Curso de Programación .NET nivel I

Unidad 4

“Relaciones entre Clases”

Contacto: consultas@elearning-total.com
Web: www.elearning-total.com

Índice

Contenido

Introducción

En esta unidad vamos a introducir conceptos más avanzados en lo que respecta a la programación orientada a Objetos.

Vamos a ver como las clases pueden interactuar entre sí lo que nos permite modelar escenarios más interesantes en los cuales un objeto puede ser parte de otro, usarse mutuamente o heredar características y acciones y así resolver problemas complejos de manera simple.

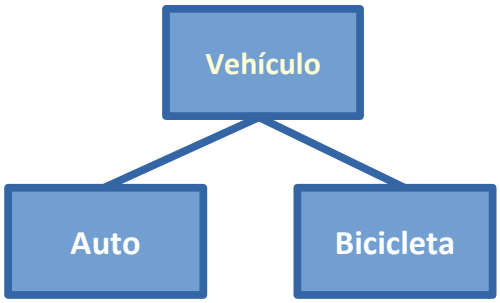
Herencia

Definición

La herencia es una forma de definir una nueva clase a partir de una ya existente. Me permite agrupar clases que estén relacionadas para controlarlas en conjunto y así fomentar la reutilización de código.

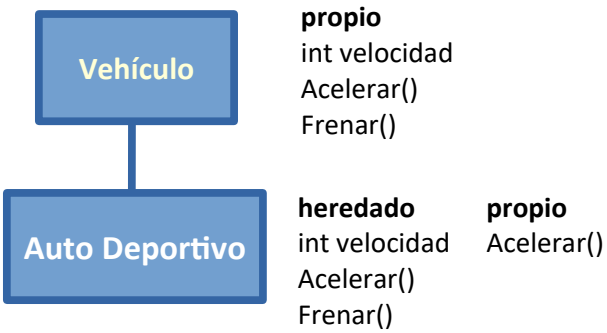
La herencia es el mecanismo a través del cual las propiedades y métodos comunes se definen en una clase (padre) y las clases subsiguientes (hijos) los heredan.

La herencia se puede representar como un árbol, en el cual en sus hojas encontramos las clases más especializadas y a medida que nos acercamos a la raíz las clases son más generales; contiene propiedades o métodos adecuados para muchas clases pero no suelen estar completas.



Si nuestro modelo esta bien definido es natural encontrar grupos de objetos relacionados entre sí, aunque puede ocurrir también que haya objetos sin herencia en casos muy específicos.

Vamos a ver en código como representar lo siguiente:



Lo que quiero es generalizar el caso del auto deportivo entonces me imagino algo menos especifico como la Clase Vehículo, que cual puede tener características que podría compartir con otras clases en el futuro: velocidad o Acelerar y Frenar.

```
class Vehiculo
    {
   protected int velocidad;

   public virtual void Acelerar(int nuevaVelocidad)
         {

      this.velocidad = nuevaVelocidad;
      Console.WriteLine("El Auto Standard acelero a " + this.velocidad);
         }

   public void Frenar()
         {
              velocidad = 0;
      Console.WriteLine("El Auto freno a "+ this.velocidad);
         }

    }
```

Hay algo nuevo en esta declaración, las palabras claves `protected` y `virtual`, ambas son palabras claves relacionadas con la herencia en cuanto al acceso y la sobre-escritura de métodos, conceptos que vamos a ver a continuación, pero antes definamos nuestra clase especifica:

```
class AutoDeportivo : Vehiculo //clase que hereda del Padre vehiculo.
 {

//sobrescribimos el método padre
    public override void Acelerar(int velocidadNueva)
    {
        velocidad = velocidadNueva * 2;
    }

 }
```

Palabras claves nuevas: `:Vehiculo y override`

Vayamos por partes.

Declaración de la herencia

Los dos puntos luego de la declaración de mi clase significa que voy a recibir una herencia de una clase, en este caso Autodeportivo recibe de herencia la propiedad velocidad y los métodos Fenar() y Acelerar()

```
class AutoDeportivo : Vehiculo
```

esto quiere decir que si yo dejo la clase AutoDeportivo vaciá igualmente puedo acelerar y frenar:

```
AutoDeportivo autoDeportivo = new AutoDeportivo();
autoDeportivo.Acelerar(100);
```

todo esto gracias a la herencia de mi clase padre `Vehiculo,`

Sobrescritura de métodos

La sobreescritura de métodos me sirve para reemplazar métodos que heredo por otros que tengan más sentido en mi objeto. Para hacer que una función sea reemplazable tengo que utilizar la palabra clave `virtual` y para sobrescribir un método `virtual` tengo que escribir la palabra clave `override` desde su hijo.

En mi caso quiero que mi auto deportivo acelere de una manera diferente a la que aceleran los demás vehículos, por lo tanto:

```
public override void Acelerar(int velocidadNueva)
```

Acceder a las propiedades del padre

Recordemos el encapsulamiento y esas normas que decían que tenemos que declarar todas las variables `private` para que otras clases no puedan acceder a sus variables.

En herencia deseamos acceder a las propiedades heredadas pero si las hago `private` los hijos no van a poder usarlas y si las hago `public` estaría rompiendo el encapsulamiento; `protected` es lo que tengo que utilizar en estos casos que quiero que el hijo pueda operar con las propiedades del padre.

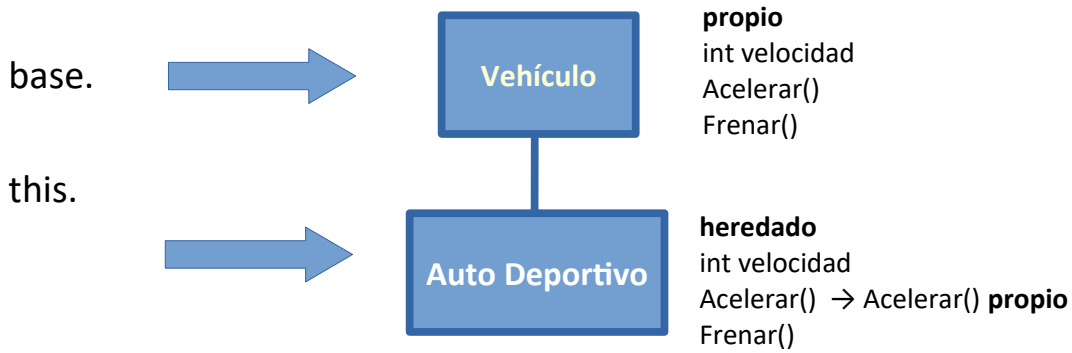
El siguiente código me va a dar error:

```
class Vehiculo
  {
     private int velocidad;
  }

  class AutoDeportivo : Vehiculo
  {

     public override void Acelerar(int velocidadNueva)
     {
           velocidad = velocidadNueva * 2;
     }

  }
```

Reemplazando `private` por `protected` es la manera correcta de declarar esta herencia.

Acceder a los métodos del padre

Si por algún motivo quiero acceder a un método que ha sido reemplazado, puedo recurrir a la palabra clave `base` la cual siempre va a apuntar a el padre. En mi caso **base**.Acelerar() llama al acelerar reemplazado.



Verificación de la herencia

Para verificar la validez de una herencia se puede utilizar la regla “es un o es una”,

El auto deportivo **es un** vehiculo

Asociación

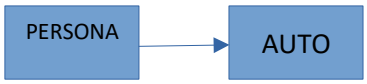
Introducción

Los objetos interactúan entre sí y estas interacciones son las que le dan vida a este modelo de objetos. De nada serviría todas las acciones o propiedades de los objetos, si no las podemos combinar para crear modelos dinámicos que resuelven situaciones complejas de la vida real.

Clasificación

Podemos dividir a estas interacciones de asociación en dos tipos:

- **Agregación ("usa un/a"):**
  Dos objetos están agregados cuando uno usa los servicios o mejor dicho los métodos del otro. En una agregación dos objetos independientes deciden colaborar para alcanzar algún objetivo. Luego de que el objetivo es logrado, ambos objetos continúan su existencia independientemente del otro.
  Por ejemplo: Una persona usa un auto, si la persona deja de existir el auto sigue teniendo sentido por sí mismo.
  La agregación es una relación menos estrecha que la relación de composición.



- **Composición ("tiene un/a")**
  Un objeto compone otro cuando su existencia depende de el otro, por eso se dice que un objeto tiene al otro.
  Por ejemplo: Un auto tiene un motor. Si el auto deja de existir, el motor de automóvil no tiene sentido por si solo.

Implementación de Composición.

Vamos a agregarle un motor a nuestro Vehículo, ya que quiero agregarle lógica a la aceleración y creo que debería modelar una nueva Clase que se encargue de todo lo referido a la aceleración.

```
class Motor
   {
  //Los motores entregan una velocidad maxima
  private int velocidadMaxima;

  public Motor(int velocidadMaxima)
        {
    this.velocidadMaxima = velocidadMaxima;
        }

  /*El encargado de saber hasta cuanto acelerar es el motor
        *Si se pasa de la velocidad, entregar solo la velocidad maxima. */
  public int Acelerar(int velocidad)
        {
    int velocidadCalculada;
    if (velocidad > velocidadMaxima)
            {
                velocidadCalculada = velocidadMaxima;
            }
    else
            {
                velocidadCalculada = velocidad;
            }
    return velocidadCalculada;
        }
  }
```

Esta lógica solo va a tener sentido con un automóvil entonces estoy ante una asociación de tipo Composición:

```
class AutoDeportivo : Vehiculo
          {
      //El Auto tiene un motor
      private Motor motor = new Motor(220);
      private bool modoTurbo;

      public override void Acelerar(int velocidadNueva)
            {
          //El Calculo de la velocidad depende del Motor
                    velocidad = motor.Acelerar(velocidadNueva);
            }
}
```

Fíjense como declaro el Motor dentro de las propiedades de mi objeto y a su vez lo estoy instanciando.
Una vez que el objeto se deja de usar, la referencia se pierde y ese motor no existe más (es lo que deseo en este caso)
Aprovechemos este caso también para ver como objetos me está ayudando a diseñar mi solución con objetos que se utilizan entre sí y esconden la complejidad. Motor es una abstracción que me sirve para ordenar mi código y que sea más entendible.

Se que con estos ejemplos simples es difícil de ver, pero imaginen un programa utilizado por toda una empresa que realiza pruebas automáticas de vehículos y tienen que procesar miles de reglas.
En ese caso desearan tener todo el código lo más ordenado posible valiéndose de las bondades que nos brinda este paradigma.

Implementación de Agregación

Vamos a designar un usuario a nuestro vehículo, este sujeto en nuestro modelo puede tener o no automóvil. Por lo tanto su existencia no depende unicamente de la tenencia o no de un auto, y de hecho si se trata de un sistema de gestión de venta deseo que mi cliente siga existiendo en mi sistema por más que no tenga más el auto que le vendí. Estamos ante una relación de agregación:

```
class Persona
    {
   //Persona USA UN auto, si la persona deja de existir el auto sigue existiendo.
   private AutoDeportivo auto;
   private string nombre;

   public Persona(string nombre)
         {
     this.nombre = nombre;
         }

   //Por ello el objeto no se crea, si no que se asigna
   public void AdjudicarVehiculo(AutoDeportivo auto)
         {
     this.auto = auto;
         }

   public void Manejar()
         {
     this.auto.Acelerar(60);
         }

    }
```

Observemos algo importante, una diferencia clave en el diseño de un tipo de asociación y la otra, en la agregación el objeto se asigna, no se crea, por lo tanto necesito un “setter” (AdjudicarVehiculo) para dicha acción. Si por motivo que fuera yo elimino el Auto, la persona puede seguir existiendo en mi sistema.

Esto es porque fue creada fuera del objeto:

```
Persona persona = new Persona("Leonel");
AutoDeportivo auto = new AutoDeportivo();
        persona.AdjudicarVehiculo(auto);
```

Clases Estáticas

Introducción

Las clases estáticas son clases como las que ya hemos visto pero con la diferencia que no pueden ser instanciadas o mejor dicho no se pueden crear objetos a partir de ellas. Sus métodos son denominados métodos de clase y pueden ser llamados directamente.

Declaración de la clase estática

Recordemos Console:

No soy un ingeniero de Microsoft® pero creo que esto está definido de esta manera:

```
Console.WriteLine("Esto es un método de una clase estatica")
static class Console
    {
  public static void WriteLine(string argumento)
        {
    // aca va el código de los ingenieros de microsoft®
        }
    }
```

La palabra clave `static` transforma a la clase en estática y dentro de ella hay que declarar los métodos también como `static`. Aunque puedo tener métodos estáticos en clases no estáticas, como ocurre con la clase Program y el método estático Main.

Utilización de la clase estática

En la práctica las clases estáticas se utilizan para los llamados **helpers** o bien conjunto de funciones que no pertenecen a ninguna clase en particular pero que me ayudan con los requerimientos no funcionales de mi solución.

Por ejemplo: Dada una fecha en un formato transformarlo a otro.

Las buenas prácticas recomiendan no abusar al uso de clases estáticas y si se requiere más complejidad en estas funciones debo pensar en modelar clases convencionales con herencia y asociación.

Clases Abstractas

Introducción

Las clases abstractas son clases que contienen al menos un método abstracto y un método abstracto es aquel método que contiene una declaración que no implementa código.

Las clases abstractas por lo tanto no son instanciables.

El código que no se implementa en la clase abstracta entonces debe ser implementado por las clases heredadas de esta clase abstracta.

Imaginemos en nuestro caso que al ser Acelerar un método tan genérico que no puedo proveer un mismo código para todos los casos, entonces tiene sentido declarar el método Acelerar como abstracto.

Implementación

```
abstract class Vehiculo
{
    protected int velocidad;

    public abstract void LlenarTanque(int cantidad);

    public void Acelerar(int nuevaVelocidad)
    {
        this.velocidad = nuevaVelocidad;
    }
}

class Gasolero : Vehiculo
{
    private int tanqueDeGas;

    public override void LlenarTanque(int cantidad)
    {
        tanqueDeGas = cantidad;
    }
}
```

Utilización de la clase Abstracta

En la practica la clase abstracta se utiliza en casos donde encuentro cierta similitud entre objetos a nivel de acciones que realizan pero no así con como las realizan.

Interfaces

Introducción

Las interfaces son estructuras que contienen declaraciones de métodos o propiedades pero que no las implementan, es decir que no contienen código. Las interfaces se aplican sobre clases para "obligarlas" a implementar los métodos y las propiedades que las interfaces contienen.

En otras palabras, la interfaz es un contrato para una clase.

Implementación

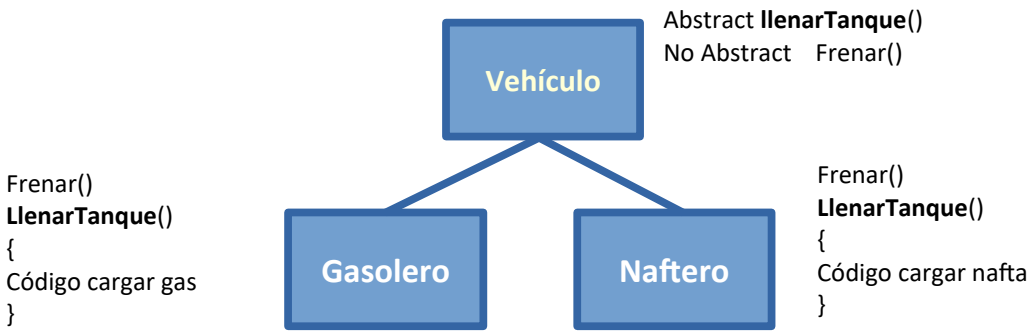
```
interface IAcelerador
 {
     void Acelerar(int nuevaVelocidad);
 }

 class Bicicleta : IAcelerador
 {
     int rotacionDelPedal = 0;

     // La norma de la interfaz IAcelerador me obliga a implementar este metodo.
     public void Acelerar(int nuevaVelocidad)
     {
         rotacionDelPedal = nuevaVelocidad;
     }
 }
```

Diferencia con la clase Abstracta y utilización

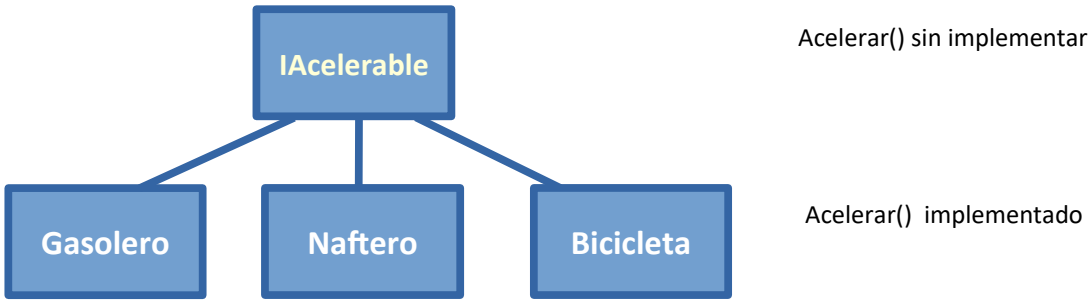
Uno de las principales discusiones es sobre la diferencia entre una interfaz y una clase abstracta:

- La interfaz no puede contener código, la clase abstracta si.
- No existe múltiple herencia en .NET, es decir que una clase no puede heredar de varias clases, en cambio si puede implementar varias interfaces.

En cuanto a sus usos en la práctica, La clase **abstracta** se utiliza cuando puedo identificar comportamiento en común entre varios objetos (métodos no abstractos) salvo algunos comportamientos que van a ser particulares (métodos abstractos). Estos métodos abstractos serán responsabilidad de las clases hijas en detallar su código. Entonces vemos que en las clases abstractas la responsabilidad es compartida.
Por ejemplo: LlenarTanque() puede ser un método abstracto ya que cada tipo particular de vehículo lo va a implementar a su manera (gas vs nafta)



En cambio las **interfaces** son contratos que delegan completamente a la clase hija o implementadora la responsabilidad de detallar el código. Se usan cuando detecto que varios objetos distintos tienen patrones comunes pero su implementación es completamente distinta.
Por ejemplo: **Acelerar()** en el caso de una bici vs un automóvil o **Caminar**() en el caso de una araña vs una persona.



Fíjense que LlenarTanque() esta mucho más estrecho a sus hijos que lo esta Acelerar() que es mucho más amplio ya que me sirve tanto para Automóviles como para Bicicletas. Por ello el primero es un método abstracto y el segundo una interfaz. Esto es valido para este caso pero puede no serlo para otros, siempre depende la complejidad de nuestro problema y su alcance.

En términos prácticos y con mucha simplificación de por medio, una interfaz es una clase 100% abstracta, Con las limitantes descriptas más arriba.