PRICIPIOS SOLID

¿QUÉ ES SOLID?

Solid es un acrónimo de palabras que describen un conjunto de normas para desarrollar software , mantenible , flexible y robusto . Está ligado a un desarrollo ágil y dinámico.

SIGLAS

Están ordenados por orden de importancia

1. Singular responsability principle

2. Open Close Principle

3. Liskov segregation principle

4.Interface segregation principle

5. Dependency inversión principle

1. Singular Renponsability Principle o Principio de Responsabilidad única.

Este principio y el más importante nos viene a decir que cada clase definida dentro de nuestro software debe tener una sola responsabilidad vamos a ver una reglas básicas.

- Cada clase debe tener una única responsabilidad , es decir que cada clase definida debe realizar una única función dentro del programa no confundir con tener varios métodos ahora veremos eso.

- No tiene una única razón para cambiar . Esto quiere decir que la misma clase pueda realizar la misma funcionalidad en diferentes casuísticas sin tener que cambiar su comportamiento para ello.

- Debe tener una alta cohesión , por ejemplo si tenemos varios métodos dentro de la clase pero no tiene una cohesión entre si , tal vez esos métodos deben estar en otra clase.

La forma de racionamiento del ser humano nos ayuda a ver las responsabilidades de cada clase por separado aun que para ello tengamos que tener muchas clases con dos líneas de código.

2.Open Close Principle o Principio de Abierto Cerrado

Este principio se aplica a todas las entidad del Sfotware es decir a todos sus componentes , el principio nos dice que toda entidad debe estar cerrada para lo modificación (se basa en el principio anterior “NO tiene una única razón para cambiar”) pero debe estar abierto a la extensión .

Significa que en nuestro software va haber una parte que va a ser propio del objetivo de ese software pero habrá puntos de accesibilidad para quien pueda usar esa clase o ese objeto pueda agregarle funcionalidades o extensión .

Un claro ejemplo es la sobrescritura de métodos utilizando herencia por ejemplo si tenemos un clase con una propiedad de la Calse Person.

```
public class Person{
  Int virtual Suma(){
   }
}
```

Utilizando la herencia podemos sobrescribir el método

```
public class Student : Person {
        Int overrride Suma(){
        }
}
```

Ahora veremos la partes abiertas y cerradas de nuestro código

```
public class Calculo {
        public var resultado{get;set;}
        //Esta seria la parte abierta del código
        public Person Alumno{
        get{
            if(this.Alumno==null)this.Alumno= new Person();
             }
             return Alumno;
        set{ Alumno = value;}
        }
        //esta la parte cerrada propia de la clase
        public void Resultado ()=> Console.WriteLine(Alumno.Suma());
}
```

Vamos a ver la implementación

```
public static void Main(string Args[]){
        var calculo = new Calculo();
// Como ves la parte abierta puede recibir diferentes extensiones o objetos .
        calculo.Alumno = new Person();
        calculo.Alumno = new Student();
//La parte cerrada realiza su funcionalidad de la misma forma
        calculo.Resultado();
}
```

Con el código queda un poco más claro

3. Liskov Sustitution Principle o Principio de Sustitución de Liskov.

Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas. Esto quiere decir fijándonos en el ejemplo anterior que si por un casual la clase Student tuviera otro método que no fuera sobreescrito tendríamos que controlarlo dentro de la clase Calcula . Ejemplo:

```
public class Student : Person {
        public void ConvertDouble(){
        }
        Int overrride Suma(){
        }
}


public class Calculo {
        public var resultado{get;set;}
        public Person Alumno{
        get{
            if(this.Alumno==null)this.Alumno= new Person();
             }
             return Alumno;
        set{ Alumno = value;}
        }
        public void Resultado (){
        //Aquí no estamos cumpliendo el principio de sustitución de Liskov ya que debemos conocer la clase hija que le pasamo para poder ejecutar el código , Liskov nos viene a decir que cualquier clase heredada puede implementarse igual que clase padre
        if(Alumno.GetType()=typeof(Student)){
                Alumno as Student;
                Alumno.ConvertDouble()
                }
        Console.WriteLine(Alumno.Suma());
        }
}
```

4. Interface Segregation Principle o Pincipio de Segragacion de Interfaces

Antes de nada apuntar que los principios están ordenados en orden de importancia pero en este caso el orden esta invertido , El principio de segregación de interfaces es el menos importante por así decirlo dentro de los principios SOLID pero se altero su orden por una mera razón acústica. Dicho esto , este principio nos dice que un cliente o una clase que implementa una interfaz no debe ser forzado a implementar un método que no va a utilizar .

Veamos el ejemplo anterior para cumplir el principio de liskov hemos creado una Interfaz

IPerson para los dos objetos :

```
public interface IPerson{
        int Suma();
        voidConvertDouble();
}

//Ahora tanto la clase Person como Student implementan la interfaz IPerson
public class Calculo {
        public var resultado{get;set;}
        //Como vemos
        public IPerson Alumno{
        get{
            if(this.Alumno==null)this.Alumno= new Person();
             }
             return Alumno;
        set{ Alumno = value;}
        }
        public void Resultado ()=>Console.WriteLine(Alumno.Suma());

}

public static void Main(string Args[]){
        var calculo = new Calculo();
        calculo.Alumno = new Person();
        calculo.Alumno = new Student();
//Ahora podemos acceder al método convert garcias a la interfaz pero estamos incumpliendo el Princinpiio de segregación de interfaces
        calculo.Alumno.ConvetDouble();
        calculo.Resultado();
}
```

Hemos forzado a la clase person a implementar la interfaz y tener un método que nunca va a usar en este caso ConvertDouble

```
public class Person : IPerson {
        Int virtual Suma(){
        }
        public void ConvertDouble(){
        }
}
```

5. Dependency Inversion Principle o Principio de Inversión de Dependencias

Este principio nos viene a decir que un módulo o clase de alto nivel no debe depender de otro módulos de alto nivel , o lo que es lo mismo no instanciar los objetos de nuestras clases con los constructores propios de las clases ya que así nos estamos casando con ellos.

En pocas palabras eliminar los new dentro de nuestras clases de negocio o funcionales.

Veamos el ejemplo: (En este caso usare una clase ajena que inyectará las instancias de los objetos)

Lo primero es crear una interfaz para cada una de nuestras clases y así no forzarlas a tener los mismos métodos.

```
public interface IPerson{
        int Suma();
}
public interface IStudent{
        int Suma();
        voidConvertDouble();
}
//Ahora cada clase solo impementa la interfaz que necesita.
public class Person : IPerson {
        Int virtual Suma(){
        }
}
public class Student : IStudent {
        Int virtual Suma(){
        }
        public void ConvertDouble(){
        }
}
```

Esta clase nos hará las veces de inyector de dependencias

```
public class InyectorClass
    {

        internal static T Find<T>() where T : class
        {
            if (typeof(T) == typeof(IPerson))
                return new Person() as T;

            if (typeof(T) == typeof(IStudent))
                return new Student() as T;

            throw new TypeLoadException($"cannot find type {typeof(T).Name}");
        }
    }
```

Ahora vemos como queda nuestra clase Calculo:

```
public class Calculo
    {
        private IPerson _IPerson;
        private IStudent _IStudent;

        public IPerson IPerson
        {
            get
            {
                if(this._IPerson == null)
                {
                    this._IPerson = InyectorClass.Find<IPerson>();
                }
                return _IPerson;
            }
            set { pageRetriever = value; }
        }

        public IStudent IStudent
        {
            get
            {
                if(this. _IStudent == null)
                {
                    this. _IStudent = InyectorClass.Find<IStudent>();
                }
                return _IStudent;
            }
            set { _IStudent = value; }
        }


        public void Process(string baseUrl)
        }
                IStudent.ConvertDouble();
                Console.WriteLine(IPerson.Suma());
        }
```

Como vemos nuestra calse Calculo o módulo de alto nivel ya no depende de otras Clases de alto nivel Como Student o Person solo de sus interfaces , por lo tanto solo con cambiar la clase InyectorClass podríamos trabajar con distintas clase que implementen la misma interfaz.

Os adjunto un link de github con un proyecto refactorizado paso a paso para cumplir los principios Solid.
https://github.com/leomicheloni/SOLID-Principles-examples