PATRONES DE DISEÑO

ABSTRACT FACTORY

El patrón de estructura Abstract Factory se emplea en los casos en los que el cliente , ya sea usuario, interfaz gráfica o una clase necesita varios productos diferentes , la Factory será la encargada de proveer estos productos al cliente , en este caso lo haremos de manera que podamos añadir tantos productos como queramos sin la necesidad de tener que modificar la estructura que sirve los productos al cliente , de esta forma cumplimos el principio de Open Close facilitando la extensión de nuestro código pero sin la necesidad de modificar el código de origen , así como el primer principio , la factoría se encarga de proveer los objetos a los clientes.

Vamos a ver un caso Practico para verlo de manera más clara:

Tenemos un concesionario y vendemos Suv y Cupes :

```
//En este caso tenemos dos tipos de productos

//Como trabajamos con dos marcas diferentes tenemos 4 productos diferentes
finales
  public abstract class Suv { }
    public abstract class Cupe { }

    public class Focus : Cupe { }
    public class EcoSport : Suv { }

    public class X1 : Suv { }
    public class Serie4 : Cupe { }
```

Ahora procedemos a crear la Abstract Factory que tendrá un interfaz que definirá su comportamiento:

```
public interface IFactory
    {
        Suv CrearSuv();
        Cupe CrearCupe();
    }
```

Como vemos la estructura de la Factoria define los métodos para crear suvs y cupes

```
  public abstract class AbstractFactory: IFactory
    {
        public Suv autos { get; set; }
        public Cupe autoc { get; set; }
        public virtual Cupe CrearCupe()
        {
            return autoc;
        }

        public  virtual Suv  CrearSuv()
        {
            return autos;
        }
    }
```

Ahora debemos definir las factorías concretas que serán las que nos devuelvan los productos que el cliente necesita

Las factorías concretas que devolverán el producto concreto .

```
  public class ConcretFactoryFord : AbstractFactory,IFactory
    {

        public override Cupe CrearCupe()
        {
            autoc = new Focus();
            return autoc;
        }

        public override Suv CrearSuv()
        {
            autos = new EcoSport();
            return autos;
        }


    }
    public  class ConcretFactoryBmw :AbstractFactory ,IFactory
    {
        public override Cupe CrearCupe()
        {
            autoc = new Serie4();
            return autoc;
        }

        public override Suv CrearSuv()
        {
            autos = new X1();
            return autos;
        }
    }
```

Ahora implementamos la factoría en nuestro Main , como veréis el objeto que el cliente necesita siempre lo declararemos con una clase abstracta o interfaz , nunca definiremos el objeto concreto , ya que es la factoría la que se encarga de producirlo.

```
static void Main(string[] args)
        {
            AbstractFactory factoryFord = new ConcretFactoryFord();
            Cupe Focus = factoryFord.CrearCupe();
            Suv EcoSport = factoryFord.CrearSuv();

            AbstractFactory factoryBmw = new ConcretFactoryBmw();
            Cupe Serie4 = factoryBmw.CrearCupe();
            Suv X1 = factoryBmw.CrearSuv();

        }
```

Como vemos si quisiéramos añadir mas marcas de coches nos seria muy fácil ya que solo tendríamos que añadir una factoría concreta para esa marca.

Obviamente este es una de las maneras de hacer Abstract Factory podemos usar interfaces en vez de clases abstractas pero eso dependerá de las características de nuestro Software.

SINGLETON

En ingeniería de software, **singleton** o instancia única es un patrón de diseño que permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

El patrón singleton se implementa creando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con modificadores de acceso como protegido o privado).

La instrumentación del patrón puede ser delicada en programas con múltiples hilos de ejecución. Si dos hilos de ejecución intentan crear la instancia al mismo tiempo y esta no existe todavía, sólo uno de ellos debe lograr crear el objeto. La solución clásica para este problema es utilizar exclusión mutua en el método de creación de la clase que implementa el patrón.

Como siempre veamos un ejemplo , Utilizaremos las provincias como ejemplo ya que suelen ser constantes .

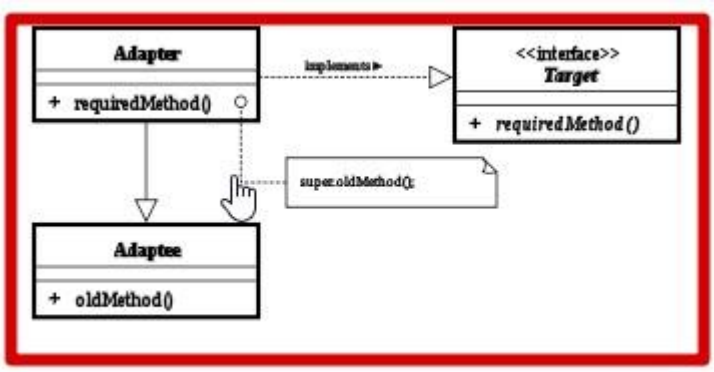
```
public sealed class Singleton
    {
        //La informacion que queremos que sea fija la definiremos en el constructor
        //y sera de solo lectura ya que no queremos que se modifique
        public readonly string[] Provincias;
        //Declaramos un objeto estatico que será la instancia del singleton , ya que solo
queremos una instancia en toda la ejecucion
        private static Singleton instance;
        //El objeto padock nos ayudará para cuando haya mas de un trhead en el programa no
puedan dos trhead instanciar el singleton a la vez
        private static Object padock = new Object();
        private Singleton()
        {
            Provincias = new string[] { "Burgos","Barcelona"};
        }
        //Si el objeto estatico instance es null lo creará , sino devuelcve el ya construido
        //lock nos ayuda con el multitrheading peromitiendo que solo un hilo de ejecucion acceda
al constructor.
        public static Singleton Instance
        {
            get

            {
                if (instance == null)
                {
                    lock (padock)
                    {
                        if (instance==null)
                        {
                            instance = new Singleton();
                        }
                    }
                }
                return instance;
            }
        }
    }
```

ADAPTER PATTERN

El patrón **adaptador** (adapter) se utiliza para transformar una interfaz en otra, de tal modo que una clase que no pueda utilizar la primera haga uso de ella a través de la segunda.

En resumen cuando una clase debe implementar dos interfaces pero son incompatibles entre si.



Vamos a ver un ejemplo rápido :

```
    //Estas dos interface son incompatibles dentro de la misma clase ya que comparten métodos
con el mismo nombre pero diferentes parametros
    public interface IEjemplo
    {
        int Suma();
        int Resta();
    }
    public interface Iejemplo2
    {
        int Suma(int x);
        int Resta(int y);

    }
//La clase adapter utiliza las dos interfaces , de manera que así son compatibles.
public class Calculo : Iejemplo2
    {

        public int Resta(int x)
        {
            throw new NotImplementedException();
        }

        public int Suma(int y)
        {
            throw new NotImplementedException();
        }
    }
    public class Adapter : IEjemplo
    {
        Iejemplo2 iejemplo2;
        int x;
        public Adapter()
        {
            iejemplo2 = new Calculo();
        }
        public int Resta()
        {
            return iejemplo2.Resta(x);
        }

        public int Suma()
        {
            return iejemplo2.Resta(x);
        }
    }
```

PATRÓN BUILDER

El patrón *Builder* es un patrón creacional cuyo objetivo es **instanciar objetos complejos** que generalmente están **compuestos por varios elementos** y que admiten **diversas configuraciones**. Cuando hablamos de "construcción" nos referimos al proceso, mientras que cuando hablamos de "representación" nos estaremos refiriendo a los datos que componen el objeto. Se encargará, por tanto, de encapsular todo el proceso de generación de modo que únicamente necesite los detalles necesarios para "personalizar" el objeto, devolviendo como resultado una instancia del objeto complejo que deseamos construir.

Veamos un ejemplo :

Tenemos una tienda de pizza que debe generar pizzas con diferentes ingredientes pero todas comparten un patrón de creación.

```
public enum TipoMasa { Fina,Pan,Queso}

public class Pizza
{//Este es el producto a construir
    public string[] Quesos{ get; set; }
    public TipoMasa masa { get; set; }
    public string Ingredinte1 { get; set; }
    public string Ingredinte2 { get; set; }
    public string Ingredinte3 { get; set; }
    public bool Gluten { get; set; }
}

public interface IPizzaBuilder
{
    //En este caso creamos una interfaz , pero podria usarse una clase abstracta
    //que ya contenga la propiedad Pizza
    void Masa();
    void CantidadQuesos();
    void Ingredientes();
    void Gluten();
    Pizza CrearPizza();
}

public class CuatroQuesosBuilder : IPizzaBuilder
{
    //Los constructores concretos generan las pizzas con diferentes caracteristicas
    private Pizza Pizza { get; }
    public void CantidadQuesos()
    {
        Pizza.Quesos=new string[]{ "Mozzarella","Gorgonzola","Grand Apadano","Parmichiano"};
    }


    public void Gluten() => Pizza.Gluten = true;

    public void Ingredientes()
    {
        Pizza.Ingredinte1="Tomate";
        Pizza.Ingredinte2 = null;
        Pizza.Ingredinte3 = null;
    }

    public void Masa() => Pizza.masa = TipoMasa.Fina;

    public Pizza CrearPizza()
    {
       return this.Pizza;
    }
}
```

```
    public class EspecialCasaBuilder : IPizzaBuilder
    {
        private Pizza Pizza { get; }
        public void CantidadQuesos()
        {
            Pizza.Quesos = new string[] { "Mozzarella" };
        }

        public Pizza CrearPizza() { return this.Pizza; }

        public void Gluten() => Pizza.Gluten = false;

        public void Ingredientes()
        {
            Pizza.Ingredinte1 = "Becon";
            Pizza.Ingredinte2 = "Peperoni";
            Pizza.Ingredinte3 = "Champiñones";
        }

        public void Masa() => Pizza.masa = TipoMasa.Pan;
    }

    public class DirectorPizaa
    {
        //El director es el encargado de llamar a el constructor específico y entrgar la Pizza
en este caso
        private IPizzaBuilder _builder;

        public DirectorPizaa(IPizzaBuilder PizzaParaCrear)
        {
            _builder = PizzaParaCrear;
        }
        public void CrearPizza()
        {
            _builder.CantidadQuesos();
            _builder.Masa();
            _builder.Gluten();
            _builder.Ingredientes();
        }
        public Pizza EntregarPizza()
        {
            return _builder.CrearPizza();
        }
    }
```

El módulo cliente solo tendría que pasarle el tipo de producto que quiere como vemos en el ejemplo:

```
//Buider
            IPizzaBuilder pizzaElegida = new EspecialCasaBuilder();
            DirectorPizaa directorPizza = new DirectorPizaa(pizzaElegida);
            directorPizza.CrearPizza();
            Pizza pedido=directorPizza.EntregarPizza();

            IPizzaBuilder pizzaElegida1 = new CuatroQuesosBuilder();
            DirectorPizaa directorPizza1 = new DirectorPizaa(pizzaElegida);
            directorPizza1.CrearPizza();
            Pizza pedido1 = directorPizza1.EntregarPizza();
```

Os dejo los ejemplos colgados en Git:

FLUENTBUILDER

Partiendo del ejemplo anterior puede llegar a ser tedioso el tener que escribir todo los métodos con el objeto que los usa en este caso podemos optar por la opción de utilizar una interfaz fluida para crear los productos del builder :

Una interfaz fluida es aquella en la que sus todos sus método devuelven el objeto actual y de esta manera sus métodos puedan ser anidados , en consecuencia se mejora el rendimiento y la consistencia del producto que genera el builder.

Veamos la implementación con el ejemplo de las pizzas:

```
public interface IFluentPizzaBuilder
    {
        //En este caso creamos una interfaz fluida, pero podria usarse una clase abstracta
        //que ya contenga la propiedad Pizza
        IFluentPizzaBuilder Begin();
        IFluentPizzaBuilder Masa();
        IFluentPizzaBuilder CantidadQuesos();
        IFluentPizzaBuilder Ingredientes();
        IFluentPizzaBuilder Gluten();
        Pizza CrearPizza();
        //El último metodo sera el que nos devuelva el producto final
    }
    public class EspecialCasaFluentBuilder : IFluentPizzaBuilder
    {
        private Pizza Pizza;
        public IFluentPizzaBuilder Begin()
        {
            Pizza = new Pizza();
            return this;
        }

        public IFluentPizzaBuilder Masa()
        {
            Pizza.masa = TipoMasa.Pan;
            return this;
        }
        public IFluentPizzaBuilder Gluten()
        {
            Pizza.Gluten = false;
            return this;
        }
        public IFluentPizzaBuilder CantidadQuesos()
        {
            Pizza.Quesos = new string[] { "Mozzarella" };
            return this;
        }
        public IFluentPizzaBuilder Ingredientes()
        {
            Pizza.Ingredinte1 = "Becon";
            Pizza.Ingredinte2 = "Peperoni";
            Pizza.Ingredinte3 = "Champiñones";
            return this;
        }


        public Pizza CrearPizza() { return this.Pizza; }
    }
    class FluentBuilder
    {
        public void Director()
        {
            //Como vemos nuestro código también se vuelve mucho mas legible
            IFluentPizzaBuilder InterfazFluente = new EspecialCasaFluentBuilder();
            Pizza selecionada = InterfazFluente.Begin()
                                               .Masa()
                                               .Gluten()
                                               .CantidadQuesos()
                                               .Ingredientes()
                                               .CrearPizza();
        }

    }
```