course: COMP6210 week: 02 lesson: 02 topic: solid-principle-2-open-closed-principle

DOWNLOAD PDF CLICK HERE

SOLID Principles

Each week in this section of the course we will look at 1 or 2 of the SOLID principles.

The principles stand for:

- Single Repsonsibility Principle
- **Open Closed Principle**
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency inversion principle

The idea behind using this principles is that you look at how you strucutre and organise your code so that it is easier to maintain. Although it will look like you need to do more work (and it is) for things that are relatively simple, it will help you to read and update your code better once it is in production.

The other takeaway from these principles is that you are going to go through a process of decoupling your code from the other code in your application.

PART 2: - SOLID Principle 2 - Open Closed Principle

Watch a video about it here

The Open Closed principle stands for "open for extension, but closed for modification"

The question you would ask when applying this principle is: If you have a working program that people are using how would you change it, without running the risk that you will break what works?

The open closed principle (OCP from here on) is designed to allow us to add extra roles and / or functionality to our application without breaking what we have already created. If you have not been applying this principle since the start of your project, then yes you most likely will need to change the code that is already in production - but once that process is completed you will have to less so, if at all.

The easiest way to implement OCP is to see what the objects you have in your program have in common. Even if you only use an object once, treat it as if there is a probability that there will be more than a single implementation of it.

This exercise has a series of code examples with it, so let's refer to those while we go through the explanation.

The code below is of a single shape - a rectangle - and in the **Program class** file we create a single instance of the rectangle and then we call the `AreaCalculator` to do the calculation.

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}

public class AreaCalculator
{
    public static List<double> Area(rectangles[] shapes)
    {
        List<double> area = new List<double>();
        foreach (var shape in shapes)
        {
            Rectangle rectangle = (Rectangle)shape;
            area.Add(rectangle.Width * rectangle.Height);
        }

        return area;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Rectangle rect1 = new Rectangle();
        rect1.Width = 24.00;
        rect1.Height = 8.00;

        Rectangle[] allTheRectangles = new Rectangle[1] { rect1 };

        Console.WriteLine(string.Join("\n", AreaCalculator.Area(allTheRectangles)));
    }
}
```

This process break OCP in many ways - because as soon as we want to add something, like another shape, we have to change the working code in to many places. But we are also in stuck that this code is not planned to allow for any extensions, so let's fix that.

What are we dealing with here

First things first, we are working with a shape, so it stands to reason that we may want to add more shapes.

In the `Main` method we instantiate the method and then add it to a collection, which we then print the results of to the screen.

If we were to create a second shape, then we would need to alter the `AreaCalculator` as well. Let's look at the process of that.

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}

public class Elipse
{
    public double Width { get; set; }
    public double Height { get; set; }
}

public class AreaCalculator
{
    public static List<double> Area(object[] shapes)
    {
        List<double> area = new List<double>();
        foreach (var shape in shapes)
        {
            if( shape is Rectangle)
            {
                Rectangle rectangle = (Rectangle)shape;
                area.Add(rectangle.Width * rectangle.Height);
            } else
            {
                Elipse elipse = (Elipse)shape;
                area.Add(elipse.Width * elipse.Height * Math.PI);
            }
        }

        return area;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Rectangle rect1 = new Rectangle();
        rect1.Width = 24.00;
        rect1.Height = 8.00;

        Elipse elipse1 = new Elipse();
        elipse1.Width = 24.00;
        elipse1.Height = 8.00;

        object[] allTheShapes = new object[2] { rect1, elipse1 };

        Console.WriteLine(string.Join("\n", AreaCalculator.Area(allTheShapes)));
    }
}
```

As you can see we had to change quite a bit of code, imagine going through that process everytime we want to add a shape - that is no good. We do now have more information about what the commonalities are between the shapes.

So now we can create an abstract class called `Shapes` and give it a method that holds the `Area` property.

```
public abstract class Shape
{
```

```
    public abstract double Area();
}
```

This class can be the parent class of `Rectangle` and `Elipse` and any other shape to allow us to store the calculations within the class and you get something like this:

```
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width * Height;
    }
}
```

Now it is completely possible that you had a `double Area()` mehtod before the use of inheritance, but because it was not required, the program would have crashed when it came to the use of the `AreaCalculator` and that data type was missing.

So for every shape that implements the `Shape` class as its base class we need to have setup how the area is calculated, but we do that within the class instead of the `AreaCalculator`.

The code for the `AreaCalculator` can then change into this:

```
public class AreaCalculator
{
    public static List<double> Area(Shape[] shapes)
    {
        List<double> area = new List<double>();
        foreach (var shape in shapes)
        {
            area.Add(shape.Area());
        }

        return area;
    }
}
```

You can see that the if statement is completely removed. You can also see that the generic `object` has been replaced into a collection of the `Shape` class. This means we need to make a small adjustment in the `Main` method by replacing the `object` type into the `Shape` type.

```
Shape[] allTheRectangles = new Shape[2] { rect1, elipse1 };
```

Now our program complies with OCP, because when we want to add another shape we don't need to change any logic that would break the original program, but only add the new data.
The format would look like this:

```
public class NameOfShape : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        // return the calculation for the area
    }
}

class Program
{
    static void Main(string[] args)
    {
        ... Rectangle shape...
        ... Elipse shape...
```

```
        NameOfShape someshape1 = new NameOfShape();
        someshape1.Width = 24.00;
        someshape1.Height = 8.00;

        Shape[] allTheShapes = new Shape[2] { rect1, elipse1, someshape1 };
    }
}
```

In the exercises it goes all the way to using interfaces, but this is the next step and bringing in some other parts of the SOLID principles which we will discuss later.