Tema 9.4

Descargar estos apuntes en pdf o html

Índice

- Profundizando en la Programación Orientada a Objetos
  - Interfaces
    - Interfaces en los diagramas de clases UML
    - Interfaces en CSharp
    - Interfaces de utilidad predefinidos en las BCL
      - IEnumerable
      - ICloneable
      - IComparable
      - IDisposable
      - Repasando la instrucción using
    - Caso de aplicación de interfaces

Profundizando en la Programación Orientada a Objetos

Interfaces

Básicamente un Interfaz es la definición de un conjunto de interfaces de métodos, accesores o mutadores (como **Propiedades**), indizadores, etc. Es muy parecido a definir una **clase abstracta pura**, pero **sin ningún tipo de atributo** o campo, constructor, ni modificador de acceso (public, private, etc…). Como en las clases abstractas, las interfaces son tipos referencia, no puede crearse objetos de ellas sino sólo de tipos que deriven de ellas, y participan del polimorfismo.

Pueden implementarse en muchos lenguajes OO con idénticas características:

- Es posible la **herencia múltiple** de interfaces.
- No pueden definir atributos pero sí propiedades.
- Un interfaz **puede heredar de otro interfaz**.
- Si una clase hereda de un interfaz. Esta, deberá invalidar todo lo que hayamos definido en el mismo.

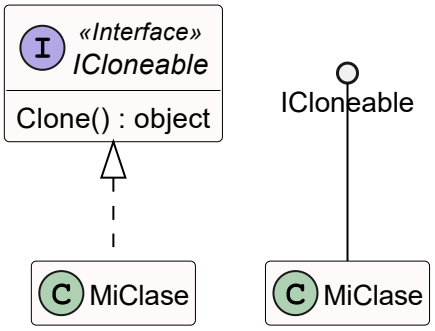
Podemos resumir diciendo que es la forma más recomendable y común de **definir la abstracción de un comportamiento**.

Interfaces en los diagramas de clases UML

Para expresar que la clase `MiClase` **implementa** el interfaz `ICloneable`.

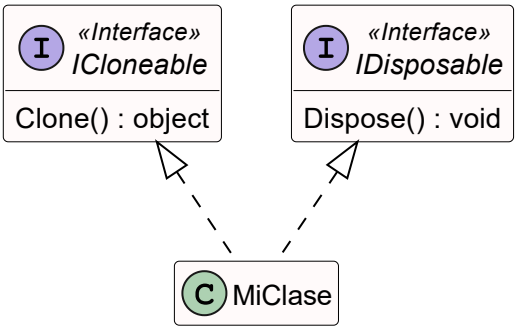
> 📌 **Nota:** Usamos la palabra **implementa** en lugar de "*hereda de*" ya que, como hemos comentado, más que responder `MiClase` a la pregunta "*es un*", un interfaz define un comportamiento abstracto que `MiClase` deberá implementar.

Podremos expresarlo de las forma siguientes formas ...



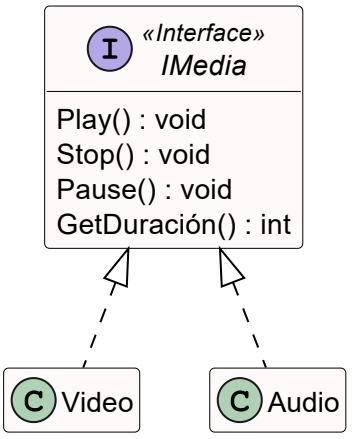
`MiClase` ahora está obligada a **implementar** el método público `Clone` con **idéntica signatura**.

Podremos hacer que una clase **implemente** o "*herede*" de más de un interfaz.



Interfaces en CSharp

Como ya habrás podido apreciar en el diagrama anterior, según el convenio de nomenclatura de C#, el identificador o nombre de la clase irá siempre precedido por la letra mayúscula `I` (`I nterface`) para distinguirlo de otro tipo de clases.




```
<modificadores> interface I<identificador> : <interfacesBase>
{
    <interfaces de métodos, propiedades o indizadores>
}
```

```
interface IMedia
{
    void Play();
    void Stop();
    void Pause();
    int Duración { get; }
}
```

Para aplicar un interfaz a una clase. Haremos que esta herede del interfaz con la sintaxis de herencia que hemos usado hasta ahora.

```
class Video : IMedia
{
    // ...

    public int Duración => 0;
    public void Pause() => Console.WriteLine("Pausando el vídeo.");
    public void Play() => Console.WriteLine("Reproduciendo el vídeo.");
    public void Stop() => Console.WriteLine("Parando el vídeo.");
}
```

Interfaces de utilidad predefinidos en las BCL

Podemos decir que me permiten definir comportamientos para mis propios tipos, que serán reconocidos por otras clases o tipos ya implementadas en las BCL.

> 📌 **Nota:** Podríamos utilizar interfaces própios para hacer lo mismo, **pero perderíamos interoperabilidad con el resto de clases de las BCL**.

IEnumerable

Lo veremos más adelante, al usar o definir colecciones.

ICloneable

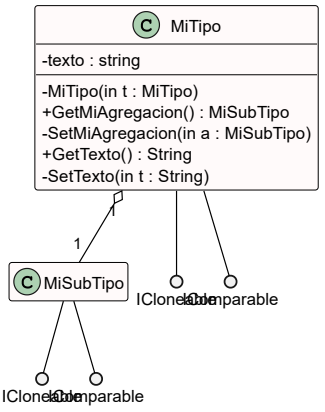
ICloneable Me indicará que puedo crear copias del objeto, puesto que me obliga a implementar un "*constructor copia*" con el interfaz `object Clone()` el cual me permitirá hacer **copias en profundidad** de objetos de tipo referencia.

IComparable

IComparable Me indicará que el objeto debe implementar el método `int CompareTo(Object otro)` que me servirá para comparar dos objetos de la misma clase y que ya usamos en el **tema 7** para comparar cadenas.

```
// Nota: Recordemos brevemente que...
Tipo o1 = new Tipo(...);
Tipo o2 = new Tipo(...);
// Si Tipo es IComparable entonces ...
int comparacion = o1.CompareTo(o2);
// comparacion = 0 si o1 y o2 son iguales.
// comparacion > 0 si o1 > o2.
// comparacion < 0 si o1 < o2.
```

Veamos un ejemplo "*genérico*" comentado el uso de este tipo de interfaces. Para ello, supongamos la siguiente agregación con tipos definidos por el usuario, donde ambos implementan los interfaces `ICloneable` e `IComparable`.



Vamos a suponer que la clase `MiSubTipo` ya la hemos definido e implementa ambos interfaces. Una implementación '*genérica*' posible para `MiTipo` podría ser ...

```
class MiTipo : ICloneable, IComparable
{
    // Propiedad autoimplementada que representa la agregación del UML.
    public MiSubTipo MiAgregacion { get; private set; }
    // Propiedad autoimplementada que representa el atributo texto con su
    // accesor y mutador
    public string Texto { get; private set; }
    // Constructor copia privado que usaremos para hacer el clonado
    // en profundidad.
    private MiTipo(MiTipo t)
    {
        // Debo hacer un downcasting porque Clone() me devuelve un object.
        MiAgregacion = t.MiAgregacion.Clone() as MiSubTipo;
        // No hay problema en igualar los string porque es inmutable.
        Texto = t.Texto;
    }
    // Estoy forzado a implementarlo por ser ICloneable
    // Llamamos al constructor copia mediante un cuerpo de expresión.
    public object Clone() => new MiTipo(this);

    // Estoy forzado a implementarlo por ser IComparable
    public int CompareTo(object obj)
    {
        // Si obj de es de MiTipo genero una excepción.
        MiTipo t = obj as MiTipo
        ?? throw new ArgumentException("No es del tipo MiClase", "obj");
        // Para realizar la comparación voy llamando a los CompareTo de cada
        // tipo en el orden adecuado.
        int comparacion = MiAgregacion.CompareTo(t.MiAgregacion);
        if (comparacion == 0)
            comparacion = Texto.CompareTo(t.Texto);
        return comparacion;
    }
}
```

IDisposable

IDisposable Me indicará que el objeto debe implementar el método `void Dispose()` que **se encargará de liberar los recursos usados por el objeto.** No confundir con el destructor `~<Tipo>()`.
Indicaremos a las BCL que nuestro objeto tiene el comportamiento de liberar recursos y lo utilizaremos junto a la **instrucción** `using`, la cual ya hemos usado para cerrar automáticamente los flujos al producirse una excepción cuando vimos la entrada y salida de datos. Esto es, porque en el fondo lo único que espera el `using` es que las clases marcadas con está instrucción o cláusula implementen este interfaz. En el fondo la clase base **Stream** lo implementa y por tanto tiene un método `Dispose()` que es llamado en el `finally`.

Repasando la instrucción using

- Se utiliza para instanciar objetos que contienen recursos no gestionados, esto es, que no son liberados por el recolector de basura.
- Como acabamos de comentar, estos objetos deben implementar el interfaz `IDisposable` y por tanto el método de liberación `Dispose()`.
- `using` garantiza que se llama a `Dispose()` **aunque se produzca una excepción**.
  > ✋ **Importante:** Dentro del bloque using, el objeto es de solo lectura y no se puede modificar ni reasignar puesto que dejaría de tener una referencia y no se liberaría.

Sintaxis básica:

```
// Podemos usar varios recursos liberables en el mismo ámbito así ...
using (var r1 = new TipoIDisposable())
using (var r2 = new TipoIDisposable())
...
using (var rN = new TipoIDisposable())
{
    // Ámbito de uso de solo lectura de r1, r2, … , rN
}

// También podremos anidarlos.
```

En **C#8.0** evolucionó el uso de esta instrucción. De tal manera que como comentamos con los flujos, **podemos aprovechar un bloque ya definido para decidir cuando va a estar disponible un recurso**.

```
// Esté código desde C#8
if (…)
{
    using var r = new TipoIDisposable();
    // Bloque...
}
```

Interpretación real de la instrucción `using`:

```
// Cuando instanciemos un objeto disposable de la siguiente manera en un método...
void Metodo()
{
    using var r = new TipoIDisposable(); // Cuerpo del método ...
}

// Realmente será un 'syntactic sugar' del siguiente código.
void Metodo()
{
    TipoIDisposable r;
    try
    {
        r = new TipoIDisposable(); // Cuerpo del método ...
    }
    finally
    {
        if (r != null) ((IDisposable)r).Dispose();
    }
}
```

Recordemos su uso a través de un **ejemplo**...

En el ejemplo siguiente creamos una clase de utilidad para generar **logs** de nuestros programas a un determinado fichero y visualizarlos.

```
static class Log
{
    public static void Escribe(string fichero, string texto)
    {
        // Se llama a dispose al salir del método.
        using TextWriter w = File.AppendText($"{fichero}.log");
        w.WriteLine(DateTime.Now.ToString("dd/MM/yyy HH:mm:ss ") + texto);
    }

    public static void Muestra(string fichero)
    {
        // Se llama a dispose al salir del método.
        using TextReader r = File.OpenText($"{fichero}.log");
        string s;
        while ((s = r.ReadLine()) != null) Console.WriteLine(s);
    }
}

static class Ejemplo
{
    static void Main()
    {
        Log.Escribe("DEBUG", "Empieza Main");
        Log.Escribe("DEBUG", "Finaliza Main");
        Log.Muestra("DEBUG");
    }
}
```

Dado que las clases `TextWriter` y `TextReader` implementan la interfaz `IDisposable`, podremos usar la instrucción `using` que garantizará que el archivo subyacente se cierre correctamente después de las operaciones de lectura o escritura.

Caso de aplicación de interfaces

🎓 **Caso de estudio:**

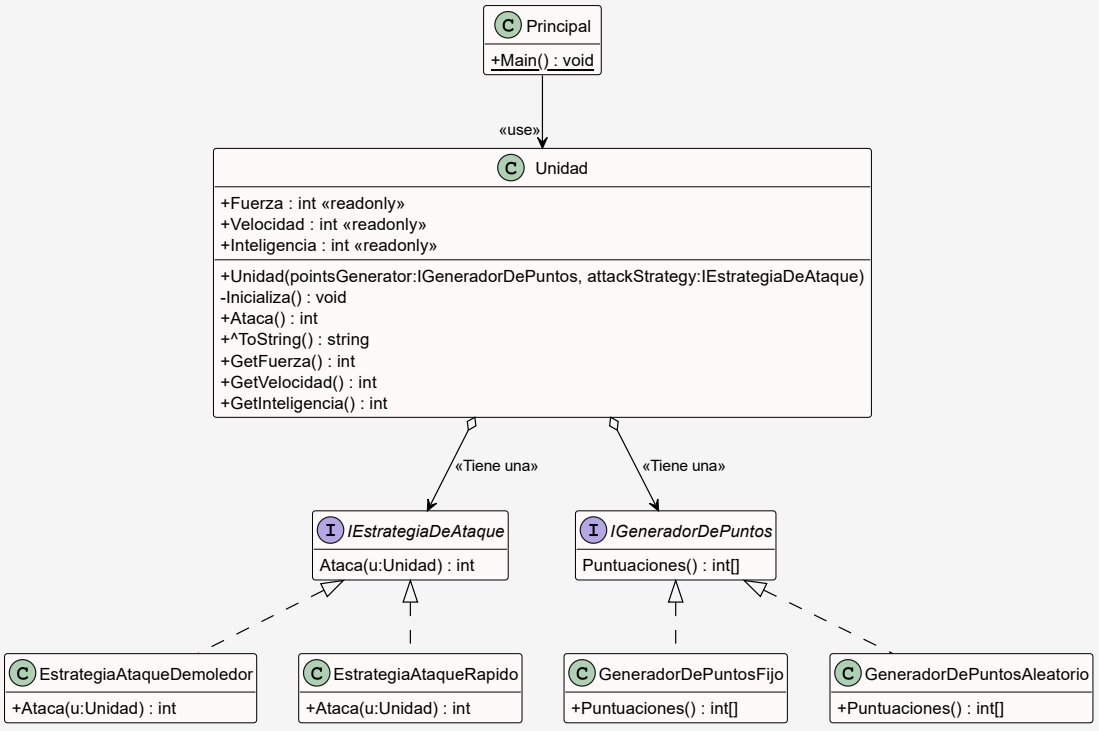
Vamos a ver un ejemplo de uso de interfaces a través de un ejemplo de aplicación del patrón de diseño **Strategy**.

📌 **Nota:** La idea no es aprender a usar el patrón, si no más bien ver un **caso práctico de uso de interfaces**.

Este patrón, nos permite definir una familia de algoritmos, encapsulando cada uno de ellos y haciéndolos intercambiables. Por tanto, nos permite que un algoritmo pueda variar de forma dinámica en el uso que hacen los clientes.

Supongamos que en un juego de estrategia en tiempo real, tenemos una **unidad** de ataque con una serie de propiedades como son `Fuerza`, `Velocidad` e `Inteligencia`. Sin embargo, hasta que no instanciemos el objeto unidad, no sabremos cómo se inicializarán estos valores y que '*estrategia*' va a seguir, es decir cómo se calcularán los puntos de ataque que va a tener la unidad de acuerdo a los valores anteriores.

Para cumplir esta especificación, implementaremos el siguiente patrón Strategy, conforme se expresa en al diagrama de clases siguiente.



Si nos fijamos, definimos 2 objetos agregados en mi `Unidad` que se corresponderán con 2 **generalizaciones** de **objetos que sabemos que implementan un determinado comportamiento** (interfaz). Uno tiene la funcionalidad de inicializar los puntos de las propiedades y otro la de calcular los puntos de ataque (estos són los algoritmos que hemos dicho antes que se definen al instanciar el objeto, en esta caso unidad).

```
// Definimos las generalizaciones de los comportamientos
public interface IEstrategiaDeAtaque
{
  int Ataca(Unidad u);
}
public interface IGeneradorDePuntos
{
  int[] Puntuaciones();
}


// Definimos las concreciones de dichos comportamientos
//    * Formas de inicializar las propiedades de la unidad.
public class GeneradorDePuntosFijo : IGeneradorDePuntos
{
    public int[] Puntuaciones() => new int[] { 6, 6, 6 };
}
public class GeneradorDePuntosAleatorio : IGeneradorDePuntos
{
    public int[] Puntuaciones()
    {
        int[] values = new int[3];
        var semilla = new Random();
        for (int i = 0; i < 3; i++)
        {
            values[i] = semilla.Next(0, 8);
        }
        return values;
    }
}
```

```
//    Formas de atacar de mi unidad y de calcular la fuerza de su ataque.
public class EstrategiaAtaqueDemoledor : IEstrategiaDeAtaque
{
    public int Ataca(Unidad u) => u.Fuerza * (u.Velocidad / 2);
}
public class EstrategiaAtaqueRapido : IEstrategiaDeAtaque
{
    public int Ataca(Unidad u) => u.Velocidad + u.Inteligencia;
}


public class Unidad
{
    private readonly IGeneradorDePuntos generadorDePuntos;
    private readonly IEstrategiaDeAtaque estrategiaDeAtaque;

    public int Fuerza { get; private set; }
    public int Velocidad { get; private set; }
    public int Inteligencia { get; private set; }


    public Unidad(IGeneradorDePuntos generadorDePuntos,
                  IEstrategiaDeAtaque estrategiaDeAtaque)
    {
        // Estos métodos definirán una forma de hacer las cosas genérica.
        this.generadorDePuntos = generadorDePuntos;
        this.estrategiaDeAtaque = estrategiaDeAtaque;
        // Este método aplica la estrategia de obtención de puntuaciones.
        Inicializa();
    }


    private void Inicializa()
    {
        // El array de puntuaciones se obtendrá según la concreción.
        int[] values = generadorDePuntos.Puntuaciones();
        Fuerza = values[0];
        Velocidad = values[1];
        Inteligencia = values[2];
    }
```

```
    public int Ataca()
    {
        // Los puntos de ataque se obtendrán según la concreción.
        return estrategiaDeAtaque.Ataca(this);
    }


    public override string ToString()
    {
        return $"Unidad\n\tF={Fuerza}\n\t" +
               $"V={Velocidad}\n\tI={Inteligencia}" +
               $"Ataque={Ataca()}\n";
    }
}
```

Fíjate que ahora instanciamos nuestra unidades de ataque con las **concreciones** que implementan los interfaces esperados por la clase `Unidad`.

```
public static void Main()
{
  var uAleatoriaDeAtaqueRapido = new Unidad(
              new GeneradorDePuntosAleatorio(),
              new EstrategiaAtaqueRapido());
  var uFijaDeAtaqueDemoledor = new Unidad(
              new GeneradorDePuntosFijo(),
              new EstrategiaAtaqueDemoledor());
  Console.WriteLine(uAleatoriaDeAtaqueRapido);
  Console.WriteLine(uFijaDeAtaqueDemoledor);
}
```