DESARROLLO DE APLICACIONES WINDOWS

# 1. Presentación de WPF

Las aplicaciones Windows, son aplicaciones de ventana que presentan datos de forma gráfica.

La tecnología de presentación **WPF** (Windows Presentation Foundaton), se incluyó desde la versión 3.0 del framework .NET.

A nivel técnico:

* La definición de la interfaz se realiza mediante el lenguaje de etiquetas **XAML** (eXtensible Application Markup Language), y no mediante código C#.
* Se **desacopla** la interfaz gráfica del código gracias a la noción de **binding**.
* La **visualización** se basa en el componente de software **DirectX**, lo que implica que algunos cálculos puede realizarlos la **CPU**.
* Todos los componentes de WPF utilizan el **diseño vectorial**.
* En un componente, el aspecto gráfico y el aspecto funcional están muy poco ligados, lo que permite modificar la parte gráfica de los componentes sin afectar a su comportamiento.

## Estructura de una aplicación WPF

En el momento de su creación, un proyecto WPF (WPF App en Visual Studio) está compuesto por varios archivos:

* Los archivos **App.xaml** y **App.xaml.cs**, que componen la declaración y la implementación de la clase principal de la aplicación: de hecho, contiene el punto de entrada al programa. El archivo .xaml contiene, a su vez, todos los recursos necesarios para el correcto funcionamiento de la aplicación, mientras que el archivo de code-behind asociado contiene los controladores de eventos relativos al ciclo de vida de la aplicación o las secciones de código necesarias para su inicio.
* Los archivos **MainWindow.xaml** y el archivo de code-behind **MainWindows.xaml.cs** contienen, respectivamente, la descripción de la ventana principal de la aplicación y el código C# asociado.

## XAML

La tecnología WPF utiliza el lenguaje XAML para la creación de interfaces gráficas. Este lenguaje es un dialecto de XML que permite instanciar objetos .NET de manera declarativa, es decir, mediante el uso de etiquetas, de manera similar a como lo hace HTML. Los distintos objetos están anidados a partir de un elemento raíz, formando un árbol lógico.

La definición de un botón mediante código XAML tiene el siguiente aspecto:

<Button Height=”30” Width=”120”>Esto es un botón</Button>

La etiqueta <Button> instancia un objeto de tipo Button, mientras que los atributos XML asignan valor a las propiedades Height y Width del botón. Como cualquier elemento XML, una etiqueta XAML puede contener otras etiquetas o un valor textual.

Es importante saber que todo lo que puede codificarse mediante XAML puede codificarse también con C#.

El equivalente imperativo del código XAML anterior es el siguiente:

Button boton=new Button();

boton.Height=30;

boton.Width=120;

boton.Content=”Esto es un botón”;

Vemos una nueva propiedad **Content** para la que no se proporcionaba ningún valor, de manera explícita, en XAML. Esta propiedad es, de hecho, la **propiedad por defecto** para el tipo Button, lo que significa que el valor textual o XAML situado en el interior de la etiqueta Button asigna el valor a la propiedad Content del objeto Button.

La **sintaxis elemento-propiedad** permite asignar un contenido XAML complejo a una propiedad. Autoriza el uso de una propiedad como elemento XML anidado. El nombre del elemento debe mostrar el tipo sobre el que se opera, seguido de un punto y, a continuación, el nombre de la propiedad correspondiente. Según esta sintaxis, el código XAML de creación de un botón que hemos visto antes puede escribirse de la siguiente manera:

<Button Height=”30” Width=”120”>

<Button.Content>Esto es un botón</Button.Content>

</Button>

En cambio, no todo el código C# tiene su equivalente en XAML, pues es imposible, en particular, invocar a un método directamente de manera declarativa. Si bien XAML es muy potente gracias a la gran cantidad de objetos que permite utilizar, los escenarios avanzados requieren, a menudo, el uso de código .NET.

### Plantillas

Los objetos se definen bajo la forma de un grafo que compone el árbol lógico de una pantalla. Este árbol es distinto del árbol visual, que es el resultado de la transformación del árbol lógico tras la aplicación de distintas plantillas sobre cada control.

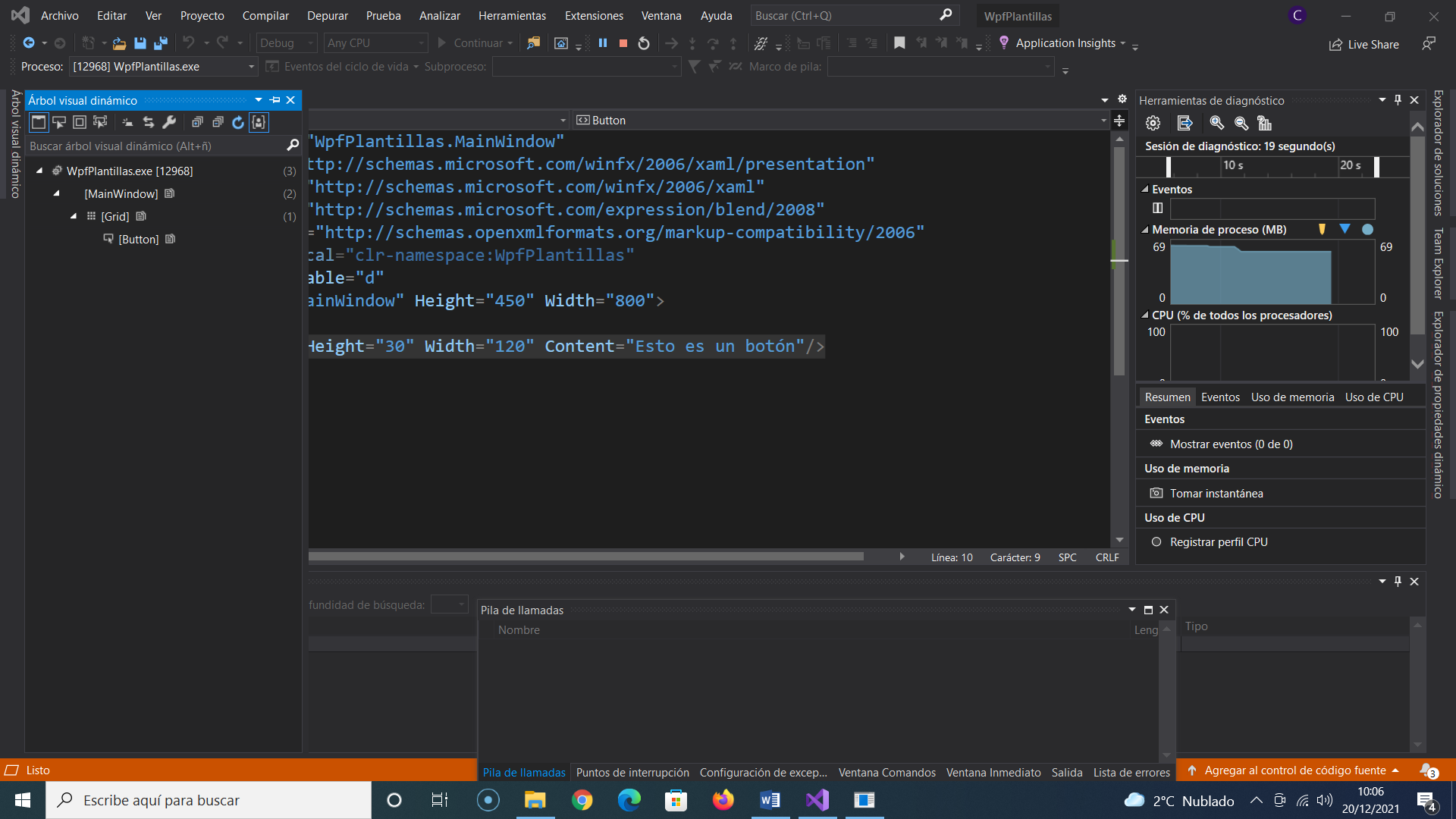
La parte visual de un control gráfico se define en XAML mediante la creación de un objeto de tipo ControlTemplate. Se trata de una plantilla de diseño que representa el resultado deseado mediante un árbol formado por objetos más simples. Se aplica al control tras su carga.

El árbol lógico siguiente es extremadamente sencillo: contiene una única etiqueta.

<Button Height="30" Width="120" Content="Esto es un botón"/>

Ejecuto el programa.

El árbol visual correspondiente es algo más complejo:



Observación:

Esta representación del árbol visual de un botón se obtiene mediante la ventana **Árbol visual dinámico** de Visual Studio. Esta puede abrirse desde la depuración de una aplicación WPF en el menú **Depurar – Ventanas – Árbol visual dinámico**.

### Espacios de nombres

Como ocurre en el código C#, es necesario importar los espacios de nombres en el código XAML para poder utilizar los tipos que contienen. Para realizar esta operación no es posible, utilizar una declaración using de C#, sino las declaraciones de espacios de nombres de XML.

Estas declaraciones son atributos XML que respetan una sintaxis particular:

xmlns:<prefijo>=”clr-namespace:<espacio de nombres>

En cada ventana WPF son obligatorios dos espacios de nombres, y deben estar presentes en el elemento raíz. El primero importa el framework WPF y su implementación de XAML, mientras que el segundo importa varios elementos globales de XAML.

Estas dos declaraciones permiten utilizar las bases de WPF en el conjunto del documento en edición.

xmlns=<http://schemas.microsoft.com/winfx/2006/xaml/presentation>

xmlns:x=<http://schemas.microsoft.com/winfx/2006/xaml>

Cuando se crea una ventana mediante la plantilla presente en Visual Studio, su declaración ya incluye estos espacios de nombres de manera predeterminada.

## Contexto de datos y binding

El concepto de **binding** es, posiblemente, el más importante para realizar un uso óptimo de las capacidades de WPF. Se corresponde con un **vínculo unidireccional o bidireccional** entre un **elemento XAML** y un **contexto de datos (DataContext)** y permite propagar datos del código C# hacia el código XAML (y/o a la inversa) sin que el desarrollador tenga que realizar una transferencia de información mediante código C#. Este mecanismo es la clave del **desacople** entre el **código** de negocio y la **interfaz gráfica**.

Se **define un vínculo** **de datos** en el código XAML mediante una **expresión** delimitada por **llaves** y cuyo primer término es la palabra clave **Binding**. Este vínculo puede estar seguido de una lista de **parámetros** que definen el **origen de datos**, la **ruta** de la **propiedad** con la que se debe establecer el vínculo o incluso el **modo de enlace**.

Observación:

Por defecto, un binding transmite el resultado de la llamada al método ToString del objeto. Para objetos complejos, el resultado que se muestra si no se ha redefinido el método, es el nombre completo de su tipo.

Crea un nuevo proyecto WPF en Visual Studio, selecciona la opción WPF App (.NET Core), y llámalo PruebasBinding.

Añade el siguiente código:

<Window x:Class="PruebasBinding.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:PruebasBinding"

mc:Ignorable="d"

Title="MainWindow" Height="450" Width="800">

<Grid>

<TextBox Height="30" Width="120" Text="{Binding Path=Nombre,

Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}">

</TextBox>

</Grid>

</Window>

Este código crea una zona para introducir texto (**TextBox**) en una ventana y asocia a su propiedad **Text** un vínculo bidireccional (**Mode=TwoWay**) hacia la **propiedad nombre** del contexto de datos (**Path=Nombre**). La transferencia de información del control hacia el contexto de datos se realizará cada vez que la propiedad sobre la que se haya definido el binding sufra alguna modificación (**UpdateSourceTrigger=PropertyChanged**).

Al ejecutar el programa, aparece el cuadro de texto vacío. No se ha definido ningún contexto de datos y, por tanto, no existe ninguna fuente de datos especificada explícitamente. Para definir el contexto de datos de la ventana es preciso asignar un valor a su propiedad DataContent.

Edita el constructor de la ventana, en **MainWindow.xaml.cs**, para **definir el contexto de datos** de la ventana como la **propia ventana**.

public MainWindow()

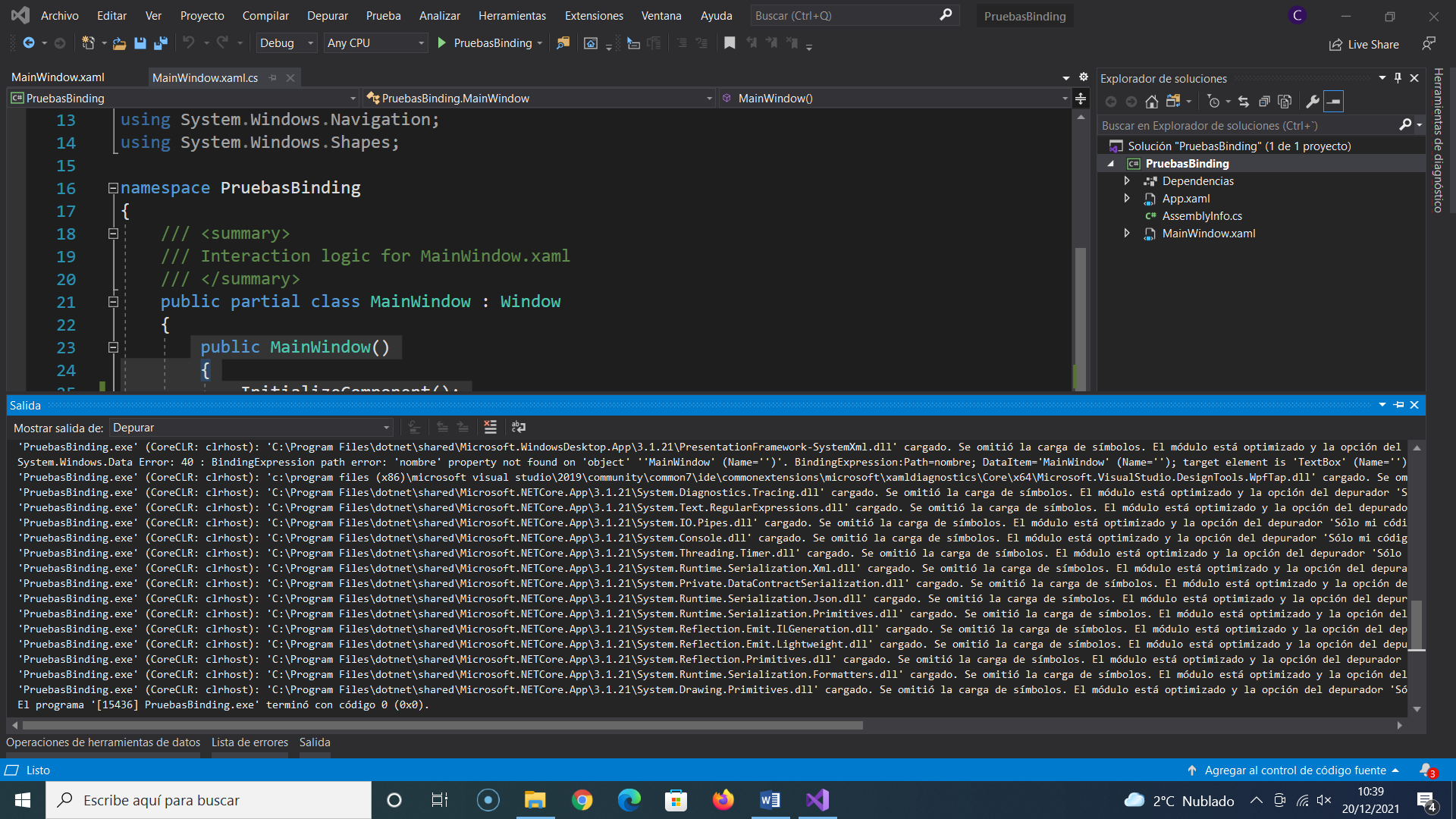
{

InitializeComponent();

this.DataContext = this;

}

Al arrancar la aplicación, la interfaz se comportará exactamente de la misma manera. En cambio, la ventana Salida de Visual Studio presenta una nueva línea de información:



En los mensajes se indica que se ha producido un **error** en la creación del vínculo de datos, pues la propiedad **Nombre** no se encuentra en el objeto **MainWindows** que se utiliza como contexto de datos. Provee, también, información que permite identificar el objeto XAML y la propiedad sobre la que se ha situado el vínculo erróneo.

Vamos a **agregar la definición de la propiedad** siguiente en la case MainWindow de cara a corregir el error.

Para visualizar el correcto funcionamiento del vínculo, agrega una asignación del valor a la propiedad Nombre en el constructor de la clase.

public partial class MainWindow : Window

{

private string nombre;

public string Nombre

{

get { return nombre; }

set { nombre = value; }

}

public MainWindow()

{

InitializeComponent();

Nombre="Celia";

this.DataContext = this;

}

}

Con esta configuración, la aplicación muestra una zona de texto en la que se presenta el valor de la propiedad Nombre, pero las modificaciones realizadas sobre este valor tras un clic en un botón, por ejemplo, no se reflejarán en el control TextBox a pesar de la existencia de un vínculo de datos.

Para comprobarlo, agrega la siguiente línea justa a continuación de la declaración del control TextBox en MainWIndow.xaml:

<Grid>

<TextBox Height="30" Width="120" Text="{Binding Path=Nombre,

Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}">

</TextBox>

<Button Height="30" Width="120" VerticalAlignment="Top"

Click="Button\_Click"

Content="Modificar el contenido del área de texto">

</Button>

</Grid>

Agregamos el controlador de eventos siguiente a la clase MainWindow:

private void Button\_Click(object sender, RoutedEventArgs e)

{

Nombre = "Nombre modificado";

}

La no propagación de la modificación es, de hecho, normal. La tecnología WPF utiliza el **evento PropertyChanged** de la **interfaz INotifyPropertyChanged** para informar a los elementos gráficos acerca de las **modificaciones** de valor en sus propiedades. Es necesario, a continuación, que el objeto que se utiliza como origen de datos (aquí, la clase MainWindow) implemente esta interfaz.

Hay que incluir la siguiente línea:

using System.ComponentModel;

A continuación, se muestra el código completo de la clase MainWindow una vez implementada esta interfaz:

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;

namespace PruebasBinding

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window, INotifyPropertyChanged

{

private string nombre;

public string Nombre

{

get { return nombre; }

set {

nombre = value;

OnPropertyChanged("Nombre");

}

}

public MainWindow()

{

InitializeComponent();

this.DataContext = this;

}

private void Button\_Click(object sender, RoutedEventArgs e)

{

Nombre = "Nombre modificado";

}

private void OnPropertyChanged(string nombrePropiedad)

{

if (PropertyChanged != null)

PropertyChanged(this, new PropertyChangedEventArgs(nombrePropiedad));

}

public event PropertyChangedEventHandler PropertyChanged;

}

}