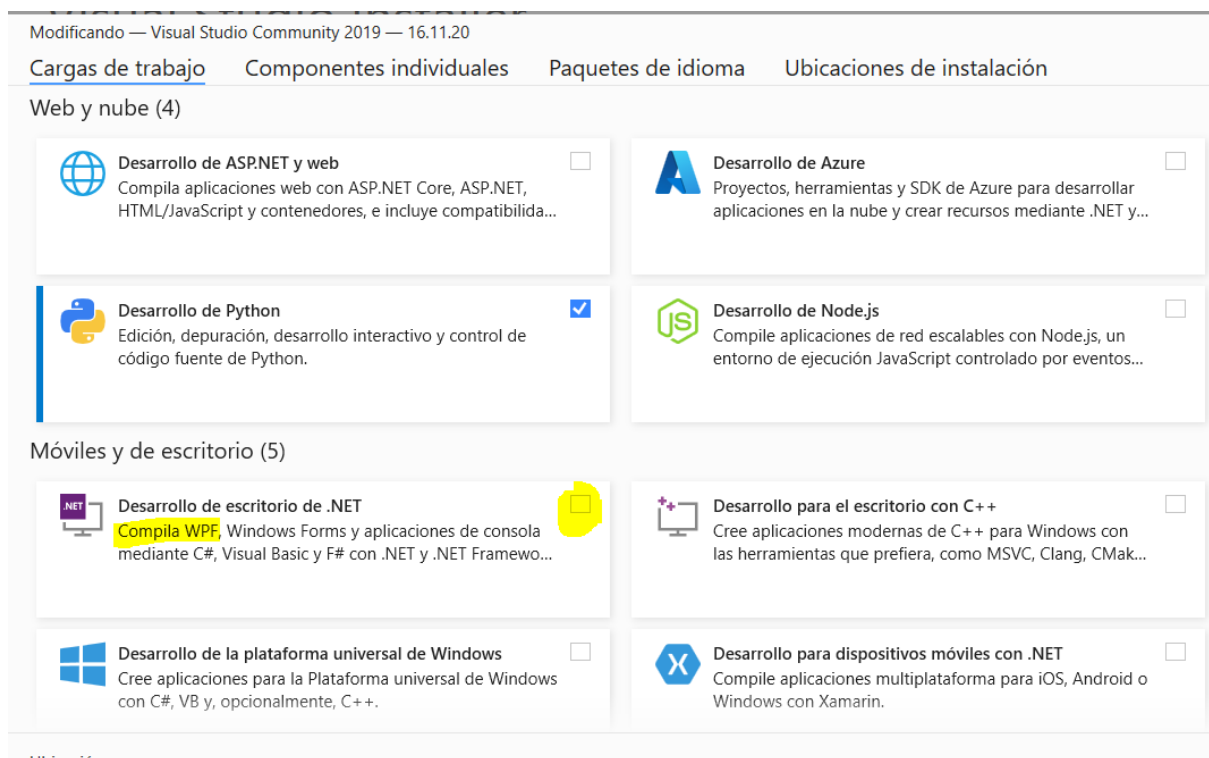
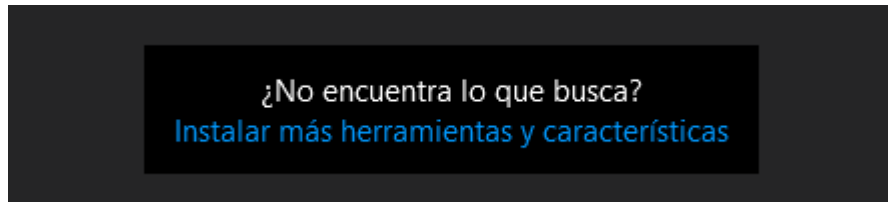


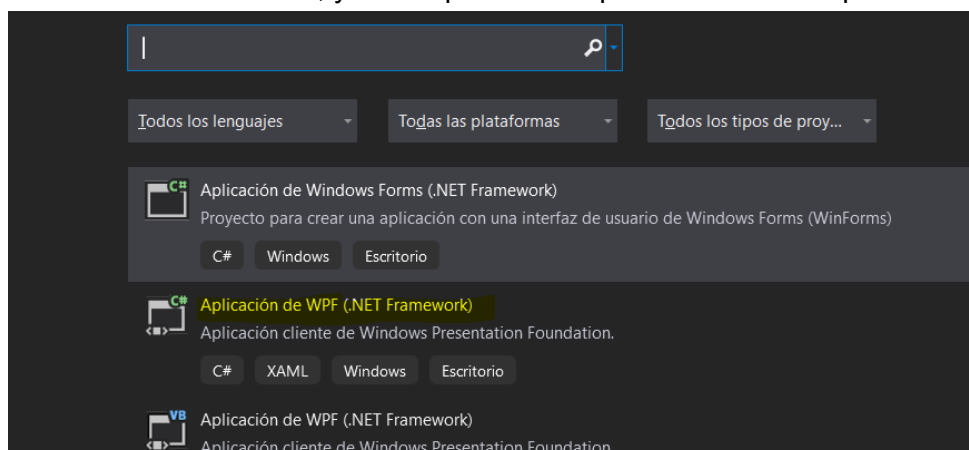
UT6 - Desarrollo de interfaces con WPF

Necesitamos Visual Studio de Microsoft

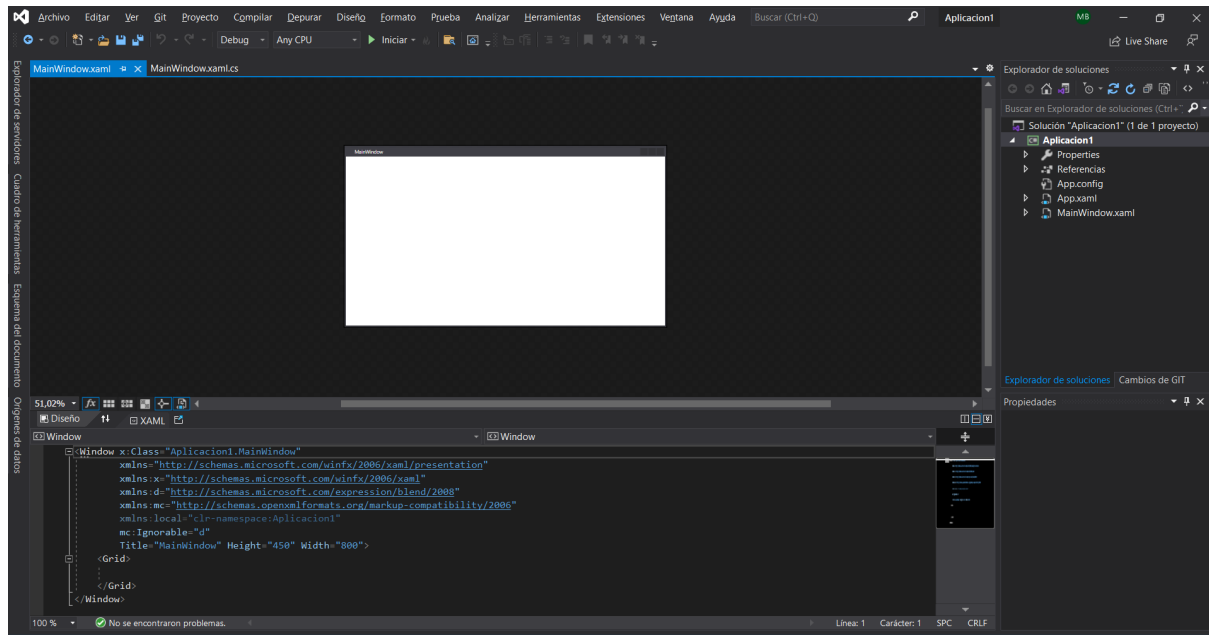
Si cuando vamos a crear un proyecto, no encontramos el tipo de proyecto WPF, vamos al final de la página de selección de tipo de proyecto y buscamos WPF:



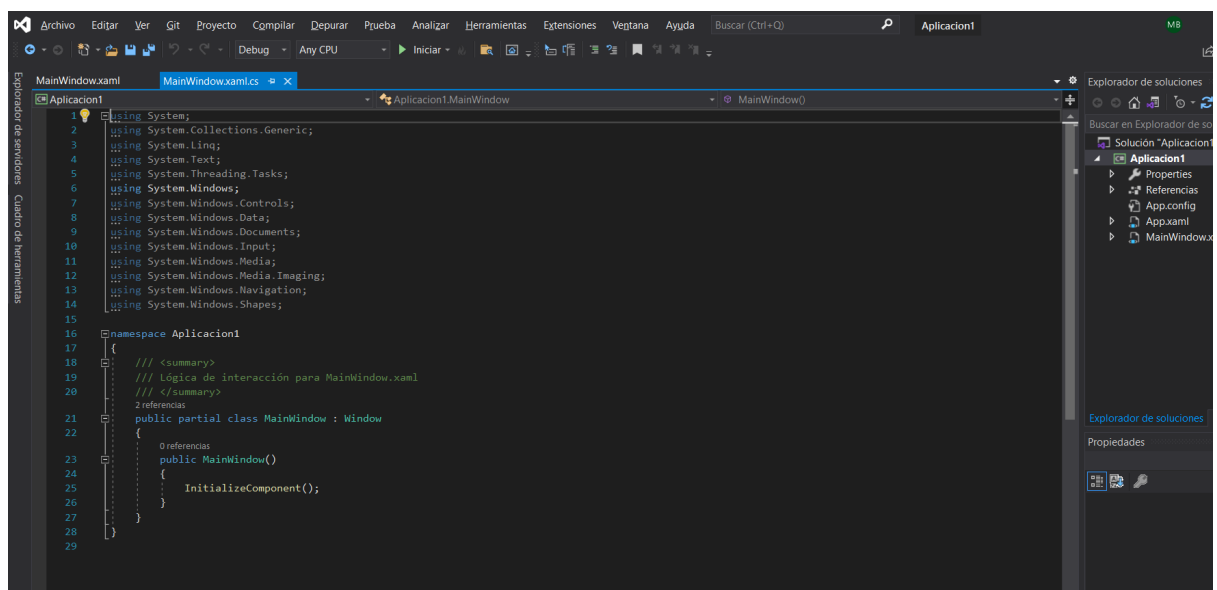
Tras instalar lo marcado, ya nos aparecerá la plantilla de crear Aplicación de WPF:



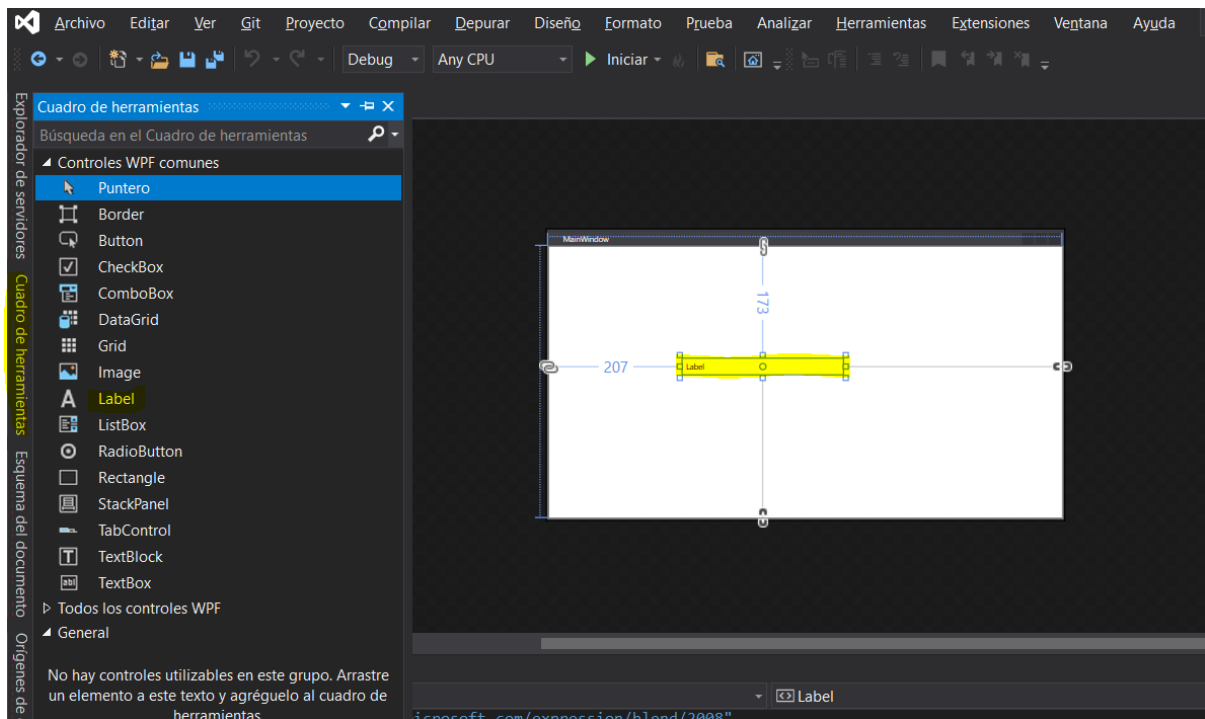
Se nos abre el editor con el explorador de proyectos a la derecha:



Abajo tenemos el código asociado a la ventana, en lenguaje XAML. Y en la parte central la vista de la pantalla y en la otra pestaña el código de en C#

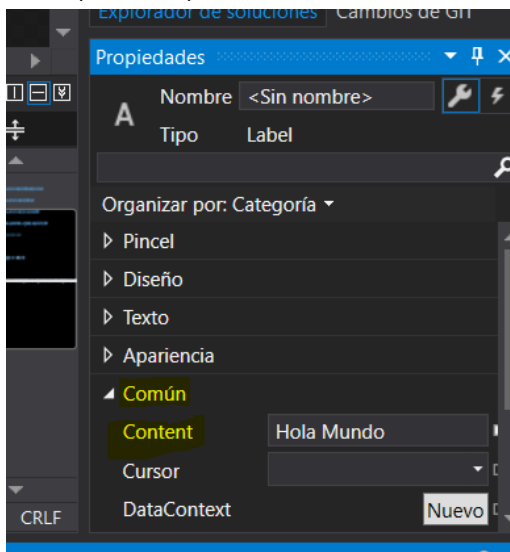


A la izquierda tenemos el cuadro de herramientas con los controles:

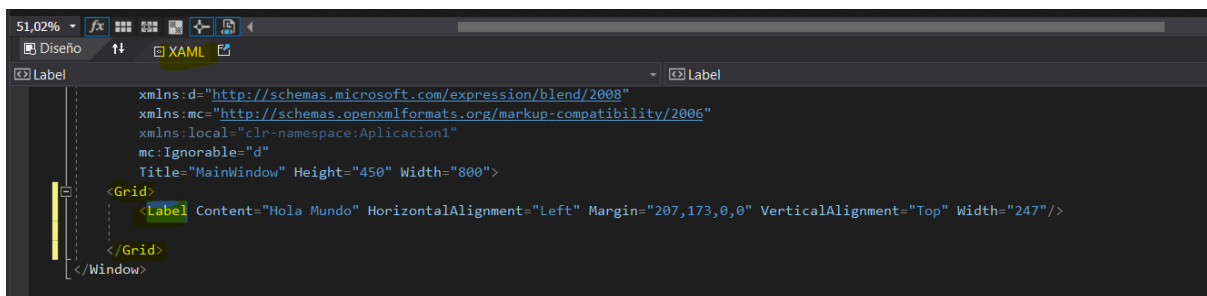


Podemos arrastrar por ejemplo una etiqueta a nuestra ventana.

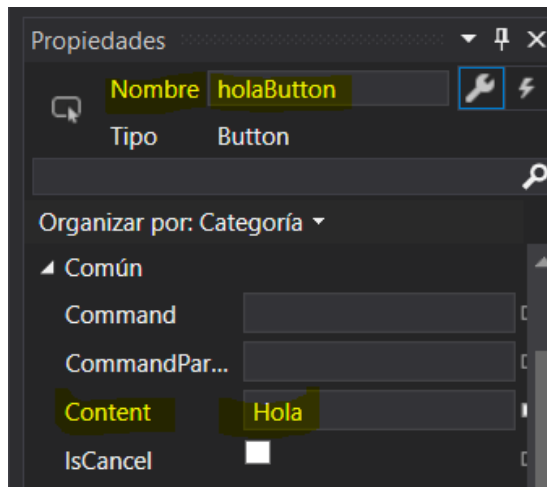
Y abajo a la derecha tenemos las propiedades de la etiqueta, donde podemos cambiar el texto (Content) a “Hola Mundo”



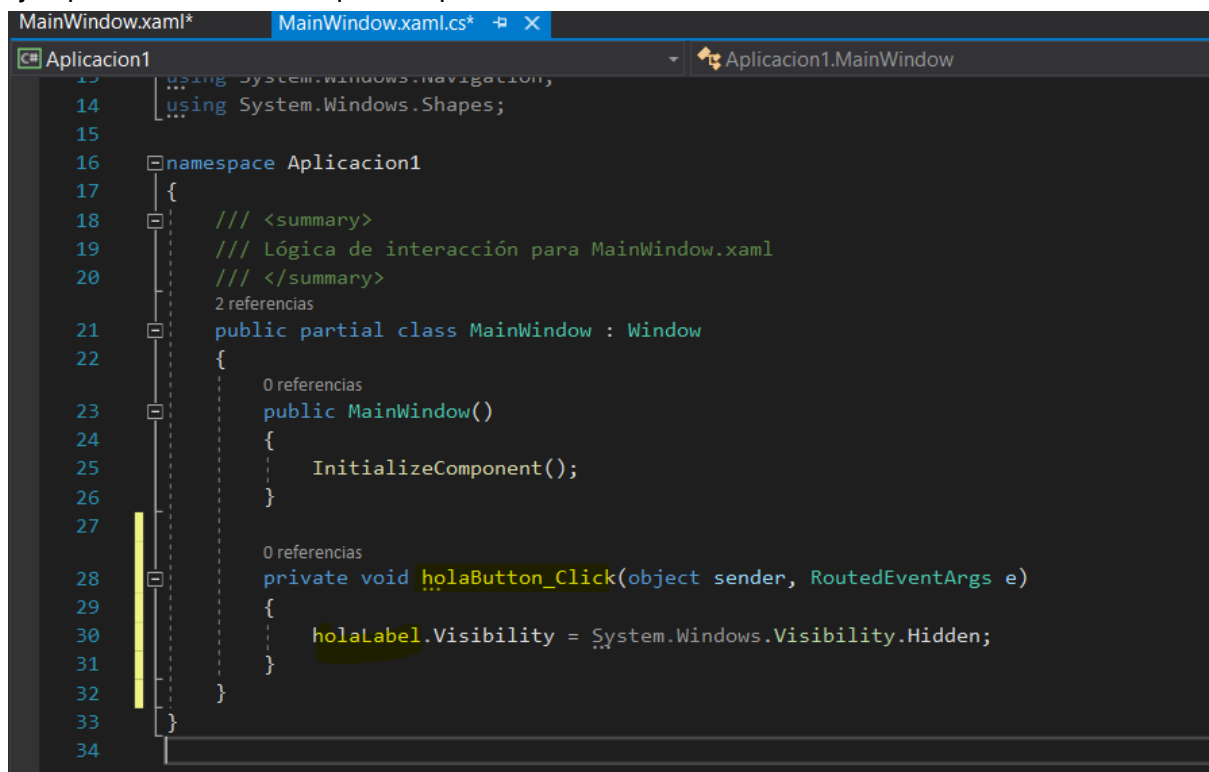
Y vemos como automáticamente se va generando código en la ventana de XAML



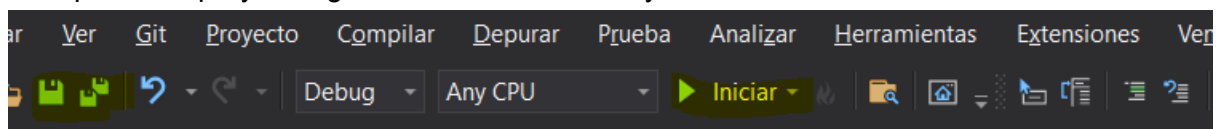
Podemos añadir un botón, que igualmente cambiamos el texto y el nombre del control en la ventana de propiedades:

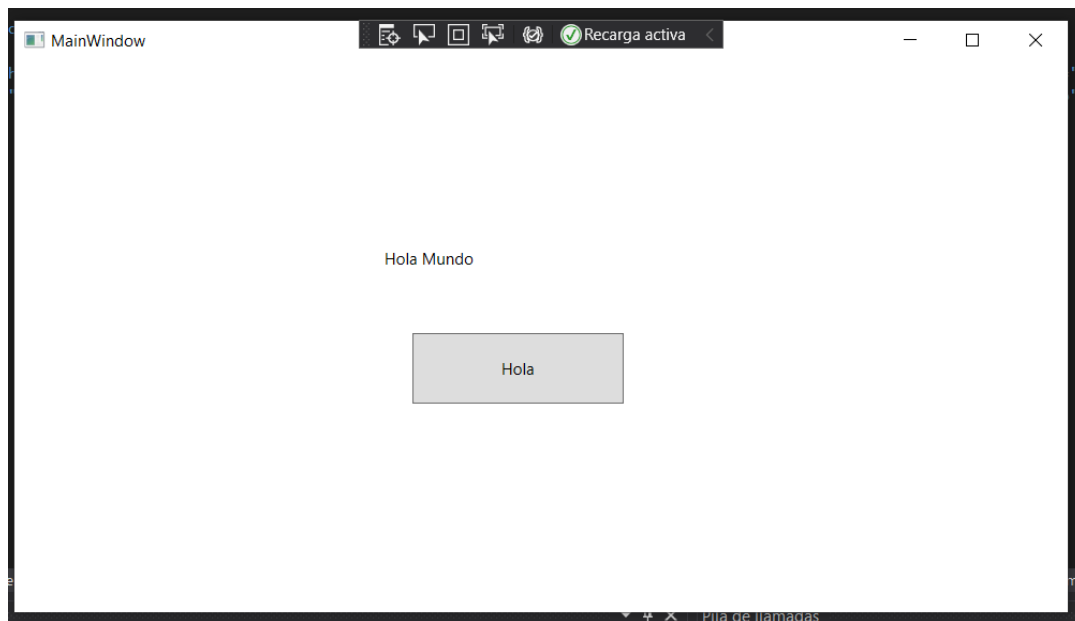


Y haciendo doble clic sobre el botón nos aparece el método del evento para definirlo. Por ejemplo, ocultamos la etiqueta al pulsar el botón.



Para probar el proyecto, guardamos los cambios y le damos a "Iniciar"

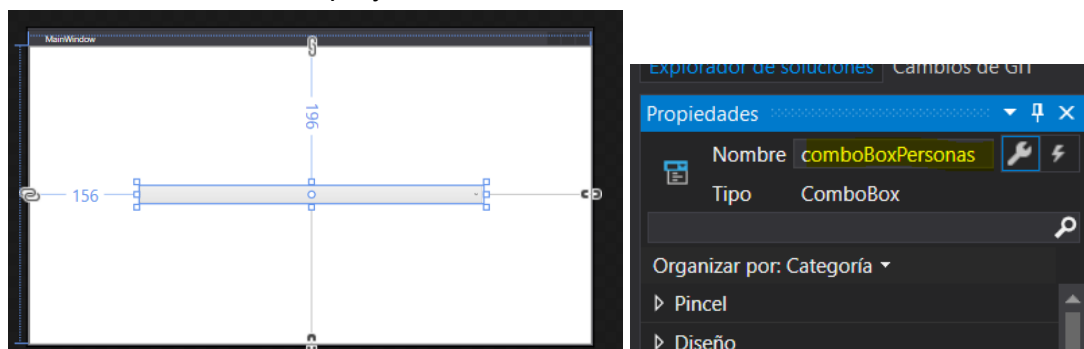




Se nos abre la ventana y al pulsar el botón se nos oculta la etiqueta.

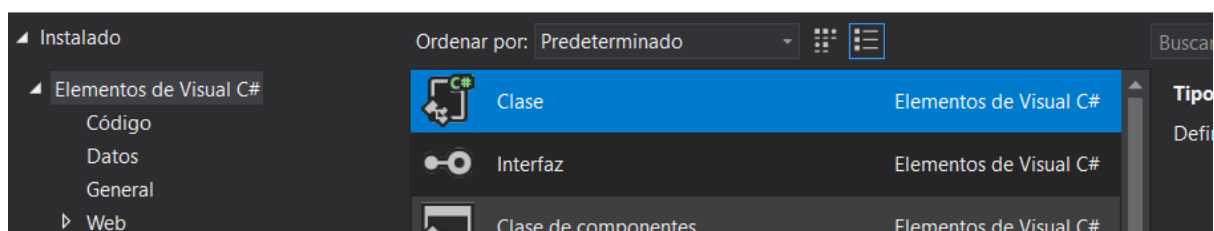
UT6 - 2 - Utilización del ComboBox en .NET WPF

Vamos a crear un nuevo proyecto, añadimos un comboBox:



Creamos una clase persona.cs dentro de una nueva carpeta (dto) para definir

Agregar nuevo elemento - WpfApp2



(si no nos aparece la clase de Visual C#, tendremos que ir a Extensiones, buscar Visual C# e instalar, y seguramente crear un nuevo proyecto si el anterior nos da error tras reiniciar.

Creamos la clase persona.cs y definimos dos propiedades: Nombre y Apellidos.

Ambas se pueden definir como se ha definido Nombre, con los get y set en la misma línea, o de forma más detallada en caso de que queramos definir una funcionalidad mayor en get y set.

```

class Persona
{
    1 referencia
    public String Nombre { get; set; }
    private String apellidos;
    0 referencias
    public String Apellidos
    {
        get
        {
            return apellidos;
        }
        set
        {
            apellidos = value;
        }
    }
    0 referencias
    public Persona(String nombre, String apellidos)
    {
        this.Nombre = nombre;
        this.Apellidos = apellidos;
    }
}

```

Luego definimos el constructor y hacemos un override del método ToString para que nos devuelva el formato que queremos:

```

0 referencias
public override string ToString()
{
    return Nombre + " " + Apellidos;
}

```

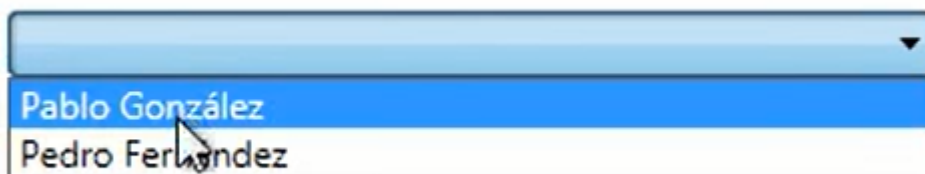
Volvemos a la pantalla principal y creamos en código los valores del combo:

```

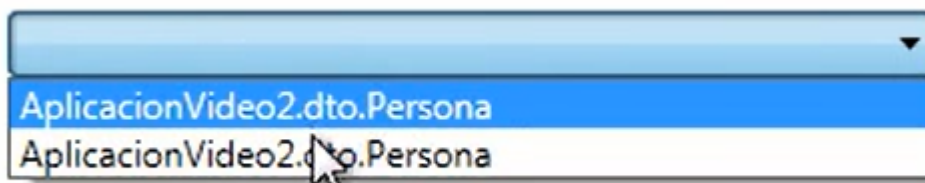
2 referencias
public partial class MainWindow : Window
{
    0 referencias
    public MainWindow()
    {
        InitializeComponent();
        List<Persona> listaPersonas = new List<Persona>();
        listaPersonas.Add(new Persona("Pablo", "González"));
        listaPersonas.Add(new Persona("Pedro", "Fernández"));
        foreach (Persona persona in listaPersonas)
        {
            ComboBoxItem cbi = new ComboBoxItem();
            cbi.Content = persona;
            comboBoxPersonas.Items.Add(cbi);
        }
    }
}

```

Si ejecutamos el programa:

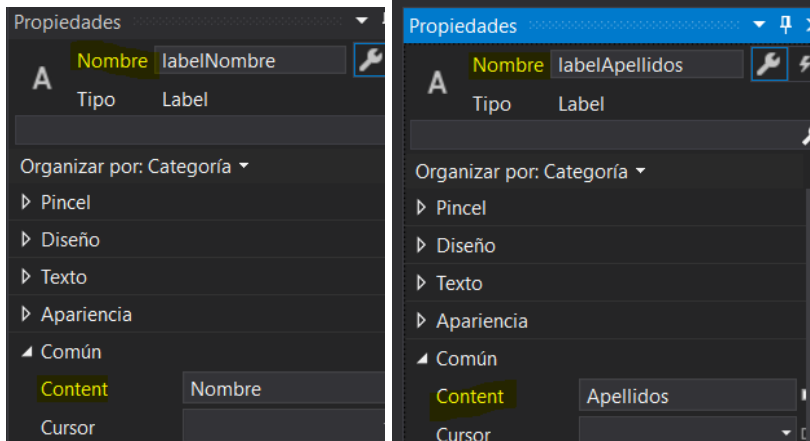
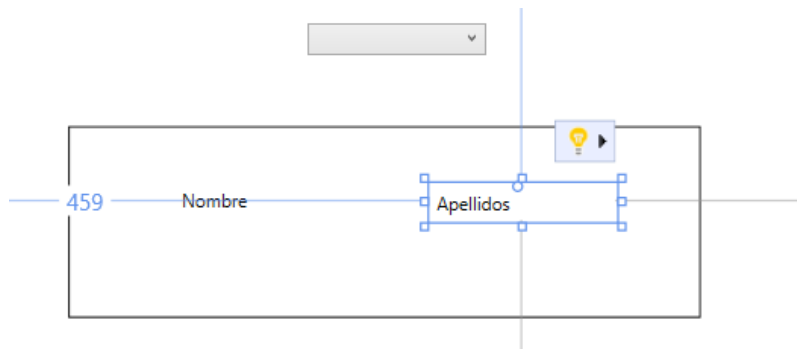


Sin el override del método ToString, nos saldría esto:

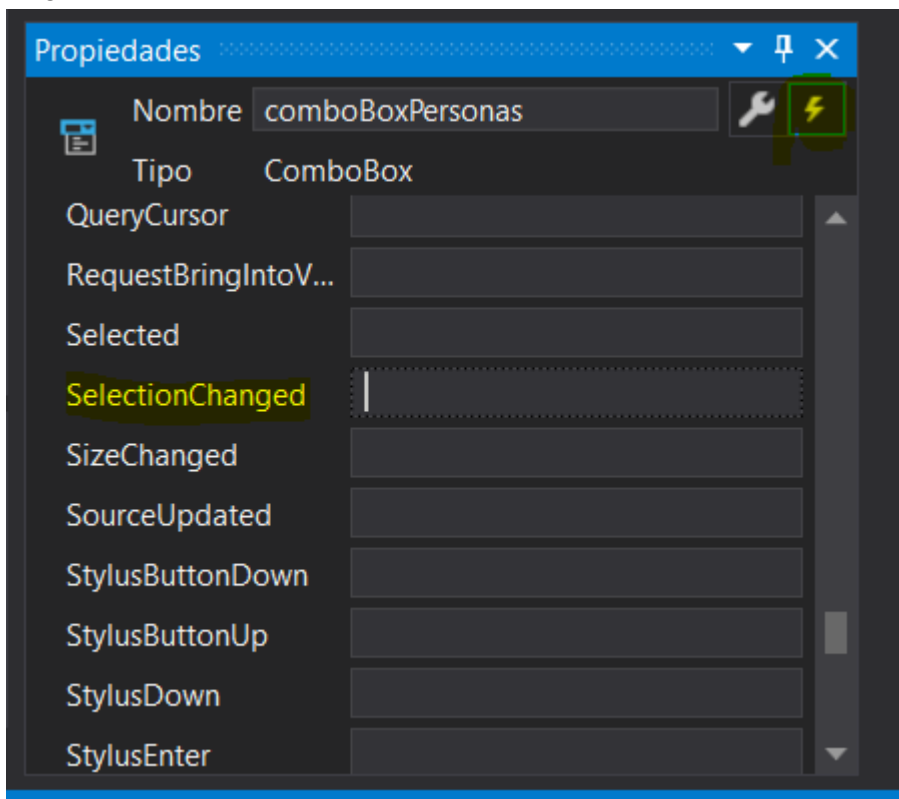


UT6 - 3 - Gestión de eventos en .NET WPF

Partiendo del proyecto anterior, vamos a añadir un par de etiquetas, para que muestren los datos seleccionados en el comboBox. Metemos las dos etiquetas en un elemento Rectángulo:



La gestión de eventos está en la ventana de propiedades, en el icono del rayo:



Si buscamos para nuestro comboBox el evento SeleccionChanged y hacemos doble clic en la caja de texto nos genera automáticamente un método para que lo definamos:


```

0 Referencias
private void comboBoxPersonas_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    ComboBoxItem cbi = (ComboBoxItem)comboBoxPersonas.SelectedItem;
    Persona persona = (Persona)cbi.Content;
    labelNombre.Content = persona.Nombre;
    labelApellidos.Content = persona.Apellidos;
}

```

Si lo ejecutamos:

Pablo González

Pablo González

UT6 - 4 - Bindings en .NET WPF

Con el binding conectamos la parte visual con la parte lógica.

Vamos a crear un proyecto donde pasamos lo que escribimos en una caja de texto a una etiqueta:

TextBox

Label

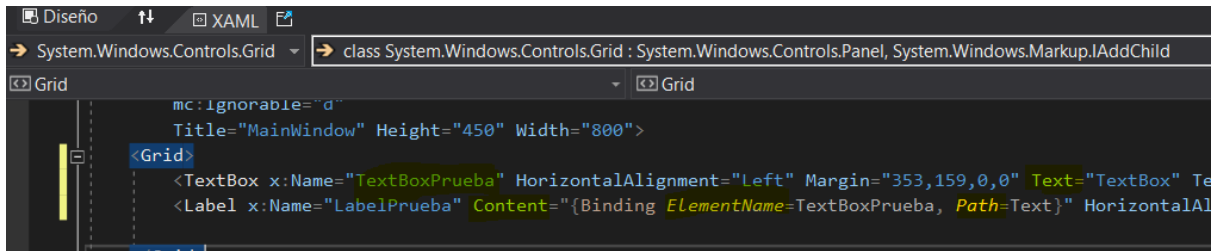
Si vemos el código asociado de XAML:

```

Diseño  ↑↓  XAML
Window.Grid  →  <Grid><TextBox x:Name= HorizontalAlignment= Margin= Text= TextWrapping= VerticalAlignment= Width=
Grid  Grid
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:WpfAppBinding"
mc:Ignorable="d"
Title="MainWindow" Height="450" Width="800">
<Grid>
<TextBox x:Name="TextBoxPrueba" HorizontalAlignment="Left" Margin="353,159,0,0" Text="TextBox" Te
<Label x:Name="LabelPrueba" Content="Label" HorizontalAlignment="Left" Margin="353,231,0,0" Verti

```

Queremos que en la propiedad "Content" de Label aparezca el valor de la propiedad "Text" del TextBox. Eso lo podemos ajustar directamente en el código con **Binding**:



En **ElementName** indicamos el nombre del control con el que queremos conectar y en **Path** indicamos la propiedad que queremos utilizar.

UT6 - 5 - Bindings en ComboBox .NET WPF

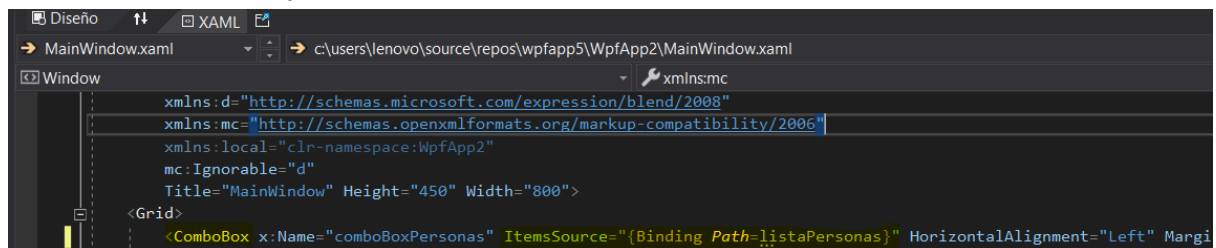
Ahora vamos a aplicar binding en la carga de los valores de un combobox, que teníamos de esta forma en un proyecto anterior, con un foreach:

```
0 referencias
public MainWindow()
{
    InitializeComponent();
    List<Persona> listaPersonas = new List<Persona>();
    listaPersonas.Add(new Persona("Pablo", "González"));
    listaPersonas.Add(new Persona("Pedro", "Fernández"));
    foreach (Persona persona in listaPersonas)
    {
        ComboBoxItem cbi = new ComboBoxItem();
        cbi.Content = persona;
        comboBoxPersonas.Items.Add(cbi);
    }
}
```

Borramos el bloque de foreach:

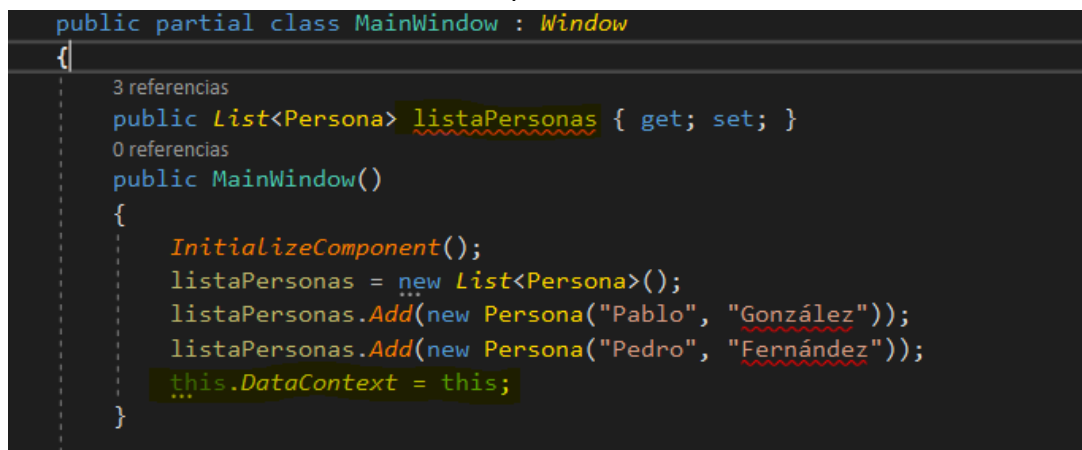
```
public partial class MainWindow : Window
{
    0 referencias
    public MainWindow()
    {
        InitializeComponent();
        List<Persona> listaPersonas = new List<Persona>();
        listaPersonas.Add(new Persona("Pablo", "González"));
        listaPersonas.Add(new Persona("Pedro", "Fernández"));
    }
}
```

y en el código XAML añadimos la propiedad ItemsSource para el binding, con el que conectamos con el objeto listaPersonas que teníamos en el archivo de cs.



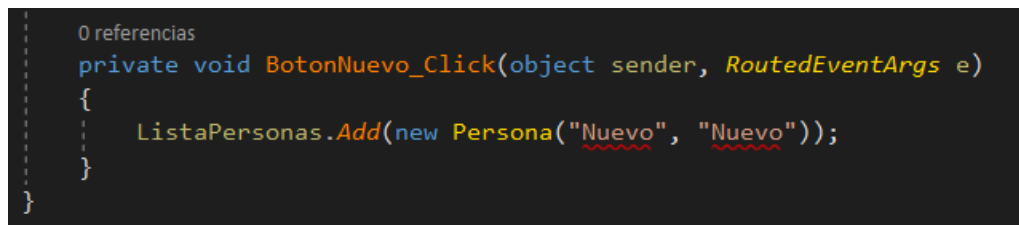
```
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:WpfApp2"
mc:Ignorable="d"
Title="MainWindow" Height="450" Width="800">
<Grid>
    <ComboBox x:Name="comboBoxPersonas" ItemsSource="{Binding Path=listaPersonas}" HorizontalAlignment="Left" Margi
```

Además, en el cs debemos añadir una referencia a DataContext para que binding pueda conectar con el atributo listaPersonas que tenemos definido.



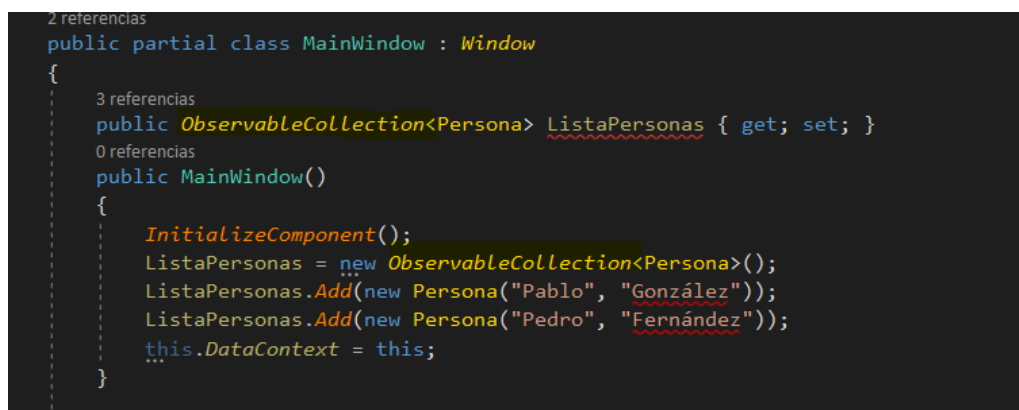
```
public partial class MainWindow : Window
{
    3 referencias
    public List<Persona> listaPersonas { get; set; }
    0 referencias
    public MainWindow()
    {
        InitializeComponent();
        listaPersonas = new List<Persona>();
        listaPersonas.Add(new Persona("Pablo", "González"));
        listaPersonas.Add(new Persona("Pedro", "Fernández"));
        this.DataContext = this;
    }
}
```

Si añadimos un botón para agregar elementos en la lista:



```
0 referencias
private void BotonNuevo_Click(object sender, RoutedEventArgs e)
{
    listaPersonas.Add(new Persona("Nuevo", "Nuevo"));
}
}
```

Tenemos que cambiar el objeto List por ObservableCollection, para que detecte cuándo se le añaden nuevos elementos a la lista.

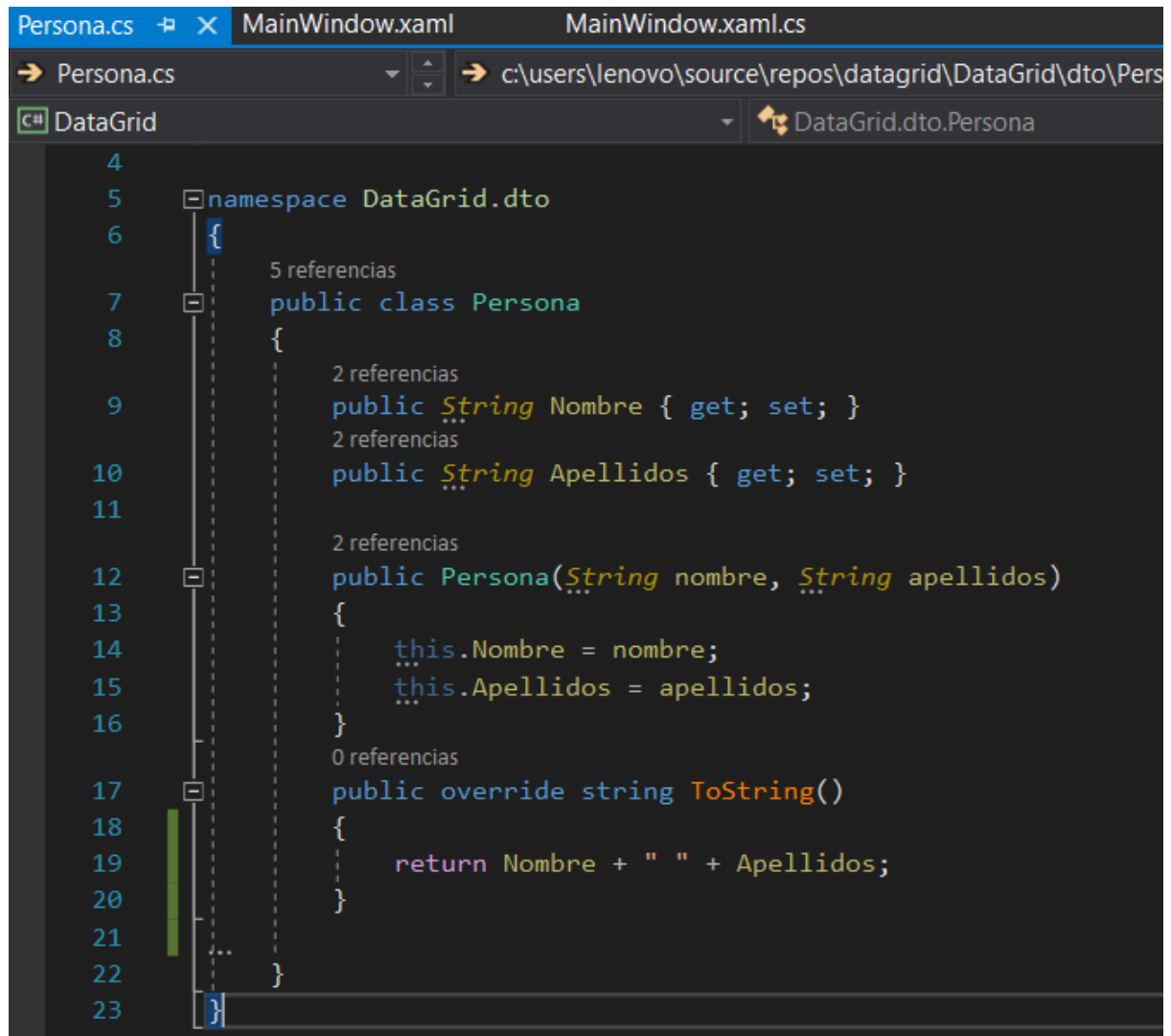


```
2 referencias
public partial class MainWindow : Window
{
    3 referencias
    public ObservableCollection<Persona> listaPersonas { get; set; }
    0 referencias
    public MainWindow()
    {
        InitializeComponent();
        listaPersonas = new ObservableCollection<Persona>();
        listaPersonas.Add(new Persona("Pablo", "González"));
        listaPersonas.Add(new Persona("Pedro", "Fernández"));
        this.DataContext = this;
    }
}
```

UT6 - 6 - Uso de DataGrid en .NET WPF

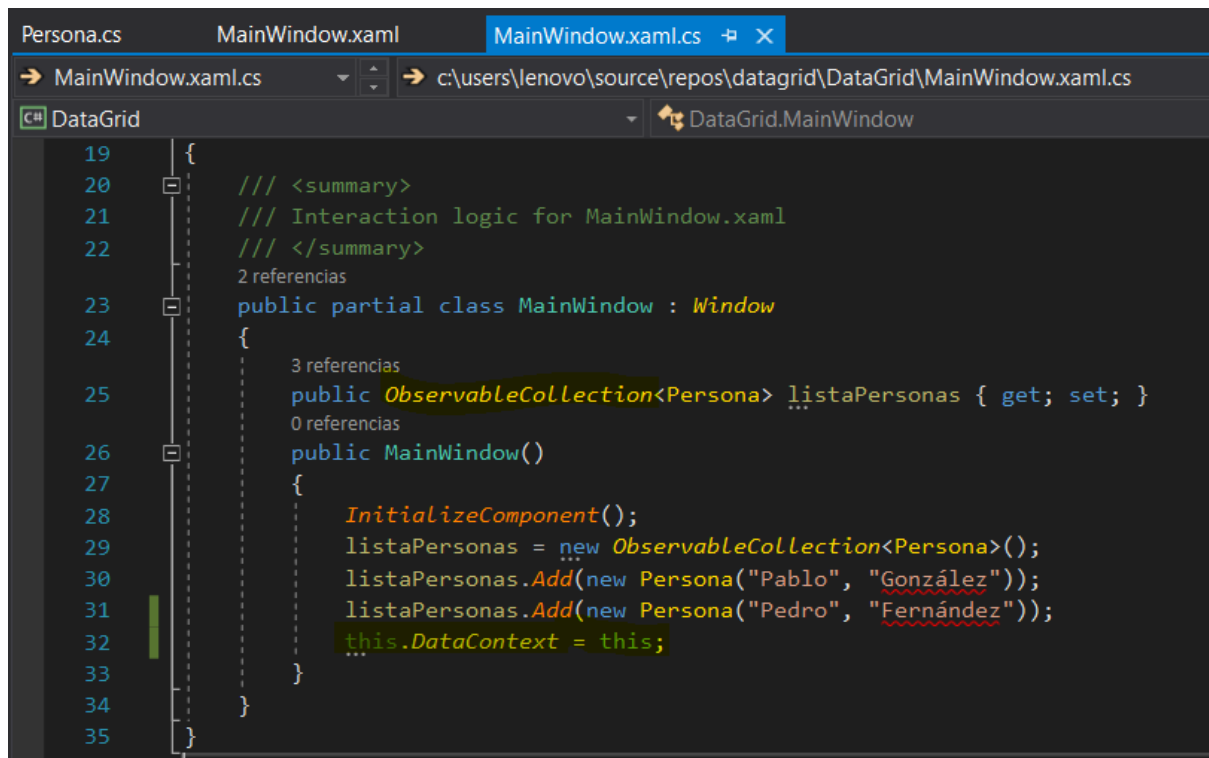
Vamos a gestionar una **tabla** en **WPF**, que son los controles **DataGrid**.

Creamos un proyecto WPF, añadimos un DataGrid y una clase C# para definir el objeto Persona.



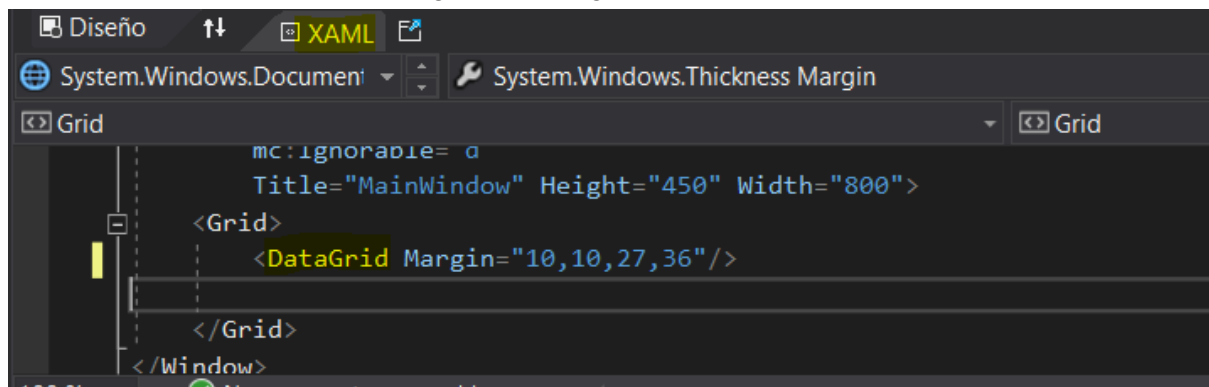
```
4
5 namespace DataGrid.dto
6 {
7     5 referencias
8     public class Persona
9     {
10         2 referencias
11         public String Nombre { get; set; }
12         2 referencias
13         public String Apellidos { get; set; }
14
15         2 referencias
16         public Persona(String nombre, String apellidos)
17         {
18             this.Nombre = nombre;
19             this.Apellidos = apellidos;
20         }
21
22         0 referencias
23         public override string ToString()
24         {
25             return Nombre + " " + Apellidos;
26         }
27     }
28 }
```

También tenemos definido un **ObservableCollection** y la línea **"this.DataContext = this"**, en el archivo CS de la ventana principal, para que cuando añadamos elementos a la lista de "personas", se actualicen los datos en la tabla.



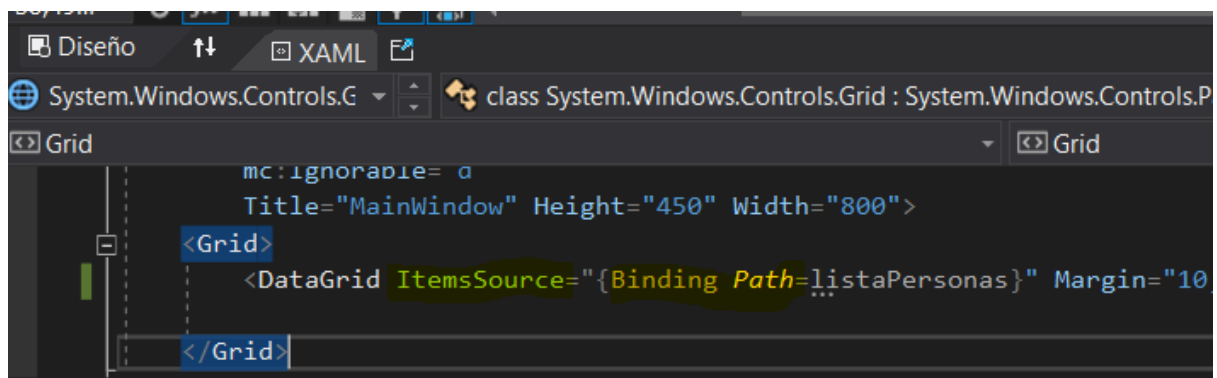
```
19 {
20     /// <summary>
21     /// Interaction logic for MainWindow.xaml
22     /// </summary>
23     2 referencias
24     public partial class MainWindow : Window
25     {
26         3 referencias
27         public ObservableCollection<Persona> listaPersonas { get; set; }
28         0 referencias
29         public MainWindow()
30         {
31             InitializeComponent();
32             listaPersonas = new ObservableCollection<Persona>();
33             listaPersonas.Add(new Persona("Pablo", "González"));
34             listaPersonas.Add(new Persona("Pedro", "Fernández"));
35             this.DataContext = this;
36         }
37     }
38 }
```

Al añadir el DataGrid vemos el siguiente código en el archivo XAML:



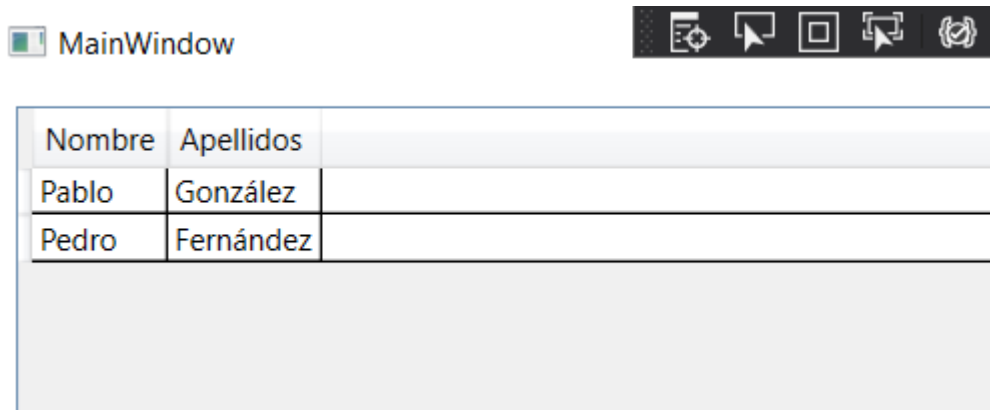
```
<Grid>
    mc:Ignorable="d"
    Title="MainWindow" Height="450" Width="800">
        <DataGrid Margin="10,10,27,36"/>
    </Grid>
</Window>
```

Sólo tenemos que añadir el atributo **ItemsSource** y engancharlo (mapearlo) con el objeto listaPersonas definido en nuestra clase.

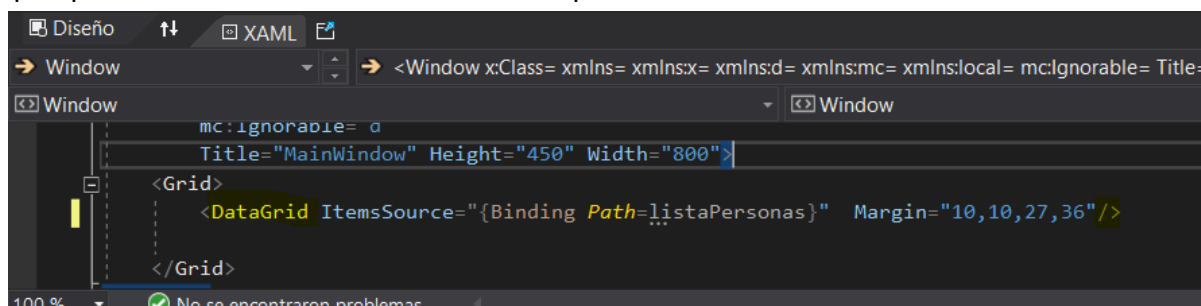


```
<Grid>
    mc:Ignorable="d"
    Title="MainWindow" Height="450" Width="800">
        <DataGrid ItemsSource="{Binding Path=listaPersonas}" Margin="10" />
    </Grid>
```

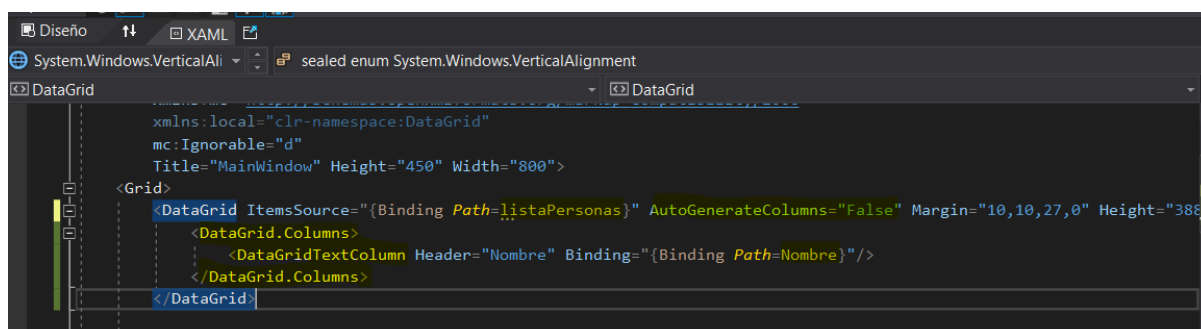
Si lo ejecutamos así, saldría lo siguiente:



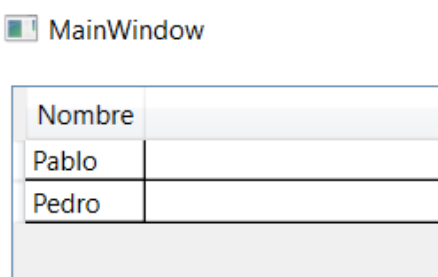
DataGrid tiene un atributo que lee los nombres de los campos de la fuente y genera las columnas necesarias automáticamente: **AutoGenerateColumns="True"**
 Esto mostraría todos los campos que existan. Si queremos controlar los campos que se muestran en vez de mostrar todos, tenemos que poner esta propiedad a "False" y cambiar la etiqueta de DataGrid de "vacía" a "apertura" y "cierre" con la información de las columnas que queremos ver anidada entre ambas etiquetas. Pasamos de esto:



a esto:



Si ejecutamos ahora saldría así:



Propuesta para hacer: agregar un botón “Añadir” con dos campos de texto para ir añadiendo registros nuevos a la tabla.

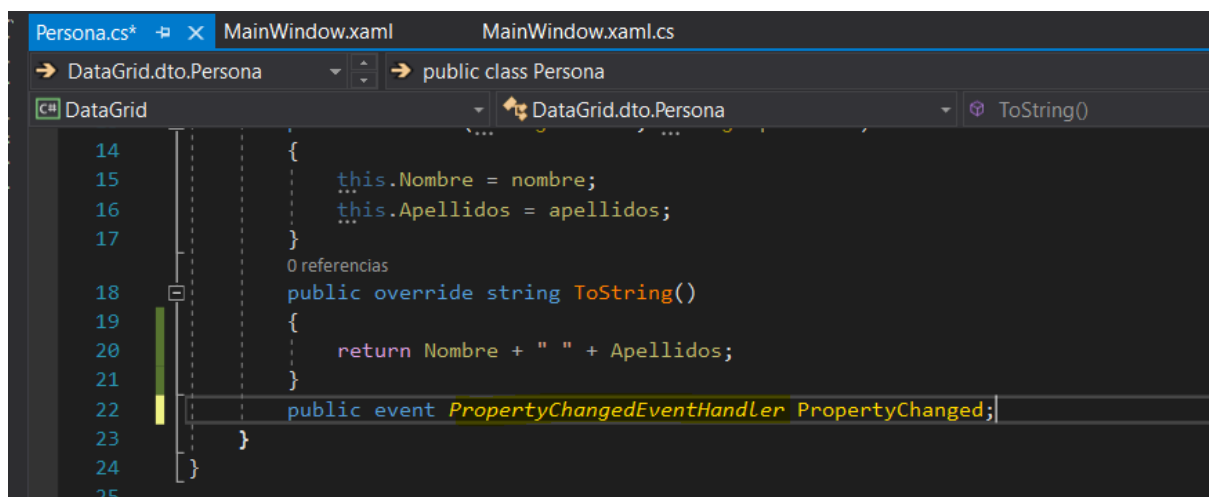
Ahora vamos a intentar **modificar elementos de la tabla** añadiendo un botón con este código en el evento click:

```
0 referencias
private void BotonModificar_Click(object sender, RoutedEventArgs e)
{
    listaPersonas.ElementAt(0).Nombre = "Juan";
}
}
```

Lo que hace es cambiar el nombre del primer registro. Si lo ejecutamos tal cual, veremos que no se refleja el cambio en la tabla, aunque sí se ha modificado internamente en el objeto Persona, pero no se ha actualizado el nuevo valor en la tabla.

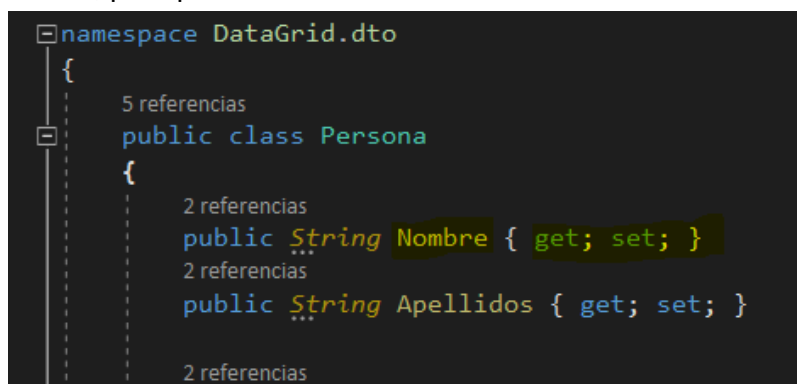
Debemos modificar el objeto “Persona” porque tal y como está definido no actualiza la información en el DataGrid cuando se modifica alguno de sus elementos.

Tenemos que añadir un **evento** de “modificación de propiedad” en la clase “Persona”:



```
Persona.cs*  MainWindow.xaml  MainWindow.xaml.cs
→ DataGrid.dto.Persona  → public class Persona
[DataGrid]  DataGrid.dto.Persona  ToString()
14  {
15      this.Nombre = nombre;
16      this.Apellidos = apellidos;
17  }
18  0 referencias
19  public override string ToString()
20  {
21      return Nombre + " " + Apellidos;
22  }
23  public event PropertyChangedEventHandler PropertyChanged;
24  }
25  }
```

y usar dicho evento en el setter de las propiedades de Persona. Como hemos definido los setter y getter de los atributos con el método simplificado, debemos cambiar a la forma normal, para poder utilizar el evento. Pasamos de esto:



```
namespace DataGrid.dto
{
    5 referencias
    public class Persona
    {
        2 referencias
        public string Nombre { get; set; }
        2 referencias
        public string Apellidos { get; set; }
        2 referencias
    }
}
```

A esto, donde hemos implementado en la clase Persona **INotifyPropertyChanged**

```

public class Persona : INotifyPropertyChanged
{
    private String nombre;
    1 referencia
    public String Nombre
    {
        get
        {
            return nombre;
        }
        set
        {
            this.nombre = value;
            this.PropertyChanged(this, new PropertyChangedEventArgs("nombre"));
        }
    }

    private String apellidos;
    0 referencias
    public String Apellidos
    {
        get
        {
            return apellidos;
        }
        set
        {
            this.apellidos = value;
            this.PropertyChanged(this, new PropertyChangedEventArgs("apellidos"));
        }
    }
}
2 referencias

```

y como ahora los atributos están en minúsculas y lo que hay en mayúsculas son el getter y el setter, tenemos que ajustar el constructor, para que utilice los atributos en minúscula:

```

2 referencias
public Persona(String nombre, String apellidos)
{
    this.nombre = nombre;
    this.apellidos = apellidos;
}
0 referencias
public override string ToString()
{
    return nombre + " " + apellidos;
}
public event PropertyChangedEventHandler PropertyChanged;
}

```

También ajustamos el método ToString.

Propuesta: crear un formulario de entrada de registros en la tabla que permita añadir, modificar y borrar.

UT6 - 7 - Desarrollo de una aplicación completa en .NET WPF

Vamos a hacer una aplicación para gestionar libros. Partimos de una pantalla vacía y un objeto libro:

```
6 namespace WpfApp7.dto
7 {
8     1 referencia
9     public class Libro : INotifyPropertyChanged
10    {
11        private String titulo;
12        0 referencias
13        public String Titulo
14        {
15            get
16            {
17                return titulo;
18            }
19            set
20            {
21                titulo = value;
22                this.PropertyChanged(this, new PropertyChangedEventArgs("titulo"));
23            }
24        }
25        private String autor;
26        0 referencias
27        public String Autor
28        {
29            get
30            {
31                return autor;
32            }
33            set
34            {
35                autor = value;
36                this.PropertyChanged(this, new PropertyChangedEventArgs("autor"));
37            }
38        }
39    }
40 }
```

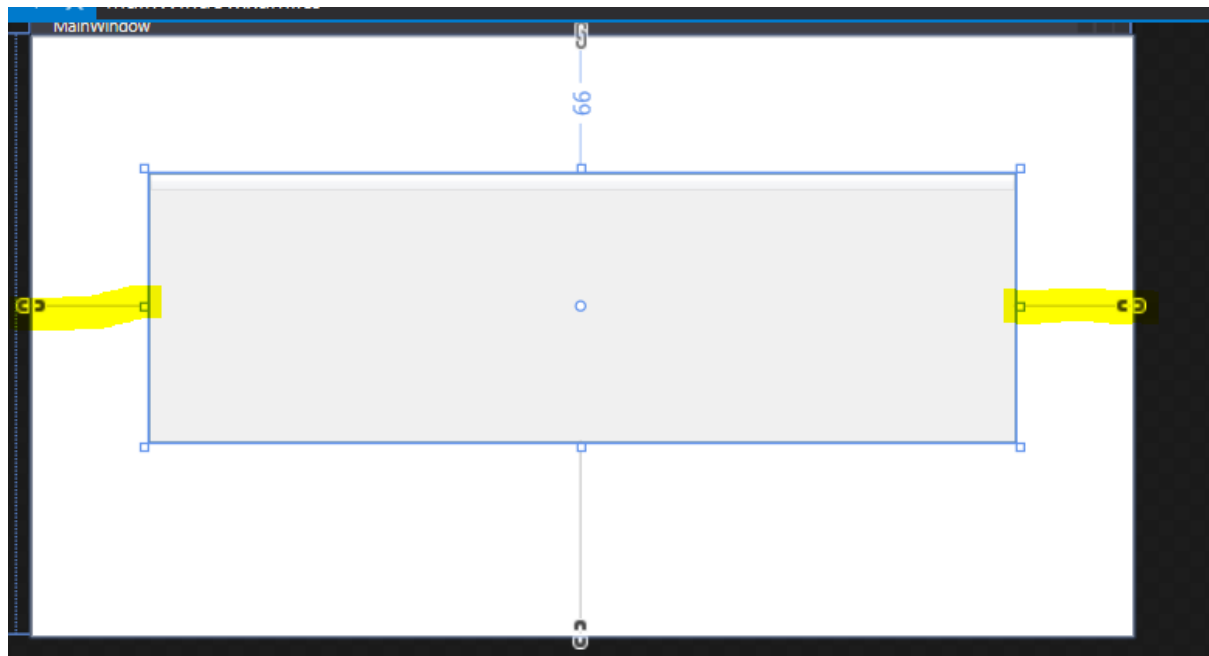
que tiene los atributos “titulo”, “autor” y “fecha de entrada” con los setter y getter adaptados a los cambios de la propiedad implementando la clase INotifyPropertyChanged, con su evento correspondiente.

```

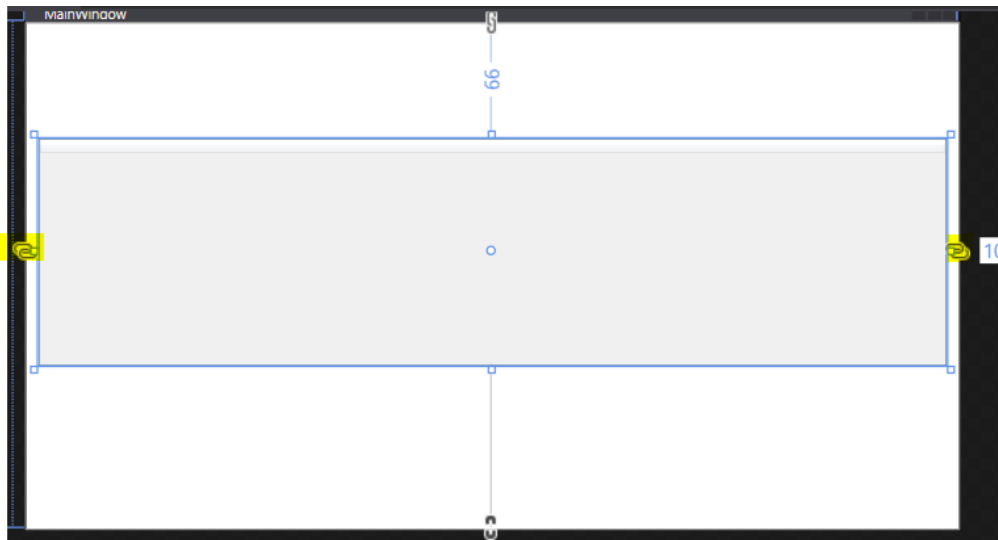
36     private DateTime fechaEntrada;
37
38
39
40     0 referencias
41     public DateTime FechaEntrada
42     {
43     get
44     {
45         return fechaEntrada;
46     }
47     set
48     {
49         fechaEntrada = value;
50         this.PropertyChanged(this, new PropertyChangedEventArgs("fechaEntrada"));
51     }
52
53     0 referencias
54     public Libro(String titulo, String autor, DateTime fechaEntrada)
55     {
56         this.titulo = titulo;
57         this.autor = autor;
58         this.fechaEntrada = fechaEntrada;
59     }
60
61     public event PropertyChangedEventHandler PropertyChanged;
62 }

```

Metemos una DataGrid en la pantalla principal:

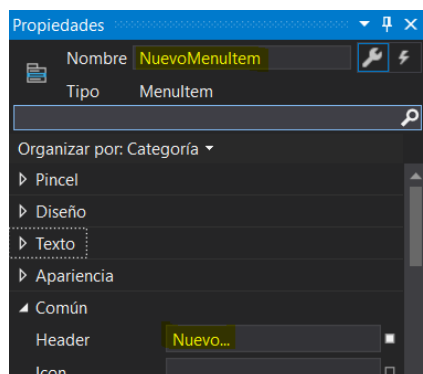
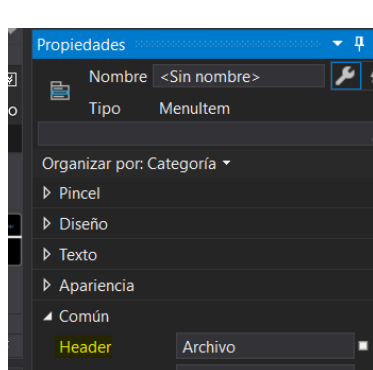
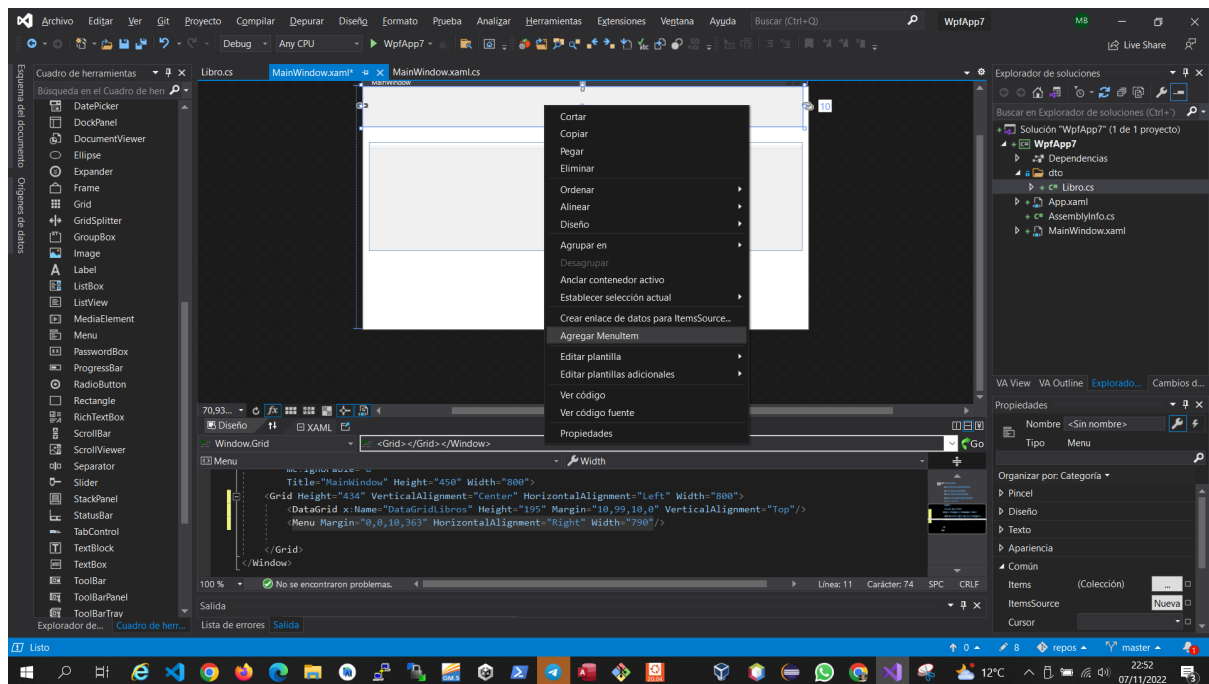


si acercamos los bordes al límite de la pantalla, podemos “enganchar los límites de la tabla a los de la pantalla, y se redimensionará junto con la pantalla.



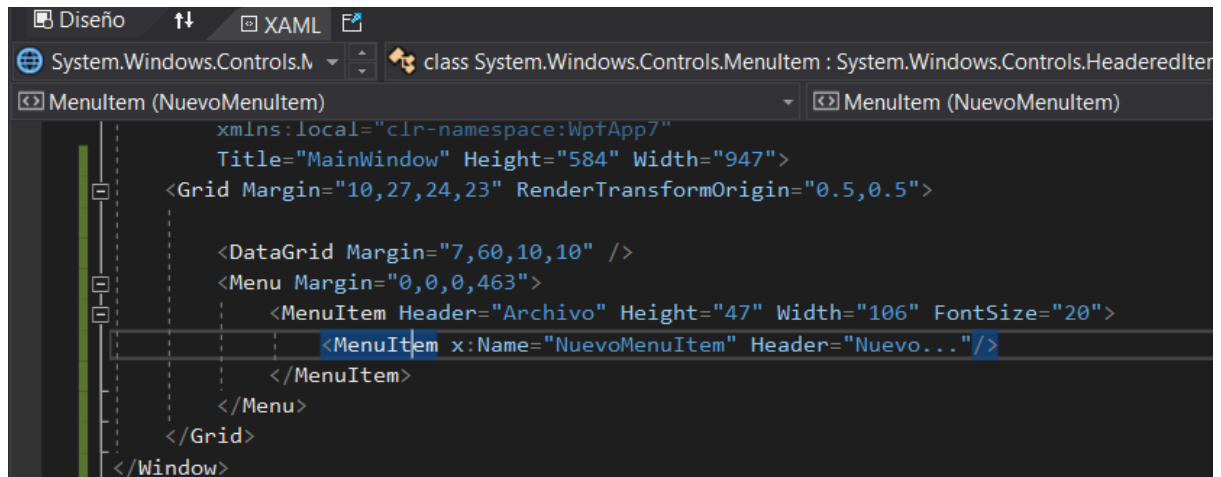
Lo llamamos DataGridLibros.

Añadimos ahora un menú, que también anclamos a los bordes de la pantalla. Y sobre el menú, botón derecho, añadimos un Item de menú:



Y sobre el MenuItem añadimos otro MenuItem, que sería un submenú. Como este submenú va a tener asociado un evento, ya le damos un nombre para poder utilizarlo en código: NuevoMenuItem.

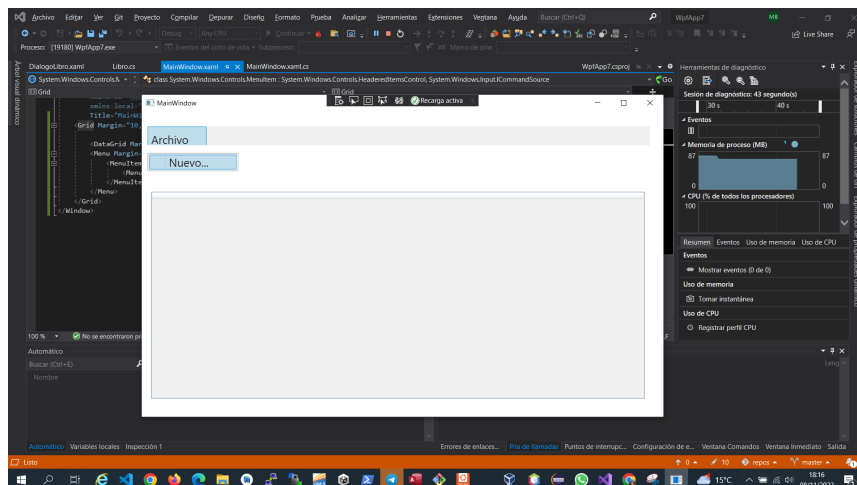
Por ahora nuestro archivo XAML tiene este aspecto:



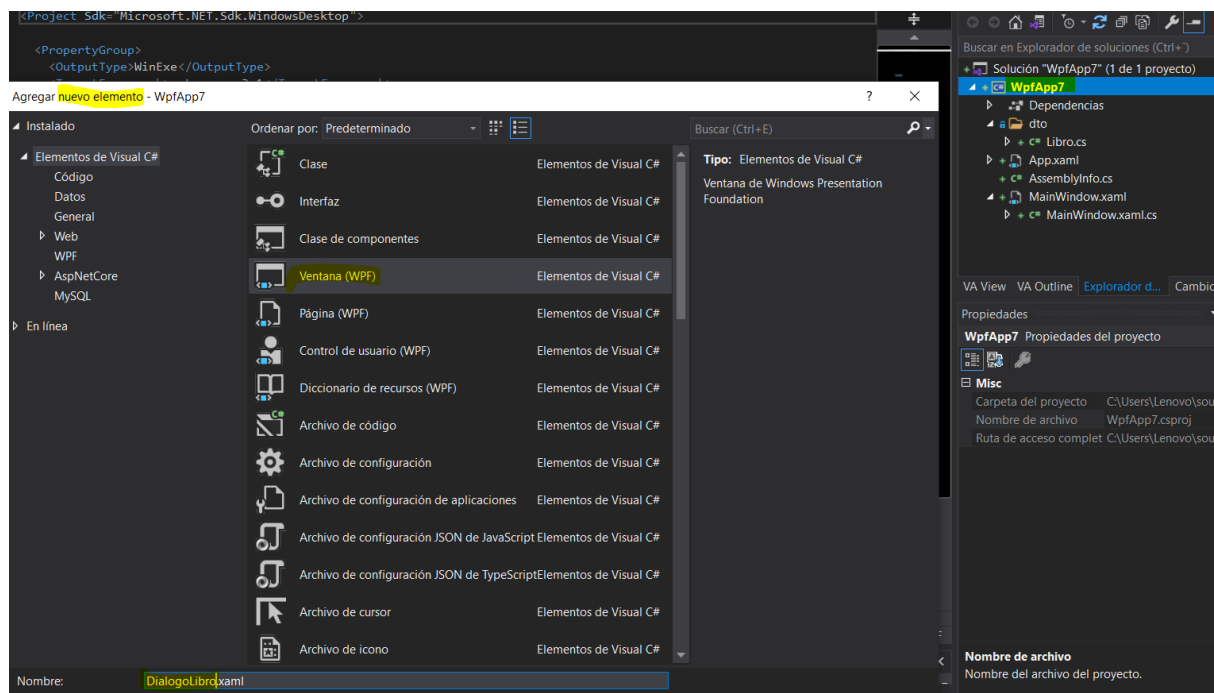
```
class System.Windows.Controls.Menuitem : System.Windows.Controls.HeaderedItem<object>
{
    public Menuitem()
    {
        this.DefaultStyleKey = typeof(Menuitem);
    }
}

<?xml version="1.0" encoding="utf-8" ?>
<xaml xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:local="clr-namespace:WpfApp7"
      Title="MainWindow" Height="584" Width="947">
  <Grid Margin="10,27,24,23" RenderTransformOrigin="0.5,0.5">
    <DataGrid Margin="7,60,10,10" />
    <Menu Margin="0,0,0,463">
      <MenuItem Header="Archivo" Height="47" Width="106" FontSize="20">
        <MenuItem x:Name="NuevoMenuItem" Header="Nuevo..." />
      </MenuItem>
    </Menu>
  </Grid>
</Window>
```

Si lo ejecutamos debe salir algo así:



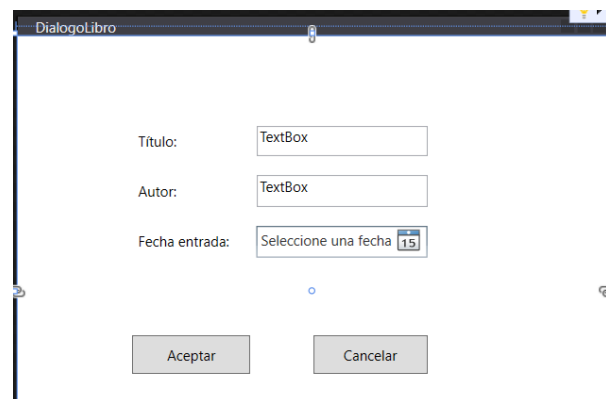
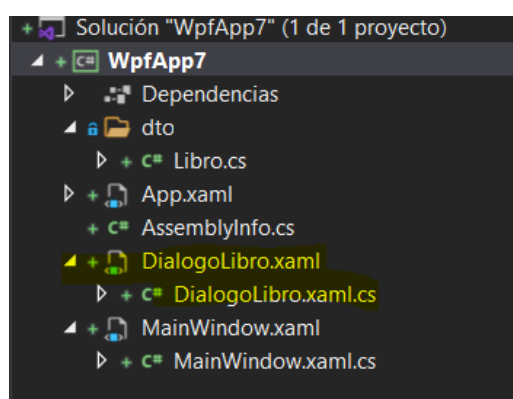
Ahora vamos a añadir un nuevo elemento al proyecto: una Ventana (WPF) que llamaremos DialogoLibro



Que utilizaremos para crear y para modificar los registros.

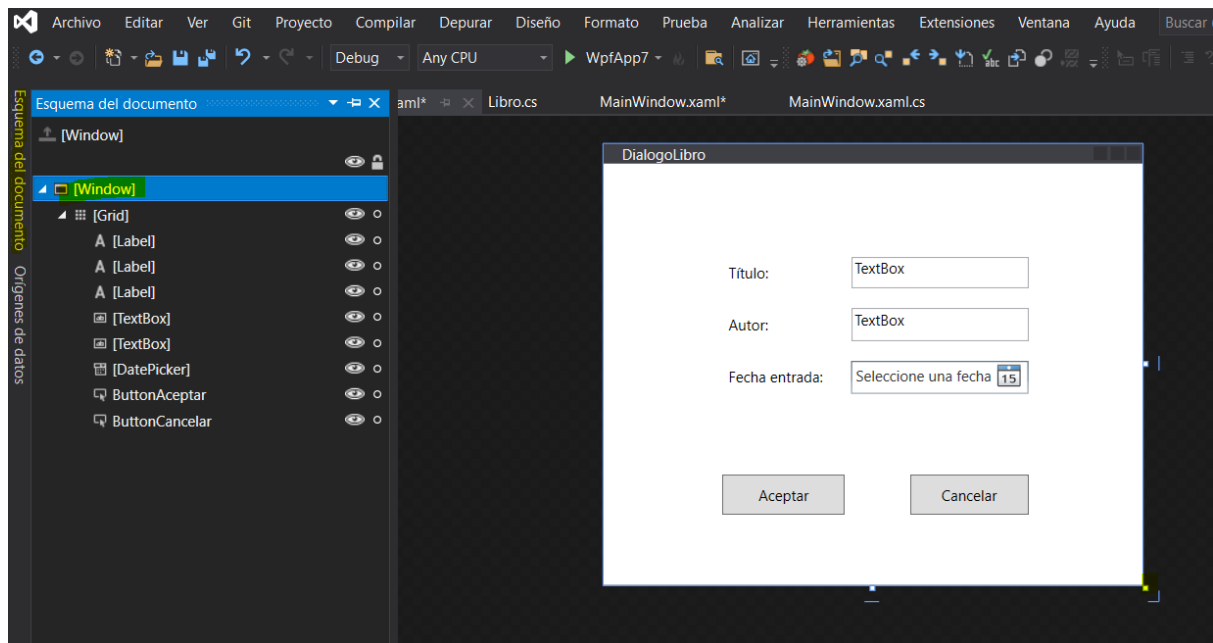
En el proyecto vemos que se han creado dos archivos: uno .xaml que sería la ventana en sí y otro xaml.cs que sería el código asociado a la ventana.

En la ventana vamos a añadir tres etiquetas (Label) para Título, Autor y Fecha de entrada. Para informar los dos primeros añadimos a la derecha un par de "TextBox" y para la fecha utilizamos un control de "DatePicker" que nos permite seleccionar la fecha de un calendario. (Para ver mejor la pantalla al posicionar los controles podemos ampliar con "Ctrl" + rueda del ratón)



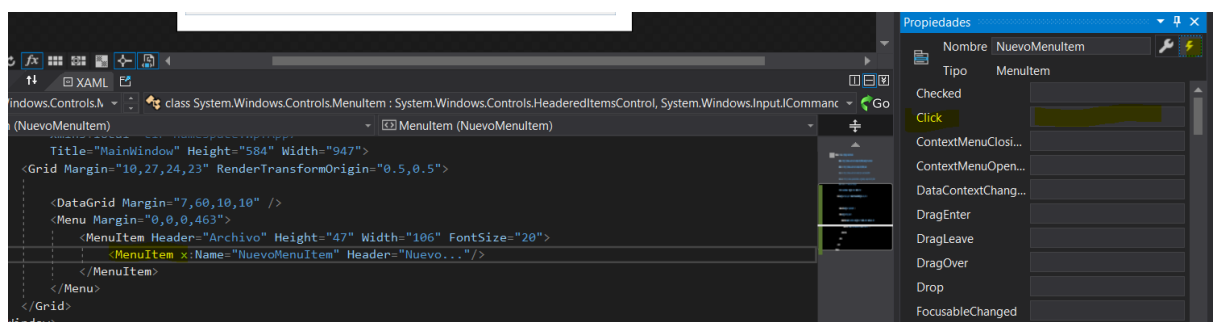
Le damos nombre a las variables: ButtonAceptar, ButtonCancelar, TextBoxTitulo, TextBoxAutor, etc.

Si queremos ajustar el tamaño de la ventana principal, podemos desplegar el menú lateral de "Esquema del documento" donde se ven todos los elementos de la pantalla:



Seleccionamos el elemento Windows y lo redimensionamos como queramos.

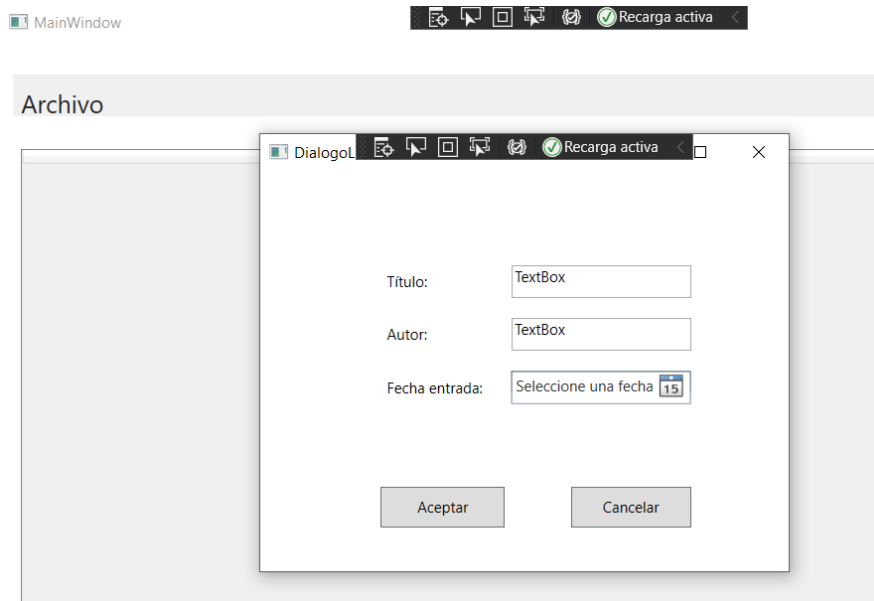
Ahora para comunicar las ventanas, solo tenemos que crear un evento en el submenu "Nuevo"



con el siguiente código:

```
private void NuevoMenuItem_Click(object sender, RoutedEventArgs e)
{
    DialogoLibro dialogoLibro = new DialogoLibro();
    dialogoLibro.Show();
}
```

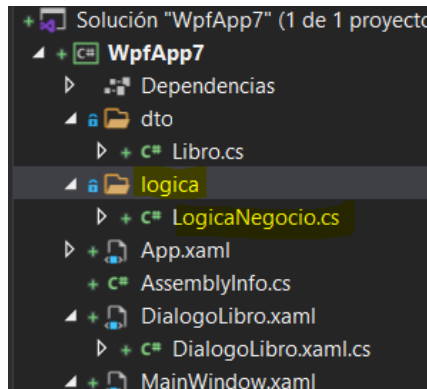
y si todo va bien, al ejecutar nos saldría la ventana principal, y por menú Archivo -> Nuevo.. nos abriría la nueva ventana:



Seguimos añadiendo funcionalidad: el botón “Cancelar” de DialogoLibro” tendría este código:

```
private void ButtonCancelar_Click(object sender, RoutedEventArgs e)
{
    this.Close();
}
```

Para implementar la lógica vamos a crear una nueva carpeta en el proyecto “logica” y un archivo C# que llamaremos LogicaNegocio:



Esta clase tendrá definida una colección de objetos libro (ObservableCollection), un constructor en el que provisionalmente vamos a añadir un libro para pruebas, y un método para añadir libros a la colección:

```

namespace WpfApp7.logica
{
    0 referencias
    public class LogicaNegocio
    {
        0 referencias
        public ObservableCollection<Libro> listaLibros { get; set; }

        0 referencias
        public LogicaNegocio()
        {
            listaLibros = new ObservableCollection<Libro>();
            listaLibros.Add(new Libro("Sistemas", "Marta", DateTime.Now));
        }

        0 referencias
        public void anadirLibro(Libro libro)
        {
            listaLibros.Add(libro);
        }
    }
}

```

Ahora tenemos que ajustar el archivo cs de DialogoLibro para que esté preparado para recibir un objeto de la clase “LogicaNegocio” por parámetro en el constructor, igual que hacíamos en java.

```

namespace WpfApp7
{
    /// <summary>
    /// Lógica de interacción para DialogoLibro.xaml
    /// </summary>
    4 referencias
    public partial class DialogoLibro : Window
    {
        private LogicaNegocio logicaNegocio;
        1 referencia
        public DialogoLibro(LogicaNegocio logicaNegocio)
        {
            InitializeComponent();
            this.logicaNegocio = logicaNegocio;
        }

        1 referencia
        private void ButtonCancelar_Click(object sender, RoutedEventArgs e)
        {
            this.Close();
        }
    }
}

```

Y ahora ajustamos el código de la ventana principal para que cuando abra la ventana de DialogoLibro (por la opción de menú) le pase por parámetro un objeto de logicaNegocio:


```

namespace WpfApp7
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    2 referencias
    public partial class MainWindow : Window
    {
        private LogicaNegocio logicaNegocio;
        0 referencias
        public MainWindow()
        {
            InitializeComponent();
            logicaNegocio = new LogicaNegocio();
        }

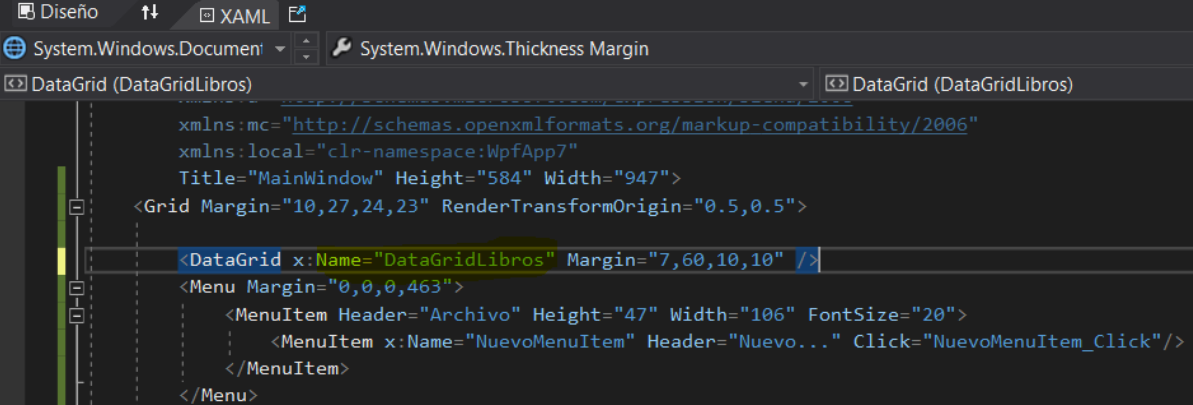
        1 referencia
        private void NuevoMenuItem_Click(object sender, RoutedEventArgs e)
        {
            DialogoLibro dialogoLibro = new DialogoLibro(logicaNegocio);
            dialogoLibro.Show();
        }
    }
}

```

Vamos a conectar ahora los elementos con “binding”:

En el código anterior añadimos: `DataGridLibros.DataContext = logicaNegocio`.

Antes de eso debemos darle nombre a la variable del DataGrid, que no lo habíamos hecho antes. Lo podemos hacer desde la ventana de propiedades seleccionando el DataGrid en el esquema del documento o en la pantalla, o en la propia etiqueta del elemento, añadiendo el atributo Name:



```

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:WpfApp7"
Title="MainWindow" Height="584" Width="947">
<Grid Margin="10,27,24,23" RenderTransformOrigin="0.5,0.5">
    <DataGrid x:Name="DataGridLibros" Margin="7,60,10,10" />
    <Menu Margin="0,0,0,463">
        <MenuItem Header="Archivo" Height="47" Width="106" FontSize="20">
            <MenuItem x:Name="NuevoMenuItem" Header="Nuevo..." Click="NuevoMenuItem_Click"/>
        </MenuItem>
    </Menu>
</Grid>

```

Ahora sí, ya podemos añadir la línea:

```

public partial class MainWindow : Window
{
    private LogicaNegocio logicaNegocio;
    0 referencias
    public MainWindow()
    {
        InitializeComponent();
        logicaNegocio = new LogicaNegocio();
        DataGridLibros.DataContext = logicaNegocio;
    }

    1 referencia
    private void NuevoMenuItem_Click(object sender, RoutedEventArgs e)
    {
        DialogoLibro dialogoLibro = new DialogoLibro(logicaNegocio);
        dialogoLibro.Show();
    }
}

```

Y el Binding lo hacemos en la etiqueta del DataGrid, a través del atributo ItemsSource:

```

<Grid Margin="10,27,24,23" RenderTransformOrigin="0.5,0.5">
    <DataGrid x:Name="DataGridLibros" ItemsSource="{Binding Path=listaLibros}" Margin="7,60,10,10" />
    <Menu Margin="0,0,0,463">
        <MenuItem Header="Archivo" Height="47" Width="106" FontSize="20">
            <MenuItem x:Name="NuevoMenuItem" Header="Nuevo..." Click="NuevoMenuItem_Click"/>
        </MenuItem>
    </Menu>
</Grid>
</Window>

```

Si ejecutamos el código, ya debería verse cargado el libro de prueba que dejamos en código:

MainWindow

Recarga activa

Archivo

Titulo	Autor	FechaEntrada	
Sistemas	Marta	11/8/2022 7:23:03 PM	

Resumiendo lo que hemos hecho:

El objeto Libro es el que almacenamos, LogicaNegocio es el que tiene las listas de objetos (libros), MainWindow que la ventana principal, tiene un DataGrid con un binding a la lista LogicaNegocio, y en DialogoLibro añadimos registros nuevos desde el botón de Aceptar.

Nos quedaría la lógica del botón aceptar para añadir más libros a la tabla. Para ello primero tenemos que añadir a nuestra clase Libro un **constructor** para crear un libro con todos los campos vacíos menos la fecha, que dejaremos la actual.

```
0 referencias
public Libro()
{
    this.fechaEntrada = DateTime.Now;
}

1 referencia
public Libro(String titulo, String autor, DateTime fechaEntrada)
{
    this.titulo = titulo;
    this.autor = autor;
    this.fechaEntrada = fechaEntrada;
}
```


Ahora en el código de DialogoLibro definimos un objeto Libro, en el constructor de DialogoLibro lo inicializamos con el constructor Libro() que acabamos de definir sin parámetros, y lo asignamos al contexto de la pantalla:

```
4 referencias
public partial class DialogoLibro : Window
{
    private LogicaNegocio logicaNegocio;
    public Libro libro;
    1 referencia
    public DialogoLibro(LogicaNegocio logicaNegocio)
    {
        InitializeComponent();
        this.logicaNegocio = logicaNegocio;
        libro = new Libro();
        this.DataContext = libro;
    }
}
```

Ahora vamos a conectar en las etiquetas de XAML las cajas de texto con el datos del contexto, que es el objeto libro. Primero borramos los textos:

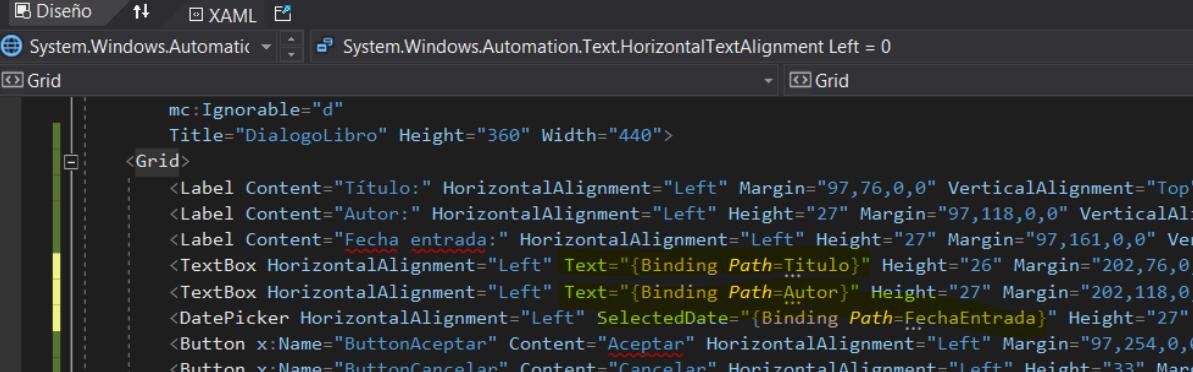
Título:

Autor:

Fecha entrada: 

porque vamos a utilizar el atributo Text para el binding y podría dar error si dejamos el mismo atributo definidos dos veces en la misma etiqueta. Y ahora ajustamos el binding:

Como el contexto de la ventana ya lo ajustamos al objeto libro, en el binding podemos enganchar directamente con los atributos del contexto (libro) -> Título, Autor y fecha. En los dos primeros campos el binding es por el atributo **Text** (el que hemos borrado antes), y en el campo de fecha el atributo de binding sería **SelectedDate**.



```

mc:Ignorable="d"
Title="DialogoLibro" Height="360" Width="440">
<Grid>
<Label Content="Título:" HorizontalAlignment="Left" Margin="97,76,0,0" VerticalAlignment="Top"
<Label Content="Autor:" HorizontalAlignment="Left" Height="27" Margin="97,118,0,0" VerticalAli
<Label Content="Fecha entrada:" HorizontalAlignment="Left" Height="27" Margin="97,161,0,0" Ver
<TextBox HorizontalAlignment="Left" Text="{Binding Path=Título}" Height="26" Margin="202,76,0,
<TextBox HorizontalAlignment="Left" Text="{Binding Path=Autor}" Height="27" Margin="202,118,0,
<DatePicker HorizontalAlignment="Left" SelectedDate="{Binding Path=FechaEntrada}" Height="27"
<Button x:Name="ButtonAceptar" Content="Aceptar" HorizontalAlignment="Left" Margin="97,254,0,0
<Button x:Name="ButtonCancelar" Content="Cancelar" HorizontalAlignment="Left" Height="33" Marg

```

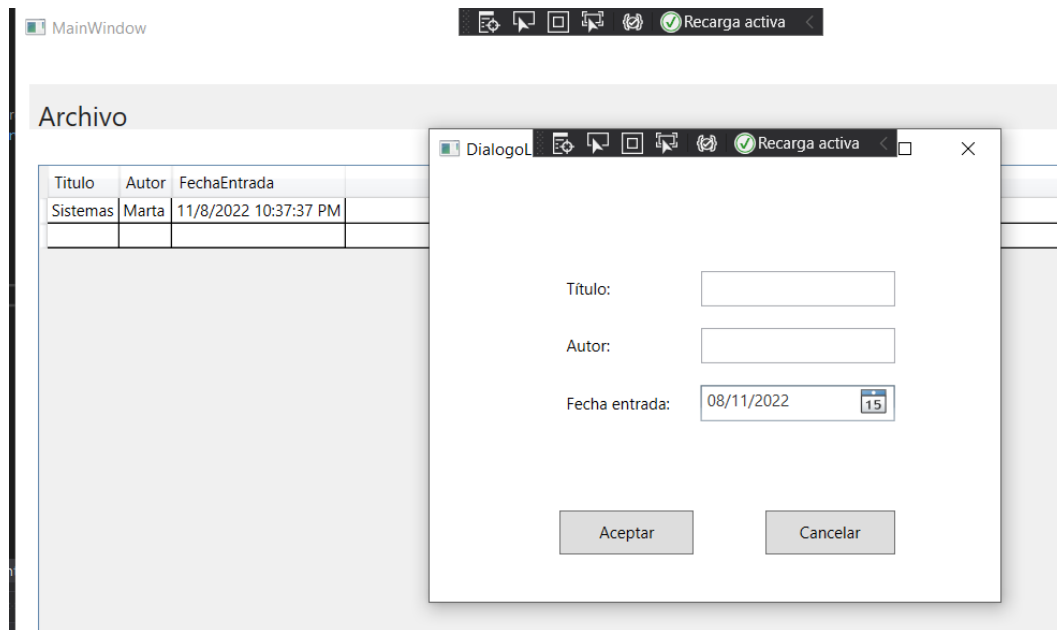
Lo últimos que nos queda ya es definir el método de Aceptar para cargar el libro en la tabla: Llamamos al método de añadir libro de la clase libro, le pasamos el objeto libro que está conectado a los campos de la pantalla de alta y luego inicializamos el objeto libro con el constructor vacío (dejamos los campos en blanco y la fecha la actualizamos a la actual) y lo volvemos a asignar al contexto de la ventana para el siguiente registro que se vaya crear:

```

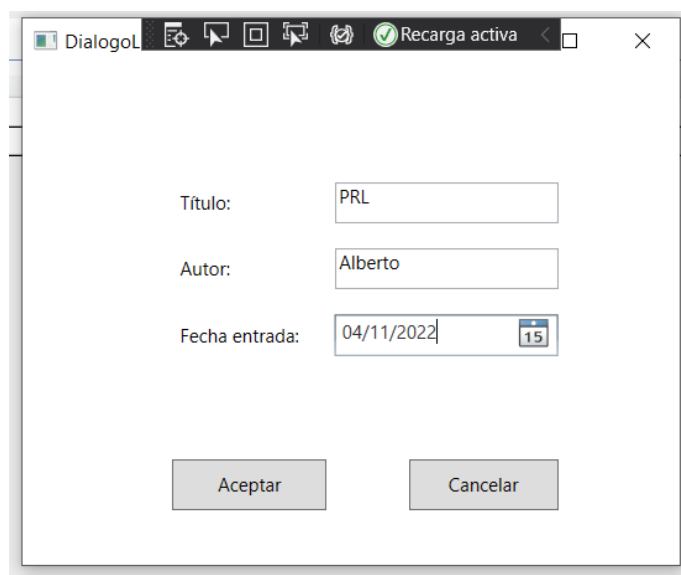
private void ButtonAceptar_Click(object sender, RoutedEventArgs e)
{
    logicaNegocio.anadirLibro(libro);
    libro = new Libro();
    this.DataContext = libro;
}

```

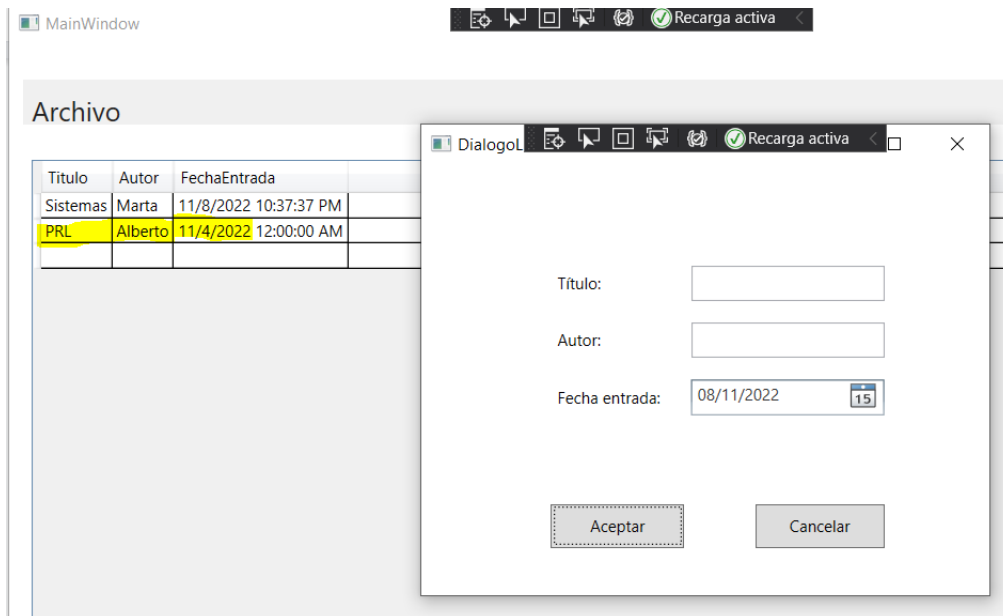
Si ejecutamos el proyecto vemos que se nos carga el registro de prueba, abrimos por menu Archivo -> Nuevo la ventana de alta, y vemos que aparece actualizada la fecha actual correctamente:



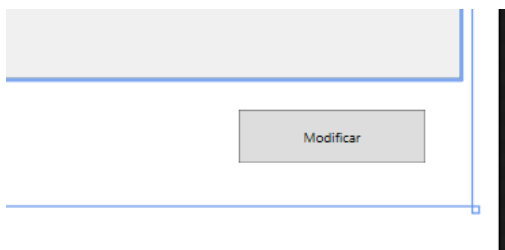
Indicamos los datos que queremos cargar:



Y al “Aceptar” vemos que los datos se carga en la tabla y se limpian los campos de texto y se refresca la fecha a la actual:



Vamos a implementar ahora la opción de modificar los registros. Para ello añadimos un botón “modificar” en la ventana principal, debajo del DataGrid, y llamaremos la misma ventana de alta, pero con los datos del libro seleccionado, al pulsar el botón modificar.



En el archivo LogicaNegocio definimos un método para modificar un libro:

```
public void modificarLibro(Libro libro, int posicion)
{
    listaLibros[posicion] = libro;
}
```

Ahora en el código C# de la ventana DialogoLibro definimos una variable booleana “modificar” para saber si estamos creando un registro o modificándolo. También declaramos una variable de posición del libro en la tabla:

```
public partial class DialogoLibro : Window
{
    private LogicaNegocio logicaNegocio;
    public Libro libro;
    private int posicion;
    private Boolean modificar;
```

Definimos otro constructor de DialogoLibro con más parámetros; además del objeto logicaNegocio, le pasamos el objeto libroModificar y la posición del mismo. Dentro del constructor antiguo inicializamos la variable modificar a false y en el constructor ampliado inicializamos la variable modificar a true:

```
public DialogoLibro(LogicaNegocio logicaNegocio)
{
    InitializeComponent();
    this.logicaNegocio = logicaNegocio;
    libro = new Libro();
    this.DataContext = libro;
    modificar = false;
}
2 referencias
public DialogoLibro(LogicaNegocio logicaNegocio, Libro libroModificar, int posicion)
{
    InitializeComponent();
    this.logicaNegocio = logicaNegocio;
    this.libro = libroModificar;
    this.posicion = posicion;
    this.DataContext = libro;
    modificar = true;
}
```

Y en el método del botón Aceptar, según el valor de “modificar” llamamos a un método o a otro y en vez de limpiar campos para añadir más libros cerramos la ventana directamente.

```
private void ButtonAceptar_Click(object sender, RoutedEventArgs e)
{
    if (modificar)
        logicaNegocio.modificarLibro(libro, posicion);
    else
        logicaNegocio.anadirLibro(libro);

    this.Close();
}
```

Por último implementamos el botón de “Modificar” de la ventana principal:

```
private void ButtonModificar_Click(object sender, RoutedEventArgs e)
{
    if(DataGridLibros.SelectedIndex != -1)
    {
        Libro libro = (Libro)DataGridLibros.SelectedItem;
        DialogoLibro dialogoLibro = new DialogoLibro(logicaNegocio, libro, DataGridLibros.SelectedIndex);
        dialogoLibro.Show();
    }
}
```

Nos aseguramos que haya algún libro seleccionado primero, y en caso afirmativo lo guardamos en un objeto libro, que luego le pasamos al constructor de la ventana DialogoLibro y mostramos la ventana.

Tal y como está, ya permitiría modificar registros, pero al utilizar binding, si en la ventana de modificación cambiamos algún dato del libro, los cambios se pasan automáticamente al libro de la tabla (al cambiar de un campo a otro), aunque luego pulsemos el botón de cancelar.

Para evitar esto vamos a utilizar la clonación del objeto libro utilizando la interfaz ICloneable:

```
12 referencias
public class Libro : INotifyPropertyChanged, ICloneable
{
```

que me genera el método Clone

```
public object Clone()
{
    throw new NotImplementedException();
}
```

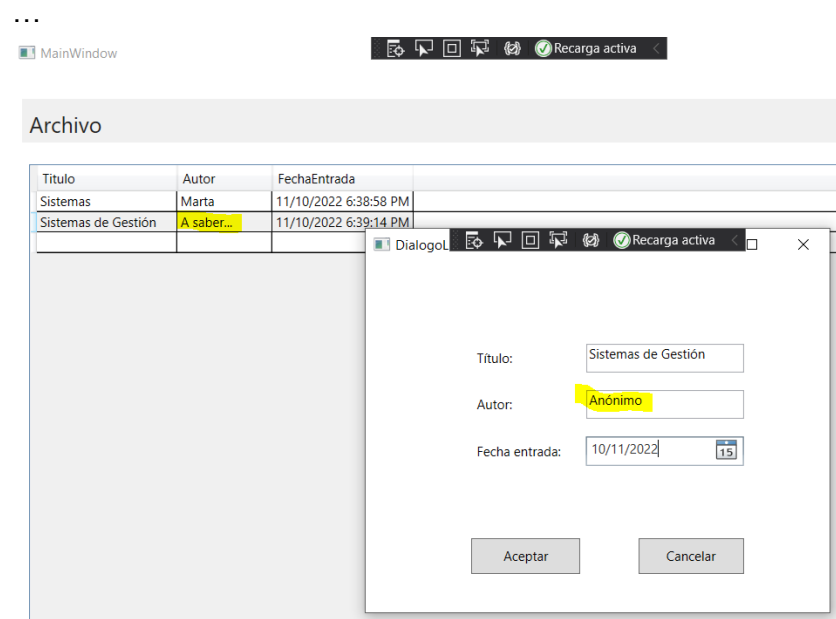
que modificamos de la siguiente forma:

```
public object Clone()
{
    return this.MemberwiseClone();
}
```

y en el constructor ampliado le pasamos un clon del libro seleccionado, de forma que las modificaciones realizadas no se pasarán al registro de la tabla hasta que pulsemos aceptar.

```
private void ButtonModificar_Click(object sender, RoutedEventArgs e)
{
    if(DataGridLibros.SelectedIndex != -1)
    {
        Libro libro = (Libro)DataGridLibros.SelectedItem;
        DialogoLibro dialogoLibro = new DialogoLibro(logicaNegocio, (Libro)libro.Clone(), DataGridLibros.SelectedIndex);
        dialogoLibro.Show();
    }
}
```

Si probamos:



Ya no se pasan los datos modificados al registro porque estamos modificando un clon del libro seleccionado, y sólo cuando pulsemos aceptar se sustituirá el libro del registro por el clon modificado.

Vamos a **validar los datos de un formulario** implementando la interfaz **IDataErrorInfo** en el objeto Libro:

```
public class Libro : INotifyPropertyChanged, ICloneable, IDataErrorInfo
```

Esta interfaz nos genera los siguientes métodos:

```
0 referencias
public string Error => throw new NotImplementedException();

0 referencias
public string this[string columnName] => throw new NotImplementedException();
```

que ajustamos de la siguiente manera:

```
public string Error
{
    get { return ""; }
}

0 referencias
public string this[string columnName]
{
    ....
    get
    {
        string result = "";
        if (columnName == "Titulo")
        {
            if (string.IsNullOrEmpty(titulo))
                result = "Debe introducir el título";
        }
        if (columnName == "Autor")
        {
            if (string.IsNullOrEmpty(autor))
                result = "Debe introducir el autor";
        }
        return result;
    }
}
```

Y en el XAML de la ventana de DialogoLibro tenemos que poner a True los atributos:

NotifyOnValidationError y ValidatesOnDataErrors en los bindings de los TextBox que estamos validando:

```
Diseño  XAML
WpfApp7.DialogoLibro public partial class DialogoLibro : Window{...}
Window
Grid
<Label Content="Título:" HorizontalAlignment="Left" Margin="97,76,0,0" VerticalAlignment="Top" RenderTransformOrigin="-0.50
<Label Content="Autor:" HorizontalAlignment="Left" Height="27" Margin="97,118,0,0" VerticalAlignment="Top" Width="64"/>
<Label Content="Fecha entrada:" HorizontalAlignment="Left" Height="27" Margin="97,161,0,0" VerticalAlignment="Top" Width="1
<TextBox HorizontalAlignment="Left" Text="{Binding Path=Título, NotifyOnValidationError=True, ValidatesOnDataErrors=True}"
<TextBox HorizontalAlignment="Left" Text="{Binding Path=Autor, NotifyOnValidationError=True, ValidatesOnDataErrors=True}" H
<DatePicker HorizontalAlignment="Left" SelectedDate="{Binding Path=FechaEntrada}" Height="27" Margin="202,161,0,0" Vertical
<Button x:Name="ButtonAceptar" Content="Aceptar" HorizontalAlignment="Left" Margin="97,254,0,0" VerticalAlignment="Top" Hei
<Button x:Name="ButtonCancelar" Content="Cancelar" HorizontalAlignment="Left" Height="33" Margin="250,254,0,0" VerticalAlig
/Grid>
```

Si probamos la validación, al abrir la ventana de DialogoLibro, aparecen los cuadros de texto con el borde en rojo porque están vacíos.

Vamos a condicionar la activación del botón Aceptar a que no haya ningún error de validación:

```
<TextBox HorizontalAlignment="Left" Validation.Error="Validation_Error" Text="{Binding Path=Título, NotifyOnValidationError=True, ValidatesOnDataErrors=True}"
<TextBox HorizontalAlignment="Left" Validation.Error="Validation_Error" Text="{Binding Path=Autor, NotifyOnValidationError=True, ValidatesOnDataErrors=True}"
<DatePicker HorizontalAlignment="Left" SelectedDate="{Binding Path=FechaEntrada}" Height="27" Margin="202,161,0,0" Vertical
```

Añadimos un atributo "Validación.Error" en los dos campos que estamos validando y le asignamos una función, que definimos en el código de DialogoLibro.

Necesitamos definir una variable errores que utiliza la nueva función:


```
public partial class DialogoLibro : Window
{
    private LogicaNegocio logicaNegocio;
    public Libro libro;
    private int posicion;
    private Boolean modificar;
    private int errores;
```

y la función quedaría:

```
private void Validation_Error(object sender, ValidationErrorEventArgs e)
{
    if (e.Action == ValidationErrorEventAction.Added)
        errores++;
    else
        errores--;


    if (errores == 0)
        ButtonAceptar.IsEnabled = true;
    else
        ButtonAceptar.IsEnabled = false;
}
```

Según los errores que se contabilicen se activará o no el botón de aceptar.

DialogoL  Recarga activa


Título:

Autor:

Fecha entrada: 10/11/2022 


Aceptar Cancelar

Si informamos los dos campos que se validan, ya se habilita el botón de aceptar:

DialogoL  Recarga activa

Título:

Autor:

Fecha entrada: 10/11/2022 

Aceptar Cancelar

