Principis SOLID

Els principis SOLID són principis de disseny de la programació orientada a objectes. Quan es combinen, faciliten al programador el desenvolupament de software fàcil de mantenir i estendre. Són part del desenvolupament de software àgil o adaptatiu.

S (Single Responsability)

A continuació veiem un exemple de codi que no segueix el principi de responsabilitat única. La classe Customer fa tasques que no li corresponen. S'hauria de limitar a fer validacions de les dades, crides a la capa de dades corresponent, etc. Ara bé, veiem que dins el catch fa una tasca de Logging. Si demà afegim un nou Logger, com el de consola, necessitarem canviar la classe Customer.

```
class Customer
{
     public void Add()
     {
         try
         {
             // Database code goes here
         }
         catch (Exception ex)
         {
                System.IO.File.WriteAllText(@"c:\Error.txt", ex.ToString());
         }
     }
}
```

Seguint el principi, movem l'activitat de Logging a una altra classe que anomenem FileLogger. Aquesta classe només s'encarrega d'activitats de Logging.

```
class FileLogger
{
     public void Handle(string error)
     {
         System.IO.File.WriteAllText(@"c:\Error.txt", error);
     }
}
```

Ara la classe Customer pot delegar l'activitat de Logging a la classe FileLogger.

```
class Customer
{
     private FileLogger obj = new FileLogger();
     publicvirtual void Add()
     {
         try
         {
             // Database code goes here
         }
         catch (Exception ex)
         {
             obj.Handle(ex.ToString());
         }
     }
}
```

O (Open-closed)

A continuació veiem un exemple de codi que no segueix el principi obert-tancat. El problema resideix en el condicional del mètode *getDiscount*. Si afegim un nou tipus de Customer, haurem d'afegir un condicional més en aquest mètode i estarem modificant la classe.

```
class Customer
{
        private int _CustType;

        public int CustType
        {
            get { return _CustType; }
            set { _CustType = value; }
        }

        public double getDiscount(double TotalSales)
        {
                if (_CustType == 1)
                {
                    return TotalSales - 100;
                }
                else
                {
                    return TotalSales - 50;
                }
        }
}
```

Cada cop que canviem la classe Customer hem d'assegurar que totes les funcionalitats prèvies al canvi i aquelles que es relacionen amb la classe, segueixen funcionant com abans. Hem de testejar-les. Si enlloc de modificar optem per estendre, cada cop que haguem d'afegir un tipus de Customer el que farem es crear noves classes com veiem tot seguit amb SilverCustomer i GoldCustomer. Així el codi de la classe no s'ha de tocar i només es testegen les noves classes.

```
class Customer
{
        public virtual double getDiscount(double TotalSales)
        {
            return TotalSales;
        }
}
class SilverCustomer : Customer
{
        public override double getDiscount(double TotalSales)
        {
            return base.getDiscount(TotalSales) - 50;
        }
}
```

```
class goldCustomer : SilverCustomer
{
        public override double getDiscount(double TotalSales)
        {
            return base.getDiscount(TotalSales) - 100;
        }
}
```

Ara la classe Customer està tancada a canvis però oberta a extensions.

L (Liskov substitution)

Suposem que el nostre sistema vol calcular descomptes per Enquiries (consultes). No es tractaria de clients reals sinó de clients potencials. És per això que de moment no volem guardar-los a la base de dades.

Així doncs, creem una nova classe anomenada Enquiry que hereta de la classe Customer. Amb *override* sobreescrivim els mètodes getDiscount i Add de la classe Customer. A més, al mètode Add llancem una excepció perquè ningú pugui afegir una Enquiry a la BD.

```
class Enquiry : Customer
{
        public override double getDiscount(double TotalSales)
        {
            return base.getDiscount(TotalSales) - 5;
        }

        public override void Add()
        {
            throw new Exception("Not allowed");
        }
}
```

La jerarquia d'herència de Customer seria la de la *Figura 1*. Customer és la classe Pare de Gold, Silver i Enquiry, que són les classes filles.

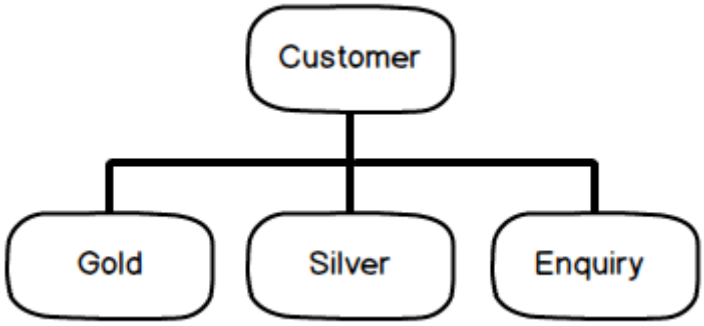

Figura 1

Segons el polimorfisme un objecte de la classe pare Customer pot apuntar a qualsevol objecte de les classes filles. En el codi següent veiem com es crea una llista d'objectes Customer i gràcies al polimorfisme s'hi poden afegir objectes Silver, Gold i Enquiry.

```
List<Customer> Customers = new List<Customer>();
Customers.Add(new SilverCustomer());
Customers.Add(new goldCustomer());
Customers.Add(new Enquiry());

foreach (Customer o in Customers)
{
     o.Add();
}
```

Per l'herència l'objecte Customer pot apuntar a qualsevol objecte dels seus fills, però la crida al mètode Add per l'objecte Enquiry causa l'error de la *Figura 2* per l'excepció que llancem.

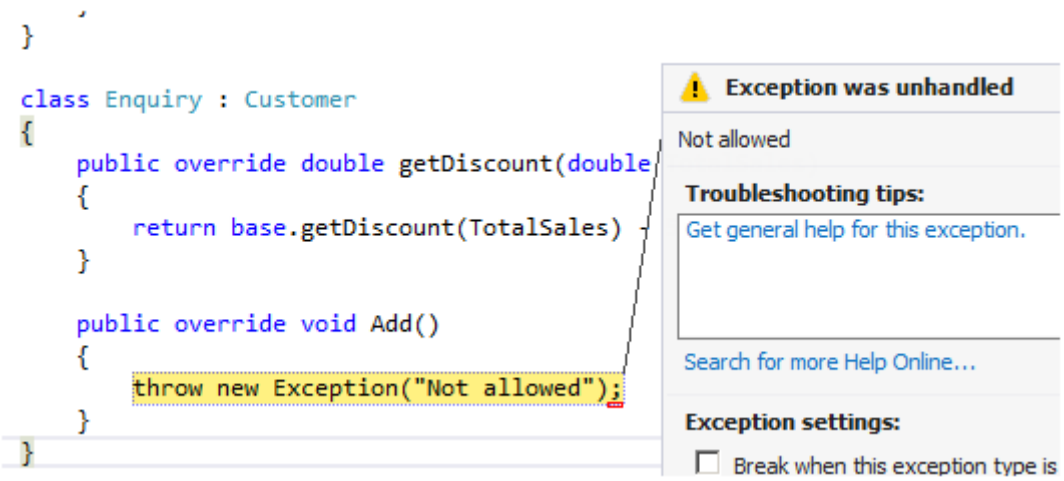

Figura 2

La consulta (Enquiry) té un càlcul del descompte, sembla un Customer però **no és un Customer**. Així doncs, el pare no pot substituir l'objecte fill. Dit d'una altra manera, Customer no és de fet el pare de la classe Enquiry. En resum, Enquiry es una entitat diferent per si mateixa.

El principi Liskov diu que el pare ha de poder substituir fàcilment l'objecte fill. Com es veu a continuació, per implementar-lo necessitem crear dues interfícies: una per obtenir el descompte i una altra per l'addició de clients a la base de dades.

```
interface IDiscount
{
        double getDiscount(double TotalSales);
}


interface IDatabase
{
        void Add();
}
```

Ara, com podem observar, la classe Enquiry només implementarà la interfície IDiscount ja que no està interessada en el mètode Add.

```
class Enquiry : IDiscount
{
        public  double getDiscount(double TotalSales)
        {
            return TotalSales - 5;
        }
}
```

En canvi, com s'observa tot seguit, la classe Customer implementarà les dues interfícies IDiscount i IDatabase ja que també vol guardar els clients a la base de dades.

```
class Customer : IDiscount, IDatabase
{
        private MyException obj = new MyException();

        public virtual void Add()
        {
            try
            {
                // Database code goes here
            }
            catch (Exception ex)
            {
                obj.Handle(ex.Message.ToString());
            }
        }

        public virtual double getDiscount(double TotalSales)
        {
            return TotalSales;
        }
}
```

Ara podem crear una llista d'objectes IDatabase i afegir objectes a ella. Si cometem un error i volem afegir un objecte de la classe Enquiry, el compilador es queixarà com es pot observar a la *Figura 3*.

```
List<IDatabase> Customers = new List<IDatabase>();
Customers.Add(new SilverCustomer());
Customers.Add(new goldCustomer());
Customers.Add(new Enquiry());

foreach (IDatabase o in Customers)
{
    o.Add();
}
```

Figura 3

I (Interface segregation)

Ara suposem que la nostra classe Customer és un component molt popular i és consumit per molts clients. La *Figura 4* ho representa.

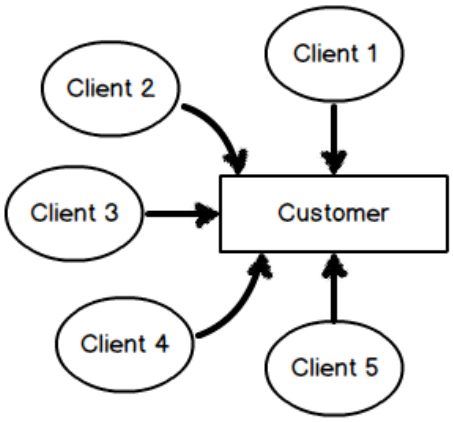

Figura 4

Imaginem que clients nous demanen un mètode que ajudi a llegir dades del Customer. La interfície IDatabase s'hauria de modificar com es veu tot seguit.

```
interface IDatabase
{
        void Add(); // old client are happy with these.
        void Read(); // Added for new clients.
}
```

Si visualitzem el nou requeriment, tenim 2 tipus de clients: els que només volen fer servir el mètode Add i els que volen utilitzar Add i Read.

Canviant la interfície, molestem els clients que no estan interessats en el nou mètode. Una millor solució seria mantenir els clients existents a part i proveir als nous clients amb el mètode Read.

Creem una interfície enlloc de modificar l'actual. D'aquesta manera mantenim la interfície actual IDatabase tal com és i tenim una nova interfície IDatabaseV1 amb el mètode Read.

```
interface IDatabaseV1 : IDatabase // Gets the Add method
{
     void Read();
}
```

Ara podem crear classes noves que implementin el mètode Read i satisfer les peticions dels nous clients i els clients anteriors no es veuen afectats pel canvi. Segueixen utilitzant la interfície antiga que no te el mètode Read.

```
class CustomerWithRead : IDatabase, IDatabaseV1
{

    public void Add()
    {
         Customer obj = new Customer();
         Obj.Add();
    }

    public void Read()
    {
         // Implements  logic for read
    }
}
```

Un exemple d'ús per les dues interfícies és el següent:

```
IDatabase i = new Customer(); // 1000 happy old clients not touched
i.Add();

IDatabaseV1 iv1 = new CustomerWithRead(); // new clients
Iv1.Read();
```

D (Dependency inversion)

Prèviament a la classe Customer havíem creat una classe Logger per satisfer el principi de responsabilitat. Suposem que es creen diferents tipus de classes Logger.

```
class Customer
{
    private FileLogger obj = new FileLogger();

    public virtual void Add()
    {
         try
         {
             // Database code goes here
         }
         catch (Exception ex)
         {
             obj.Handle(ex.ToString());
         }
    }
}
```

Per tenir control creem una interfície comuna i la utilitzem per crear nous tipus de Logger.

```
interface ILogger
{
        void Handle(string error);
}
```

A continuació tenim 3 tipus de Logger diferents.

```
class FileLogger : ILogger
    {
        public void Handle(string error)
        {
            System.IO.File.WriteAllText(@"c:\Error.txt", error);
        }
    }
```

```
class EverViewerLogger : ILogger
    {
        public void Handle(string error)
        {
            // log errors to event viewer
        }
    }
```

```
class EmailLogger : ILogger
  {
      public void Handle(string error)
      {
          // send errors in email
      }
  }
```

Ara en funció dels paràmetres de configuració en un moment donat utilitzarem classes de Logger diferents. A continuació veiem que per aconseguir-ho, a la classe Customer, hem fet servir un condicional que decideix quina classe Logger s'ha d'utilitzar.

```
class Customer : IDiscount, IDatabase
{
     private IException obj;

     public virtual void Add(int Exhandle)
     {
           try
           {
                // Database code goes here
           }
           catch (Exception ex)
           {
                if (Exhandle == 1)
                {
                    obj = new MyException();
                }
                else
                {
                    obj = new EmailException();
                }
                obj.Handle(ex.Message.ToString());
           }
     }
}
```

El codi anterior, tot i així, viola el principi de responsabilitat única. Ho fa en el sentit de decidir quins objectes s'han de crear. No és responsabilitat del Customer prendre aquesta decisió.

El problema més gran es la paraula *new*. En aquest sentit Customer està agafant responsabilitats extra sobre quin objecte s'ha de crear. Si invertim o deleguem aquesta responsabilitat a algú altre resolem aquest problema.

Aplicant la inversió tenim el següent codi. Ara el constructor de Customer rep l'objecte ILogger d'un altre. Per tant, és responsabilitat del client que consumeix l'objecte Customer decidir quin Logger s'utilitza.

```
class Customer : IDiscount, IDatabase
 {
        private Ilogger obj;
        public Customer(ILogger i)
        {
            obj = i;
        }
}
```

El client passarà com a paràmetre l'objecte Logger i el Customer ja no ha de tenir les condicions IF que decideixen quin Logger s'usa. La crida a la constructora de Customer quedaria com es veu tot seguit.

```
IDatabase i = new Customer(new EmailLogger());
```

Bibliografia

koirala, S. (2018). SOLID architecture principles using simple C# examples - CodeProject. [en línia] Codeproject.com. Disponible a: https://www.codeproject.com/Articles/703634/SOLID-architecture-principles-using-simple-Csharp [Accedit 16/04/2018].