Principios del Diseño Básicos Orientado a Objetos

"Así como conocer las reglas del ajedrez no te hace un buen jugador, lo mismo pasa con la programación orientada a objetos: se puede conocer los conceptos pero no saber utilizarlos."
(Adrian Paredes)

Inicial	Significado	Concepto
S	SRP	Principio de Única Responsabilidad (Single responsibility principle): *A class should have one, and only one, reason to change*.
O	OCP	Principio Abierto/Cerrado: *You should be able to extend a classes behavior, without modifying it*.
L	LSP	Principio de sustitución de Liskov: *Derived classes must be substitutable for their base classes.* Ver también diseño por contrato.
I	ISP	Principio de Segregación de la Interface (Interface segregation principle): *Make fine grained interfaces that are client specific.*
D	DIP	Principio de Inversión de Dependencia (Dependency inversion principle): Depend on abstractions, not on concretions. La Inyección de Dependencias es una de los métodos que siguen este principio.

THE OPEN-CLOSED PRINCIPLE

(Fuente: http://joelabrahamsson.com/entry/simple-example-of-the-open-closed-principle)

"SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR MODIFICATION".

Modules that conform to the open-closed principle have two primary attributes.

1. They are "Open For Extension". This means that the behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.
2. They are "Closed for Modification". The source code of such a module is inviolate. No one is allowed to make source code changes to it.

It would seem that these two attributes are at odds with each other. The normal way to extend the behavior of a module is to make changes to that module. A module that cannot be changed is normally thought to have a fixed behavior. How can these two opposing attributes be resolved?

It's a principle for object oriented design first described by Bertrand Meyer that says that *"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"*.

At first thought that might sound quite academic and abstract. What it means though is that **we should strive to write code that doesn't have to be changed every time the requirements change**. How we do that can differ a bit depending on the context, such as our programming language. When using Java, C# or some other statically typed language the solution often involves inheritance and polymorphism, which is what this example will illustrate.

An example – calculating area

Let’s say that we’ve got a Rectangle class. As most rectangles that I’ve encountered it has a width and a height.

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

Now our customer, Aldford (which apparently means “old river-ford”, did you know that?), wants us to build an application that can calculate the total area of a collection of rectangles. That’s not a problem for us. We learned in school that the area of a rectangle is it’s width multiplied with it’s height and we mastered the for-each-loop a long time ago.

```
public class AreaCalculator{
   public double Area(Rectangle[] shapes)    {
      double area = 0;
      foreach (var shape in shapes)       {
         area += shape.Width*shape.Height;
      }

      return area;
   }
}
```

We present our solution, the AreaCalculator class to Aldford and he signs us his praise. But he also wonders if we couldn’t extend it so that it could calculate the area of not only rectangles but of circles as well. That complicates things a bit but after some pondering we come up with a solution where we change our Area method to accept a collection of objects instead of the more specific Rectangle type. Then we check what type each object is of and finally cast it to it’s type and calculate it’s area using the correct algorithm for the type.

```
public double Area(object[] shapes){
   double area = 0;
   foreach (var shape in shapes)    {
      if (shape is Rectangle)       {
         Rectangle rectangle = (Rectangle) shape;
         area += rectangle.Width*rectangle.Height;
      }       else       {
         Circle circle = (Circle)shape;
         area += circle.Radius * circle.Radius * Math.PI;
      }
   }
   return area;
}
```

The solution works and Aldford is happy.

Only, a week later he calls us and asks: “extending the AreaCalculator class to also calculate the area of triangles isn’t very hard, is it?”. Of course in this very basic scenario it isn’t but it does require us to modify the code. That is, AreaCalculator isn’t **closed for modification** as we need to change it in order to extend it. Or in other words: it isn’t **open for extension**.

In a real world scenario where the code base is ten, a hundred or a thousand times larger and modifying the class means redeploying it’s assembly/package to five different servers that can be a pretty big problem. Oh, and in the real world Aldford would have changed the requirements five more times since you read the last sentence :-)

A solution that abides by the Open/Closed Principle
One way of solving this puzzle would be to create a base class for both rectangles and circles as well as any other shapes that Aldford can think of which defines an abstract method for calculating it’s area.

```
public abstract class Shape{
   public abstract double Area();
}
```

Inheriting from Shape the Rectangle and Circle classes now looks like this:

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

public class Circle : Shape{
   public double Radius { get; set; }
   public override double Area()    {        return Radius*Radius*Math.PI;    }
}
```

As we’ve moved the responsibility of actually calculating the area away from AreaCalculator’s Area method it is now much simpler and robust as it can handle any type of Shape that we throw at it.

```
public double Area(Shape[] shapes){
   double area = 0;
   foreach (var shape in shapes)    {       area += shape.Area();    }
   return area;
}
```

In other words we’ve closed it for modification by opening it up for extension.

When should we apply the Open/Closed Principle?

If we look back our previous example, where did we go wrong? Clearly even our first implementation of the Area wasn't open for extension. Should it have been? I'd say that it all depends on context. If we had had very strong suspicions that Aldford would ask us to support other shapes later on we could probably have prepared for that from the get-go. However, often it's not a good idea to try to anticipate changes in requirements ahead of time, as at least my psychic abilities haven't surfaced yet and preparing for future changes can easily lead to overly complex designs. Instead, I would suggest that we focus on writing code that is well written enough so that it's easy to change if the requirements change.

Once the requirements do change though it's quite likely that they will change in a similar way again later on. That is, if Aldford asks us to support another type of shape it's quite likely that he soon will ask for support for a third type of shape.

So, in other words, I definitely think we should have put some effort into abiding by the open/closed principle once the requirements started changing. Before that, in most cases, I would suggest limiting your efforts to ensuring that the code is well written enough so that it's easy to refactor if the requirements starts changing.

Observación.
El uso más común de extensión es mediante la **herencia** y la reimplementación de métodos. Existe otra alternativa que consiste en utilizar métodos que acepten una **interface** de manera que podemos ejecutar cualquier clase que implemente ese interface. En todos los casos, el comportamiento de la clase cambia sin que hayamos tenido que tocar código interno.

Ejemplo No 1:

Supongamos que estoy programando un reproductor de audio, y tengo una clase `Archivo` con un método `reproducir`, que consiste en un `switch` que llama a un método u otro de la clase dependiendo de si el archivo es un MP3, un WMA o un OGG. Si quiero añadir un nuevo formato más tarde, tendré que modificar el `switch`, añadiendo una nueva expresión que llame a otra función. Si en lugar de utilizar un `switch` definimos `reproducir` como abstracto, y movemos la implementación a varias clases hijas, una por cada formato, cuando queramos añadir un nuevo formato, sólo necesitaremos crear una nueva clase hija.

Ejemplo No.2
Supongamos que estamos trabajando en nuestra versión del juego *PACMAN*. Una de las primeras cosas que podríamos hacer, es crear una clase que represente los personajes del juego. La clase **Personaje**, tendría un atributo que representa la posición actual del personaje en el escenario y una acción que le permita avanzar de posición siempre que sea posible:

```
public abstract class Personaje {
   public Posicion Posicion { get; set; }
   public void AvanzarCasillero(Posicion nuevaPosicion){}
}
```

Otra función que debería cumplir es la de comer otros personajes, por ejemplo *Pacman* puede comer fantasmas (siempre y cuando coma la píldora grande), en tanto que los *fantasmas* tienen como objetivo comerse a Pacman. Además de comer otros personajes, *Pacman* puede comer píldoras. Por lo tanto, todas estas acciones deberíamos incluirlas también:

```
public abstract class Personaje{
   public Posicion Posicion { get; set; }
   public void AvanzarCasillero(Posicion nuevaPosicion){ }
   public void ComerPersonaje(Personaje personaje){ }
   public void ComerPildora(Pildora pildora) { }
 }
```

Podemos ver que la clase **Personaje** tiene más de una razón para el cambio, ya que "comer píldoras" es responsabilidad únicamente de Pacman, y no del Fantasma.

Fuentes:

- https://docs.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/edit?pli=1
- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- http://blog.sanaulla.info/2011/11/19/solid-open-closed-principle/
- http://www.oodesign.com/open-close-principle.html
- http://www.palentino.es/blog/interesante-video-sobre-los-principios-solid/
- http://sebys.com.ar/tag/solid/
- http://www.slideshare.net/JuanjoFuchs/solid-cmo-lo-aplico-a-mi-cdigo