Unidad 16

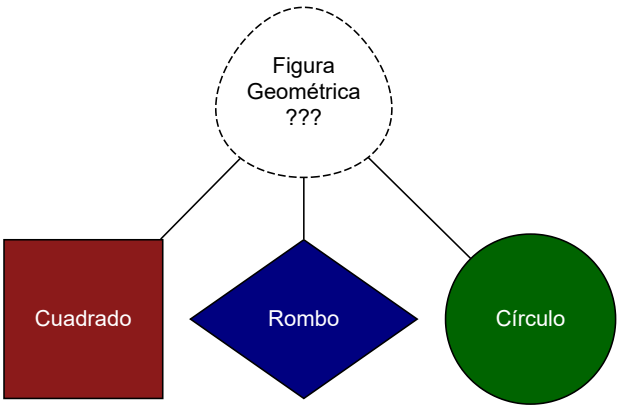
Descargar estos apunte en pdf o html

Índice

Abstracción

Concepto de clase asbtracta

En la mayoría de los casos, al buscar el polimorfismo con la herencia de superclases se nos darán **generalizaciones que no tienen sentido como objetos**. A este tipo de clases se les denominará **Clases Abstractas** y de las mismas **no podremos definir objetos**, y sí objetos para sus subclases.
El típico ejemplo, que muestra el diagrama, es el de **figuras geométrica**, no tiene sentido crear un objeto de la clase `Figura`. En cambio, sí que tiene sentido crear objetos de las subclases como `Cuadrado`, `Rombo` o `Circulo`.

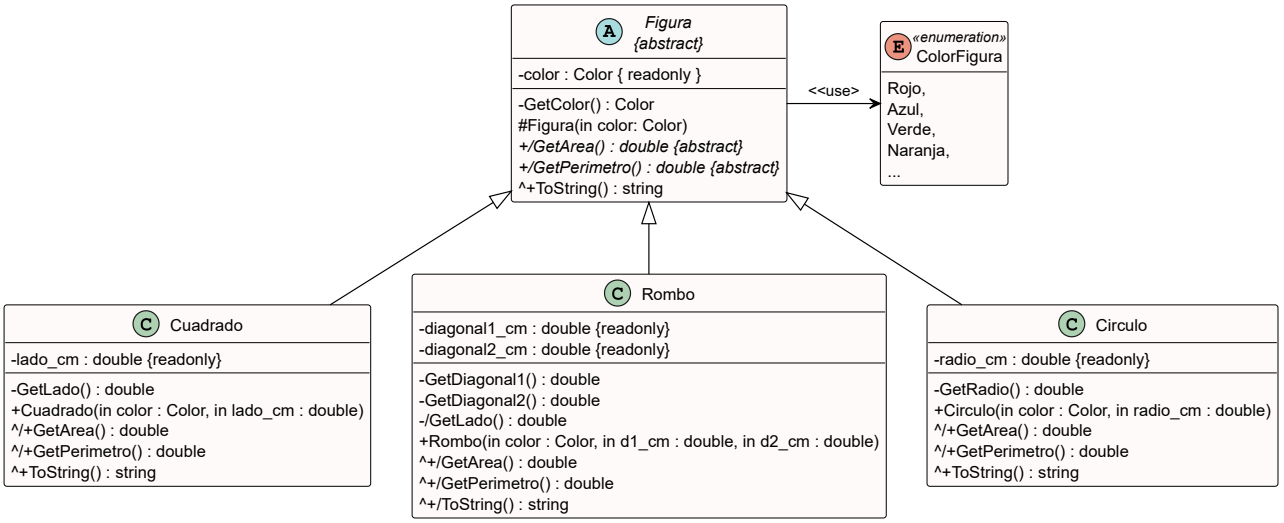


Definiciones desde el punto de vista de la POO tradicional:

- Cómo hemos comentado, no podremos **instanciar objetos** de una clase abstracta, pues no tiene sentido la existencia de dicha abstracción sin una especificación.
- Deberían, pero no es necesario, tener **al menos un campo o propiedad** común a todas las subclases.
- Deberían, tener **al menos un método abstracto o virtual puro**.
  - Un **método abstracto** o virtual puro **no definirá un '*cuerpo de método*'** y por tanto dejaremos su implementación en manos de las subclases o especificaciones.
  - Este método abstracto **deberá ser redefinido obligatoriamente en la subclases**.
- A las clases abstractas con **todos sus métodos abstractos** se les denomina ***'clases abstractas puras'***.

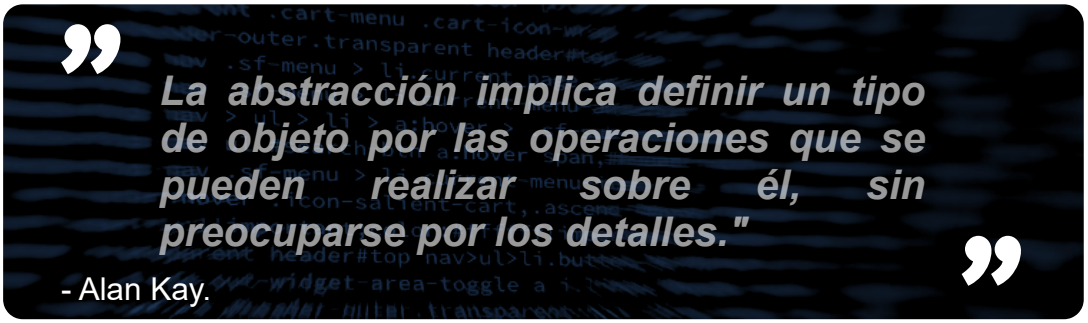
Modelando la abstracción en UML

Un posible modelo de clases para representar la **jerarquía de figuras representada en el diagrama del ejemplo anterior**, podría ser el siguiente:



Fíjate que en el diagrama UML hemos indicado que la clase es abstracta, además de usando el modificador `{abstract}` hemos puesto en *cursiva* el nombre, y en los get de las propiedades abstractas (virtuales) además de la cursiva hemos puesto el modificador `{abstract}`. Lo mismo sucedería con cualquier otro método.

> **Nota**
>
> Aunque, la cursiva ya no es necesaria a partir de la versión 2.5 de UML aún sigue siendo ampliamente aceptada.



Clases asbtractas en CSharp

La sintaxis será muy similar a la de la herencia, pero ampliando las condiciones que hemos descrito con anterioridad.

Vamos a ver la sintaxis a través de la implementación de nuestro **ejemplo con figuras**. Puedes descargar el código de este ejemplo del siguiente enlace: **abstraccion_figuras.cs**.

Paso 1: Definimos el tipo enumerado `ColorFigura` que representará los colores de las figuras que es usada por la clase `Figura` y por tanto sus subclases y por tanto **hay una dependencia de dicho tipo**.

```
public enum ColorFigura { Rojo, Azul, Verde, Naranja }
```

Paso 2: En primer lugar vamos a implementar la clase abstracta `Figura`.

```
public abstract class Figura
{
    private ColorFigura Color { get; }

    protected Figura(ColorFigura color)
    {
        Color = color;
    }

    abstract public double Area_cm2 { get; }

    abstract public double Perimetro_cm { get; }

    public override string ToString() => $"""
                    Color: {Color}
                    Area: {Area_cm2:F2} cm²
                    Perímetro: {Perimetro_cm:F2} cm
                    """;
}
```

Definimos la clase anteponiendo el modificador `abstract` y después
añadimos una **propiedad privada de solo lectura** común a todas las especificaciones que en este caso es el `Color`. Será privada pues es responsabilidad de la clase y así no repetimos código en las concreciones manteniendo la encapsulación.

Aquellas **propiedades calculadas que no podemos implementar hasta que no sepamos la especificación** que son `Area_cm2` y `Perimetro_cm`, les anteponemos el modificador `abstract` y las declaramos como `public` para que sean accesibles desde las subclases.

Los constructores que definamos pueden ser `protected` pues **solo tiene sentido usarlos desde las subclases**.
Nunca podré hacer un `Figura f = new (color: ColorFigura.Rojo);`

Pero, **¿Cómo puede ser que `Figura` use `Area_cm2` o `Perimetro_cm` si no están implementados?**.
De hecho, estoy accediendo a su valor desde el `ToString()` pero realmente no se implementan.

Porque, tendré siempre la seguridad de que un si tengo un objeto de tipo `Figura` habré realizado una sustitución de tipo y por tanto **tendré un objeto de una subclase** que **implementa esas propiedades**

o métodos de forma obligatoria y por tanto, **siempre se producirá un enlace dinámico** a la propiedad de la concreción que la implementa.

Paso 3: Implementación de las concreciones (subclases)

```
public class Circulo : Figura
{
    public double Radio_cm { get; }

    public Circulo(
           ColorFigura color,
           double radio_cm) : base(color)
    {
        Radio_cm = radio_cm;
    }

    public override double Area_cm2 =>
        Math.PI * Math.Pow(Radio_cm, 2);

    public override double Perimetro_cm =>
        Math.PI * Radio_cm * 2;

    public override string ToString() => $"""
                    ____Círculo____
                    Radio: {Radio_cm} cm
                    {base.ToString()}
                    """;
}
```

Desde el constructor de la subclase `Circulo` llamamos al constructor de la superclase `Figura` para inicializar el color y es posible porque lo hemos definido como `protected`.

Nos obligará a implementar las propiedades `Area_cm2` y `Perimetro_cm`. Fíjate que las hemos implementado como propiedades de calculadas.

Además, hemos implementado el método `ToString()` que nos permite mostrar la información propia de la figura como el Color que no es accesible desde las subclases. En ocasiones, esto no es posible y deberíamos marcar `Color` como `protected` para que las subclases puedan acceder a él.

El resto de implementaciones como `Cuadrado` y `Rombo` serán análogas a `Circulo`...

```
public class Cuadrado : Figura
{
    private double Lado_cm { get; }

    public Cuadrado(ColorFigura color, double lado_cm) : base(color)
    {
        Lado_cm = lado_cm;
    }
    public override double Area_cm2 => Lado_cm * Lado_cm;
    public override double Perimetro_cm => Lado_cm * 4d;
    public override string ToString() => $"""
                    ____Cuadrado____
                    Lado: {Lado_cm} cm
                    {base.ToString()}
                    """;
}
```

```
public class Rombo : Figura
{
    private double Diagonal1_cm { get; }
    private double Diagonal2_cm { get; }

    public Rombo(ColorFigura color, double d1_cm, double d2_cm) : base(color)
    {
        Diagonal1_cm = d1_cm;
        Diagonal2_cm = d2_cm;
    }
    private double Lado_cm => Math.Sqrt(Math.Pow(Diagonal1_cm / 2d, 2d) + Math.Pow(Diagonal2_cm / 2d, 2d));
    public override double Area_cm2 => Diagonal1_cm * Diagonal2_cm / 2d;
    public override double Perimetro_cm => Lado_cm * 4d;
    public override string ToString() => $"""
                    ____Rombo____
                    Diagonal1: {Diagonal1_cm} cm
                    Diagonal2: {Diagonal2_cm} cm
                    Lado: {Lado_cm:F2} cm
                    {base.ToString()}
                    """;
}
```

El polimorfismo de datos lo vamos a poder realizar igual que en el caso de `Articulo` pero **no vamos a poder crear ningún artículo directamente**.

```
public static void Main()
{
    List<Figura> figuras =
    [
        new Cuadrado(
            color: ColorFigura.Rojo,
            lado_cm: 2),
        new Rombo(
            color: ColorFigura.Azul,
            d1_cm: 2,
            d2_cm: 2),
        new Circulo(
            color: ColorFigura.Verde,
            radio_cm: 2)
    ];

    foreach (Figura f in figuras)
        Console.WriteLine(f);
}
```

Mostrará por consola:

```
____Cuadrado____
Lado: 2 cm
Color: Rojo
Area: 4,00 cm²
Perímetro: 8,00 cm
____Rombo____
Diagonal1: 2 cm
Diagonal2: 2 cm
Lado: 1,41 cm
Color: Azul
Area: 2,00 cm²
Perímetro: 5,66 cm
____Círculo____
Radio: 2 cm
Color: Verde
Area: 12,57 cm²
Perímetro: 12,57 cm
```

🎓 Caso de estudio

Supongamos el siguiente modelo simplificado, en el cual, un **Sicólogo** crea un gabinete o **consulta** psicológica, donde cada día atiende a los **pacientes** que le llegan.
Los pacientes **irán entrando a la consulta** y en un momento dado el sicólogo les **pasará consulta** por orden de llegada.
Los pacientes tendrán un **nombre** y **de momento**, nuestro sicólogo solo sabe atender a dos tipos de pacientes:

1. Pacientes **alegres**
2. Pacientes **tristes**

En el momento en que el sicólogo **atiende a un paciente**, se producirá un diálogo con el mismo que empezará igual para todos los pacientes:

```
- Sicólogo: Buenos días!. ¿Cómo se llama?
- Paciente: Soy <Nombre>
- Sicólogo: Dígame <Nombre>!.. ¿Qué siente?
```

Pero dependiendo del tipo de paciente obtendremos un tipo de **respuesta** diferente dependiendo de si es un paciente alegre o triste respectivamente...

```
- Paciente:  Pues... ahora estoy alegre.

- Paciente:  Pues... ahora estoy triste.
```

El sicólogo realizará un **diagnóstico** diferente dependiendo de si es un paciente alegre o triste respectivamente...

```
- Sicólogo:  le veo estupendamente. Enhorabuena!! no necesita más terapia.

- Sicólogo:  tome fluoxetina 20mg y vuelva en un mes.
```

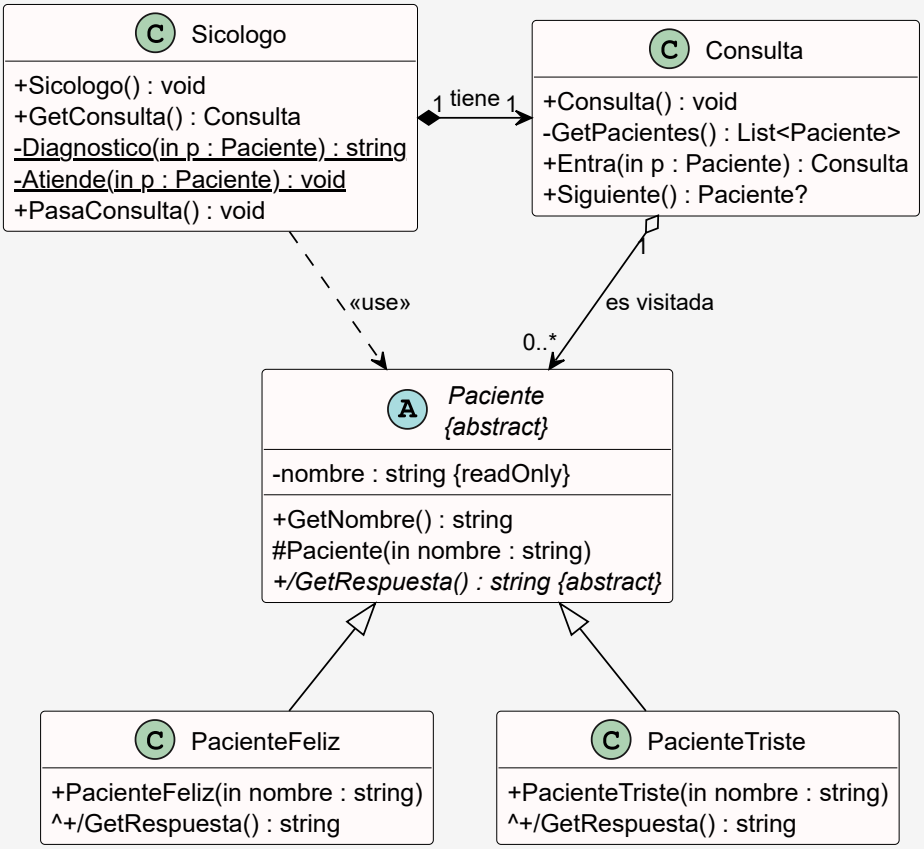
Tras realizar el diagnostico el sicólogo dira ...

```
- Sicólogo: Que pase el siguiente !!!
```

Atendiendo a otro paciente si hay aún pendientes en consulta.

Piensa en un posible modelado o diseño de clases para representar las especificaciones anteriores.

Si no se te ocurre ningúno, aquí tienes una propuesta de implementación.



Antes de ver la implementación comentada de esta prupuesta, intenta realizarla tú y así posteriormente puedes ver la propuesta de solución para lo que no has sabido resolver. Puedes descargar el código de la propuesta de solución del siguiente enlace: **abstraccion_sicologo.cs**.

Propuesta de solución al caso de estudio:

Paso 1: Crearemos la clase abstracta `Paciente`. Será abstracta porque los pacientes del sicólogo responden de forma diferente dependiendo de su especificación.

```
public abstract class Paciente
{
    public string Nombre { get; }
    public abstract string Respuesta { get; }

    protected Paciente(string nombre)
    {
        Nombre = nombre;
    }
}
```

Marcamos la clase como `abstract` y definimos una propiedad `Nombre` común a todos los pacientes y una **propiedad abstracta** `Respuesta` que será implementada por las **concreciones de paciente**.

Paso 2: Definiremos las **concreciones de** `Paciente` para que respondan según su estado de ánimo.

```
public class PacienteAlegre : Paciente
{
    public override string Respuesta
    => "Pues... ahora estoy alegre.";

    public PacienteAlegre(string nombre)
    : base(nombre) { ; }
}

public class PacienteTriste : Paciente
{
    public override string Respuesta
    => "Pues... ahora estoy triste.";

    public PacienteTriste(string nombre)
    : base(nombre) { ; }
}
```

Tanto `PacienteAlegre` como `PacienteTriste` están obligados a invalidar la propiedad `Respuesta`.

Además, en el constructor de cada concreción **llamamos al constructor protegido de la superclase** para inicializar el nombre. Sin embargo, el cuerpo del constructor de las concreciones está vacío pues no hay nada más que inicializar. Un convenio, para indicar a otros programadores q**ue lo hemos dejado vacío a propósito**, es poner una instrucción vacía `{ ; }`.

Paso 3: Definiremos la clase `Consulta`

```
public class Consulta
{
    private List<Paciente> Pacientes { get; }
    public Consulta()
    {
        Pacientes = [];
    }

    public Consulta Entra(Paciente p)
    {
        Pacientes.Add(p);
        return this;
    }

    public Paciente? Siguiente
    {
        get
        {
            Paciente? p = Pacientes.FirstOrDefault
            if (p != null)
                Pacientes.RemoveAt(0);
            return p;
        }
    }
}
```

Representamos la agregación multiple con una lista de pacientes que inicializamos en el constructor.

Definimos el método `Entra` que añade un paciente a la consulta y devuelve la propia consulta para poder encadenar llamadas de forma **fluida**.

El métodos `Siguiente` devuelve el siguiente paciente en la lista o `null` si no hay pacientes. Fíjate que el método `FirstOrDefault()` devuelve el primer elemento o `null` si la lista está vacía. Además, si devuelve un paciente, lo elimina de la lista.

Paso 4: Definiremos la clase `Sicologo`

> **Nota**
>
> Tiene un dependencia débil de uso (`<<use>>`) con `Paciente` en el UML, ya que necesita conocer la clase. Normalmente no se suele indicar en el UML, pero es importante que lo tengas en cuenta.

```
public class Sicologo
{
    public Consulta Consulta { get; }
    public Sicologo(Consulta consulta)
    {
        Consulta = consulta;
    }
    private static string Diagnostico(Paciente p) => p switch
    {
        PacienteAlegre _ => $"{p.Nombre} le veo estupendamente. Enhorabuena!! no necesita más terapia.",
        PacienteTriste _ => $"{p.Nombre} tome fluoxetina 20mg y vuelva en un mes.",
        _ => $"{p.Nombre} déjeme que estudie un poco más su caso y vuelva la semana que viene."
    };
    private static void Atiende(Paciente p)
    {
        StringBuilder proceso = new();
        proceso.AppendLine("- Sicólogo: Buenos días!. ¿Cómo se llama?")
                .AppendLine($"- Paciente: Soy {p.Nombre}")
                .AppendLine($"- Sicólogo: Dígame {p.Nombre}!.. ¿Qué siente?")
                .AppendLine($"- Paciente: {p.Respuesta}")
                .AppendLine($"- Sicólogo: {Diagnostico(p)}")
                .AppendLine("- Sicólogo: Que pase el siguiente !!!");
        Console.WriteLine(proceso);
    }
    public void PasaConsulta()
    {
        Paciente? p;
        while ((p = Consulta.Siguiente) != null)
            Atiende(p);
    }
}
```

Fíjate que `Sicologo` tiene una **composición** con `Consulta` y por tanto el sicólogo es responsable de la misma. Además, su único método público es `PasaConsulta()` que atiende a todos los pacientes que hay en la consulta. Siendo el resto de métodos **privados y de clase (estáticos)**, pues no necesitan acceder a ningún campo o propiedad de instancia y simplemente es una **modularización de la tarea de pasar consulta**.

Paso 5: Vamos a crear un sencillo programa principal de test...

```
public static void Main()
{
    Sicologo sicologo = new (consulta : new ());
    sicologo.Consulta
            .Entra(new PacienteAlegre("Xusa"))
            .Entra(new PacienteAlegre("Juanjo"))
            .Entra(new PacienteTriste("Carmen"));
    sicologo.PasaConsulta();
}
```

Cómo sucedía en otros casos podemos crear una nueva especificación de `Paciente` por ejemplo...

```
public class PacienteSociopata : Paciente
{
    public override string Respuesta => "Vas a morir .. muuhaaahahahaha !!";
    public PacienteSociopata(string nombre) : base(nombre) { ; }
}
```

El **consultorio seguiría funcionando**. Pero, esta vez el `Sicologo` debería actualizarse para saber tratar a este tipo de pacientes.

```
private static string Diagnostico(Paciente p) => p switch
{
    // ... código omitido por abreviar
    PacienteSociopata _ => $"Lo siento!. Debo aplicarte una decarga de 10000V justo ahora.",
};
```

Reto

¿Se te ocurre cómo crear **diferentes tipos de sicólogos que diagnostiquen de forma diferente** o alguna forma de hacer que sepamos que tenemos que actualizar al sicólogo si hay nuevos pacientes?

El Patrón "*Union Type*" o "*Sum Type*"

En programación, un "union type" (también conocido como "sum type", tipo de unión, tipo de suma o unión etiquetada) es un tipo de dato que **puede contener un valor de varios tipos diferentes, pero solo uno a la vez**.

El nombre "***Union Type***" proviene de la **teoría de tipos**, donde el conjunto total de posibles valores del tipo es la "suma" (o unión) de los conjuntos de valores de sus tipos constituyentes. Este concepto **es común en lenguajes de programación funcional** como `Haskell` y `F#` donde permiten modelar datos de una manera muy expresiva y segura evitando errores en tiempo de ejecución.

Con el tiempo, los beneficios de este patrón se hicieron evidentes y **los lenguajes de programación más modernos, incluidos los orientados a objetos**, han ido adoptando formas de implementarlo como el caso de como `C#`, `Java`, `Kotlin`, `Swift` o `Rust`.

Por resumir y poner un símil de algo que ya conocemos, **podríamos decir que es un tipo enumerado excluyente mejorado**. Donde en lugar de tener un conjunto de constantes, tenemos una enumeración de tipos diferentes unificados por una abstracción y que guardarán un estado inmutable diferente al poder tener propiedades distintas. Por todo ello, en un momento dado, una instancia de la abstracción, **solo podrá ser de uno de ellos de forma excluyente**. De hecho, en lenguajes como `Swift` o `Rust` se definen con la palabra reservada `enum`.

"*Union Type*" en CSharp

En `C#` no existe una sintaxis específica para definir un "*Union Type*" pero podemos implementarlo usando **clases abstractas y herencia**. Vamoslo a través a ver un **ejemplo sencillo**.

Supongamos que queremos retornar en una solo tipo una validación sobre una entrada de datos. Hasta ahora, devolvíamos una tupla con un booleano y un mensaje de error `(bool valido, string? mensaje)` y si el booleano era verdadero, el mensaje estaría a null. En el ejemplo validamos que **un entero con la edad tuviese 18 años o más**.

```
static public (bool valido, string? mensaje) ValidaEdad(int edad)
{
    bool valido = edad >= 18;
    string? mensaje = !valido
                      ? "El usuario debe ser mayor de 18 años."
                      : null;
    return (valido, mensaje);
}
public static void Main()
{
    (bool valido, string? _) = ValidaEdad(13);
    if (valido)
        Console.WriteLine("Edad válida");
}
```

Pero, si lo pensamos bien realmente hay dos posibles tipos de resultado y uno de ellos **tiene asociado un estado de error** (el mensaje) y el otro no y esa consistencia debe saberla y gestionarla el que llama al método.

```
public abstract record Validacion
{
    public record Exito() : Validacion;
    public record Error(string Mensaje) : Validacion;
}
```

Una posible solución es definir un **value object abstracto por referencia denominado** `Validacion` con dos **tipos anidados** a modo de enumeración de casos que en nuestro ejemplo podrían ser `Exito` y `Error`.

Cada tipo anidado será a su vez un **record** que hereda de `Validacion` y que podrá tener sus propias propiedades. En este caso, `Error` tiene una propiedad `Mensaje`.

Imaginemos ahora, que tenemos una clase de utilidad llamada `Validador` que tiene varios métodos de validación que devuelven una `Validacion` ...

```
public static class Validador
{
    public static Validacion ValidaEdad(int edad) => edad switch
    {
        >= 18 => new Validacion.Exito(),
        _ => new Validacion.Error(Mensaje: "El usuario debe ser mayor de 18 años.")
    };
    public static Validacion ValidaNombre(string nombre) => string.IsNullOrWhiteSpace(nombre) switch
    {
        false => new Validacion.Exito(),
        true => new Validacion.Error(Mensaje: "El nombre no puede estar vacío.")
    };
}
```

Fíjate que en cada método de utilidad, si la validación es correcta devolvemos una instancia de `Validacion.Exito` y si no lo es, devolvemos una instancia de `Validacion.Error` con el mensaje de error correspondiente. Esto es, retornamos en un solo tipo dos posibles tipos de resultado con su propio estado inmutable.

Un caso de uso del ejemplo que nos hemos planteado podría ser una clase `Acceso` que representa el **acceso de un usuario a una web** y que tiene una propiedad `Validacion` que valida el nombre y la edad del usuario.

```
public record class Acceso
{
    public DateTime FechaHora { get; }
    public string Nombre { get; }
    public int Edad { get; }

    public Validacion Validacion =>
        Validador.ValidaNombre(Nombre) is Validacion.Error errorNombre
        ? errorNombre
        : Validador.ValidaEdad(Edad);

    public Acceso(string nombre, int edad, DateTime? fechaHora = null)
    {
        FechaHora = fechaHora ?? DateTime.Now;
        Nombre = nombre;
        Edad = edad;
    }

    public override string ToString()
    => $"{Nombre} ({Edad} años) el {FechaHora:dd/MM/yyyy} a las {FechaHora:HH:mm}";
}
```

Fíjate que la propiedad `Validacion` usa el operador `is` para comprobar si la validación del nombre ha devuelto un error y en ese caso, devuelve dicho error. Si no ha habido error en el nombre, entonces valida la edad y devuelve el resultado de dicha validación que puede ser un éxito o un error.

Si ejecutamos el siguiente **programa principal de test** donde verificamos los casos posibles de acceso...

```
public static void Main()
{
    List<Acceso> accesos =
    [
        new (nombre: "", edad: 30, fechaHora: new DateTime(2026, 3, 22, 10, 35, 0)),
        new (nombre: "Luis", edad: 17, fechaHora: new DateTime(2026, 3, 22, 10, 39, 0)),
        new (nombre: "Marta", edad: 22, fechaHora: new DateTime(2026, 3, 22, 10, 50, 0)),
    ];

    foreach (var acceso in accesos)
    {
        string mensaje = acceso.Validacion switch
        {
            Validacion.Error error => $"Acceso denegado a {acceso}.\nMotivo: {error.Mensaje}\n",
            _ => $"Acceso permitido a {acceso}\n",
        };
        Console.WriteLine(mensaje);
    }
}
```

Mostrará por consola:

```
Acceso denegado a  (30 años) el 22/03/2026 a las 10:35.
Motivo: El nombre no puede estar vacío.

Acceso denegado a Luis (17 años) el 22/03/2026 a las 10:39.
Motivo: El usuario debe ser mayor de 18 años.

Acceso permitido a Marta (22 años) el 22/03/2026 a las 10:50
```

Nota

Puedes descargar el código de este ejemplo del siguiente enlace: **abstraccion_uniontype.cs**.

Interfaces

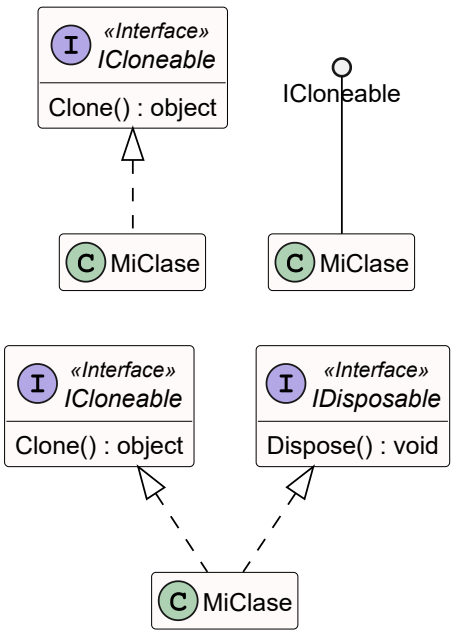
Básicamente un **Interfaz** es la definición de un conjunto de **interfaces de métodos** y accesores o mutadores (como **Propiedades**). Es muy parecido a definir una **clase abstracta pura**, pero **sin ningún tipo de campo**, constructor, **ni modificador de acceso** (public, private, etc…). Como en 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.

De hecho, es el caso de abstracción más puro que existe, pues **solo define el comportamiento que debe tener una clase** que la **implemente** el interfaz y por tanto **es la forma más común y recomendable de definir la abstracción** de un comportamiento. En otras palabras, siempre que no haya un **campo** o **propiedad no calculada** común a todas las especificaciones, la forma correcta de definir la abstracción es mediante un interfaz.

Pueden implementarse en muchos lenguajes OO con idénticas características:

- Es posible la **herencia múltiple** de interfaces.
- No pueden definir campos pero sí propiedades calculadas.
- 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.

Representación en los diagramas de clases UML



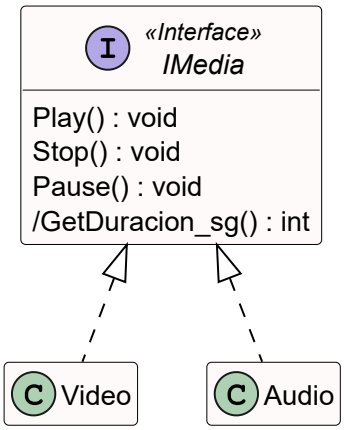
Representa que la clase `MiClase` **implementa** el interfaz `ICloneable`.

Fíjate que 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**.

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

Además como vemos en este segundo diagrama. Podemos hacer que una clase **implemente** o "*herede*" de más de un interfaz.

Interfaces en CSharp



Como ya habrás podido apreciar en los diagramas, 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>
}
```

Por tanto, si queremos definir el interfaz anterior y aplicarlo a una `Video`. Haremos que `Video` herede del interfaz con la sintaxis de herencia que hemos usado hasta ahora y esta se verá obligada a implementar todo lo que hayamos definido en el interfaz, **sin necesidad de anteponer el modificador** `override`.

```
interface IMedia
{
    void Play();
    void Stop();
    void Pause();

    int Duracion_sg { get; }
}
```

```
class Video : IMedia
{
    // ...
    public int Duracion_sg => 20;
    public void Pause() => Console.WriteLine("Pausando el vídeo.");
    public void Play() => Console.WriteLine("Reproduciendo el vídeo.");
    public void Stop() => Console.WriteLine("Parando el vídeo.");
}
```

> **Aviso**
>
> Fíjate que no hemos definido ni en el UML ni en el código ningún modificador de acceso en el interfaz. Por defecto, **todos los métodos y propiedades de un interfaz son públicos**. Es un error muy común entre los programadores noveles el poner el modificador `public` y esto **será considerado en el examen como un error**.

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**. Aunque podríamos utilizar interfaces propios para hacer lo mismo, no lo hacemos porque **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 básicamente implementan muchos tipos como las cadenas, números, fechas, etc... y que es usado por las BCL para ordenar listas o arrays de objetos.

Recordatorio de uso:

```
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.
```

IDisposable

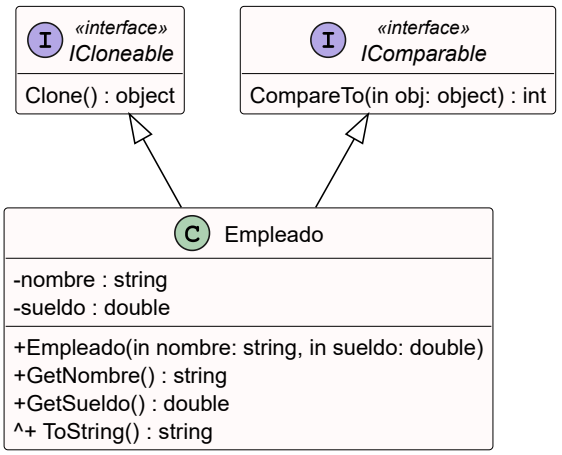
IDisposable Me indicará que el objeto debe implementar el método `void Dispose()` que **se encargará de liberar los recursos usados por el objeto.** No debemos confundirlo con el destructor `~<Tipo>()`.

Indicaremos a las BCL que nuestro objeto tiene el comportamiento de liberar recursos y lo utilizaremos junto a la **instrucción** `using` que veremos más adelante.

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`.

Ejemplo de uso de interfaces predefinidos en las BCL

Vamos a definir una clase `Empleado` que tiene una propiedad `Nombre` de solo lectura y una `Sueldo` que debe estar entre 1200 y 3000 euros. Además, **implementa los interfacesde las BCL** `IComparable` e `ICloneable`. De tal manera que la comparación entre empleados **se hará por su sueldo**.



Puedes descargar el código de este ejemplo del siguiente enlace: **interfaces_empleado_comparable_y_clonable.cs**.

Fíjate que podemos implementar más de un interfaz en la misma clase y los indicaremos separados por comas.

```
public class Empleado : IComparable, ICloneable
{
    private double _sueldo;
    public double Sueldo
    {
        get => _sueldo;
        set
        {
            Debug.Assert(
                condition: value >= 1200D && value <= 3000D,
                message: "El sueldo debe estar entre 1200 y 3000 euros");
            _sueldo = value;
        }
    }

    public string Nombre { get; }

    public Empleado(string nombre, double sueldo)
    {
        Nombre = nombre;
        Sueldo = sueldo;
    }

    public override string ToString() => $"Nombre: {Nombre,-8}Sueldo: {Sueldo:F0}";
}
```

Ahora estaremos obligados a implementar los métodos y propiedades definidos en los interfaces.

> **Tip**
>
> En **VSCode**, si situas el cursor sobre un interfaz que implementa una clase y pulsas `Ctrl + .` te mostrará un menú contextual con la opción de ***"Implementar interfaz"*** que generará automáticamente los métodos y propiedades que debes implementar.

Puesto que vamos a comparar por `Sueldo` que es un tipo básico que ya implementa `IComparable`, podemos usar su método `CompareTo` para implementar el nuestro.

```
public int CompareTo(object? obj) => Sueldo.CompareTo((obj as Empleado)!.Sueldo);
```

Fíjate que comparamos el `Sueldo` del objeto que llama a `CompareTo` (`this`) con el sueldo del objeto que se pasa como parámetro. Pero hemos hecho un **downcasting** con `as` y el operador de supresión de nulabilidad `!` para indicarle al compilador genere un error de ejecución si el objeto no es un `Empleado` .

Para implementar el método `Clone` simplemente llamamos al constructor de la clase con los mismos parámetros que el objeto que llama a `Clone` .

```
public object Clone() => new Empleado(Nombre, Sueldo);
```

> **Aviso**
>
> Si alguno de los objetos que componen la clase **fuera un tipo referencia mutable**, deberíamos hacer una **copia en profundidad** de los mismos para evitar que el objeto clonado comparta referencias con el original. Esto es, deberíamos llamar al método `Clone` de dichos objetos si implementan `ICloneable` o crear nuevas instancias de los mismos copiando sus valores. En este caso, como `string` es inmutable y `double` es un tipo valor, no tenemos que preocuparnos por pasarlos tal cual al constructor.

Veamos ahora un sencillo programa principal de test donde probar la clase `Empleado` y los interfaces que implementa.

```
static void Main()
{
    Empleado e1 = new (nombre: "Juanjo", sueldo: 2000);
    Empleado e2 = new (nombre: "Carmen", sueldo: 2800);
    Empleado e3 = new (nombre: "Xusa", sueldo: 2400);

    // Aunque sabemos que e1 es ICloneable, podemos siempre
    // preguntar a un objeto si implementa un interfaz
    // para poder llamar a sus métodos en esta caso Clone()
    // Como clone devuelve un object, debemos hacer un downcasting.
    Empleado e4 = e1 is ICloneable
                    ? (e1.Clone() as Empleado)!
                    : e1;
    // Modificamos el sueldo de e4 y no debería afectar a e1.
    e4.Sueldo = 2900;

    Empleado[] empleados = [e1, e2, e3, e4];

    // Puesto que Empleado es IComparable
    // podemos usar Array.Sort() para ordenar el array por sueldo.
    // Si no lo fuera, se produciría un error al ejecutar.
    Array.Sort(empleados);
    Console.WriteLine(string.Join<Empleado>("\n", empleados));
}
```

Mostrará por consola:

```
Nombre: Juanjo  Sueldo: 2000
Nombre: Xusa    Sueldo: 2400
Nombre: Carmen  Sueldo: 2800
Nombre: Juanjo  Sueldo: 2900
```

Fíjate que, además de **mostrar los empleados ordenados por sueldo**, al clonar `e1` en `e4` y cambiar su sueldo, no hemos modificado el sueldo de `e1`. Esto es porque hemos hecho una **copia en profundidad** del objeto.

El Patrón "*Strategy*"

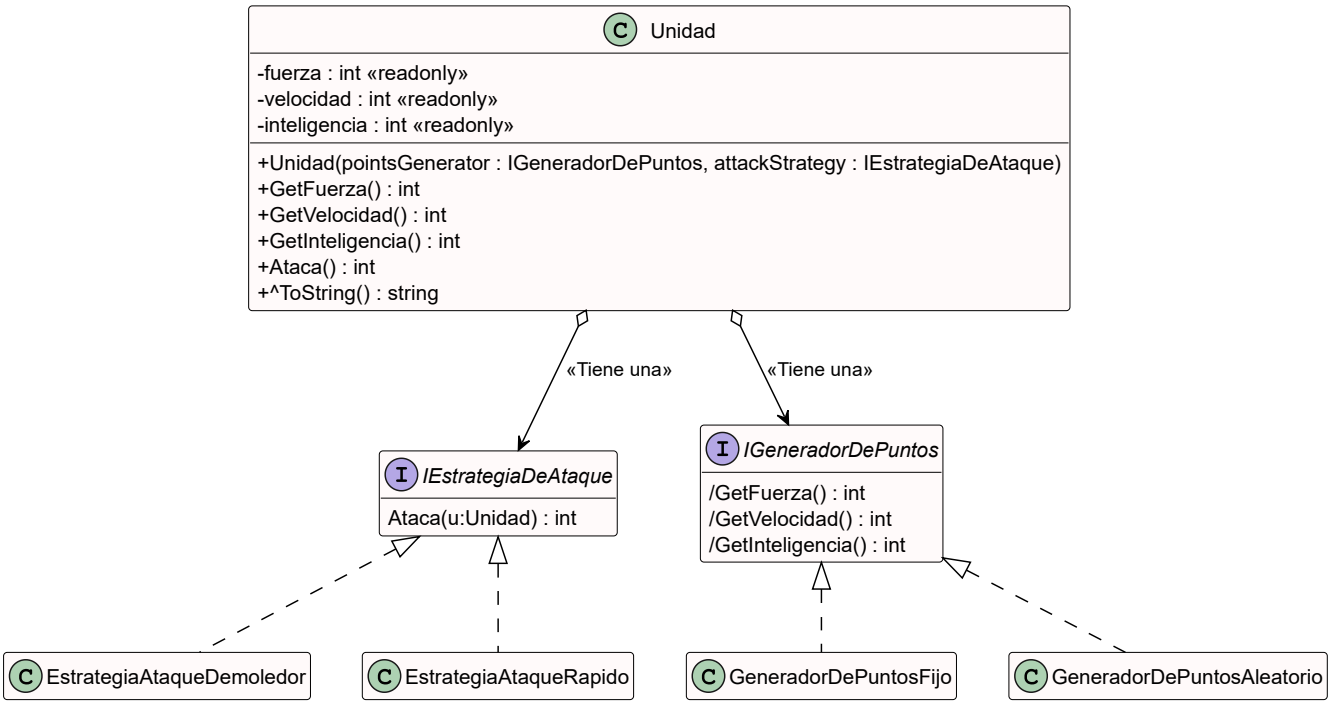
Los interfaces son muy usados en la industria del software para definir abstracciones de comportamientos y **patrones de diseño**. Uno de los más usados es el patrón **Strategy** que **será base de muchos otros patrones de diseño** y será básico para aplicar muchos aspectos de la POO moderna como los principios SOLID, la inyección de dependencias, etc...

Este patrón nos permite abstraer un comportamiento que puede tener múltiples implementaciones y que se definen en tiempo de ejecución. Es decir, el algoritmo o comportamiento concreto que se va a usar, **no se define hasta que se instancia el objeto**. Determinadas clases **deberán exponer en su constructor las abstracciones colaboradoras** (interfaces) que definen estos comportamientos para que el usuario de la clase pueda decidir qué implementación concreta usar al instanciar el objeto.

Vamos a ver pues **un ejemplo de uso de interfaces** a través del uso de este patrón de diseño. Tratando de aislar el concepto de otras consideraciones.

Para mostrar su uso, nada mejor que a través de juego de estrategia en tiempo real. En él, tendremos 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).

Abastracción y concrecciones de la estrategia de obtención de puntuaciones:

```
public interface IGeneradorDePuntos
{
    int Fuerza { get; }
    int Velocidad { get; }
    int Inteligencia { get; }
}

public class GeneradorDePuntosFijo : IGeneradorDePuntos
{
    public int Fuerza => 6;
    public int Velocidad => 6;
    public int Inteligencia => 6;

}

public class GeneradorDePuntosAleatorio : IGeneradorDePuntos
{
    private static int PuntosAlestorios => new Random().Next(0, 8);
    public int Fuerza => PuntosAlestorios;
    public int Velocidad => PuntosAlestorios;
    public int Inteligencia => PuntosAlestorios;
}
```

Abastracción y concrecciones de la estrategia de ataque:

```
public interface IEstrategiaDeAtaque
{
    int Ataca(Unidad u);
}

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
{
    public int Fuerza { get; }
    public int Velocidad { get; }
    public int Inteligencia { get; }
    private IGeneradorDePuntos GeneradorDePuntos { get; }
    private IEstrategiaDeAtaque EstrategiaDeAtaque { get; }

    // La clase siempre expondrá sus dependecias de abtracciones
    // colaboradoras a través del constructor.
    public Unidad(
        IGeneradorDePuntos generadorDePuntos,
        IEstrategiaDeAtaque estrategiaDeAtaque)
    {
        GeneradorDePuntos = generadorDePuntos;
        EstrategiaDeAtaque = estrategiaDeAtaque;
        Fuerza = GeneradorDePuntos.Fuerza;
        Velocidad = GeneradorDePuntos.Velocidad;
        Inteligencia = GeneradorDePuntos.Inteligencia;
    }

    public int Ataca() => EstrategiaDeAtaque.Ataca(this);

    public override string ToString()
    => $"Unidad con {GeneradorDePuntos.GetType().Name} y {EstrategiaDeAtaque.GetType().Name} " +
       $"F={Fuerza} V={Velocidad} I={Inteligencia} A={Ataca()}";
}
```

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

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

Mostrará por consola:

```
Unidad con GeneradorDePuntosAleatorio y EstrategiaAtaqueRapido F=2 V=4 I=4 A=8
Unidad con GeneradorDePuntosFijo y EstrategiaAtaqueDemoledor F=6 V=6 I=6 A=18
```

> **Idea principal** 💡
>
> Lo más importante de este patrón es que **podemos definir nuevas estrategias** de generación de puntos o de ataque sin necesidad de modificar la clase `Unidad` y por tanto, **sin necesidad de recompilarla**. Simplemente, creamos nuevas concreciones que implementen los interfaces definidos.

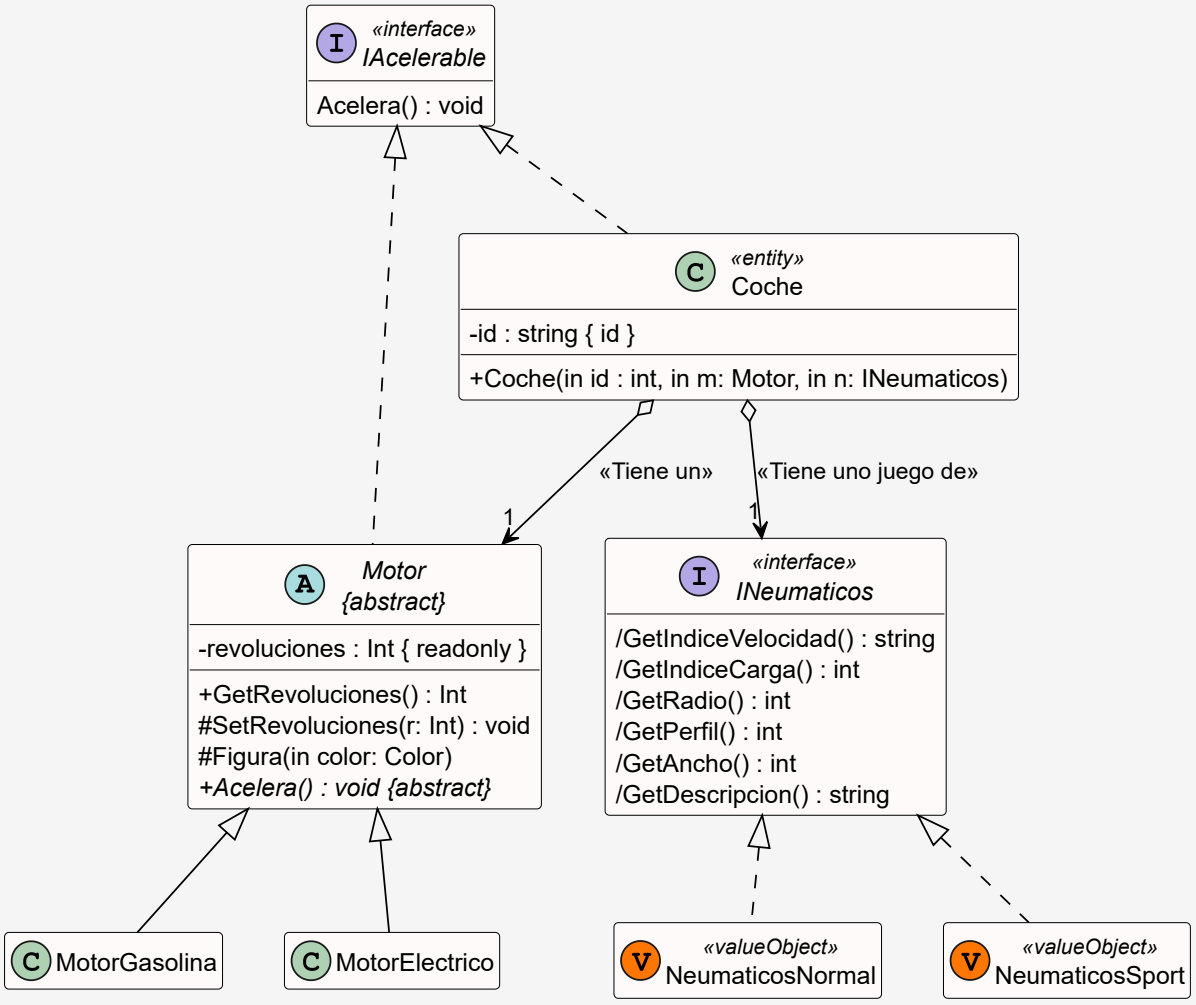
Puedes descargar el código de este ejemplo del siguiente enlace: **interfaces_satrategy_unidades.cs**.

🎓 **Caso de estudio:**

Vamos a aplicar alguno de los conceptos vistos en esta unidad para modelar una clase `Coche` que tendrá un `Motor` y unos `Neumáticos`. El motor podrá ser de gasolina o eléctrico y los neumáticos normales o deportivos. Además, el coche podrá acelerar y la forma de hacerlo dependerá del tipo de motor que tenga.

Para ello, vamoa a abdtraer el comportamiento de acelerar en un interfaz y haremos que el `Motor` y el `Coche` lo implemente. Además, haremos que el `Coche` implemente un patrón **Strategy** para definir el tipo de motor y neumáticos que tendrá en tiempo de ejecución a través de abstracciones que expondremos en el constructor.

Una posible modelización UML del caso de estudio podría ser la siguiente:



¿Sabrías implementarlo en C#?. Si no es así, puedes seguir la explicación a continuación y descargar el código de este caso de estudio del siguiente enlace: **interfaces_strategy_coche.cs**.

Definimos el interfaz `IAcelerable` que define el comportamiento de acelerar:

```
public interface IAcelerable
{
    void Acelera();
}
```

Definimos la abstracción del `Motor` como una clase abstracta que implementa el interfaz `IAcelerable`. La hemos hecho ambstracta pues tiene un campos común que define un estado común a todas las especificaciones (**las revoluciones**).

```
public abstract class Motor : IAcelerable
{
    // Inicializando la propiedad evito definir un constructor.
    public int Revoluciones { get; protected set; } = 0;

    // Hasta que no concretemos un tipo de motor
    // no sabremos cómo se acelera.
    abstract public void Acelera();
}

public class MotorGasolina : Motor
{
    // Comportamiento inventado de acelerar en un motorg gasolina.
    // Cambia las revoluciones en +2
    public override void Acelera()
    {
        Console.Write("inyectando gasolina para explosión");
        Revoluciones += 2;
    }
}
public class MotorElectrico : Motor
{
    // Comportamiento inventado de acelerar en un motorg gasolina.
    // Cambia las revoluciones en +2
    public override void Acelera()
    {
        Console.Write("aumentando potencia eléctrica");
        Revoluciones += 6;
    }
}
```

Definimos las propiedades que debería tener un neumático a través del interfaz `INeumaticos` y dos concreciones que lo implementan coma `value objects` por referencia.

> **Importante**
>
> Fíjate que hemos definido una **propiedad calculada** llamada `Descripcion` que nos devuelve una cadena con la descripción completa del neumático. Es decir, **es un método con cuerpo**.
>
> En muchos lenguajes de programación orientada a objetos, como es el caso de `C#`, **las interfaces pueden definir métodos con cuerpo**. Esto es una característica que no está en todos los lenguajes OO y que puede ser muy útil para evitar repetir código en las concreciones del interfaz. **Vendría a ser una implementación por defecto, que podría ser sobreescrita en las concreciones si fuera necesario**.
>
> **Al tener una cuerpo de defecto, no es obligatorio que las concreciones lo implementen**. De esta manera, como hemos comentado, evitamos repetir código en las concreciones.

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

La implementación de la clase `Coche` siguiendo el patrón **Strategy** y el diagrama UML anterior podría ser ...

```
public class Coche : IAcelerable
{
    public string Id { get; }
    public Motor Motor { get; }
    public INeumaticos Neumaticos { get; }

    public Coche(string id, Motor motor, INeumaticos neumaticos)
    {
        Id = id;
        Motor = motor;
        Neumaticos = neumaticos;
    }
    public void Acelera()
    {
        Console.Write($"Coche {Id} ");
        Motor.Acelera();
        Console.WriteLine($" a {Motor.Revoluciones} r.p.m.");
    }
    public override string ToString() =>
        $"Coche {Id} {Motor.GetType().Name} y neumaticos {Neumaticos.Descripcion}";
}
```

Podemos definir un sencillo programa principal de test ...

```
public class Principal
{
    public static void Main()
    {
        Coche c1 = new (id: "C1",
            motor: new MotorGasolina(),
            neumaticos: new NeumaticosNormal());
        Console.WriteLine(c1);
        c1.Acelera();
        c1.Acelera();
        Console.WriteLine();

        Coche c2 = new (id: "C2",
            motor: new MotorElectrico(),
            neumaticos: new NeumaticosSport());
        Console.WriteLine(c2);
        c2.Acelera();
        c2.Acelera();
    }
}
```

Mostrará por consola:

```
Coche C1 MotorGasolina y neumaticos 205/55 R16 88H
Coche C1 inyectando gasolina para explosión a 2 r.p.m.
Coche C1 inyectando gasolina para explosión a 4 r.p.m.

Coche C2 MotorElectrico y neumaticos 225/40 R18 92Y
Coche C2 aumentando potencia eléctrica a 6 r.p.m.
Coche C2 aumentando potencia eléctrica a 12 r.p.m.
```