



.NET

Teoría 7

Interfaces



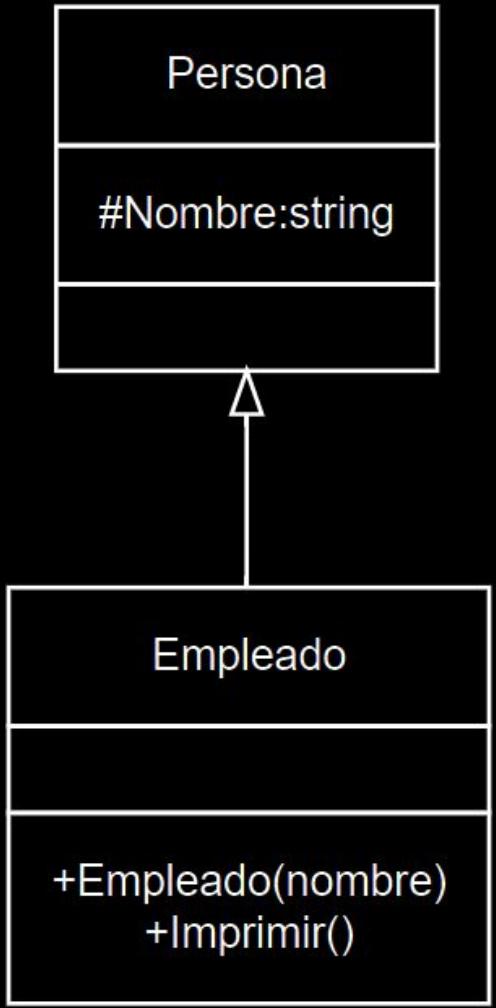
Vamos a presentar el concepto por medio de un ejemplo



1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `Teoria7`
4. Abrir `Visual Studio Code` sobre este proyecto



Codificar las clases Persona y Empleado



- La clase **Persona**, debe tener un campo protegido de tipo **string** llamado **Nombre**
- La clase **Empleado** debe derivar de **Persona** y contar con un **constructor** que reciba su nombre como parámetro y un método público **Imprimir()** para imprimirse en la consola

Interfaces - Presentación de caso

```
----- Persona.cs -----
```

```
namespace Teoria7;

class Persona
{
    protected string Nombre = "";
}
```

```
----- Empleado.cs -----
```

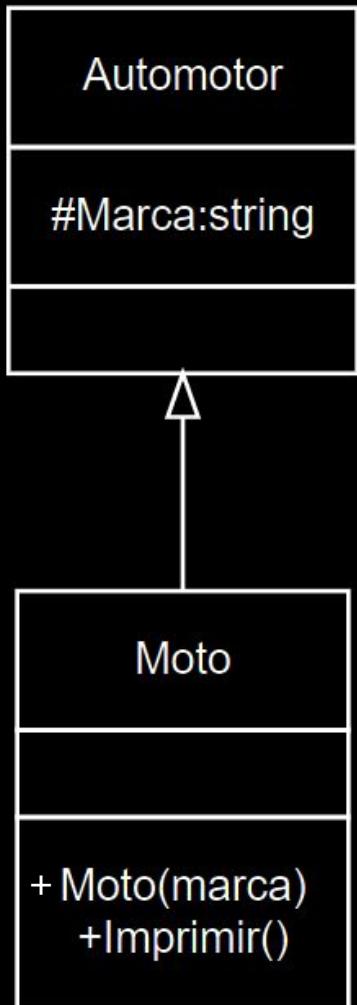
```
namespace Teoria7;

class Empleado : Persona
{
    public Empleado(string nombre)
        => Nombre = nombre;
    public void Imprimir()
        => Console.WriteLine($"Soy el empleado {Nombre}");
}
```

Código en el archivo
07_Teoria-Recursos.txt



Codificar las clases Automotor y Moto



- La clase **Automotor**, debe tener un campo protegido de tipo **string** llamado **Marca**
- La clase **Moto** debe derivar de **Automotor** y contar con un **constructor** que reciba su marca como parámetro y un método público **Imprimir()** para imprimirse en la consola

Interfaces - Presentación de caso

```
----- Automotor.cs -----
```

```
namespace Teoria7;

class Automotor
{
    protected string Marca = "";
}
```

```
----- Moto.cs -----
```

```
namespace Teoria7;

class Moto : Automotor
{
    public Moto(string marca)
        => Marca = marca;
    public void Imprimir()
        => Console.WriteLine($"Soy una moto {Marca}");
}
```

Código en el archivo
07_Teoria-Recursos.txt



Completar Program.cs invocando el método
Imprimir de todos los elementos del vector



```
using Teoria7;  
  
object[] vector = [  
    new Moto("Zanella"),  
    new Empleado("Juan"),  
    new Moto("Gilera")  
];
```

object es el
ancestro común
más cercano
entre Moto y
Empleado

```
foreach (object o in vector)  
{  
    . . .  
}
```

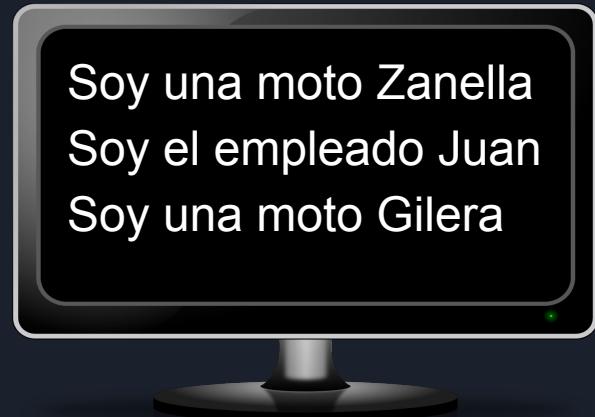
Completar

Possible solución

```
 . . .
foreach (object o in vector)
{
    if (o is Empleado e)
    {
        e.Imprimir();
    }
    else if (o is Moto m)
    {
        m.Imprimir();
    }
}
. . .
```

equivale a

```
if (o is Empleado)
{
    Empleado e = (o as Empleado);
    e.Imprimir();
}
```



Solución poco eficiente

```
...  
foreach (object o in vector)  
{  
    if (o is Empleado e)  
    {  
        e.Imprimir();  
    }  
    else if (o is Moto m)  
    {  
        m.Imprimir();  
    }  
}  
...
```

No hay
polimorfismo



Dificultad para usar polimorfismo

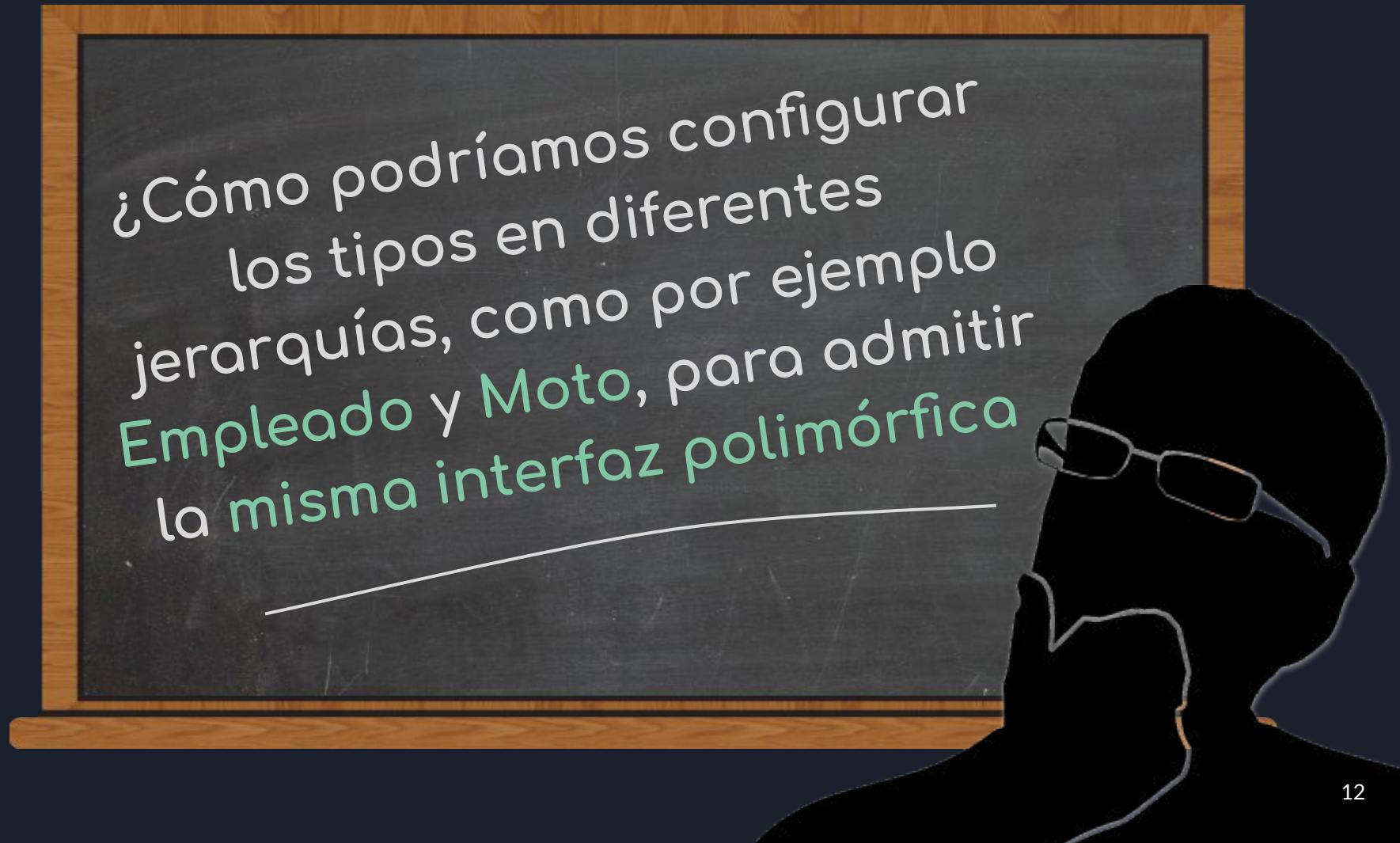
La interfaz polimórfica establecida por una clase base sólo es aprovechada por los tipos derivados

Sin embargo, en sistemas de software más grandes, es común desarrollar múltiples jerarquías de clases que no tienen un padre común más allá de System.Object.

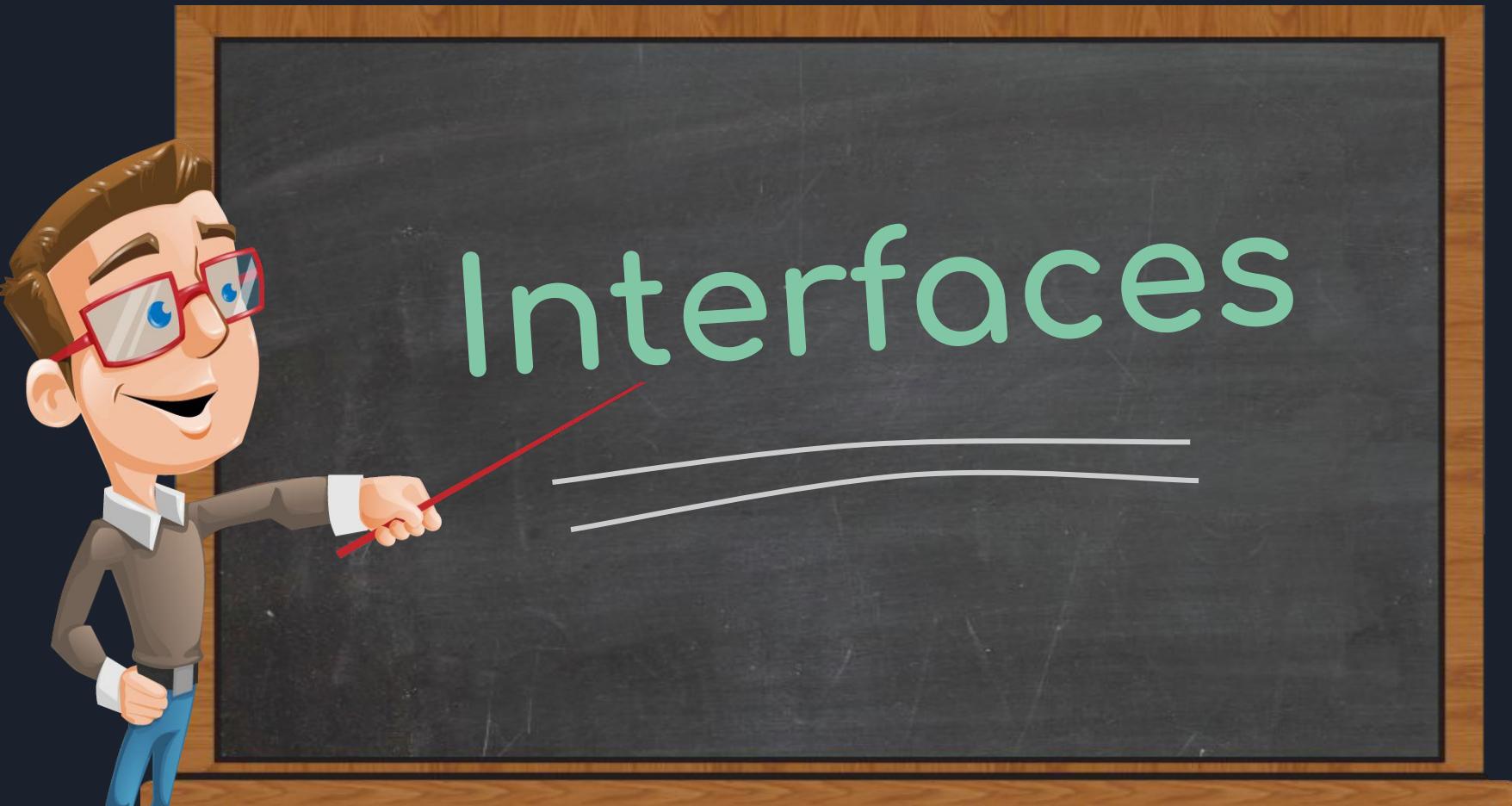


Interrogante

¿Cómo podríamos configurar los tipos en diferentes jerarquías, como por ejemplo Empleado y Moto, para admitir la misma interfaz polimórfica



Respuesta



¿Qué es una Interfaz ?

- Es un tipo de referencia que especifica un conjunto de funciones sin implementarlas.
- Pueden especificar métodos, propiedades, indizadores y eventos de instancia, sin implementación (o con implementación predeterminada a partir de c# 8.0).
- En lugar del código que los implementa llevan un punto y coma (;
- Por convención comienzan con la letra “I” (i latina mayúscula)

¿Qué es una Interfaz ?

- A partir de C# 8.0, una interfaz puede definir una implementación predeterminada de miembros.
- A partir de C# 8.0, una interfaz puede definir miembros estáticos (con implementación)
- Una interfaz no puede contener campos de instancia, constructores de instancia ni finalizadores

Declarando una interfaz

Los miembros de las interfaces **son públicos de manera predeterminada**. En versiones del lenguaje anteriores a C# 8.0 no se permite utilizar modificadores de acceso (ni siquiera **public**)

```
public interface IMiInterface  
{  
    public void UnMetodo();  
}
```



Atención, no usar modificadores si se está utilizando una versión anterior a C# 8.0

Implementación de interfaces

Las clases *derivan* de otras clases y opcionalmente *implementan* una o más interfaces

Implementación de interfaces

Si una clase implementa una interfaz debe implementar todos los miembros de la interfaz que no tienen implementación predeterminada.

Si una clase deriva de otra clase y además implementa algunas interfaces, la clase debe ser la primera en la lista después de los dos puntos

```
class Rombo : Figura, IImprimible, IGrandable
{
    ...
}
```

Clase base

Interfaces

Interfaces - Implementación de interfaces

```
interface IImprimible
{
    void Imprimir();
}
```

```
interface IGrandable
{
    void Agrandar(double factor);
    double TamañoMaximo { get; set; }
}
```

```
class Rombo : Figura, IImprimible, IGrandable
```

```
{  
    public void Imprimir()  
    { . . . }
```

Obligado a
implementarlo

```
    public void Agrandar(double factor)  
    { . . . }
```

```
    public double TamañoMaximo  
    {  
        get { . . . }  
        set { . . . }  
    }  
}
```

Interfaces - Implementación de interfaces

```
interface IImprimible
{
    void Imprimir();
}
```

```
interface IGrandable
{
    void Agrandar(double factor);
    double TamañoMaximo { get; set; }
}
```

```
class Rombo : Figura, IImprimible, IGrandable
{
```

```
    public void Imprimir()
    { . . . }
```

```
    public void Agrandar(double factor)
    { . . . }
```

```
    public double TamañoMaximo
    {
        get { . . . }
        set { . . . }
    }
}
```

Obligado a implementarlos

Utilización de interfaces

Es posible definir y utilizar variables de tipo interfaz.
Por ejemplo:

```
Rombo r1 = new Rombo();  
Figura r2 = new Rombo();  
IAgrandable r3 = new Rombo();  
IIImprimible r4 = new Rombo();
```

Las siguientes son sentencias son válidas

```
r3.TamañoMaximo = 100;  
r3.Agrandar(1.2);  
r4.Imprimir();  
(r3 as IIImprimible)?.Imprimir();
```

Utilización de interfaces

Las interfaces **son tipos de referencia**, por lo tanto es posible utilizar el operador **as** (ya lo vimos). Es habitual combinar su uso con el del operador **is** de la siguiente manera:

```
. . .
object o;
. . .
if (o is IImprimible)
{
    (o as IImprimible)?.Imprimir();
}
. . .
```

```
. . .
object o;
. . .
if (o is IImprimible imp)
{
    imp.Imprimir();
}
. . .
```

↑
facilidad
incorporada en la
versión 7.0 de C#

Utilización de interfaces

No es posible crear una instancia de una interface

```
IImprimible imp = new IImprimible();
```



No está permitido



Utilización de interfaces

Sí se puede hacer esto

```
IImprimible[] vector = new IImprimible[10];
```

Acá no instanciamos ningún
objeto **IImprimible**.

Los elementos que agreguemos al
vector (inicialmente todos null)
tendrán que implementar la
interface **IImprimible**.



Utilización de interfaces

También es posible utilizar tipos Interfaz para declarar propiedades, indizadores y métodos (argumentos y valor de retorno)

```
IImprimible Elemento {get;set;}  
IImprimible this [int index]...  
void EstablecerElemento(IImprimible e) ...  
IImprimible ObtenerElemento() ...
```



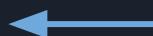


Resolver el ejercicio inicial de manera polimórfica



----- IImprimible.cs -----

```
namespace Teoria7;  
interface IImprimible  
{  
    void Imprimir();  
}
```



----- Moto.cs -----

```
class Moto : Automotor, IImprimible  
{  
    . . .  
}
```

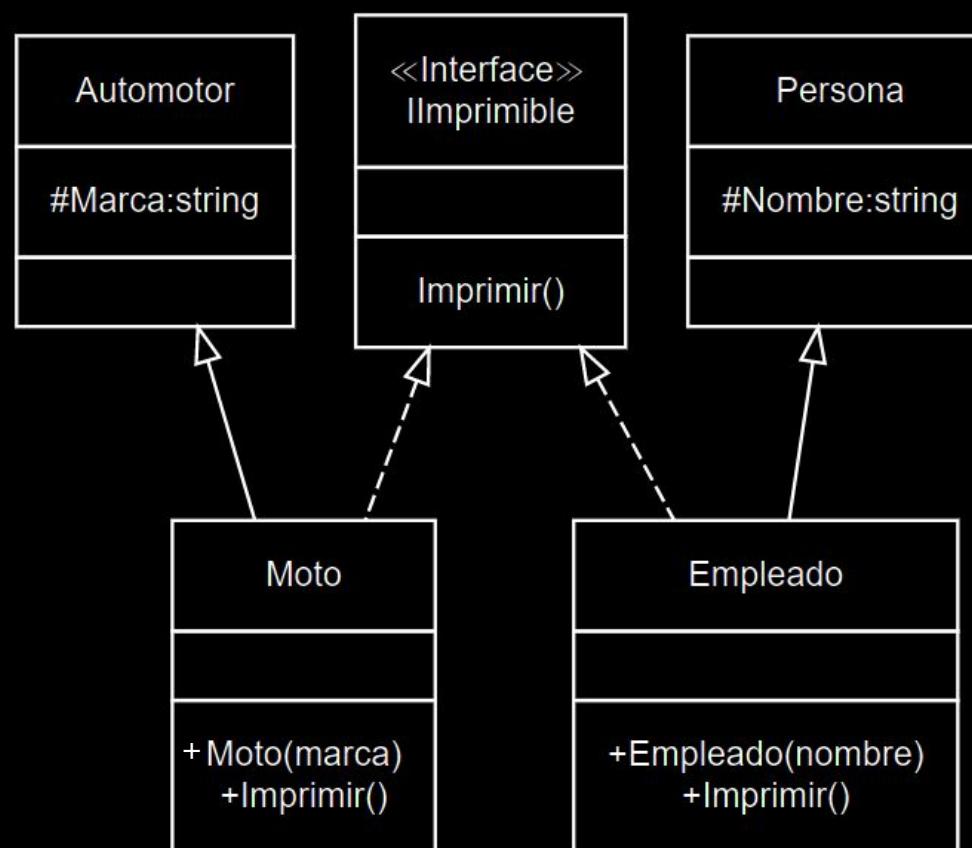


----- Empleado.cs -----

```
class Empleado : Persona, IImprimible  
{  
    . . .  
}
```



Resolución del ejercicio inicial utilizando interfaces





Codificar Program.cs para obtener una Solución polimórfica



```
using Teoria7;
```

```
object[] vector = [  
    new Moto("Zanella"),  
    new Empleado("Juan"),  
    new Moto("Gilera")  
];
```

```
foreach (IIImprimible imp in vector)  
{  
    imp.Imprimir();  
}
```



Acá hay una conversión de tipo de `object` a `IIImprimible`



Completar el cuerpo del for



```
using Teoria7;

object[] vector = [
    new Moto("Zanella"),
    new Empleado("Juan"),
    new Moto("Gilera")
];

for (int i = 0; i < vector.Length; i++)
{
    ...
}
```

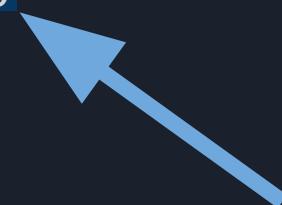
A large blue arrow points from the word "... " in the code towards the opening brace of the for loop, indicating where the body of the loop should be added.

Solución: Debemos convertir explícitamente de object a Iimprimible

```
using Teoria7;

object[] vector = [
    new Moto("Zanella"),
    new Empleado("Juan"),
    new Moto("Gilera")
];

for (int i = 0; i < vector.Length; i++)
{
    (vector[i] as Iimprimible)?.Imprimir();
}
```



Esta es una mejor solución

```
using Teoria7;

IImprimible[] vector = [
    new Moto("Zanella"),
    new Empleado("Juan"),
    new Moto("Gilera")
];

for (int i = 0; i < vector.Length; i++)
{
    vector[i].Imprimir();
}
```

Vector de elementos **IImprimible**
Esta solución es más segura, se aprovecha la verificación de tipo que hace el compilador

Funciona sin necesidad de conversión alguna porque los elementos son **IImprimible**

File System

File System Espacio de nombres System.IO

La BCL incluye todo un espacio de nombres llamado **System.IO** especialmente orientado al trabajo con archivos. Entre las clases más utilizadas de este espacio están:

- Path
- Directory
- DirectoryInfo
- File
- FileInfo

La clase Path

La clase `Path` incluye un conjunto de miembros estáticos diseñados para realizar cómodamente las operaciones más frecuentes relacionadas con rutas y nombres de archivos.

Con los campos públicos `VolumeSeparatorChar`, `DirectorySeparatorChar`, `AltDirectorySeparatorChar` y `PathSeparator`, se obtiene el carácter específico de la plataforma que se utiliza para separar unidades, carpetas y archivos, y el separador de múltiples rutas.

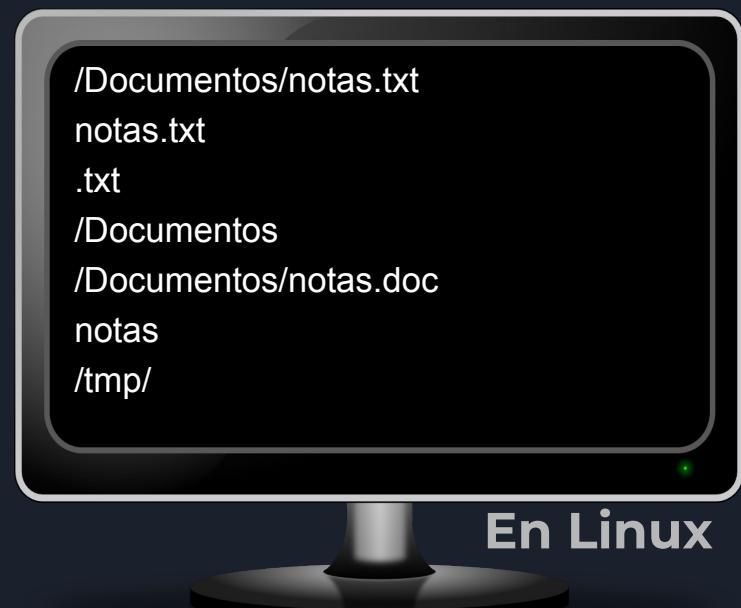
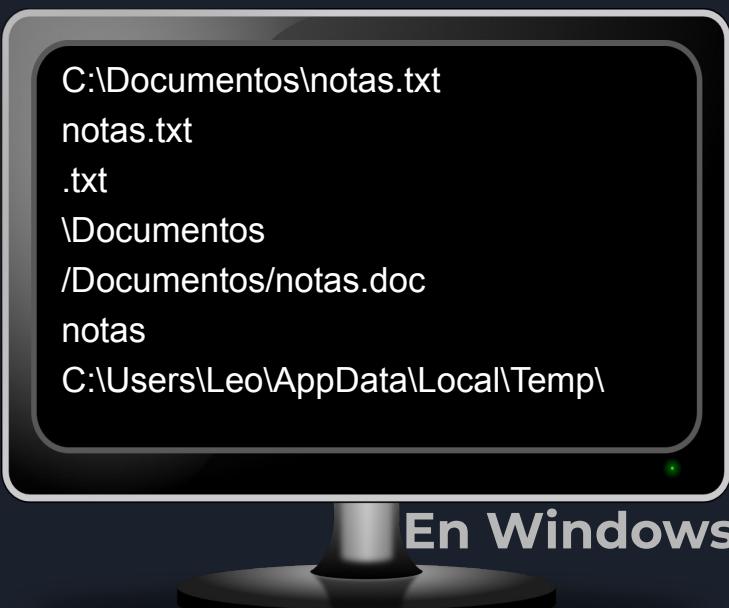
- Con Windows, estos caracteres son `:`, `\`, `/` y `;`
- Con Linux, estos caracteres son `/`, `/`, `/` y `:`

File System - System.IO - Path

Ejemplo:

```
string archivo = "/Documentos/notas.txt";
Console.WriteLine(Path.GetFullPath(archivo));
Console.WriteLine(Path.GetFileName(archivo));
Console.WriteLine(Path.GetExtension(archivo));
Console.WriteLine(Path.GetDirectoryName(archivo));
Console.WriteLine(Path.ChangeExtension(archivo, "doc"));
Console.WriteLine(Path.GetFileNameWithoutExtension(archivo));
Console.WriteLine(Path.GetTempPath());
```

Muchos métodos
sólo procesan
strings

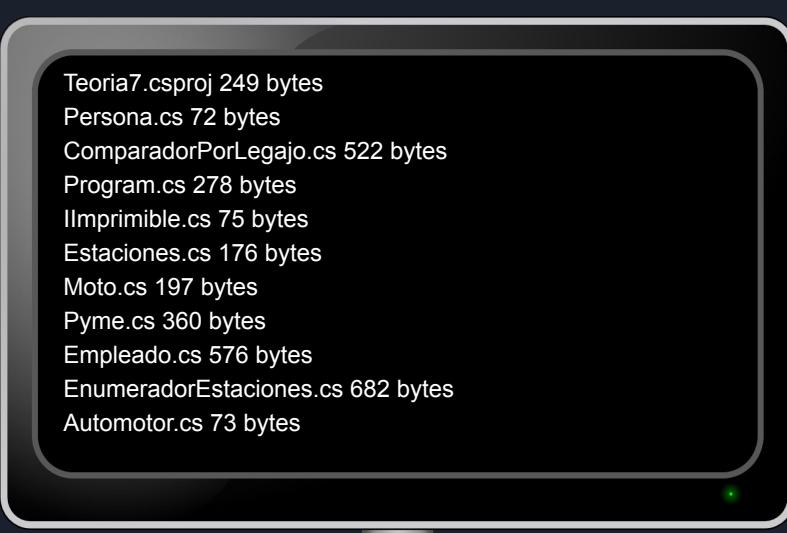


Las clases DirectoryInfo, FileInfo, Directory y File

- Para trabajar con archivos se utilizan objetos de la clase `FileInfo` y para trabajar con directorios objetos de la clase `y DirectoryInfo`.
- Las clases `File` y `Directory` que sólo tienen métodos estáticos (al igual que `Path`) son útiles para realizar tareas sencillas. No requieren la creación de ningún objeto pero son menos poderosas y menos eficientes.

DirectoryInfo y FileInfo ejemplo

```
string stDir = Environment.CurrentDirectory;
DirectoryInfo dirInfo = new DirectoryInfo(stDir);
FileInfo[] archivos = dirInfo.GetFiles();
foreach (FileInfo archivo in archivos)
{
    string st = $"{archivo.Name} {archivo.Length} bytes";
    Console.WriteLine(st);
}
```



The image shows a computer monitor with a dark frame and a light-colored base. On the screen, there is a list of file names and their sizes, enclosed in a rounded rectangle. The list includes:

- Teoria7.csproj 249 bytes
- Persona.cs 72 bytes
- ComparadorPorLegajo.cs 522 bytes
- Program.cs 278 bytes
- IlImprimible.cs 75 bytes
- Estaciones.cs 176 bytes
- Moto.cs 197 bytes
- Pyme.cs 360 bytes
- Empleado.cs 576 bytes
- EnumeradorEstaciones.cs 682 bytes
- Automotor.cs 73 bytes

Archivos de texto

El trabajo con archivos en .NET está ligado al concepto de **stream** o flujo de datos, que consiste en tratar su contenido como una **secuencia ordenada** de datos.

El concepto de **stream** es aplicable también a otros tipos de almacenes de información tales como **conexiones de red** o **buffers en memoria**.

La **BCL** proporciona las clases **StreamReader** y **StreamWriter**. Los objetos de estas clases facilitan la lectura y escritura de **archivos de textos**.

StreamReader

- Para facilitar la **lectura de flujos de texto** **StreamReader** ofrece una familia de métodos que permiten leer sus caracteres de diferentes formas:
- **De uno en uno:** El método **int Read()** devuelve el próximo carácter del flujo. Tras cada lectura la posición actual en el flujo se mueve un carácter hacia delante.

StreamReader

- Por líneas: El método `string ReadLine()` devuelve la siguiente línea del flujo (y avanza la posición en el flujo). Una línea de texto es cualquier secuencia de caracteres terminada en '`\n`', '`\r`' ó "`\r\n`", aunque la cadena que devuelve no incluye dichos caracteres.
- Por completo: `string ReadToEnd()`, que nos devuelve una cadena con todo el texto que hubiese desde la posición actual hasta el final (y avanza hasta el final del flujo).

StreamWriter

StreamWriter ofrece métodos que permiten:

- Escribir cadenas de texto: El método Write() escribe cualquier cadena de texto en el destino que tenga asociado. Pueden utilizarse formatos compuestos.
- Escribir líneas de texto: El método WriteLine() funciona igual que Write() pero añade un indicador de fin de línea. Pueden utilizarse formatos compuestos

StreamWriter

- Dado que el indicador de fin de línea depende de cada sistema operativo, **StreamWriter** dispone de una propiedad string **NewLine** mediante la que puede configurarse este indicador. Su valor por defecto es el “**\r\n**” en **Windows** y “**\n**” en **Linux**.

Ejemplo 1 - leyendo y escribiendo archivo de texto fuente en destino

```
StreamReader sr = new StreamReader("fuente.txt");
StreamWriter sw = new StreamWriter("destino.txt");
string? linea;
while (!sr.EndOfStream)
{
    linea = sr.ReadLine();
    sw.WriteLine(linea);
}
sr.Close(); sw.Close();
```



El método `close()` libera los recursos de manera explícita, invocando un método `Dispose()`

Ejemplo 2 - Manejando excepciones

```
StreamReader? sr = null;  
StreamWriter? sw = null;  
try  
{  
    sr = new StreamReader("fuente.txt");  
    sw = new StreamWriter("destino.txt");  
    sw.WriteLine(sr.ReadToEnd()); ←  
}  
catch (Exception e)  
{  
    Console.WriteLine(e.Message);  
}  
finally ←  
{  
    sr?.Dispose();  
    sw?.Dispose();  
}
```

Esta línea hace todo el trabajo

Es recomendable proveer manejo de excepciones y liberar los recursos en un bloque finally

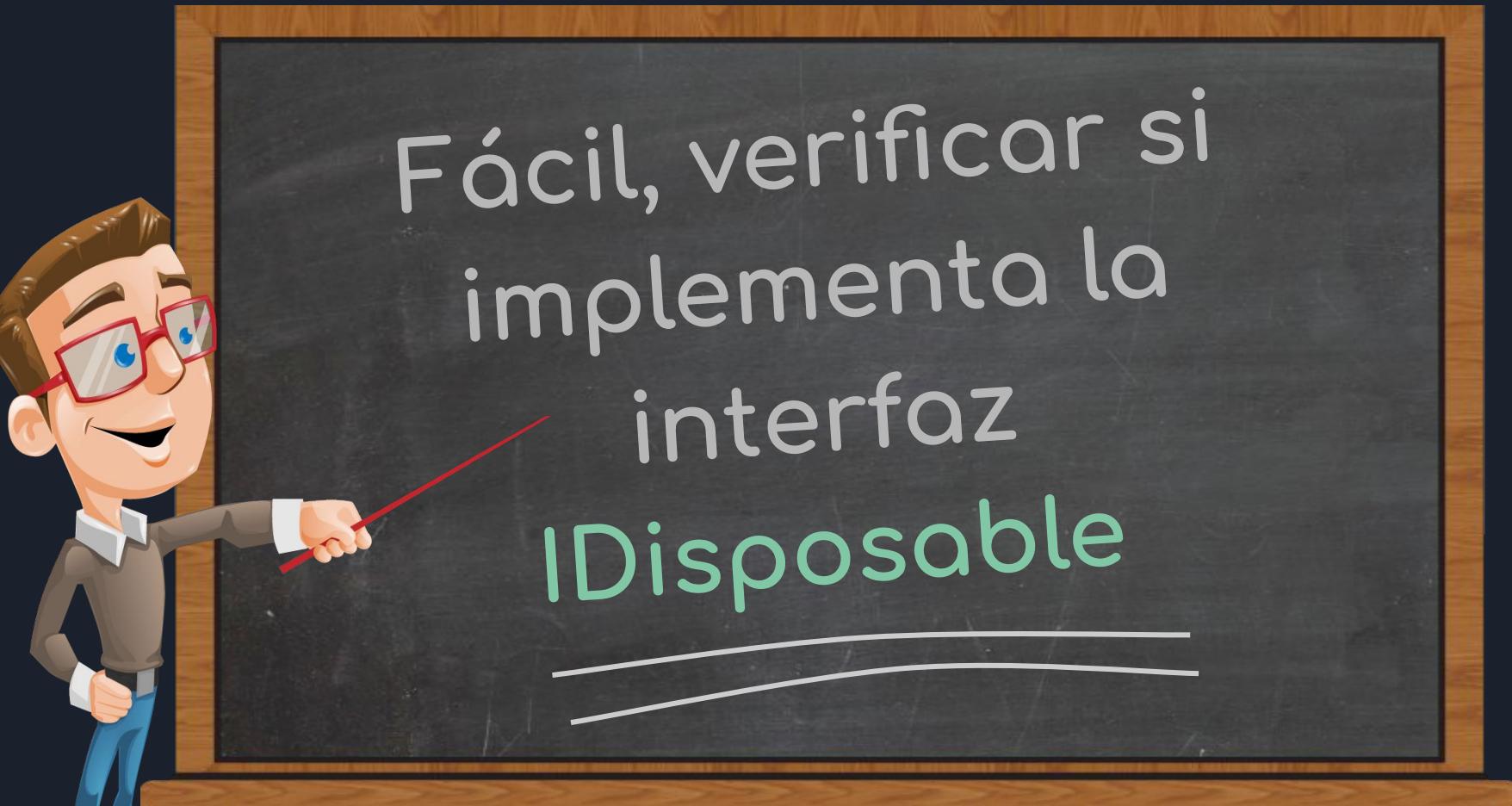
Se puede usar `Dispose()` en lugar de `Close()`

Interrogante

¿ Cómo podemos saber si es necesario liberar recursos explícitamente cuando dejamos de usar un objeto de la BCL en nuestro código ?



Respuesta



Interfaz IDisposable

- En C #, la alternativa recomendada al uso de finalizadores, es implementar la interfaz `System.IDisposable` que posee un único método.

```
public interface IDisposable
{
    void Dispose();
}
```

- `IDisposable` define un mecanismo **determinista** para liberar recursos no administrados y evita los problemas relacionados con el **recolector de basura** inherentes a los finalizadores.

Interfaz IDisposable

- Cuando se termina de usar un objeto que implementa `IDisposable`, se debe invocar el método `Dispose()` del objeto. Hay dos maneras de hacerlo:
 - Mediante un bloque `try/finally` como lo hicimos en el último ejemplo de manejo de archivos de texto.
 - Mediante la instrucción `using` (no es la directiva `using` que venimos usando para hacer referencia a los espacios de nombres)

Instrucción using

```
using(TipoDisposable recurso = new TipoDisposable(...))  
{  
    bloque de sentencias  
}
```

es equivalente a

```
TipoDisposable recurso = new TipoDisposable(...)  
try {  
    bloque de sentencias  
} finally {  
    if (recurso != null) recurso.Dispose();  
}
```

Cuidado!
No hay
bloque catch

Instrucción using

- En una instrucción using se pueden instanciar más de un objeto del mismo tipo, por ejemplo:

```
using (StreamReader f1 = new StreamReader("file1.txt"),
       f2 = new StreamReader("file2.txt"))
{
    ...
}
```

Instrucción using - Ejemplo

- Si se trata de distintos tipos los using se pueden anidar, como se observa en el siguiente ejemplo:

```
try
{
    using (StreamReader sr = new StreamReader("fuente.txt"))
    {
        using (StreamWriter sw = new StreamWriter("destino.txt"))
        {
            sw.WriteLine(sr.ReadToEnd());
        }
    }
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```



Sin embargo, C# 8.0
introdujo las
declaraciones using que
proveen una sintaxis
más limpia

Declaración using

- Una declaración `using` es una declaración de variable precedida de la palabra clave `using`. El compilador invocará el `Dispose()` sobre la variable declarada al final del ámbito de inclusión. El ejemplo de la diapositiva anterior puede escribirse de la siguiente manera:

```
try
{
    using StreamReader sr = new StreamReader("fuente.txt");
    using StreamWriter sw = new StreamWriter("destino.txt");
    sw.WriteLine(sr.ReadToEnd());
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

Principio de inversión de dependencias (DIP) y Patrón de Inyección de Dependencias

Principio de inversión de dependencias

Las aplicaciones web ASP.NET Core hacen uso intensivo del principio de inversión de dependencias.

Sin embargo este principio es muy poderoso y deberíamos aprovecharlo en cualquier tipo de aplicación

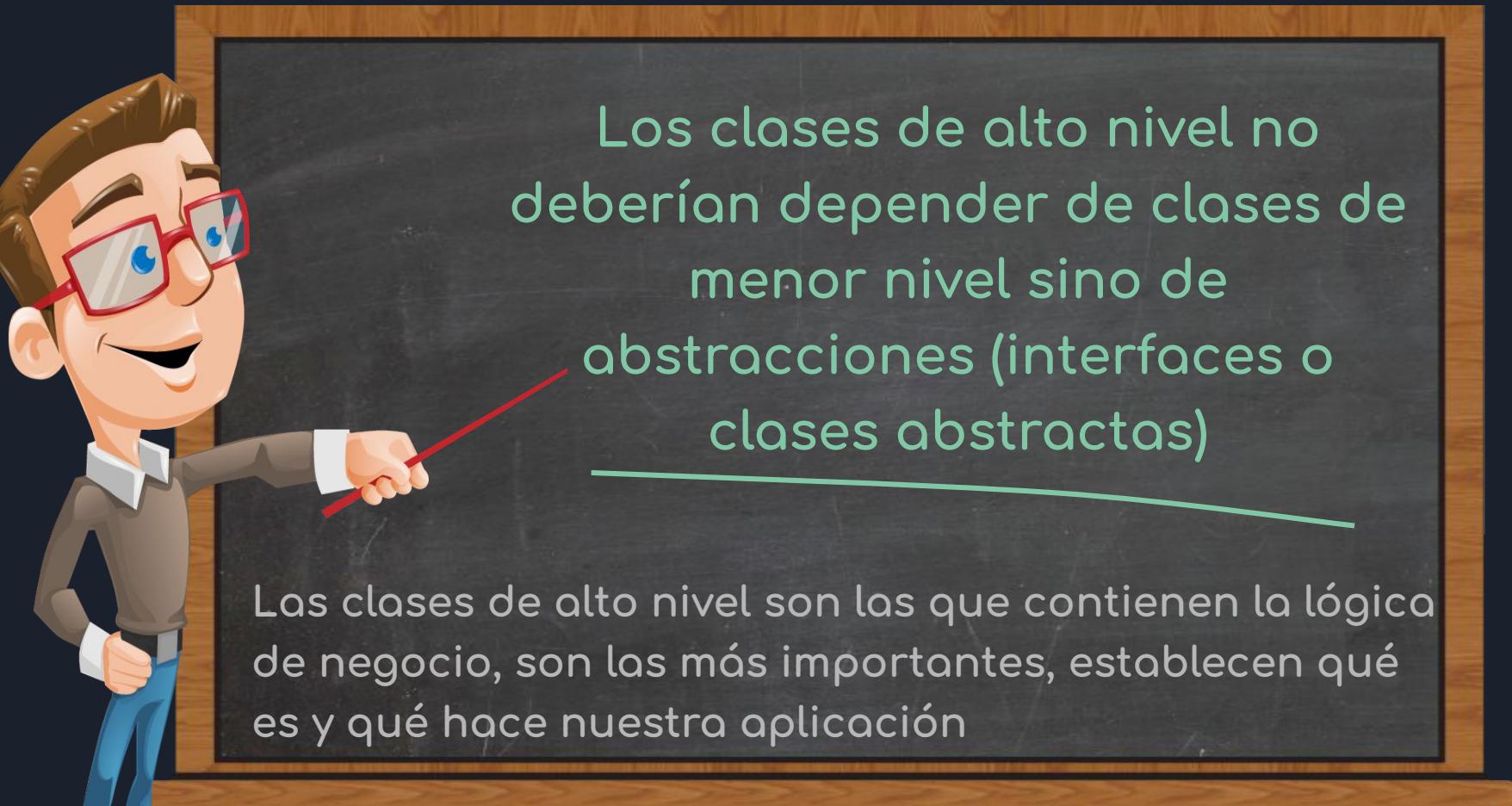


Principio de inversión de dependencias

- El Principio de inversión de dependencias (**DIP**) es uno de los 5 principios **SOLID**.
- El principio **Open/Close**, que vimos cuando introdujimos el tema del polimorfismo, también es uno de los principios **SOLID**.
- Los principios **SOLID** facilitan el **mantenimiento**, **extensión** y **reusabilidad** del código.

Principio de inversión de dependencias

Reinterpretación en POO

A cartoon illustration of a teacher with brown hair and red-rimmed glasses, wearing a brown sweater over a white collared shirt and blue jeans. He is standing on the left, pointing a red stick at a chalkboard with his right hand. The chalkboard has a wooden frame and contains text in Spanish. The background behind the chalkboard is dark.

Los clases de alto nivel no
deberían depender de clases de
menor nivel sino de
abstracciones (interfaces o
clases abstractas)

Las clases de alto nivel son las que contienen la lógica
de negocio, son las más importantes, establecen qué
es y qué hace nuestra aplicación

Principio de inversión de dependencias

- Vamos a mostrar por medio de código como podemos implementar el principio de inversión de dependencias utilizando el patrón de Inyección de Dependencias.
- Primero codificaremos una solución que no cumple DIP y luego la iremos transformando gradualmente hasta que lo cumpla.
- Se trata de una aplicación que procesa un cálculo simple y registra los eventos durante su ejecución (log).



Codificando una solución sin DIP



1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `CalculoSimple`
4. Abrir `Visual Studio Code` sobre este proyecto



Codificar la clase Logger



```
namespace CalculoSimple;  
public class Logger  
{  
    public void Log(string mensaje)  
    {  
        Console.WriteLine(mensaje);  
    }  
}
```



Codificar la clase Calculador



```
namespace CalculoSimple;  
class Calculador  
{  
    Logger _logger = new Logger();  
    public void Calcular(int n)  
    {  
        int resul = (n + 5) * (n + 7);  
        _logger.Log($"Fin de Calculo - (resul={resul})");  
    }  
}
```



Codificar Program.cs de la siguiente manera y ejecutar



-----Program.cs-----

```
using CalculoSimple;  
Calculador calc = new Calculador();  
calc.Calcular(3);
```

Principio de inversión de dependencias

```
-----Program.cs-----
```

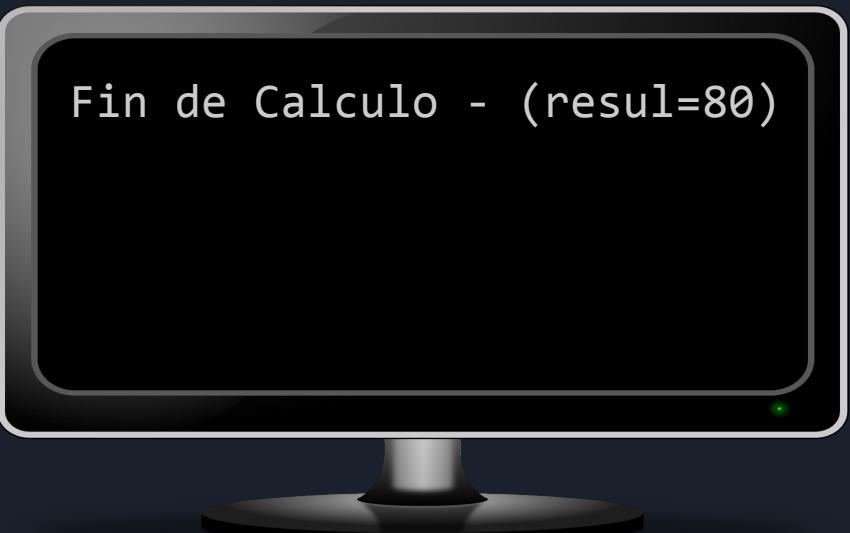
```
using CalculoSimple;
Calculador calc = new Calculador();
calc.Calcular(3);
```

```
-----Calculador.cs-----
```

```
namespace CalculoSimple;
class Calculador
{
    Logger _logger = new Logger();
    public void Calcular(int n)
    {
        int resul = (n + 5) * (n + 7);
        _logger.Log($"Fin de Calculo - (resul={resul})");
    }
}
```

```
-----Logger.cs-----
```

```
namespace CalculoSimple;
class Logger
{
    public void Log(string mensaje)
    {
        Console.WriteLine(mensaje);
    }
}
```



Fin de Calculo - (resul=80)

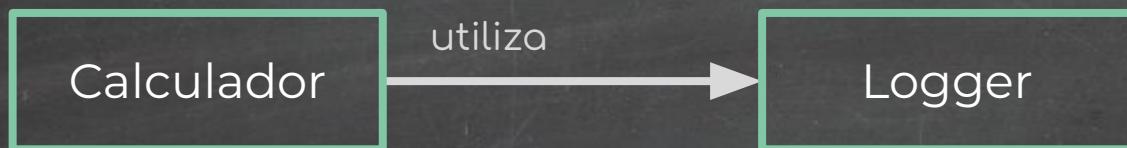
Relación entre las clases de la solución anterior

Un objeto **Calculador** usa a un objeto **Logger** invocando un método público de este último

Podemos considerar que la clase **Logger** provee un servicio y que la clase **Calculador** lo consume



Relación entre las clases de la solución anterior



- **Calculador** es cliente de **Logger**
- **Calculador** depende de **Logger**
- Pero **Calculador** es una clase de alto nivel (implementa lógica de negocio) y **Logger** es una clase de bajo nivel

“No se cumple con DIP”



Análisis de Dependencias

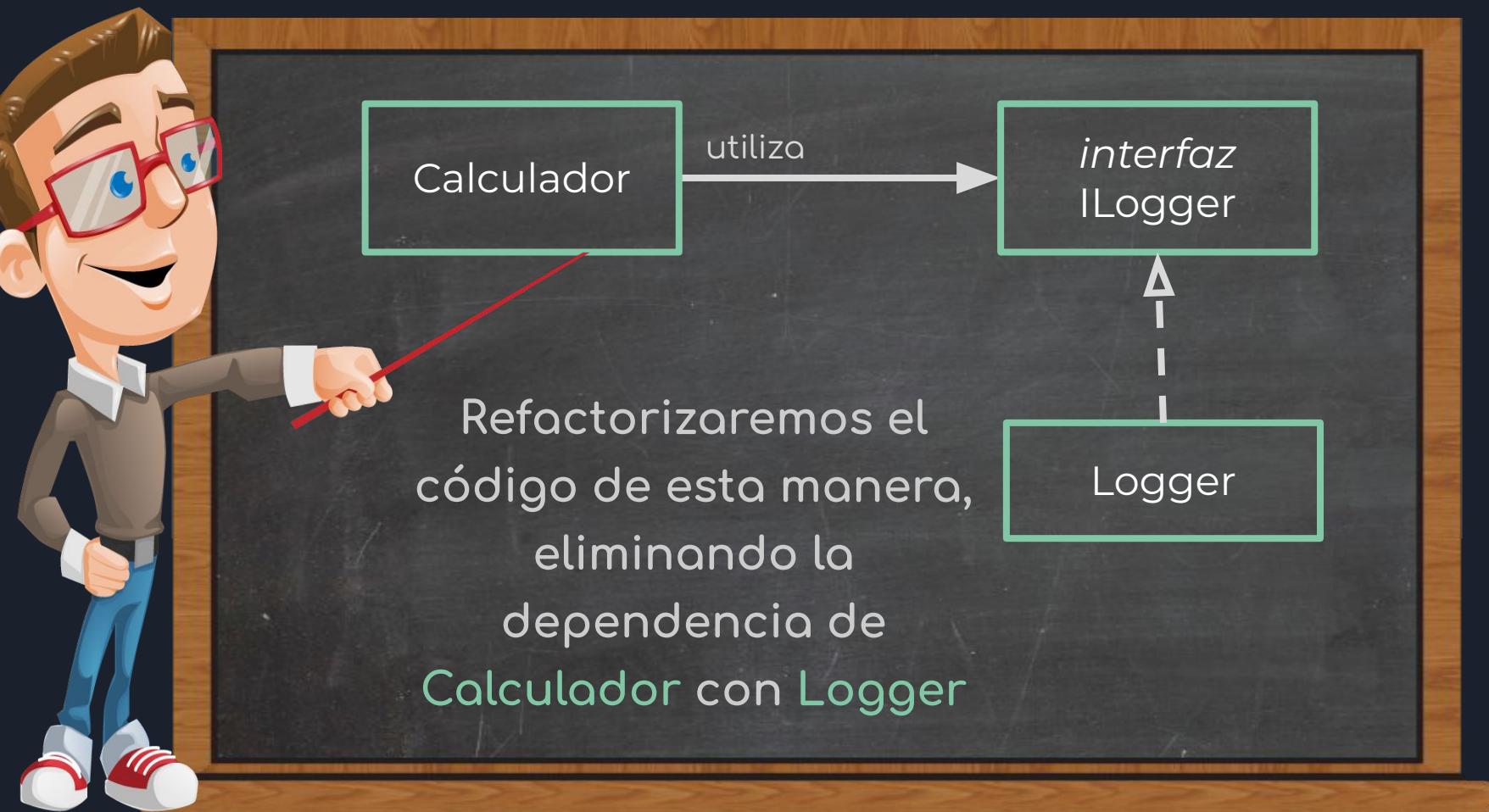
El análisis de las dependencias es estático, se hace sin correr el programa, analizando el código.

La mayoría de las clases que implementemos tendrán dependencias con tipos de la plataforma como `string`, `Array`, `List<T>`, `int`, `double`, `char`, etc. Esto no es problemático, esos tipos son muy estables y además no los modificamos nosotros

El análisis de dependencias que debemos realizar es entre los tipos definidos por nosotros



Refactorización de la solución anterior





Codificar la interfaz ILogger e indicar que la clase Logger implementa esta interfaz



-----ILogger.cs-----

```
namespace CalculoSimple;  
interface ILogger  
{  
    void Log(string mensaje);  
}
```

-----Logger.cs-----

```
namespace CalculoSimple;  
class Logger : ILogger  
{  
    public void Log(string mensaje)  
    {  
        Console.WriteLine(mensaje);  
    }  
}
```



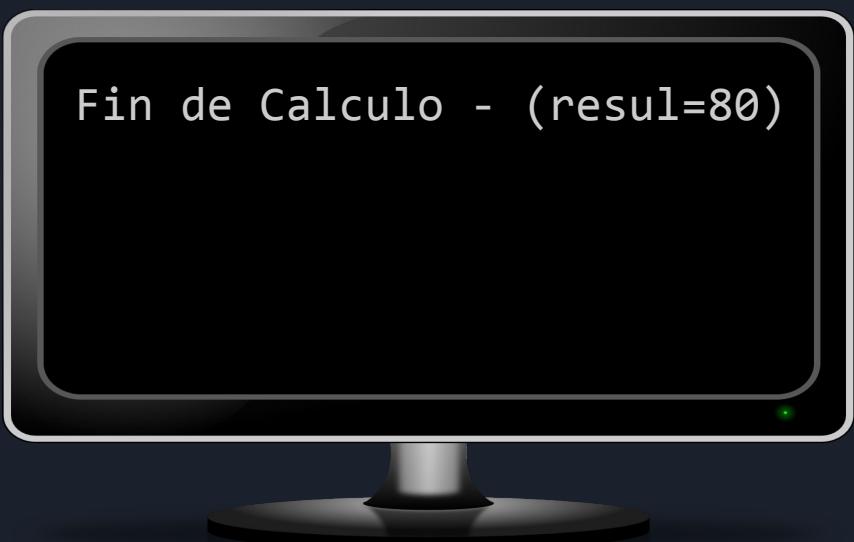
En la clase Calculador declarar la variable _logger de tipo ILogger y ejecutar



```
namespace CalculoSimple;  
class Calculador  
{  
    ILogger _logger = new Logger();  
    public void Calcular(int n)  
    {  
        int resul = (n + 5) * (n + 7);  
        _logger.Log($"Fin de Calculo - (resul={resul})");  
    }  
}
```

Principio de inversión de dependencias

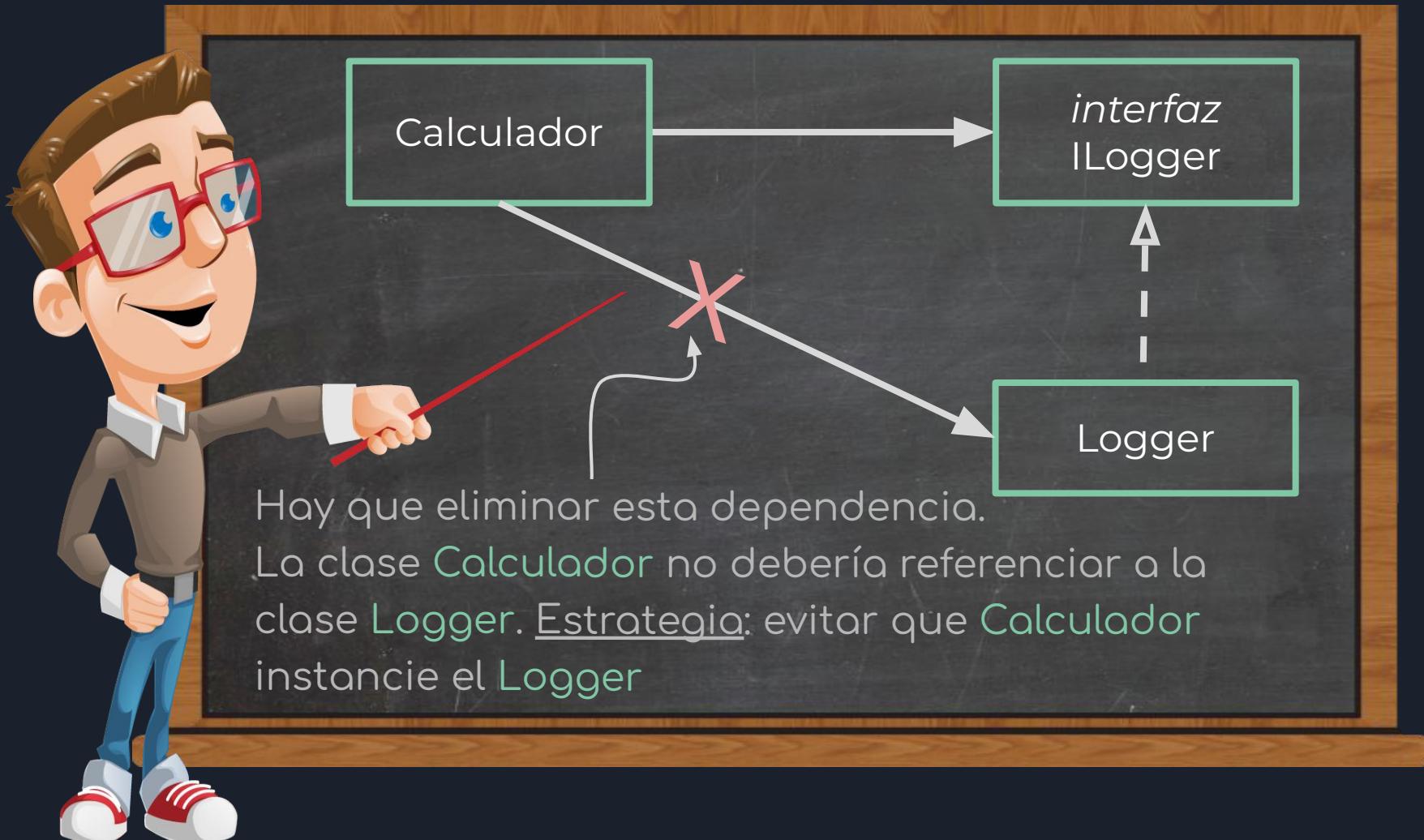
```
namespace CalculoSimple;  
class Calculador  
{  
    ILogger _logger = new Logger();  
    public void Calcular(int n)  
    {  
        int resul = (n + 5) * (n + 7);  
        _logger.Log($"Fin de Calculo - (resul={resul})");  
    }  
}
```



La aplicación está funcionando correctamente pero todavía no eliminamos la dependencia de Calculador con Logger

Al crear esta instancia se está referenciando directamente a la clase concreta Logger, por lo tanto Calculador ahora depende de ILogger y de Logger

Refactorización de la solución anterior



Principio de inversión de dependencias

- Para evitar que la clase `Calculador` cree una instancia de la clase `Logger` vamos a utilizar el patrón conocido como `Inyección de Dependencias`
- Otra clase debe hacerse responsable de crear la instancia requerida e injectarla en `Calculador`
- La inyección podría hacerse de varias maneras (por medio de `método`, `propiedad` o `constructor`)
- Vamos a utilizar la `inyección por constructor`



Modificar la clase Calculador (agregar constructor)



```
namespace CalculoSimple;  
class Calculador  
{  
    ILogger _logger;  
    public Calculador(ILogger logger)  
    {  
        _logger = logger;  
    }  
    public void Calcular(int n)  
    {  
        int resul = (n + 5) * (n + 7);  
        _logger.Log($"Fin de Calculo - (resul={resul})");  
    }  
}
```



Inyección de dependencia, en este caso una instancia de cualquier clase que implemente ILogger



Modificar el método Main de la clase Program
y ejecutar



```
using CalculoSimple;
```

```
ILogger logger = new Logger();
Calculador calc = new Calculador(logger);
calc.Calcular(3);
```



Inyección de
dependencia

Principio de inversión de dependencias

```
-----Program.cs-----  
using CalculoSimple;  
ILogger logger = new Logger();  
Calculador calc = new Calculador(logger);  
calc.Calcular(3);
```

```
-----Calculador.cs-----
```

```
namespace CalculoSimple;  
class Calculador  
{  
    ILogger _logger;  
    public Calculador(ILogger logger)  
    {  
        _logger = logger;  
    }  
    public void Calcular(int n)  
    {  
        int resul = (n + 5) * (n + 7);  
        _logger.Log($"Fin de Calculo - (resul={resul})");  
    }  
}
```

```
-----ILogger.cs-----
```

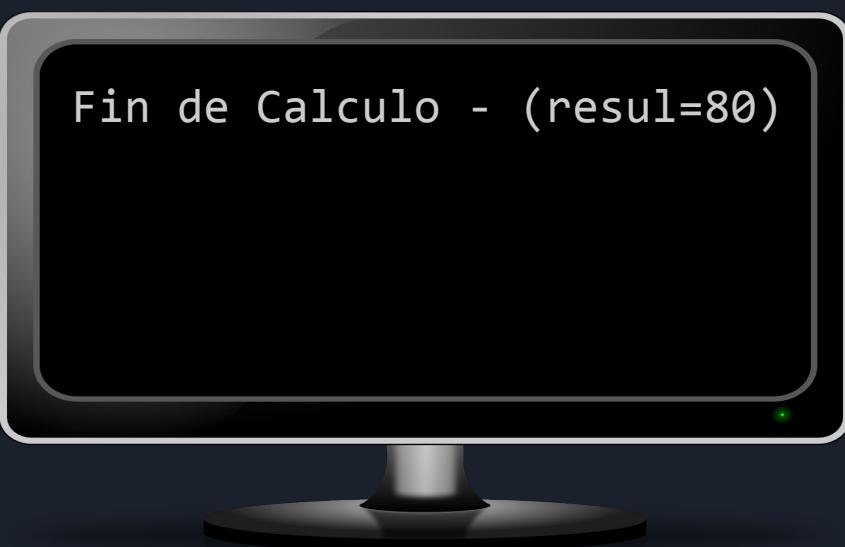
```
namespace CalculoSimple;  
public interface ILogger  
{  
    void Log(string mensaje);  
}
```

```
-----Logger.cs-----
```

```
namespace CalculoSimple;  
public class Logger : ILogger  
{  
    public void Log(string mensaje)  
    {  
        Console.WriteLine(mensaje);  
    }  
}
```

Hemos desacoplado completamente las clases **Calculador** y la clase **Logger**.

Ahora resulta fácil inyectar en **Calculador** una instancia de otra clase que implemente **ILogger**. **Calculador** no lo advertirá, seguirá funcionando convenientemente sin requerir ninguna modificación (**principio Open/Closed**)



Fin de Calculo - (resul=80)



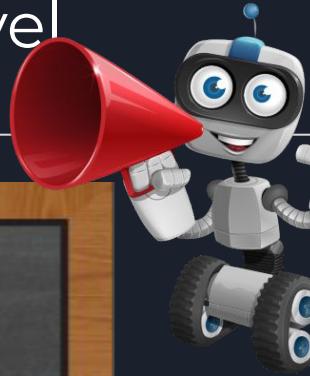
Modificar la clase Calculador usando constructor primario (válido a partir de C# 12)



```
namespace CalculoSimple;  
class Calculador(ILogger logger)  
{  
    public void Calcular(int n)  
    {  
        int resul = (n + 5) * (n + 7);  
        logger.Log($"Fin de Calculo - (resul={resul})");  
    }  
}
```

Al utilizar el constructor primario de `Calculador` se reduce la cantidad de líneas de código necesarias

La clase que contiene a Main es de bajo nivel



- La clase **Program** se considera una clase de muy **bajo nivel**.
- Es el **punto de entrada** inicial del sistema. Nada, salvo el sistema operativo, depende de ella. Por lo tanto se toleran las dependencias con otras clases de la aplicación sin afectar el principio de inversión de dependencias.
- En esta clase podemos establecer las configuraciones iniciales y luego entregar el control a módulos abstractos de alto nivel.

Principio de inversión de dependencias

- El Principio de inversión de dependencias favorece el acoplamiento débil en el código
- El acoplamiento es el grado de dependencia que existe entre los módulos (clases o estructuras en POO)
- El acoplamiento débil hace más fácil la modificación o reemplazo de un módulo por otro

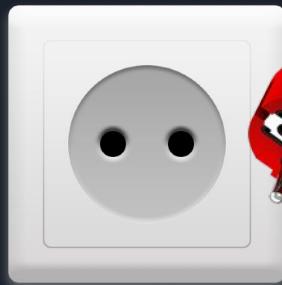
Diseño fuertemente acoplado Una muy mala idea



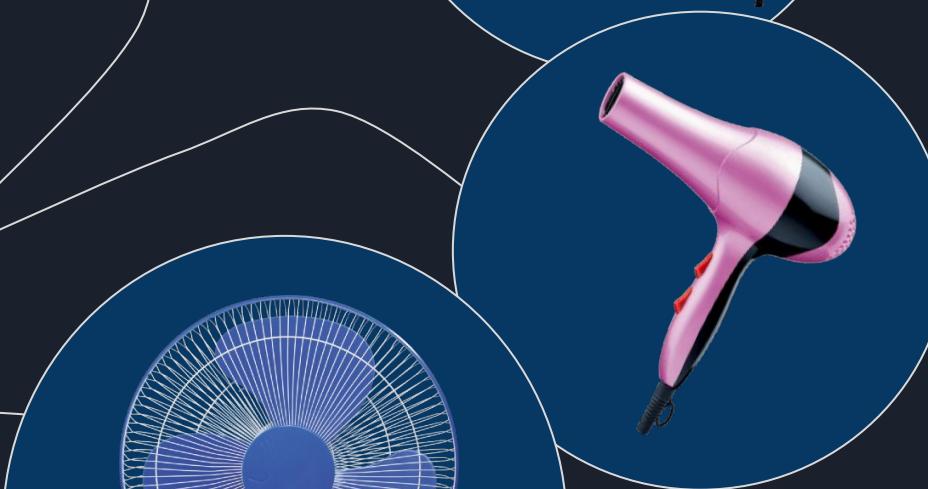
¿Y si queremos usar otra cosa distinta de este secador de pelo ?

Imagen extraída del libro “Dependency Injection in .NET” de Mark Seemann

Diseño débilmente acoplado



Con una interfaz adecuada resulta fácil reemplazar el secador por otro secador o algo distinto, sólo hace falta que posea la misma ficha para poder conectarlo al tomacorriente



Principio de inversión de dependencias

- Al igual que en una pared podemos conectar distintos dispositivos eléctricos por medio del tomacorriente (interfaz), a la clase `Calculador` podemos conectarle (inyectando) distintas clases que implementen la interfaz `ILogger`
- Por ejemplo podríamos querer que los mensajes de log se graben en un archivo en el disco (la consola no siempre está disponible, por ejemplo en una aplicación de escritorio `WinForm` o `WPF`)

Principio de inversión de dependencias

Alcanza con agregar la clase LoggerArchivo

```
class LoggerArchivo : ILogger
{
    public void Log(string mensaje)
    {
        using var sw = new StreamWriter("registro.log", true);
        sw.WriteLine($"{DateTime.Now} {mensaje}");
    }
}
```

Y configurar su uso adecuadamente en Programs.cs

```
using CalculoSimple;
ILogger logger = new LoggerArchivo();
Calculador calc = new Calculador(logger);
calc.Calcular(3);
```

Principio de inversión de dependencias

Alcanza con agregar la clase `Logger`

```
class LoggerArchivo : ILogger
{
    public void Log(string mensaje)
    {
        using var sw = new StreamWriter("log.txt");
        sw.WriteLine($"{DateTime.Now} - {mensaje}");
    }
}
```

Esta es la única
modificación de código
que se realizó
Todo lo demás es
código nuevo

Y configurar su uso adecuadamente en el código

```
using CalculoSimple;
	ILogger logger = new LoggerArchivo();
	Calculador calc = new Calculador(logger);
	calc.Calcular(3);
```



Contenedor de Inyección de Dependencias

Hemos cambiado el comportamiento del programa agregando nuevo código y modificando sólo el área donde se configuran los servicios

¡¡ Principio OPEN/CLOSE !!



Principio de inversión de dependencias

Aplicar DIP hace que el código sea mantenible. Los programas pequeños, como el anterior, son inherentemente mantenibles, por ello aplicar DIP en ejemplos simples tiende a parecer una ingeniería excesiva.

Cuanto mayor sea el tamaño del código, más visibles serán los beneficios de DIP



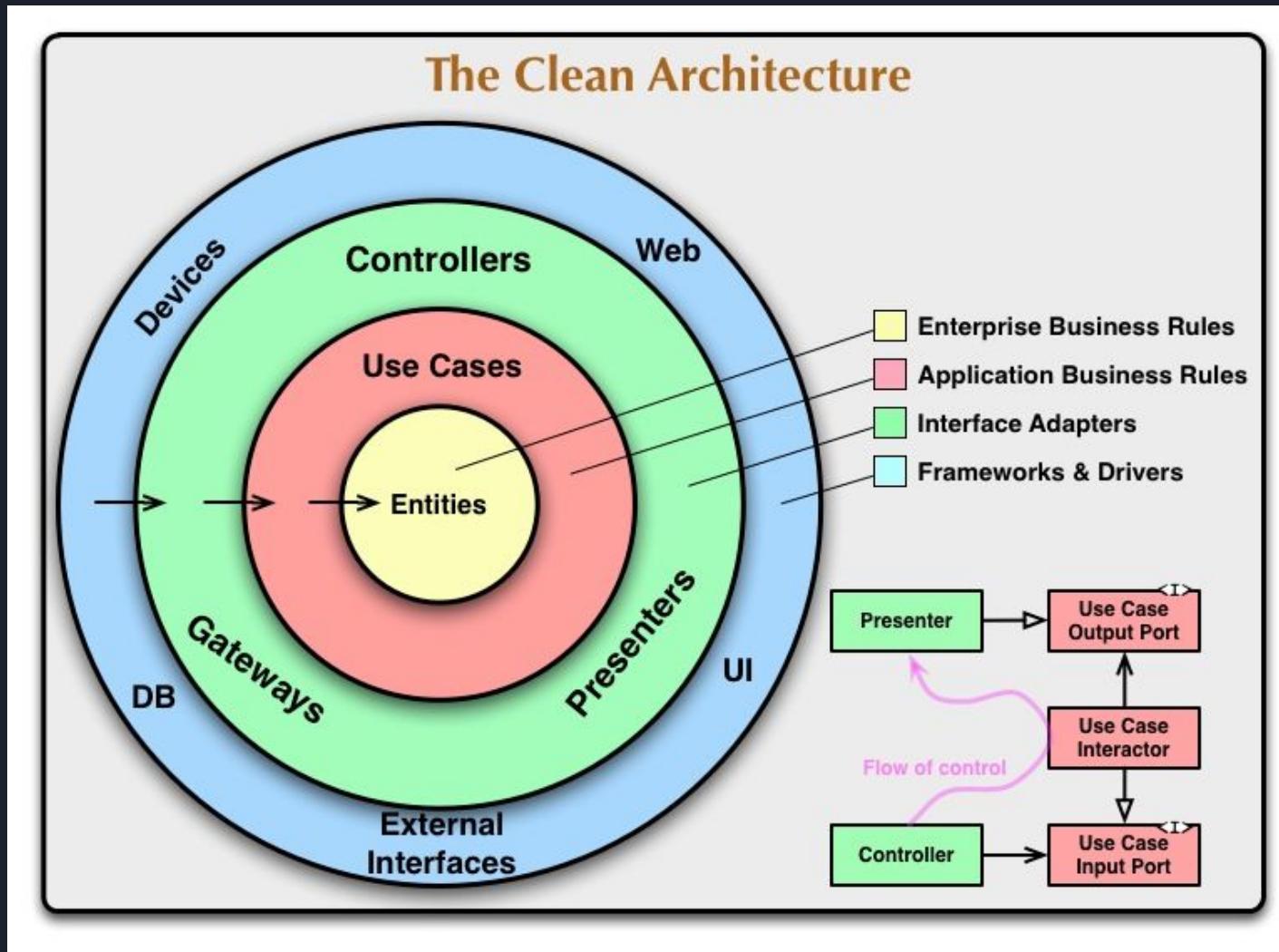
Arquitectura Limpia



Para que nuestras
aplicaciones puedan escalar
fácilmente

Arquitectura limpia

- Se refiere a organizar el proyecto para que sea fácil de entender y cambiar a medida que el proyecto crece.
- Basada en la separación de responsabilidades o intereses. Para ello divide el software en capas.
- En los últimos años han aparecido varias propuestas de arquitecturas limpias como DDD, Arquitectura Hexagonal y Arquitectura de Cebolla.
- En su blog, Robert C. Martin (el tío Bob) expone una idea general de una Arquitectura Limpia



<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

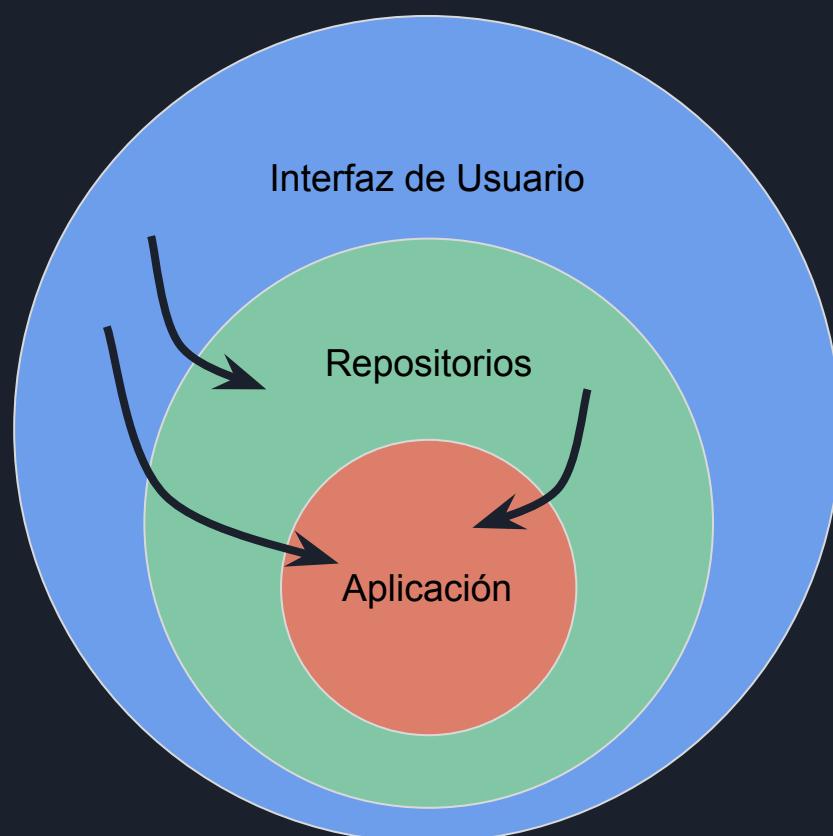
Separa los elementos de un diseño en niveles de anillo. Los anillos externos dependen de los internos y nunca al revés. El código en las capas internas no puede tener conocimiento de las funciones en las capas externas.

Arquitectura limpia

Para trabajar en este curso proponemos la siguiente versión simplificada

Aplicación y Repositorios
serán proyectos de
biblioteca de clases

Interfaz de Usuario será un
proyecto ejecutable (por ej.
una aplicación de consola
o Blazor)

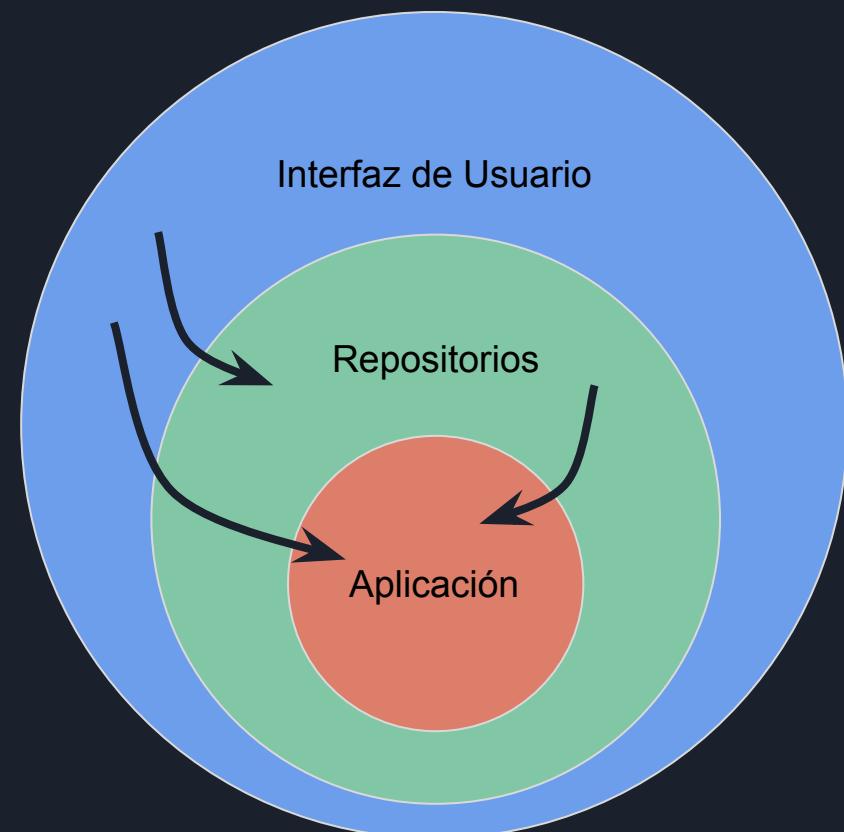


Arquitectura limpia

Interfaz de usuario hará referencia a los proyectos **Repositorios** y **Aplicación**

Repositorios (persistencia de los datos, definiremos al menos un repositorio por cada entidad que debamos persistir) hará referencia a **Aplicación**

Aplicación (lógica de la aplicación, entidades, casos de uso) no hará referencia a ningún otro proyecto





Ejemplo: Vamos a codificar una solución con Arquitectura Limpia



- Abrir una terminal del sistema operativo
- Cambiar a la carpeta `proyectosDotnet`
- Crear la solución `Almacen` con el siguiente comando:
`dotnet new sln -o Almacen`
- Cambiar a la carpeta `Almacen`
`cd Almacen`
- Crear los proyectos siguientes proyectos:
`dotnet new classlib -o Almacen.Aplicacion`
`dotnet new classlib -o Almacen.Repositorios`
`dotnet new console -o Almacen.Consola`

Interfaz de Usuario



Luego de ejecutar estos comando abrir Visual Studio Code (en la carpeta de la solución Almacen)



- Agregar los 3 proyectos a la solución

```
dotnet sln add Almacen.Aplicacion
```

```
dotnet sln add Almacen.Repositorios
```

```
dotnet sln add Almacen.Consola
```

- Establecer las referencias entre proyectos para que `Almacen.Consola` conozca a los otros 2 proyectos

```
dotnet add Almacen.Consola reference Almacen.Aplicacion
```

```
dotnet add Almacen.Consola reference Almacen.Repositorios
```

- Establecer la referencia para que `Almacen.Repositorios` conozca a `Almacen.Aplicacion`

```
dotnet add Almacen.Repositorios reference Almacen.Aplicacion
```

Ejemplo de solución con Arquitectura limpia

- Vamos a crear una entidad `Producto` y dos casos de uso `AgregarProducto` y `ListarProductos` (los casos de uso también los vamos a modelar con clases)
- Vamos a crear una clase `RepositorioProductoTXT` para manejar la persistencia de los productos en un archivo de texto plano
- Vamos a usar inversión de dependencias, los casos de uso no van a depender `RepositorioProductoTXT` sino de una interfaz `IRepositorioProducto`



En el proyecto Almacen.Aplicacion crear la clase Producto



```
namespace Almacen.Aplicacion;

public class Producto
{
    public int Id { get; set; }
    public string Nombre { get; set; } = "";
    public double Precio { get; set; }
    public override string ToString()
    {
        return $"{Nombre} ${Precio} (Id:{Id})";
    }
}
```



En el proyecto Almacen.Aplicacion crear la interfaz IRepositoryProducto



```
namespace Almacen.Aplicacion;  
  
public interface IRepositoryProducto  
{  
    void AgregarProducto(Producto producto);  
    List<Producto> ListarProductos();  
}
```

En el proyecto Almacen.Aplicacion crear la clase AgregarProductoUseCase

```
namespace Almacen.Aplicacion;

public class AgregarProductoUseCase(IRepositoryProducto repo)
{
    public void Ejecutar(Producto producto)
    {
        repo.AgregarProducto(producto);
    }
}
```

Inyección de dependencias!

En el proyecto Almacen.Aplicacion crear la clase ListarProductoUseCase

```
namespace Almacen.Aplicacion;

public class ListarProductosUseCase(IRepositoryProducto repo)
{
    public List<Producto> Ejecutar()
    {
        return repo.ListarProductos();
    }
}
```

Inyección de dependencias!



En el proyecto Almacen.Repositorios crear la clase RepositorioProductoTXT



```
namespace Almacen.Repositorios;
using Almacen.Aplicacion;
public class RepositorioProductoTXT : IRepositoryProducto
{
    readonly string _nombreArch = "productos.txt";
    public void AgregarProducto(Producto producto)
    {
        using var sw = new StreamWriter(_nombreArch, true);
        sw.WriteLine(producto.Id);
        sw.WriteLine(producto.Nombre);
        sw.WriteLine(producto.Precio);
    }
    public List<Producto> ListarProductos()
    {
        var resultado = new List<Producto>();
        using var sr = new StreamReader(_nombreArch);
        while (!sr.EndOfStream)
        {
            var producto = new Producto();
            producto.Id = int.Parse(sr.ReadLine() ?? "");
            producto.Nombre = sr.ReadLine() ?? "";
            producto.Precio = int.Parse(sr.ReadLine() ?? "");
            resultado.Add(producto);
        }
        return resultado;
    }
}
```

Código en el archivo
07 Teoria-Recursos.txt

Hemos implementado un repositorio concreto



Acabamos de implementar un repositorio concreto que persiste los productos en un archivo de texto.

Sin embargo, gracias a la inversión de dependencia, `Almacen.Aplicacion` no depende de este repositorio concreto sino de una abstracción (una interfaz)

En el futuro podremos cambiar este repositorio concreto fácilmente (por otro que acceda a una base de datos, por ejemplo) sin necesidad de modificar el código de `Almacen.Aplicacion`

Codificar Program.cs del proyecto Aplicacion.Consola de esta manera y ejecutar

```
using Almacen.Aplicacion;
using Almacen.Repositorios;

//configuro las dependencias
IRepositorioProducto repo = new RepositorioProductoTXT();

//creo los casos de uso
var agregarProducto = new AgregarProductoUseCase(repo);
var listarProducto = new ListarProductosUseCase(repo);

//ejecuto los casos de uso
agregarProducto.Ejecutar(new Producto() { Id = 1,Nombre="Yerba",Precio=1000});
agregarProducto.Ejecutar(new Producto() { Id = 2,Nombre="Azúcar",Precio=500});
var lista = listarProducto.Ejecutar();

foreach(Producto p in lista)
{
    Console.WriteLine(p);
}
```

Código en el archivo
07 Teoria-Recursos.txt

Salida por la consola

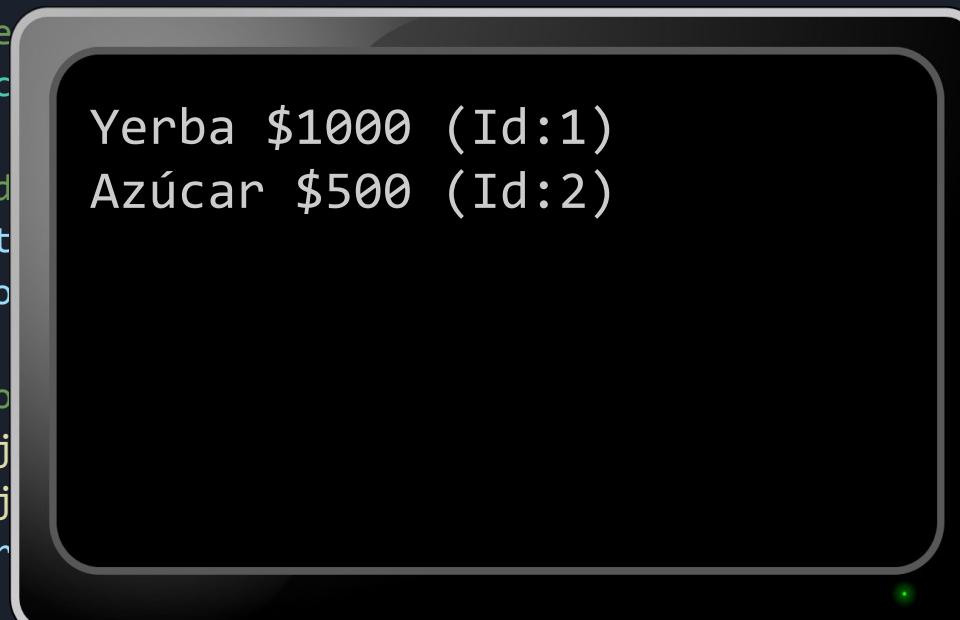
```
using Almacen.Aplicacion;
using Almacen.Repositorios;

//configuro las dependencias
IRepositorioProducto repositorioProducto = new RepositorioProducto();
var agregarProducto = new AgregarProducto(radiculo);
var listarProductos = new ListarProductos(radiculo);

//creo los casos de prueba
var producto1 = new Producto("Yerba", 1000);
var producto2 = new Producto("Azúcar", 500);

agregarProducto.Ejecutar(producto1);
agregarProducto.Ejecutar(producto2);
var lista = listarProductos.Ejecutar();

foreach(Producto p in lista)
{
    Console.WriteLine(p);
}
```



The monitor displays the following text:
Yerba \$1000 (Id:1)
Azúcar \$500 (Id:2)

Hemos implementado una interfaz de usuario



Acabamos de implementar una interfaz de usuario de consola.

Sin embargo, ni `Almacen.Aplicacion` ni `Almacen.Repositorios` depende de la interfaz de usuario.

En el futuro podremos cambiar esta interfaz de usuario fácilmente (por una interfaz web, por ejemplo), sin necesidad de modificar el código de `Almacen.Aplicacion` ni de `Almacen.Repositorios`.

Si volvemos a ejecutar notaremos que no controlamos que los Ids sean únicos

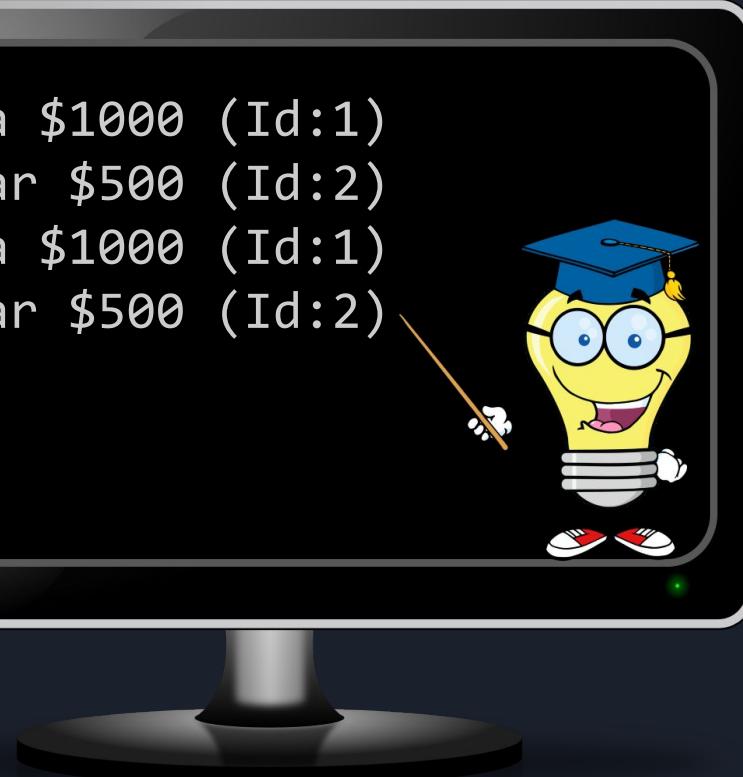
```
using Almacen.Aplicacion;
using Almacen.Repositorios;

//configuro las dependencias
IRepositorioProducto repositorioProducto = new RepositorioProducto();
var agregarProducto = new AgregarProducto(radiculo);
var listarProducto = new ListarProducto();

//creo los casos de prueba
var producto1 = new Producto("Yerba", 1000);
var producto2 = new Producto("Azúcar", 500);

agregarProducto.Ejecutar(producto1);
agregarProducto.Ejecutar(producto2);
var lista = listarProducto.Ejecutar();

foreach(Producto p in lista)
{
    Console.WriteLine(p);
}
```



The image shows a computer monitor displaying a list of products. The monitor is a dark grey rectangle with a thin black border. To its right is a cartoon character of a lightbulb with a face, wearing a blue graduation cap and holding a wooden pointer stick. The character has large, expressive eyes and a wide smile. The background behind the monitor is dark.

Nombre	Precio	Id
Yerba	\$1000	Id:1
Azúcar	\$500	Id:2
Yerba	\$1000	Id:1
Azúcar	\$500	Id:2

```
    var producto1 = new Producto("Yerba", 1000);
    var producto2 = new Producto("Azúcar", 500);

    agregarProducto.Ejecutar(producto1);
    agregarProducto.Ejecutar(producto2);
    var lista = listarProducto.Ejecutar();

    foreach(Producto p in lista)
    {
        Console.WriteLine(p);
    }
```



Tarea complementaria extra clase

- Hacer que el `Id` del producto sea establecido por `RepositorioProductoTXT` al momento de agregar el producto, garantizando que sea único, asignando los `Id` de manera incremental.
- Completar `IRepositorioProducto` y `RepositorioProductoTXT` con la siguiente funcionalidad (también agregar los use cases necesarios):

```
Producto? GetProducto(int id);  
void ModificarProducto(Producto producto);  
void EliminarProducto(int id);
```

- Usar `try/catch` para prevenir excepciones en `RepositorioProductoTXT`

Haciendo validaciones



¿Cómo podríamos validar que el producto cumpla con ciertas restricciones en el caso de uso AgregarProductoUseCase?

```
public class AgregarProductoUseCase(IRepositoryProducto repo)
{
    public void Ejecutar(Producto producto)
    {
        repo.AgregarProducto(producto); ← Queremos validar
    }                                         antes de invocar
}                                         esta sentencia
```

Además, queremos desacoplar el código de la validación en otro objeto que también vamos a injectarle AgregarProductoUseCase

En el proyecto Almacen.Aplicacion crear la clase ProductoValidador

```
namespace Almacen.Aplicacion;  
public class ProductoValidador  
{  
    public bool Validar(Producto producto, out string mensajeError)  
    {  
        mensajeError = "";  
        if (string.IsNullOrWhiteSpace(producto.Nombre))  
        {  
            mensajeError = "Nombre del producto inválido.\n";  
        }  
        if (producto.Precio <= 0)  
        {  
            mensajeError += "El Precio debe ser un valor mayor que cero.\n";  
        }  
        return (mensajeError == "");  
    }  
}
```

Código en el archivo
07 Teoria-Recursos.txt



En el proyecto Almacen.Aplicacion modificar la clase AgregarProductoUseCase



```
namespace Almacen.Aplicacion;  
public class AgregarProductoUseCase(IRepositoryProducto repo,  
                                         ProductoValidador validador)  
{  
    public void Ejecutar(Producto producto)  
    {  
        if (!validador.Validar(producto, out string mensajeError))  
        {  
            throw new Exception(mensajeError);  
        }  
        repo.AgregarProducto(producto);  
    }  
}
```



Probar este código en Codificar Program.cs del proyecto Aplicacion.Consola



```
using Almacen.Aplicacion;
using Almacen.Repositorios;

//creo el caso de uso
var agregarProducto = new AgregarProductoUseCase(new RepositorioProductoTXT(),
                                                     new ProductoValidador());

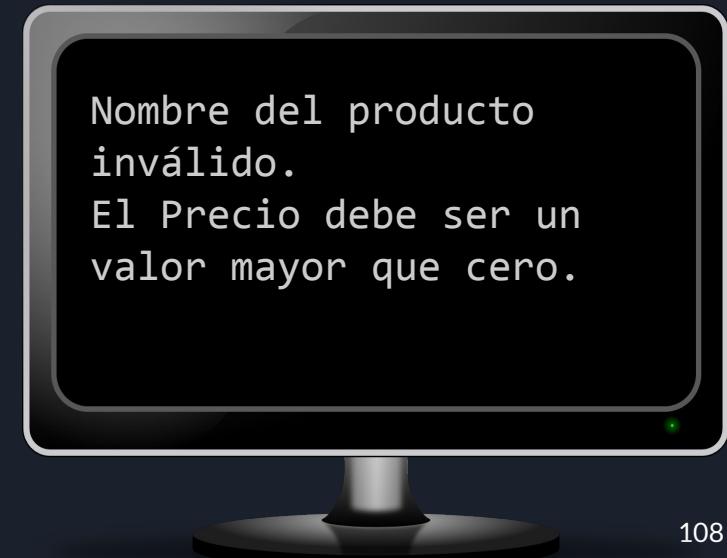
try
{
    agregarProducto.Ejecutar(new Producto());
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

Salida por la consola

```
using Almacen.Aplicacion;
using Almacen.Repositorios;

//creo el caso de uso
var agregarProducto = new AgregarProductoUseCase(new RepositorioProductoTXT(),
                                                    new ProductoValidador());

try
{
    agregarProducto.Ejecutar(new Producto());
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```



Fin Teoría 7

Práctica sobre la teoría 7

Práctica sobre la teoría 7

1) Codificar las clases e interfaces necesarias para modelar un sistema que trabaja con las siguientes entidades: Autos, Libros, Películas, Personas y Perros. Algunas de estas entidades pueden ser: alquilables (pueden ser alquiladas y devueltas por una persona), vendibles (pueden ser vendidas a una persona), lavables (se pueden lavar y secar) reciclables (se pueden reciclar) y atendibles (se pueden atender). A continuación se describen estas relaciones:

- Son Alquilables: Libros y Películas
- Son Vendibles: Autos y Perros
- Son Lavables: Autos
- Son Reciclables: Libros y Autos
- Son Atendibles: Personas y Perros

Completar el código de la clase estática Procesador:

```
static class Procesador
{
    public static void Alquilar(IAlquilable x, Persona p) => x.SeAlquilaA(p);
    public static . . .
    . . .
}
```

Para que el siguiente código produzca la salida indicada:

Práctica sobre la teoría 7

```
Auto auto = new Auto();
Libro libro = new Libro();
Persona persona = new Persona();
Perro perro = new Perro();
Pelicula pelicula = new Pelicula();
Procesador.Alquilar(pelicula, persona);
Procesador.Alquilar(libro, persona);
Procesador.Atender(persona);
Procesador.Atender(perro);
Procesador.Devolver(pelicula, persona);
Procesador.Devolver(libro, persona);
Procesador.Lavar(auto);
Procesador.Reciclar(libro);
Procesador.Reciclar(auto);
Procesador.Secar(auto);
Procesador.Vender(auto, persona);
Procesador.Vender(perro, persona);
```

Salida por consola

```
Alquilando película a persona
Alquilando libro a persona
Atendiendo persona
Atendiendo perro
Película devuelta por persona
Libro devuelto por persona
Lavando auto
Reciclando libro
Reciclando auto
Secando auto
Vendiendo auto a persona
Vendiendo perro a persona
```

- 2) Incorporar al ejercicio anterior la posibilidad también de lavar a los perros. También se debe incorporar una clase derivada de Película, las “películas clásicas” que además de alquilarse pueden venderse. Estos cambios deben poder realizarse sin necesidad de modificar la clase estática Procesador. El siguiente código debe producir la salida indicada:

Práctica sobre la teoría 7

```
Auto auto = new Auto();
Libro libro = new Libro();
Persona persona = new Persona();
Perro perro = new Perro();
Pelicula pelicula = new Pelicula();
Procesador.Alquilar(pelicula, persona);
Procesador.Alquilar(libro, persona);
Procesador.Atender(persona);
Procesador.Atender(perro);
Procesador.Devolver(pelicula, persona);
Procesador.Devolver(libro, persona);
Procesador.Lavar(auto);
Procesador.Reciclar(libro);
Procesador.Reciclar(auto);
Procesador.Secar(auto);
Procesador.Vender(auto, persona);
Procesador.Vender(perro, persona);
Procesador.Lavar(perro);
Procesador.Secar(perro);
PeliculaClasica peliculaClasica = new PeliculaClasica();
Procesador.Alquilar(peliculaClasica, persona);
Procesador.Devolver(peliculaClasica, persona);
Procesador.Vender(peliculaClasica, persona);
```

Salida por consola

```
Alquilando película a persona
Alquilando libro a persona
Atendiendo persona
Atendiendo perro
Película devuelta por persona
Libro devuelto por persona
Lavando auto
Reciclando libro
Reciclando auto
Secando auto
Vendiendo auto a persona
Vendiendo perro a persona
Lavando perro
Secando perro
Alquilando película clásica a persona
Película clásica devuelta por persona
Vendiendo película clásica a persona
```

NOTA: Esta práctica se complementa con la primera entrega del trabajo de programación solicitado