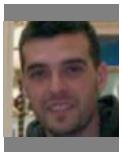
Principio de Sustitución de Liskov

Tercer principio de programación SOLID. En esta ocasión presentamos los fundamentos del Principio de Sustitución de Liskov y cómo la aplicación de este principio tiene una repercusión directa sobre las jerarquías de herencia entre clases.



José Miguel Torres

MVP de Device Application Development

En nuestra entrega anterior, hablábamos de lo útil que puede ser el Principio Open/Closed para desarrollar código fácil de mantener y reusar mediante el diseño de clases abiertas a la extensión y cerradas a la modificación, y para conseguirlo nos basamos en dos características clave de la Programación Orientada a Objetos (POO): la abstracción y el polimorfismo.

En lenguajes OO como C# o VB.NET, la clave para conseguir la abstracción y polimorfismo de entidades es mediante la herencia, y es precisamente en esta característica en la que se basa el Principio de Sustitución de Liskov (*Liskov Substitution Principle*, LSP). Cuáles son los fundamentos básicos de diseño que debe seguir la herencia en un caso particular, o cuál es la mejor forma de crear jerarquías de herencia entre clases, son algunas de las preguntas a las que responde dicho principio.

El Principio de Sustitución de Liskov

Echemos un vistazo al código del listado 1. Este código trata de calcular los impuestos de un vehículo en base a la matricula (antigüedad) y la cilindrada del mismo.

En 1987, **Barbara Liskov** presentó en una conferencia sobre jerarquía y abstracción de datos la siguiente definición:

> *Si por cada objeto **o1** del tipo **S** existe un objeto **o2** del tipo **T** tal que para todos los programas **P** definidos en términos de **T**, el comportamiento de **P** permanece invariable cuando **o1** es sustituido por **o2**, entonces **S** es un subtipo de **T**.*

Básicamente, LSP afirma que si tenemos dos objetos de tipos diferentes –`Coche` y `Ciclomotor`– que derivan de una misma clase base –`Vehiculo`–, deberíamos poder reemplazar cada uno de los tipos –`Coche`/`Ciclomotor` y viceversa– allí dónde el tipo base –`Vehiculo`– esté implementado. En el ejemplo anterior tenemos un claro caso de violación del LSP, ya que la ejecución del método `CalcularImpuesto` generará una excepción de conversión de tipo si el objeto pasado por parámetro es de tipo `Ciclomotor` en lugar de `Coche`, pese a que ambas clases derivan de la misma clase base `Vehiculo`.

Podríamos pensar en solucionar el problema de la forma que se expone en el listado 2. Pese a que el compilador no genere ninguna excepción de conversión de tipo, esta clase aún viola el LSP. Esto es debido a que estamos forzando a un objeto `Vehiculo` pasado como parámetro a comportarse como `Ciclomotor` o `Coche`. Además, esta aproximación vulnera el Principio Open/Closed que vimos en la entrega anterior, ya que ante cualquier nueva entidad que derive de `Vehiculo` deberemos modificar el método `CalcularImpuesto`.

```
class Vehiculo
{
  public string Marca { get; set; }
  public string Modelo { get; set; }
  public int Cilindrada { get; set; }
}

class Ciclomotor: Vehiculo
{
  public string ObtenerNumLicencia()
  {
      // Devuelve número de licencia
  }
}

class Coche: Vehiculo
{
  public string ObtenerMatricula()
  {
      // Devuelve matrícula
  }
}

class Impuestos
{
  public void CalcularImpuesto(Vehiculo vehiculo)
  {
    string matricula = ((Coche) vehiculo).ObtenerMatricula();
    ServicioCalculoImpuestos(matricula, vehiculo.Cilindrada);
  }
}
```

Listado 1

```
public void CalcularImpuesto(Vehiculo vehiculo)
{
  string matricula = string.Empty;

  if (vehiculo.GetType().Name == "Coche")
    matricula = ((Coche) vehiculo).ObtenerMatricula();
  else if (vehiculo.GetType().Name == "Ciclomotor")
    matricula = ((Ciclomotor)vehiculo).ObtenerNumLicencia();

  ServicioCalculoImpuestos(matricula, vehiculo.Cilindrada);
}
```

Listado 2

Un ejemplo clásico que encontraremos si buscamos referencias acerca del LSP en Internet o en nuestra biblioteca es el de herencia entre dos clases **Rectangulo** y **Cuadrado**; este ejemplo refleja lo que denominamos una incorrecta implementación de la herencia. Fijémonos en el código del listado 3 y el diagrama de clases de la figura 1.

> En el caso general, las clases incorporan métodos, que son los que aportan la clave diferencial en la POO: el comportamiento

Un objeto también es comportamiento

Equivocadamente, tendemos a pensar que una clase únicamente representa datos; eso es cierto solamente en el caso en los objetos de transferencia de datos (DTO) y poco más. En el caso general, las clases incorporan métodos, que son los que aportan la clave diferencial en la POO: el comportamiento.

En el caso anterior, probablemente no tendríamos problemas de herencia si obviáramos los métodos de las clases **Coche** y **Ciclomotor**, pero no es el caso. Los métodos **ObtenerMatricula** y **ObtenerNumLicencia** probablemente se conecten a un servicio o repositorio de datos externo, o calculen sus resultados según la fecha de matriculación y por tanto ese dato no se almacena en la clase. Dichos métodos otorgan un comportamiento a la clase, y es en la implementación de la herencia dónde puede verse modificado el comportamiento de una clase.

Figura 1

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

    public override int Alto
    {
        get
        {
            return base.Alto;
        }
        set
        {
            base.Ancho = value;
            base.Alto = value;
        }
    }
}
```

Listado 3

> Al sobrescribir métodos en las clases heredadas, debemos especificar una precondición menos restrictiva y una poscondición más restrictiva que las especificadas en la clase base

Si ejecutamos el código del listado 4, podremos apreciar cómo se vulnera el LSP. La idea subyacente es que en realidad la clase derivada **Cuadrado** no solo debe "ser un" sino que también debe "comportarse como un" **Rectangulo**, y efectivamente no lo hace, ya que la propiedad **Cuadrado.Alto** modifica tanto la altura como el ancho.

Este ejemplo demuestra además la estrecha relación que existe entre LSP y el **Diseño por Contratos** (*Design by Contract* – DbC) expuesto por Bertrand Meyer. Utilizando DbC, declaramos en los métodos unas precondiciones y poscondiciones; la precondición debe ser cierta antes de ejecutar el método y tras la ejecución, mientras que el propio método debe garantizar que la poscondición se cumpla.

```
[Test]
public void AreaRectangulo()
{
    Rectangulo r = new Cuadrado { Ancho = 5, Alto = 2 };
    // Fallará, pues Cuadrado establece
    // a 2 el ancho y el alto
    Assert.IsEqual(r.Ancho * r.Alto, 10); // false
}
```

Listado 4

Pues bien, existe una pequeña variación de DbC cuando se aplica en la herencia. Los métodos de la clase base implementan sus propias precondiciones y poscondiciones; sin embargo, cuando sobrescribimos dichos métodos en las clases heredadas **debemos especificar una precondición menos restrictiva** y **una poscondición más restrictiva** que las especificadas respectivamente en la clase base - de otra forma, se violaría el LSP. La razón es que el cliente de la llamada conoce la precondición y poscondición de la clase base, pero no la de la clase heredada, y por lo tanto no podemos suponer que el cliente conozca la precondición del método de la clase heredada, y por tanto debemos ser menos restrictivos en la precondición. Debido a la baja restricción de la precondición y para asegurar el comportamiento correcto tras la ejecución del método, la poscondición debe ser más restrictiva.

El ejemplo del **Rectangulo** y el **Cuadrado**, en la propiedad **Rectangulo.Alto** podríamos establecer la poscondición como:

```
POSTCONDICIÓN ->  (Alto == value && Ancho == Ancho)
```

Es decir, el valor de **Alto** debe contener el nuevo valor y el valor de **Ancho** debe permanecer inalterado. Por lo tanto, la poscondición de **Cuadrado.Alto** será menos restrictiva al no cumplir el segundo predicado **Ancho==Ancho**, y en consecuencia **Cuadrado.Alto** viola el contrato de la clase base y por consiguiente el LSP.

Conclusión

No olvidemos que una clase son datos más comportamiento, y que dicho comportamiento no debe ser sacrificado entre herencias. Por tanto, minimizaremos el impacto de una incorrecta implementación y por tanto de la modificación del comportamiento aplicando el Principio de Sustitución de Liskov.