Inyección de Dependencias

Como colofón de la serie de cinco artículos dedicados a los principios SOLID, en esta ocasión toca hablar del Principio de Inyección de Dependencias (*Dependency Inyection*, DI), abordando la problemática de los modelos altamente acoplados y mostrando por qué este principio está tomando cada vez más relevancia en nuestros desarrollos.

Introducción

Si nos remontamos a los primeros años de la programación, nos encontraremos con programas rígidos repletos de código monolítico y lineal. La propia evolución hizo aparecer conceptos hoy por hoy imprescindibles como la modularidad y la reutilización de componentes, conceptos fundamentales en el paradigma de la Programación Orientada a Objetos.

La modularidad y reutilización de clases conlleva un flujo de comunicación entre instancias cuyo mal uso deriva en un hándicap que limita la flexibilidad, robustez y reusabilidad del código debido a la dependencia o alto acoplamiento entre las clases.

En la figura 1 podemos ver un sencillo diagrama de clases de un sistema de adquisición y control de datos meteorológicos. Existen dos clases participantes: una para la captura de la temperatura, y otra que representa a la estación meteorológica. Ambas tienen una responsabilidad a la hora de mostrar los datos, como puede apreciarse en el listado 1.

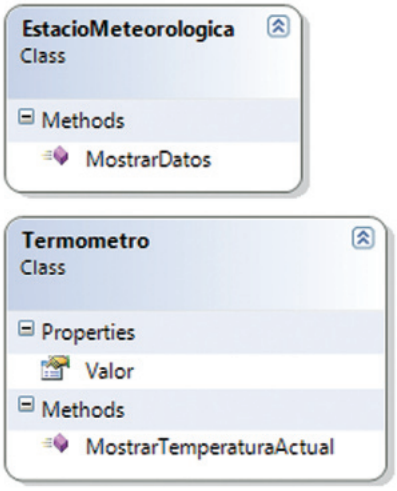
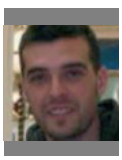


Figura 1

```
public class EstacioMeteorologica
{
  public void MostrarDatos()
  {
    Console.WriteLine(
       string.Format("Datos a {0} \n", DateTime.Now));
    Termometro termometro = new Termometro();
    termometro.MostrarTemperaturaActual();
  }
}

public class Termometro
{
  public int Valor { get; set; }
  public void MostrarTemperaturaActual ()
  {
    Console.WriteLine(
        string.Format("Temperatura: {0} º", Valor));
  }
}
```

Listado 1



José Miguel Torres
MVP de Device Application Development

Identificando el problema

Cuando hablamos en términos de calidad, solemos utilizar los adjetivos "bueno" o "malo" para definir la calidad de un diseño. Sin embargo, no siempre utilizamos los argumentos o criterios que sustentan la afirmación "éste es un mal diseño". Existe un conjunto de criterios más allá del siempre subjetivo TNTWIWHDI (*That's Not The Way I Would Have Done It*, "Yo no lo habría hecho así") acuñado por **Robert C. Martin**, y son los que miden el nivel de rigidez, la fragilidad y la inmovilidad del sistema.

En nuestro ejemplo de la estación meteorológica, podemos afirmar que el diseño es **rígido**, porque cualquier cambio será difícil de llevar a cabo, ya que no conocemos el impacto que la modificación de una clase de bajo nivel (clase **Termometro**) tendrá sobre la clase de alto nivel (clase **EstacioMeteorologica**).

Cuando los cambios tienen una repercusión en otras entidades, no necesariamente dependientes, se dice que un sistema o aplicación es **frágil**. Si nos fijamos en el listado 1, la clase **EstacioMeteorologica** depende tanto de **Termometro** como de **System.Console**. Un cambio del flujo de salida de datos del programa (por ejemplo, a una impresora en lugar de **System.Console**) repercutiría en las clases de bajo nivel.

El término **inmóvil** lo utilizamos para medir el nivel de dependencia entre una parte del diseño y otros datos no directos. El ejemplo es inmóvil porque la clase **EstacioMeteorologica** depende de las clases **Termometro** y **System.Console** para mostrar los datos. Dicho en otras palabras, no podríamos extraer la clase de mayor nivel y utilizarla con otras entidades. Lo mismo pasaría con la clase de bajo nivel por su dependencia de **System.Console**.

```
public class EstacioMeteorologica
{
  public void MostrarDatos()
  {
      Termometro termometro = new Termometro();
      string temperatura =
          termometro.MostrarTemperaturaActual();
      Console.WriteLine(
          string.Format("Datos a {0} \n{1}",
              DateTime.Now, temperatura));
  }
}
public class Termometro
{
  public int Valor { get; set; }
  public string MostrarTemperaturaActual ()
  {
      return string.Format("Temperatura:{0} º", Valor);
  }
}
```

Listado 2

> Entre los criterios que permiten determinar si un diseño es bueno o malo están los que miden su nivel de rigidez, fragilidad e inmovilidad

Planteemos un nuevo diseño a nuestro sistema. En primer lugar, eliminemos la dependencia que la clase **Termometro** tiene de **System.Console**, ya le que estamos otorgando la responsabilidad de salida por pantalla cuando realmente no le corresponde. El resultado sería el que se muestra en el listado 2.

Ahora la clase **Termometro** ha quedado libre de dependencias, y por tanto es reutilizable. Sin embargo, aún **EstacioMeteorologica** depende tanto de **System.Console** como de **Termometro**. Por otro lado, la clase **Termometro** no es más que una representación de un valor referencial meteorológico cualquiera; por tanto, podríamos abstraer la interfaz **IMeteoReferencia**, tal y como se muestra en el listado 3, y hacer que la clase **Termometro** la implemente. Esto es un ejemplo de aplicación del patrón Fachada (*Façade*), mediante el cual simplificamos la firma de varias clases a través de una única interfaz.

```
public interface IMeteoReferencia
{
    int Valor { get; set; }
    string Mostrar();
}

public class Termometro : IMeteoReferencia
{
    public int Valor { get; set; }
    public string Mostrar()
    {
        return string.Format("Temperatura:{0} º", Valor);
    }
}
```

Listado 3

Ahora que hemos abstraído la interfaz, ésta nos servirá como contrato para las clases que quieran utilizarla. Esto nos permitirá desacoplar la clase **EstacioMeteorologica** de **Termometro**, tal y como muestra el listado 4.

Sin embargo, aún no hemos solucionado el problema, pese a que estamos más cerca. Lo que pretendemos es eliminar completamente la instanciación de la clase **Termometro**, y la solución pasa por inyectar la dependencia directamente a través del constructor, como se muestra en el listado 5.

```
public class EstacioMeteorologica
{
    private IMeteoReferencia termometro;

    public EstacioMeteorologica()
    {
        termometro = new Termometro();
    }

    public void MostrarDatos()
    {
        Console.WriteLine(
           string.Format("Datos a {0}", DateTime.Now));
        Console.WriteLine(termometro.Mostrar());
    }
}
```

Listado 4

```
public class EstacioMeteorologica
{
    private IMeteoReferencia termometro;

    public EstacioMeteorologica(
        IMeteoReferencia termometro)
    {
        this.termometro = termometro;
    }

    public void MostrarDatos()
    {
        Console.WriteLine(
           string.Format("Datos a {0}", DateTime.Now));
        Console.WriteLine(termometro.Mostrar());
    }
}
```

Listado 5

El Principio de Inyección de Dependencias

Robert C. Martin afirma en el Principio de Inyección de Dependencias:

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

Imaginemos por un momento la solución inicial de la estación meteorológica (listado 1). La clase de alto nivel **EstacioMeteorologica** depende de la clase de bajo nivel **Termometro** (o **Barometro**, **Anemometro**, etc.). Toda la lógica de la solución se implementaría en la clase de alto nivel, y cualquier modificación en las clases de bajo nivel tendría repercusión no únicamente sobre la definición de la clase de alto nivel, sino sobre la propia lógica de la aplicación, llegando incluso a forzar cambios en la misma, cuando debería ser la clase de alto nivel la que debería forzar el cambio a las clases de bajo nivel sin comprometer la lógica de la aplicación; es decir, justamente lo contrario. Además, la clase de alto nivel sería difícilmente reusable debido a este acoplamiento. Sencillamente, y resumiendo, la clase **EstacioMeteorologica** no debe depender de la clase **Termometro**; en todo caso, al contrario.

Existen tres formas de implementación de la Inyección de Dependencias:

- por constructor
- por *setter*
- por interfaz.

El primer caso lo hemos visto en la sección anterior, donde hemos inyectado la dependencia a través del constructor de la clase; el listado 6 muestra una generalización. La inyección por *setter* se realiza a través de una propiedad de la clase (listado 7); y por último, la inyección por interfaz se realiza a través de un método, recibiendo como parámetro el objeto a inyectar (listado 8).

```
IMeteoReferencia referencia = ObtenerReferencia();
EstacioMeteorologica estacion =
       new EstacioMeteorologica(referencia);
```

Listado 6. Inyección por constructor

```
EstacioMeteorologica estacion = new EstacioMeteorologica();
estacioMeteorologica.Referencia = ObtenerReferencia();
```

Listado 7. Inyección por setter

```
EstacioMeteorologica estacion = new EstacioMeteorologica();
estacioMeteorologica.LecturaContador(ObtenerReferencia());
```

Listado 8. Inyección por interfaz

Inversión de control y contenedores

No podemos hablar de DI sin dejar de hablar de la Inversión de control (*Inversion of Control*, IoC). IoC también es conocido como Principio de Hollywood, nombre derivado de las típicas respuestas de los productores de cine a los actores noveles: "no nos llames; nosotros lo haremos".

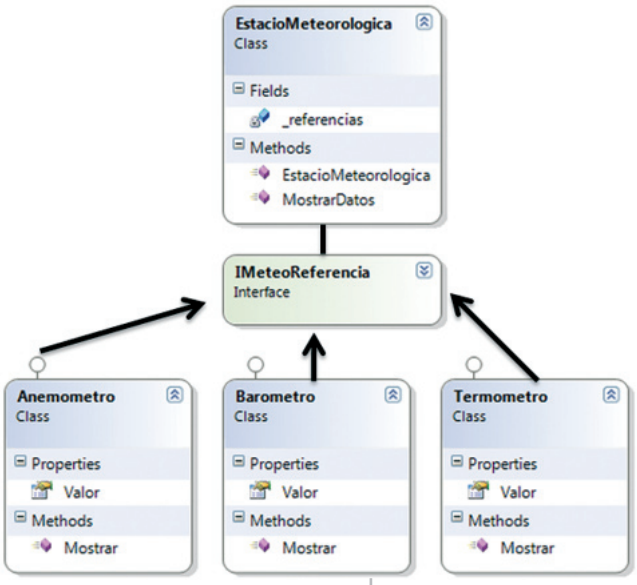


Figura 2

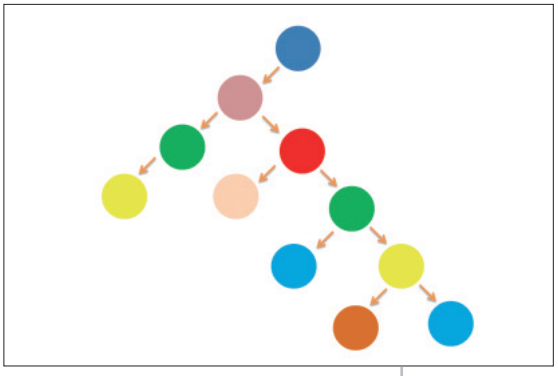


Figura 3.Diagrama de jerarquía de dependencias.

IoC invierte el flujo de control de un sistema en comparación con la programación estructurada y modular. En el fondo, DI es una implementación de IoC. Aún hoy existe la discusión acerca de si IoC es un principio, un patrón o ambas cosas a la vez. IoC, en definitiva, es una característica fundamental de un *framework*, y de hecho lo que lo hace realmente diferente a una librería de funciones.

En escenarios de producción, las clases no son tan triviales como la que hemos presentado en este artículo. Imagine por un momento que la interfaz `IMeteoReferencia` tiene una implementación de `IEntradaDatos` e `IVerificador`, y éstas a su vez implementan otras interfaces. En realidad, obtendremos una jerarquía de dependencias (figura 3), cuyo manejo en tiempo de diseño es imposible de gestionar "manualmente"; es aquí donde entra a jugar el término **contenedor IoC** (*IoC Container*).

El principal cometido de un contenedor IoC, a diferencia de una factoría, es el de gestionar el ciclo de vida de los objetos. El contenedor IoC registra una implementación específica para cada tipo de interfaz y retorna una instancia de objeto. Esta resolución de objetos tiene lugar en un único punto de las aplicaciones; normalmente, a nivel de infraestructura.

Conclusión

Con este artículo, hemos tratado de mostrar de una forma práctica la relación existente entre dependencias, detalles y abstracciones. Con el Principio de Inyección de Dependencias, ponemos fin a la última de las siglas que componen SOLID. Existen libros íntegros que hablan de este principio, y podrá encontrar en Internet una gran cantidad de recursos relacionados ([1]-[4]).

A lo largo de esta serie sobre los principios SOLID, hemos presentado aspectos muy importantes que debemos tener en cuenta ante cualquier nuevo desarrollo, y hemos visto cómo muchas de las problemáticas lógicas del diseño pueden ser reducidas mediante la aplicación de estos principios. Trate de entender cada uno de los principios desde un punto de vista práctico. Algunos de ellos (y lo digo por experiencia) son realmente complejos de llevar a la práctica; recuerde además que son principios, no reglas.

Para finalizar, agradecer a **Hadi Hariri**, quien me ha servido de "enciclopedia de consulta" para esta serie, por su apoyo y ayuda en todo momento.

Referencias

[1] http://msdn.microsoft.com/en-us/magazine/cc163739.aspx#S3.

[2] http://weblogs.asp.net/sfeldman/archive/2008/02/14/understanding-ioc-container.aspx.

[3] http://martinfowler.com/articles/injection.html.

[4] http://msdn.microsoft.com/en-us/library/aa973811.aspx.