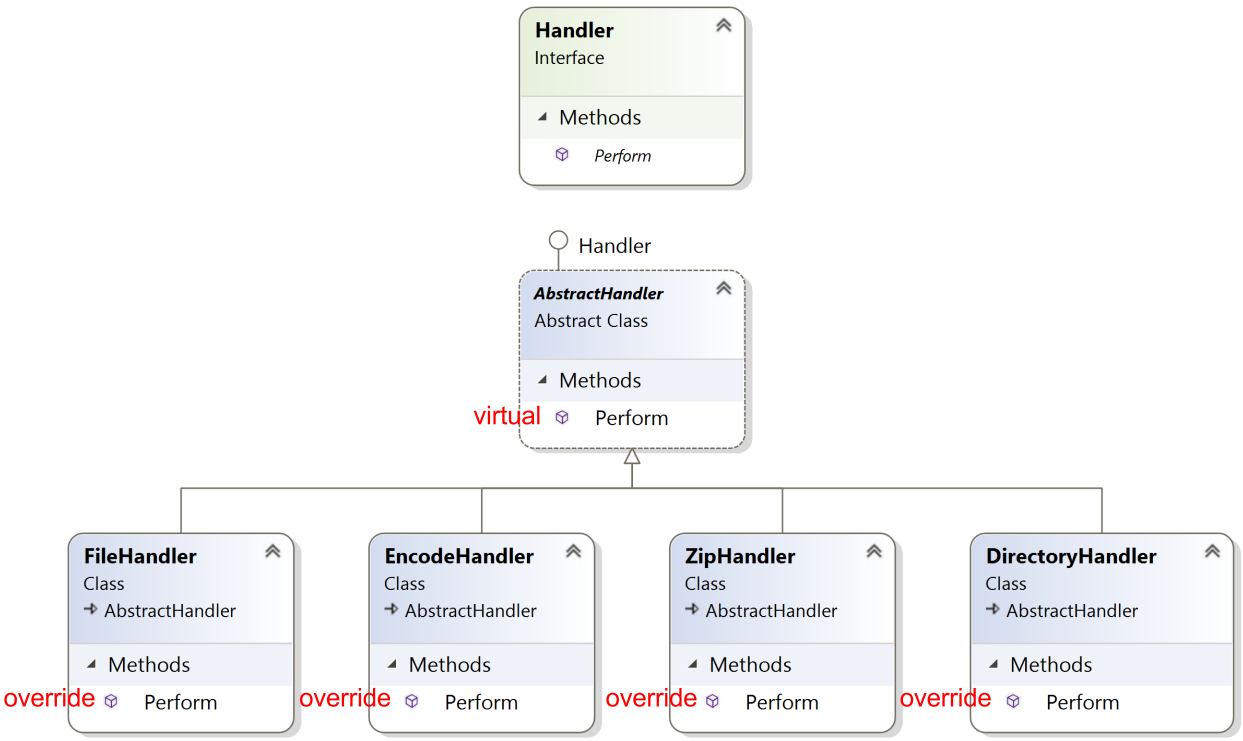

IFoo
Interface
Methods
Foo
IMoo
Interface
Methods
Moo
IFoo
IMoo
Parent
Class
Methods
Foo
Moo
IFoo
Child
Class

```
public interface IFoo
{
    void void Foo();
}

public interface IMoo
{
    void void Moo();
}

public class Parent : IFoo, IMoo
{
    public void Foo() { ... }
    public void Moo() { ... }
}

public class Child : IFoo
{
    public void Foo() { ... }
}
```

`Child` 不再有 `Moo()` 的空實作，且 `Parent` 與 `Child` 對於 `IFoo` 仍然遵循 `里氏替換原則`。

> OOP 心法
>
> 1. 子類別必須完全實現父類別 method
> 2. 子類別可以擁有自己不同的 property 與 method
> 3. 若子類別對父類別有空實作，可拆成更小 interface

正確使用 LSP



- 不要因為要 code reuse 某一個 class 的功能，而去繼承該 class，然後又任意 override 不需要的功能。因為子類別已經 `覆蓋` 父類別，而不是 `取代` 父類別，這種繼承 `無法` 使用 `多型`
- 應該要定義 interface，將共用的程式碼抽到 abstract class，然後各 class 實踐 interface，因為子類別只是 `實現` 父類別，一定可 `取代` 父類別，這種繼承 `可以` 使用 `多型`

Conclusion

- 要以 `多型` 使用 `繼承`，而不要以 `code reuse` 來使用 `繼承`
- 違反 LSP，也會同時違反 OCP，物件導向就崩潰了