



Tema 9.6

Descargar estos apuntes en [pdf](#) o [html](#)

Índice

▼ Principios SOLID

- Conceptos básicos
- S → Principio de responsabilidad simple (SRP)
- O → Principio de Abierto/Cerrado (OCP)
- L → Liskov Substitution Principle (LSP)
- I → Principio de Segregación de Interfaces (ISP)
- ▼ D → Principio de Inversión de Dependencias (DIP)
 -  Patrón creacional Factory Method
 -  Patrón creacional Fluent Builder

Principios SOLID

Conceptos básicos

- La experiencia en el desarrollo usando el paradigma POO, dio lugar a cinco reglas que todo desarrollador OO debería seguir para crear un sistema que sea **fácil de mantener** y **escalable** a través del tiempo.
- Estas **cinco reglas** son conocidas como los principios **SOLID** y las vamos a enumerar a continuación.
- Algunas ya las hemos visto indirectamente durante el curso.



S → Principio de responsabilidad simple (SRP)

Definición: Descrito por **Robert C. Martin**. Este principio establece que un componente del software (método, clase o módulo) debe estar centrado en una **única tarea** (tener solo una responsabilidad).

Cómo ocurría en programación modular, que si un módulo hacía más de una cosa genera acoplamiento. Lo mismo sucederá con nuestras clases.

👉 Importante: No siempre es tan obvio su incumplimiento. Por eso, siempre que añadamos un método a una clase deberemos pensar si realmente debería ir en la misma o realmente se está convirtiendo en un **'Cajón Desastre'**.

Vemos cómo **'darnos cuenta'** de su incumplimiento a través de un par de ejemplos...

1. Supongamos la siguiente clase **Rectangulo** donde hemos añadido dos métodos de utilidad sobre un conjunto de objetos de la clase.

```
1 public class Rectangulo
2 {
3     public double Alto { get; }
4     public double Ancho { get; }
5     public Rectangulo(double alto, double ancho)
6     {
7         Alto = alto;
8         Ancho = ancho;
9     }
10    public static double SumaÁreas(Rectangle[] Rectangulos) {...}
11    public static double SumaPerímetros(Rectangle[] Rectangulos) {...}
12 }
```

Por algún tipo de requerimiento, estos sumatorios han acabado formando parte de nuestra lógica de la aplicación. En principio, podríamos pensar que como la única clase que interviene es **Rectangulo** y además, vamos a realizar operaciones de utilidad sobre un conjunto de objetos de este tipo, su lugar más apropiado sería la clase Rectangulo. Además, muchas veces al programar somos 'vagos' y hacemos la opción más rápida y simple '*sin pensar*'.

Sin embargo, estas operaciones no forman parte de la lógica que podría tener un Rectangulo en la vida real, esto es, no definen operaciones sobre un objeto **Rectangulo**. Una pista importante, es que en lugar de uno solo Rectangulo, al método llegan un conjunto de ellos y eso nos debería '*chirriar*'. Es más, si fuera una operación sobre un solo objeto Rectangulo, no tendría sentido que fuera un método de clase sino uno de 'instancia' de un determinado Rectangulo.

Para cumplir con el principio, quitamos la funcionalidad de sumatorios de la clase **Rectangulo** e introducimos un par de clases encargadas de realizar las operaciones sobre el los conjuntos de Rectangulos, una posible implementación podría ser la siguiente...

```
public class CalculosSobreAreas
{
    public static double Suma(Rectangulo[] Rectangulos) {...}
}
public class CalculoSobrePerimetros
{
    public static double Suma(Rectangulo[] Rectangulos) {...}
}
```

Entonces así cada clase tiene una sola responsabilidad: una representa un Rectangulo y las otras se encargan de hacer operaciones relacionadas con ellos.

2. Veamos otro ejemplo sutil, pero interesante de la aplicación de este principio ...

Supongamos que el estado de un objeto persona queremos formatearlo a cadena de diferentes formas para representarlo en diferentes sitios.

```
class Persona
{
    public string Nombre { get; private set; }
    public short Edad { get; private set; }
    ...
    public string FomateaALinea() => $"Nombre: {Nombre}, Edad: {Edad}";
    public string FomateaATabla() => $"{"Nombre", -8}{ "Edad", -8}\n{Nombre, -8}{Edad, -8}";
}
```

Aparentemente no debería haber problema porque **esta vez parecen operaciones sobre objetos de la clase** y además acceden a la propiedades del mismo. Pero, la clase puede empezar a ser '**pesada**' si, por requerimientos del cliente, tendremos que añadir muchos métodos de Formato y por tanto de alguna manera también estaríamos incumpliendo este principio.

Una posibilidad, sería llevarnos la responsabilidad del formateo a otras clases haciendo más '**liviana**' de la siguiente manera...

```
1 class FomateaPersonaATabla
2 {
3     private Persona Persona{ get; set; }
4     public FomateaPersonaATabla(Persona persona) { ... }
5
6     public override string ToString() =>
7     $"{ "Nombre", -8 } { "Edad", -8 } \n { Persona.Nombre, -8 } { Persona.Edad, -8 }";
8 }
9
10 class FomateaPersonaALinea
11 {
12     private Persona Persona { get; set; }
13     public FomateaPersonaALinea(Persona persona) { ... }
14
15     public override string ToString() =>
16     $"Nombre: {Persona.Nombre}, Edad: {Persona.Edad}";
17 }
```

O → Principio de Abierto/Cerrado (OCP)

Definición: Establece que el diseño debe ser **abierto para poderse extender**, pero **cerrado para poderse modificar**.

En otras palabras, el Software debe ser diseñado pensando en el crecimiento de la aplicación, pero **el nuevo código debe requerir el menor número de cambios en el código existente**. Otra forma simple de expresarlo podría sere: *'Abierto a lo nuevo, cerrado para lo viejo'*.

Formas de aplicarlo en POO:

1. El **uso más común** de extensión es **mediante el mecanismo de herencia y la invalidación** o sobrescritura de métodos.
2. Utilizando una **abstracción**, por ejemplo utilizando métodos que acepten una **interface**, de manera que podemos pasar cualquier clase que lo implemente.
3. Usando el Patrón **Decorator** cómo vimos en el tema y salida por ejemplo cuando vimos el *'decorator'* `BufferedStream` sobre un `FileStream`. De esta manera podremos añadir futuros *'decoradores'* sin modificar la clase original.

Vamos a ver a través de un **ejemplo** como aplicar diferentes estrategias.

Supongamos que realizamos la siguiente abstracción de ordenador para tener el coste de diferentes configuraciones de ordenadores.

```
1 public class OrdenadorBásico
2 {
3     public virtual double Procesador => 56D;
4     public virtual double HDD => 30D;
5     public virtual double Gráfica => 41.99D;
6     public virtual double RAM => 23.5D;
7     public virtual double Precio => Procesador + HDD + Gráfica + RAM;
8 }
9 public class OrdenadorOficina : OrdenadorBásico
10 {
11     public override double Procesador => 100D;
12     public override double HDD => 150D;
13     public override double Gráfica => 110D;
14     public override double RAM => 90D;
15 }
```

Podríamos extender nuevas configuraciones además de `OrdenadorOficina` tales como: `OrdenadorGammer`, `OrdenadorServidor` y así todas las que se nos ocurran en el futuro.

```
static void Main()
{
    OrdenadorBásico setup = new OrdenadorOficina();
    Console.WriteLine($"Precio: {setup.Precio}");
}
```

Hasta aquí podemos decir que estamos cumpliendo **OCP**, ya que si queremos combinar otros precios solo tenemos que heredar de `OrdenadorBásico`. Sin embargo, nuestras configuraciones solo admiten **4 componentes básicos** y si quisiéramos definir modelos con algún componente nuevo en forma de propiedad como `Refrigeración`. **¿Cómo lo haríamos?**

Bueno pues aplicando herencia, invalidaciones y ligadura dinámica de la siguiente forma...

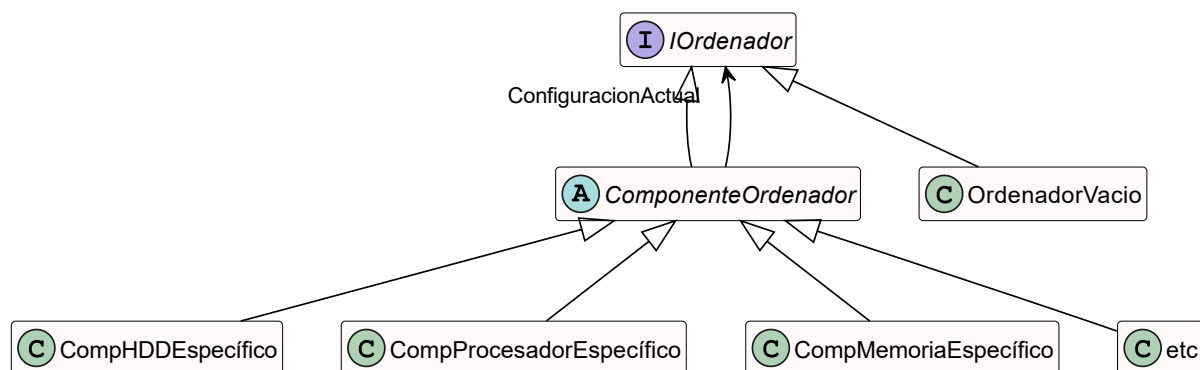
```
1 public class OrdenadorGaming : OrdenadorBásico
2 {
3     public override double Procesador => 3600;
4     public override double HDD => 255.200;
5     public override double Gráfica => 1200;
6     public override double RAM => 1100;
7     public virtual double Refrigeración => 900;
8     public override double Precio => base.Precio + Refrigeración;
9 }
10
11 static void Main()
12 {
13     OrdenadorBásico setup = new OrdenadorOficina();
14     Console.WriteLine($"Precio: {setup.Precio}");
15     setup = new OrdenadorGaming();
16     Console.WriteLine($"Precio: {setup.Precio}");
17 }
```

👉 **Importante:** Seguimos teniendo la clase **OrdenadorBásico** 'cerrada' a la modificación, pero en ocasiones esta estrategia puede ser poco 'flexible', sobre todo en casos en que las combinaciones posibles son muchísimas y nos obligarían a crear muchas de extensiones de funcionalidad. En nuestro ejemplo, no hemos contemplado que la cantidad de componentes posibles es enorme y puede darnos lugar a decenas de definiciones para nuestras configuraciones.

En ese caso una alternativa más 'flexible', sería usar una versión del **Patrón Decorator**.

🚧 **Nota:** La idea de este ejemplo, **no es aprendernos el patrón**, sino más bien ver una forma nueva de extender la funcionalidad de una clase ya dada y ver posibles usos de interfaces y tipos abstractos..

Su esquema básico sería el siguiente...



Para ello, definiremos la abstracción de la funcionalidad de **Ordenador** a través de un interfaz.

```
public interface IOrdenador
{
    double Precio { get; }
}
```

Definiremos el decorador componente que extenderá de **IOrdenador** y que referenciará a la configuración del ordenador actual (**acumulada**). Esto le obliga a implementar la propiedad **Precio** donde añada a esta configuración el precio de dicho componente '**acumulado**'.

```
public abstract class ComponenteOrdenador : IOrdenador
{
    protected IOrdenador ConfiguracionActual { get; set; }
    public abstract double Precio { get; }
}
```

Definiremos ahora extensiones de **ComponenteOrdenador** que envolverán objetos con la funcionalidad final del ordenador **IOrdenador**. La cual referenciará **la configuración 'acumulado' hasta el momento** y la '**decorará**' añadiéndole el precio de un componente.

```
public class SSDRapido : ComponenteOrdenador
{
    public SSDRapido(IOrdenador c) => ConfiguracionActual = c;
    public override double Precio => ConfiguracionActual.Precio + 255.20D;
}
public class Procesador8Nucleos : ComponenteOrdenador
{
    public Procesador8Nucleos(IOrdenador c) => ConfiguracionActual = c;
    public override double Precio => ConfiguracionActual.Precio + 360D;
}
public class Memoria16GB3600 : ComponenteOrdenador
{
    public Memoria16GB3600(IOrdenador c) => ConfiguracionActual = c;
    public override double Precio => ConfiguracionActual.Precio + 110D;
}
```

De forma análoga añadiríamos todos los componentes que queramos a posteriori **sin modificar el código anterior** y así cumpliendo **OCP**.

Ahora, definimos una clase ordenador con una **configuración básica o vacía** que iremos '**decorando**'

```
public class OrdenadorVacio : IOrdenador
{
    public double Precio => 0D;
}
```

Por último, podremos definir la configuración del ordenador de la siguiente manera...

```
static void Main()
{
    IOrdenador setup = new OrdenadorVacio();
    setup = new SSDRapido(setup);
    setup = new Procesador8Nucleos(setup);
    setup = new Memoria16GB3600(setup);

    // Aquí podríamos añadir nuevos componentes futuros a la configuración

    Console.WriteLine($"Precio: {setup.Precio}");
}
```


L → Liskov Substitution Principle (LSP)

Nos sonará en nombre porque ya lo hemos usado. Principalmente lo que buscaremos es definir bien sus subclases para que también puedan ser tratadas como la propia abstracción.

Definición: Establece que las subclases deberían poder extender a su superclase sin cambiar el comportamiento de la misma.

En otras palabras, podríamos decir que después de la sustitución, no debería ser necesario ningún otro cambio para que el programa continúe funcionando como lo hacía originalmente.

Veamos un **ejemplo ilustrativo** de concreción que no nos permite aplicar **LSP** de forma correcta...

En la vida real tenemos '*claro*' que un cuadrado **es un** Rectángulo con los dos lados iguales. Por lo cual, podríamos suponer en un principio que si cuadrado hereda de Rectángulo sería un caso de herencia válido. Veamos que pasa si intentamos implementar este modelo...

```
class Rectangulo
{
    public virtual int Ancho { get; set; }
    public virtual int Alto { get; set; }
    public int Area => Ancho * Alto;
}
```

Invalidamos las propiedades para cuando cambiemos **Ancho** y **Alto** **siga siendo un cuadrado**.

```
class Cuadrado : Rectangulo
{
    public override int Ancho { set { base.Ancho = value; base.Alto = value; } }
    public override int Alto { set { base.Ancho = value; base.Alto = value; } }
}
```

Ahora si pasamos el siguiente test...

```
1  static void TestArea(Rectangulo r)
2  {
3      r.Ancho = 5; r.Alto = 4;
4      Debug.Assert(r.Area == 20, $"Área {r.Area} y no 20.");
5  }
6  static void Main()
7  {
8      TestArea(new Rectangulo());
9      // Creamos una instancia de Cuadrado y hacemos una sustitución
10     // de Liskov hacia la superclase la cual no pasará el test
11     TestArea(new Cuadrado());
12 }
```

Vemos como el **LSP no se cumple** en el segundo Test, ya que al asignar 4 a **Alto** el **Ancho** pasa a ser también 4 para cumplir la restricción de cuadrado y **el área me devolverá 16 en lugar de 20** como esperaba en el test.

Una posible solución es hacer nuestro objeto **inmutable**, este es, no dar la posibilidad de que el **Ancho** y el **Alto** cambien después de su creación.

I → Principio de Segregación de Interfaces (ISP)

Definición: Este principio viene a decir, que ninguna clase debería implementar métodos definidos en un interface que luego no va a usar o no tienen sentido.

Para cumplirlo, evitaremos interfaces grandes ('*fat interfaces*') que definan muchos métodos. Por tanto, **las reduciremos a aquellos que siempre se van a dar claramente en todas las clases que la implementen**.

También podremos usar el mecanismo de extensión o herencia de interfaces para realizar dicha segregación.

Veamos un **ejemplo** de incumplimiento de dicho principio. Para ello, supongamos la siguiente interfaz para **Ave**.

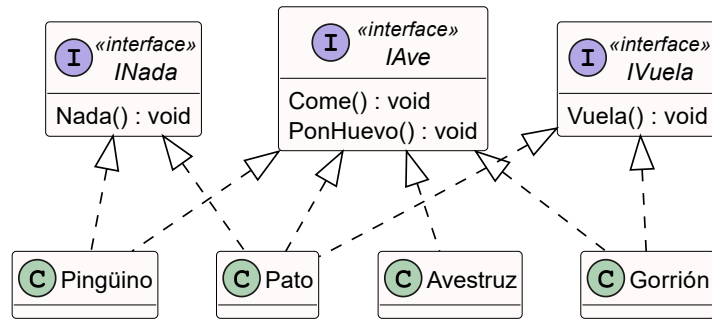
```
interface IAve
{
    void Vuela();
    void Nada();
    void Come();
    void PonHuevo();
}
```

Ahora definimos las siguientes clases que implementan la interfaz...

```
1 class Pato : IAve
2 {
3     public void Come()      => Console.Write("Soy un pato comiendo");
4     public void Nada()      => Console.Write("Soy un pato nadando.");
5     public void PonHuevo()  => Console.Write("Soy un pato poniendo un huevo");
6     public void Vuela()     => Console.Write("Soy un pato volando");
7 }
8 class Gorrión : IAve
9 {
10     public void Come()      => Console.Write("Soy un gorrión comiendo");
11     public void Nada()      => throw new NotImplementedException("Los gorriónes no nadamos :'(");
12     public void PonHuevo()  => Console.Write("Soy un gorrión poniendo un huevo");
13     public void Vuela()     => Console.Write("Soy un gorrión volando");
14 }
15 class Avestruz : IAve
16 {
17     public void Come()      => Console.Write("Soy un avestruz comiendo");
18     public void Nada()      => throw new NotImplementedException("Los avestruces no nadamos :'(");
19     public void PonHuevo()  => Console.Write("Soy un avestruz poniendo un huevo");
20     public void Vuela()     => throw new NotImplementedException("Los avestruces no volamos :'(");
21 }
22 class Pingüino : IAve
23 {
24     public void Come()      => Console.Write("Soy un pingüino comiendo");
25     public void Nada()      => Console.Write("Soy un pingüino nadando.");
26     public void PonHuevo()  => Console.Write("Soy un pingüino poniendo un huevo");
27     public void Vuela()     => throw new NotImplementedException("Los pingüinos no volamos :'(");
28 }
```

Si nos fijamos hay **operaciones que no somos capaces de implementar** en ciertas concreciones. Estamos obligados a definirlas por heredar del interfaz. Pero seguramente generaremos algún tipo de excepción al llamarlas y en ese caso no estaremos siguiendo el principio de **ISP**.

La mejor solución es **segregar los interfaces** y que cada **Ave** implemente aquellos que le corresponden.



D → Principio de Inversión de Dependencias (DIP)

Es uno de los más importantes y muy a tener en cuenta en nuestros diseños. El hacerlo, nos **facilitará la labor de creación de test unitarios**.

Definición: Este principio establece que:

1. Las clases de alto nivel **no deberían depender** de las clases de bajo nivel. Ambas deberían depender de las abstracciones.
2. Las abstracciones **no deberían depender** de los detalles. Los detalles deberían depender de las abstracciones.

En otras palabras, podemos decir que el objetivo de este principio es el uso de interfaces para conseguir que una clase interactúe con otras clases, sin que las conozca directamente. De esta manera conseguiremos desacoplar las clases lo máximo posible.

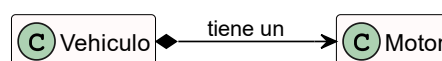
Existen diferentes patrones muy usados en frameworks actuales tales como la **inyección de dependencias** o **service locator** que nos permitirán invertir el control.

Supongamos la siguiente relación todo parte en la que un **Vehiculo** está **compuesto por** un **Motor**.

🚩 **Nota:** El siguiente ejemplo es un modelo super simplificado para que entendamos el concepto de **DIP**.

```
1 public class Motor
2 {
3     public int Revoluciones { get; private set; }
4     public void Acelera()
5     {
6         Console.WriteLine("Inyecta combustible");
7         Revoluciones += RELACION_INYECCION_MARCHA_CILINDRADA_ETC;
8     }
9 }
10
11 public class Vehiculo
12 {
13     private Motor Motor { get; }
14
15     public Vehiculo()
16     {
17         Motor = new Motor();
18     }
19     public void Acelera() => Motor.Acelera();
20     public int Revoluciones => Motor.Revoluciones;
21 }
```

Tenemos claramente una clase de alto nivel **Vehiculo** que **depende** de otra de Bajo nivel **Motor**. Esto significa que **Vehiculo** tiene que **'conocer'** una clase **Motor** 'específica' por la relación de composición (**línea 17**).



Pero... **¿Qué pasa si queremos concretar más el tipo de motor?** Por ejemplo queremos indicar que el vehiculo tiene un motor de gasolina (explosión).

Tendríamos que modificar la clase `Vehiculo` incumpliendo así principio **OCP** y refactorizar la clase `Vehiculo` de la siguiente forma ...

```
1 public class MotorGasolina
2 {
3     public int Revoluciones { get; private set; }
4     public void Acelera()
5     {
6         Console.WriteLine("Inyecta gasolina para explosión");
7         Revoluciones += RELACION_INYECCION_MARCHA_CILINDRADA_ETC_GASOLINA;
8     }
9 }
10
11 public class Vehiculo
12 {
13     private Motor MotorGasolina { get; }
14
15     public Vehiculo()
16     {
17         Motor = new MotorGasolina();
18     }
19     public void Acelera() => Motor.Acelera();
20     public int Revoluciones => Motor.Revoluciones;
21 }
```

Pero... ¿Qué pasa si queremos añadir nuevos tipos de motor?. Por ejemplo, diesel o eléctrico.

La respuesta es que no podremos, ya que creamos directamente el objeto de bajo nivel `MotorGasolina` dentro de la clase `Vehiculo` y eso nos genera una dependencia directa.

Para que cumpla **DIP** haremos que `Vehiculo` dependa de una abstracción de `Motor` la cual tendremos que pasar o **'inyectar'** ya sea como parámetro a través del constructor de vehículo o a través de un setter. De esta manera conseguiremos que nuestro vehículo tenga distintos tipos de motor sin modificar la clase `Vehiculo`.

👉 A este proceso se le denominará **'Inversión de Dependencias'**.

```
1 public abstract class Motor
2 {
3     public int Revoluciones { get; protected set; }
4     abstract public void Acelera();
5 }
6
7 public class Vehiculo
8 {
9     private Motor Motor { get; }
10    public Vehiculo(Motor m) => Motor = m;
11    public void Acelera() => Motor.Acelera();
12    public int Revoluciones => Motor.Revoluciones;
13 }
```

Ahora ya podremos implementar todas las concreciones de bajo nivel de `Motor` que se nos ocurran y dependerán de la abstracción `Motor` tal y como postula **DIP**.

```

public class MotorGasolina : Motor
{
    public override void Acelera()
    {
        Console.WriteLine("Inyecta gasolina para explosión");
        Revoluciones += RELACION_INYECCION_MARCHA_CILINDRADA_ETC_GASOLINA;
    }
}

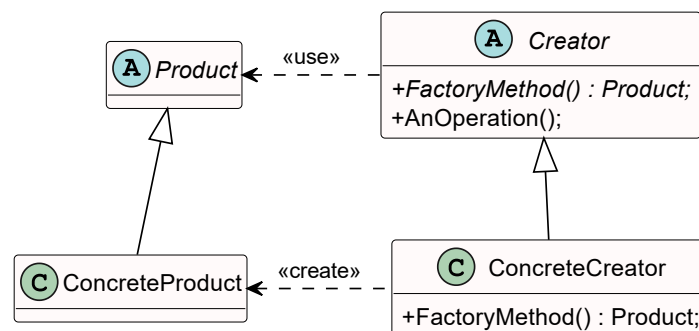
public class MotorDiesel : Motor
{
    public override void Acelera()
    {
        Console.WriteLine("Inyecta gasóleo para combustión");
        Revoluciones += RELACION_INYECCION_MARCHA_CILINDRADA_ETC_DIESEL;
    }
}

```

Patrón creacional Factory Method

Pero... ¿Cómo realizar esta inversión de dependencias teniendo en cuenta la relación de **composición** entre **Vehiculo** y **Motor** ?. Una posibilidad es usar el patrón **Method Factory**, el cual se encargará de relacionar las dos clases.

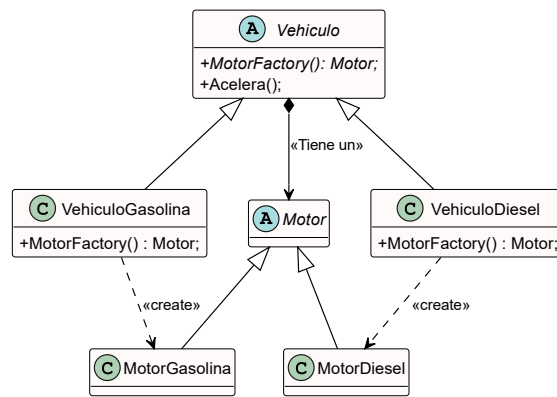
El esquema **tradicional** y básico del patrón será el siguiente:



Una posible descripción de las clases y objetos que participan en este patrón podría ser:

- **Product** Define la interfaz de objetos que crea el '*Factory Method*' en nuestros ejemplo queremos crear la abstracción de motor de vehículo **Motor** .
- **ConcreteProduct** Define las concreciones actuales y futuras de la abstracción **Product** en nuestro caso **MotorGasolina** y **MotorDiesel** .
- **Creator** Abstracción que declara el **método abstracto de de fábrica** (Method Factory), que devuelve un objeto de tipo **Product** . En nuestro caso esta clase de creación será **Vehiculo** pues definirá el método de fabrica que los crea con los diferentes motores.
- **ConcreteCreator** Concreción del creador que invalida el método abstracto de fábrica para devolver una instancia de **ConcreteProduct** . En nuestro caso esta clase podría representarse a través de los tipos **VehiculoGasolina** y **VehiculoDiesel** .

De tal manera que si tuviéramos que sustanciar una implementación del patrón '*Factory Method*' podría ser la siguiente.



✦ **Nota:** Aunque aquí estamos implementando la versión **clásica** del patrón con clases abstractas. Las abstracciones son más comunes hacerlas con **Interfaces** en la actualidad.

```


public abstract class Vehiculo
{
    public Motor Motor { get; private set; }

    protected Vehiculo()
    {
        Motor = MotorFactory();
    }
    public void Acelera() => Motor.Acelera();
    public int Revoluciones => Motor.Revoluciones;
    public abstract Motor MotorFactory();
}

public class VehiculoGasolina : Vehiculo
{
    public override Motor MotorFactory() => new MotorGasolina();
}

public class VehiculoDiesel : Vehiculo
{
    public override Motor MotorFactory() => new MotorDiesel();
}

public class Principal
{
    static void Main()
    {
        const int NUMERO_VEHICULOS_DE_MI_GARAJE = 10;
        Vehiculo[] miGaraje = new Vehiculo[NUMERO_VEHICULOS_DE_MI_GARAJE];
        for (int i = 0; i < miGaraje.Length; i++)
        {
            miGaraje[i] = (i % 2) switch
            {
                0 => new VehiculoDiesel(),
                _ => new VehiculoGasolina(),
            };
        }
        foreach (Vehiculo v in miGaraje) v.Acelera();
    }
}
  
```

 **Importante:** Si nos fijamos a efectos prácticos seguirá cumpliéndose la composición porque las referencias a motor desaparecerán junto con el vehículo.

Además:

1. Cumplimos **DIP** porque la clase de alto nivel **Vehículo** depende solo de la abstracción **Motor** y así como las especificaciones de bajo nivel de motor.
2. El *'Factory Method'* es la que tiene dependencias de uso (`<<use>>`) con las clases implicadas y se encargara de crear instancias de los diferentes tipos de vehículos asegurándonos que al desaparecer un vehículo desaparezca su motor.
3. La clase **Principal** usa las clases subclasses *'factoría'* para instanciar objetos vehículo de forma simple y además podríamos usar esta misma de **forma simple** en nuestros test para crear **mocks** de vehículo para hacer pruebas unitarias.
4. Cumplimos el principio **OCP** porque si añadiésemos un nuevo tipo de motor, las definiciones Vehiculo e Motor no tendríamos que modificarlas (Cerradas para Modificación) y solo tendríamos que definir nuevas concreciones (Abiertas para ampliación). Por ejemplo si añadimos un tipo de motor llamado **MotorElectrico** tendríamos solo que definir las nuevas concreciones de las abstracciones ...

```
public class MotorElectrico : Motor
{
    public override void Acelera()
    {
        Console.WriteLine("Aumenta intensidad corriente");
        Revoluciones += 6;
    }
}
public class VehiculoElectrico : Vehiculo
{
    public override Motor MotorFactory() => new MotorElectrico();
}
public class Principal
{
    static void Main()
    {
        const int NUMERO_VEHICULOS_DE_MI_GARAJE = 10;
        Vehiculo[] miGaraje = new Vehiculo[NUMERO_VEHICULOS_DE_MI_GARAJE];
        for (int i = 0; i < miGaraje.Length; i++)
        {
            miGaraje[i] = (i % 3) switch
            {
                0 => new VehiculoDiesel(),
                1 => new VehiculoElectrico(),
                _ => new VehiculoGasolina(),
            };
        }
        foreach (Vehiculo v in miGaraje) v.Acelera();
    }
}
```

Sin embargo, si añadiésemos nuevos componentes al vehículo como por ejemplo neumáticos de varios tipos. Las combinaciones de creación podrían crecer y posiblemente necesitemos definir muchos tipos nuevos a modo de *'factoría'*. Algo similar a la que nos sucedía con los componentes de ordenador y que solucionamos con un patrón Decorator.

Otro patrón creacional usado sobretodo por programadores de Java para facilitar sus test es el denominado **Builder**.

Patrón creacional Fluent Builder

Existe un [Classical Builder Pattern](#) pero es este caso vamos a implementar una versión más simplificada e incluso más usada, sobre todo entre programadores de Java, que nos hará tener una mejor idea de la utilidad del patrón.

Supongamos como comentábamos, que queremos añadir neumáticos a nuestros vehículos. Podemos definir la siguiente abstracción de los datos de un neumático con sus abstracciones...

```
public interface INeumaticos
{
    string IndiceVelocidad { get; }
    int IndiceCarga { get; }
    int Radio { get; }
    int Perfil { get; }
    int Ancho { get; }
}
public class NeumaticosNormal : INeumaticos
{
    public string IndiceVelocidad => "H";
    public int IndiceCarga => 88;
    public int Radio => 16;
    public int Perfil => 55;
    public int Ancho => 205;
}
public class NeumaticosSport : INeumaticos
{
    public string IndiceVelocidad => "Y";
    public int IndiceCarga => 92;
    public int Radio => 18;
    public int Perfil => 40;
    public int Ancho => 225;
}
```

Vamos a modificar la definición de vehículo.

```
public class Vehiculo
{
    public Motor Motor { get; private set; }
    public INeumaticos Neumaticos { get; private set; }
    public Vehiculo(Motor motor, INeumaticos neumaticos)
    {
        Motor = motor;
        Neumaticos = neumaticos;
    }
    public void Acelera() => Motor.Acelera();
    public int Revoluciones => Motor.Revoluciones;
}
```

Fíjate que **ya no es una clase abstracta** y que además el constructor tiene más sentido al recibir las abstracciones para cumplir **DIP**. Es una clase más simple y sin definir métodos raros fuera de su '*responsabilidad*'.

Definiremos ahora una clase **Builder** a modo de '*obrero de la construcción*' a la que le iremos pasando los componentes para ensamblar un determinado tipo de vehículo. Dicha clase le podemos llamar **VehiculoBuilder**

```

1 public class VehiculoBuilder
2 {
3     // Debe definir propiedades autoimplementadas con los campos o
4     // 'piezas' que necesita el 'obrero' para ensamblar un objeto de tipo vehículo.
5     private Motor Motor { get; set; }
6     private INeumaticos Neumaticos { get; set; }
7
8     // Definimos Setters para cada una de las piezas pero de con un
9     // interfaz fluido, para facilitar la sintaxis de construcción.
10    // Es la forma en que le pasamos al 'obrero' las piezas necesarias para ensamblar el vehículo.
11    public VehiculoBuilder SetMotor(Motor m)
12    {
13        Motor = m;
14        return this;
15    }
16    public VehiculoBuilder SetNeumaticos(INeumaticos n)
17    {
18        Neumaticos = n;
19        return this;
20    }
21    // Por último, definiremos un método llamado Build() que devolverá un objeto vehículo creado.
22    // Fíjate que si no está definido uno de los componentes o piezas del vehículo,
23    // el obrero creará una por defecto o que puede definirse en función de las otras.
24    // Por ejemplo, podríamos decidir que si el motor es gasolina y no me han
25    // definido los neumáticos, podría montar por defecto neumáticos sport para ese tipo de motor.
26    public Vehiculo Build() => new Vehiculo(
27        Motor ?? new MotorGasolina(), Neumaticos ?? new NeumaticosNormal()
28    );
29 }

```

Ahora crear, difenetes combinaciones de vhículos sería bastante simple e intuitiva.

```

1 static void Main()
2 {
3     // Definimos el 'obrero que va a construir los vehículos'.
4     VehiculoBuilder builder = new VehiculoBuilder(); // 🧑
5
6     // Creamos un vehículo con motor de gasolina y neumáticos sport.
7     Vehiculo gasolinaSport = builder.SetMotor(new MotorGasolina())
8         .SetNeumaticos(new NeumaticosSport())
9         .Build();
10    // Le cambiamos el componente de neumáticos a obrero para que ahora
11    // me haga un vehículo con motor de gasolina y neumáticos normales.
12    Vehiculo gasolinaNormal = builder.SetNeumaticos(new NeumaticosNormal())
13        .Build();
14 }

```

Si te fijas bien en el código, podremos definir todas las combinaciones de motor y neumático que queramos, simplemente definiendo nuevas concreciones y sin tener que definir una factoría específica de la combinación. Además cumpliremos **OCP** y **DIP** ya que al definir nuevos tipos de neumático o motor no tendremos que redefinir la clase `Vehículo`, ni ninguna de los 'ensamblajes' de vehículos ya creadas.

🚧 **Nota:** Esta sería una versión muy simplificada del patron '*Fluent Builder*' para que entendamos la idea. Exsiten otras propuestas de implementación donde el builder solo puede construir un objeto y además el set de los diferentes componentes del mismo está formazo ha hacerse y además en un determinado orden. De esa forma no podemos dejar ningún componente con un valor por defecto como en nuestra propuesta.