# ACTIVIDAD GUIADA 1
## Introducción a WPF



---

*Windows Presentation Foundation* (**WPF**) es un sistema de visualización gráfica para Windows. Supone un cambio radical respecto a las tecnologías anteriores, incluyendo cambios como la aceleración de hardware y la independencia de la resolución de pantalla.

WPF nos permite crear aplicaciones enriquecidas de escritorio en Windows prescindiendo de los antiguos controles **Windows Forms** que daban una apariencia plana a las aplicaciones. En comparación con **Metro** (que sólo permite construir aplicaciones para Windows 8), **WPF** permite construir aplicaciones para Windows 7, Vista, 8, e incluso Windows XP.

WPF es una tecnología madura, que ha ido pasando por distintas versiones. El número de versión se corresponde con la versión de .NET Framework sobre la cual funciona. La versión actual es WPF 4.5, luego funciona sobre la **versión 4.5 del Framework .NET.**

Microsoft ha incluido .NET en todas las versiones de Windows desde Vista, y las nuevas versiones del framework se actualizan a través de **Windows Update.** Es decir, los usuarios de Windows podrán ejecutar las aplicaciones WPF.

# Evolución de los gráficos en Windows

## Antecedentes

Antes de WPF, los desarrolladores Windows pasaron casi 15 años utilizando la misma tecnología de pantalla. Eso es porque Windows siempre se basaba en dos componentes que no evolucionaron demasiado desde sus primeras versiones:

- **User32**: Proporciona el aspecto tradicional de Windows a elementos como ventanas, botones, cajas de texto, etc.
- **GDI/GDI+:** Permite el dibujado de formas, texto e imágenes. Las imágenes son convertidas a mapa de bits, y enviadas en ese formato a la tarjeta gráfica. Esto provoca una merma en el rendimiento gráfico de las aplicaciones.

Con los años, ambas tecnologías se refinaron, y las APIs que los desarrolladores utilizaban para interactuar con ellas evolucionaron. Sin embargo, detrás de una aplicación, siempre teníamos estas tecnologías que limitaban las posibilidades de la misma.

## DirectX: El nuevo motor gráfico

**DirectX** comenzó como un framework para la creación de juegos en la plataforma Windows. Uno de sus objetivos era la velocidad, luego Microsoft trabajó estrechamente con fabricantes de tarjetas gráficas para que DirectX dispusiera de la aceleración hardware necesaria para recrear texturas complejas, efectos especiales (como la transparencia parcial) y gráficos tridimensionales.

A lo largo de los años DirectX ha madurado. Ahora es parte integral de Windows, con soporte para todas las tarjetas de vídeo modernas, y su estabilidad ha mejorado notablemente.

Sin embargo, la API de programación para DirecX todavía refleja sus raíces de kit de herramientas para desarrollo de juegos. Debido a su gran complejidad, DirectX casi nunca se usa en aplicaciones tradicionales para Windows.

En WPF, la tecnología de gráficos subyacente se sustituye por **DirectX**, de forma que disponemos de aceleración hardware en nuestras aplicaciones de escritorio (DirectX envía todo el trabajo posible a la unidad de procesamiento gráfico (GPU), liberando a la CPU). Además, como resultado, incluso en las aplicaciones empresariales más simples podemos utilizar efectos como la transparencia.

# WPF: Una API de alto nivel

WPF incluye una serie de servicios de alto nivel diseñados para los programadores de aplicaciones. Los siguientes son algunos de los cambios más importantes que WPF introduce en el mundo de la programación en Windows.

- Un **modelo de diseño** similar a la Web. En lugar de fijar los controles en el lugar con coordenadas estáticas, WPF usa **coordenadas relativas**. El resultado es una interfaz de usuario que escala perfectamente.
- Un **modelo de dibujo** mucho más rico que GDI+. En lugar de pintar píxeles, todo lo que pintamos en WPF se reduce a **primitivas** (formas básicas como elipses, cuadrados, etc.)

con las que se construyen los controles. También disponemos de nuevas características de dibujo, tales como el soporte 3D nativo.

- Un nuevo **modelo de texto enriquecido** Podemos mostrar texto enriquecido con varios formatos en cualquier lugar de la interfaz de usuario. Si necesitamos mostrar grandes cantidades de texto, podemos usar funciones de visualización de documentos avanzadas.
- Soporte para **animaciones, audio y vídeo**. Se incluye soporte nativo para la reproducción de cualquier archivo de audio o vídeo compatible con Windows Media Player.
- **Estilos y plantillas** que permiten estandarizar el formato y volver a utilizarlo en toda la aplicación.
- Uso de **comandos** para permitir que distintos elementos de la interfaz realicen la misma tarea.
- **Aplicaciones basadas en páginas**. WPF permite crear aplicaciones similares a las que usamos en los navegadores Web, formadas por una colección de páginas, con botones de avance y retroceso.
- **Interfaz de usuario declarativa**. Aunque podemos construir una ventana de WPF con código, habitualmente utilizamos un enfoque algo distinto. Se serializa el contenido de cada ventana en un conjunto de etiquetas XML, formando un archivo XAML. La ventaja es que la interfaz de usuario está completamente **separada** de su código, y los diseñadores gráficos pueden utilizar herramientas específicas para editar la interfaz gráfica (archivos XAML) y refinar la apariencia de la aplicación sin necesidad de tocar el código de ésta.

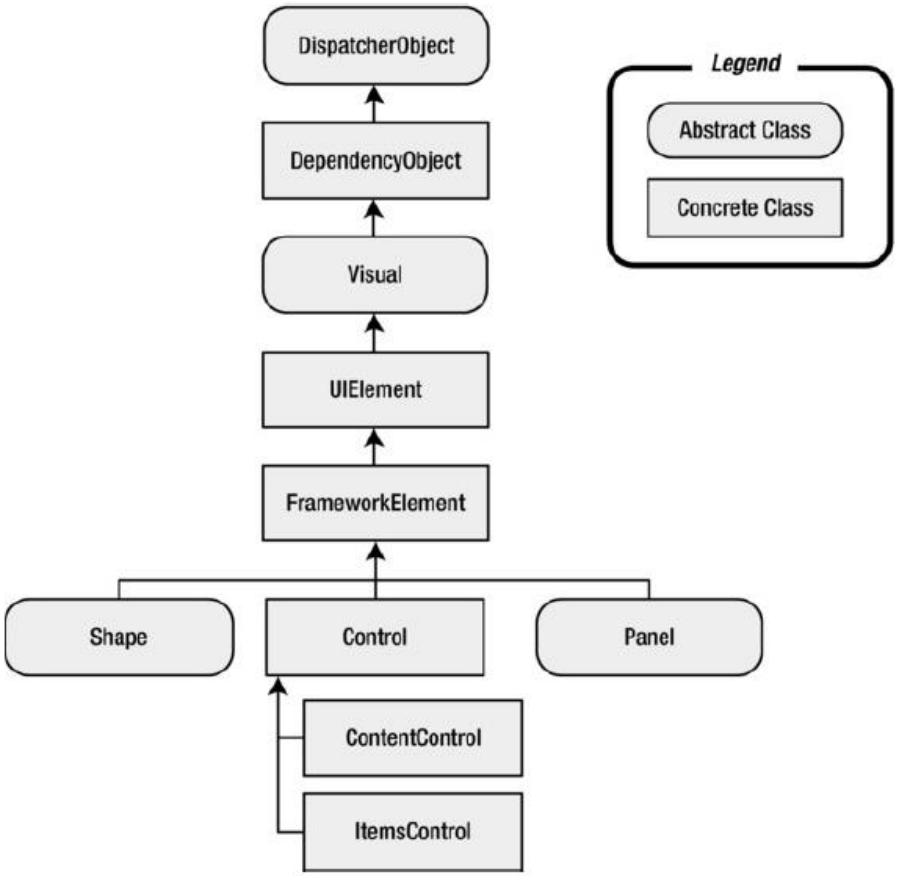
## Independencia de la resolución

El gran problema en las aplicaciones tradicionales de Windows es que la interfaz de usuario no es escalable. Como resultado, si usamos un monitor de alta resolución con muchos más píxeles por pulgada, la ventana de la aplicación tendrá controles pequeños y difíciles de leer. Esto se traduce en una gran fatiga visual para el usuario. El escenario opuesto se produce cuando diseñamos una aplicación pensando en un monitor de gran resolución, y luego es usada en un monitor con una resolución inferior. La aplicación puede llegar a no caber en pantalla.

WPF basa su escala en la configuración de PPP del sistema, de modo que ajusta el tamaño real de los controles a las preferencias del usuario. Tanto las ventanas como WPF se miden *unidades independientes del dispositivo*, que adaptan los tamaños de los elementos correctamente a la resolución. Una unidad independiente del dispositivo se corresponde con 1/96 de pulgada. El tamaño en píxeles de los elementos dependerá de la densidad de píxeles que tenga el dispositivo donde representamos la interfaz.

## Jerarquía de clases

Antes de comenzar a programar en WPF, es útil echar un primer vistazo a la **jerarquía de clases** que contiene el conjunto básico de controles de WPF. La siguiente figura muestra una visión general básica con algunas de las ramas principales de la jerarquía. A medida que vayamos desarrollando los distintos controles de WPF, iremos adquiriendo una perspectiva más en detalle de estas clases.




A continuación se describen algunas de las clases que muestra el diagrama. Varias de ellas son usadas en una multitud de elementos, como formas, paneles y controles.

- **`System.Threading.DispatcherObject`**

  WPF usa el modelo de afinidad de un solo hilo (**STA** – *Single Thread Affinity*), lo que significa que toda la interfaz de usuario es propiedad de un solo hilo. Este modelo hace que no podamos acceder con seguridad a los controles WPF desde un hilo distinto.

  Para facilitar este modelo, cada aplicación WPF se rige por un despachador (*dispatcher*) que coordina los mensajes (provenientes principalmente de la entrada de teclado y movimientos de ratón). **`DispatcherObject`** permite verificar si el código que ejecutamos se ejecuta en el mismo hilo que la interfaz gráfica, accediendo de manera segura al mismo.

- **`System.Windows.Media.Visual`**

  Cada elemento que aparece en una ventana de WPF es, en el fondo, un objeto Visual. Podemos pensar en la clase **`Visual`** como un objeto de dibujo único que encapsula las instrucciones de dibujo, detalles adicionales sobre cómo se debe realizar el dibujo (recorte, opacidad y configuración de transformación) y funcionalidades básicas (como pruebas de posicionamiento).
  Si queremos crear una interfaz de usuario utilizando una API más ligera que el marco de nivel superior de WPF, se puede programar directamente usando los objetos **`Visual`**.

- **`System.Windows.UIElement`**

  **`UIElement`** añade soporte para algunos elementos esenciales de WPF como el diseño, la entrada, el foco y los eventos (que el equipo de WPF denomina como VIDA).
  Por ejemplo, aquí es donde se organiza el proceso de diseño. También es aquí donde se transforman en eventos las pulsaciones de ratón y de teclado.

- **`System.Windows.FrameworkElement`**

  Implementa algunos miembros que **`UIElement`** únicamente define. Por ejemplo, **`UIElement`** establece las bases para el sistema de diseño de WPF, pero **`FrameworkElement`** incluye las propiedades clave (como **`HorizontalAligment`**) que lo apoyan. También añade soporte para el enlace de datos, animación y estilos.

- **`System.Windows.Shapes.Shape`**

  Las formas geométricas básicas como rectángulo, polígono, elipse, línea y path derivan de esta clase. Dichas formas pueden utilizarse junto con otros controles tradicionales de Windows, tales como botones y cuadros de texto.

- **`System.Winows.Controls.Control`**

  Un control es un elemento que puede interactuar con el usuario. Obviamente las clases **`TextBox`**, **`Button`**, **`ListBox`** y similares heredan directa o indirectamente de la clase **`Control`**.
  La clase **`Control`** agrega propiedades adicionales para configurar el tipo de letra y los colores de primer plano y de fondo. Pero el detalle más interesante que ofrece es modificar el estilo del control, lo cual nos permite cambiar la apariencia de forma uniforme.
  Cuando programamos en Windows Forms, cada elemento visual en una forma se refiere como un control. En WPF, esto no es así. Los elementos visuales son denominados simplemente **elementos**, y algunos de ellos son denominados **controles** (los que pueden recibir el foco e interactuar con el usuario). Para hacer que este sistema sea aún más confuso, muchos elementos (como la clase **`Panel`**) se definen en el espacio de nombres **`System.Windows.Controls`**, a pesar de que no se derivan de **`System.Windows.Controls.Control`** y no se consideran controles.

- **`System.Windows.Controls.ContentControl`**

  Esta es la clase base para todos los controles que tienen una única pieza de contenido (desde los **`Label`** hasta los **`Window`**).

- **`System.Windows.Controls.ItemsControl`**

  Es la clase base para todos los controles que muestran una colección de elementos, como el **`ListBox`** y el **`TreeView`**. Los controles de esta lista son muy flexibles, por ejemplo podemos transformar una **`ListBox`** en una lista de botones de radio. De hecho, en WPF menús, barras de herramientas y barras de estado son en realidad listas especializadas y las clases que implementan todos ellos se derivan de **`ItemsControl`**.

- **`System.Windows.Controls.Panel`**

  Es la clase base para todos los contenedores de elementos (pueden contener uno o más hijos). Estos contenedores son la base del sistema de diseño de WPF, y su uso es clave para la organización de su contenido de la forma más atractiva y flexible posible.

# XAML

## Concepto

**XAML** (siglas de *eXtensible Application Markup Language*) es el lenguaje basado en XML que nos permite crear e inicializar objetos .NET con relaciones jerárquicas entre sí. Si bien se ha diseñado para WPF, se puede utilizar para cualquier árbol de objetos

Todo lo que se puede hacer en XAML también se puede hacer mediante código. XAML es simplemente otro modo de crear e inicializar objetos. También podemos usar WPF sin usar XAML, si bien el uso de XAML tiene algunas ventajas:

- El código XAML es **corto y se lee claramente**.
- Separamos el código de diseño y la lógica, **separando los roles de diseñador y desarrollador.**

## XAML vs código

En el siguiente ejemplo construimos un componente llamado `StackPanel` que contiene un bloque de texto y un botón:

```
<StackPanel>
    <TextBlock Margin="20">Bienvenido a XAML</TextBlock>
    <Button Margin="10" HorizontalAlignment="Right">OK</Button>
</StackPanel>
```

Para expresar lo mismo en C# necesitaríamos el código siguiente:

```
// Creamos el StackPanel
StackPanel stackPanel = new StackPanel();
this.Content = stackPanel;

// Creamos el TextBlock
TextBlock textBlock = new TextBlock();
textBlock.Margin = new Thickness(10);
textBlock.Text = "Bienvenido a XAML";
stackPanel.Children.Add(textBlock);

// Creamos el Button
Button button = new Button();
button.Margin= new Thickness(20);
button.Content = "OK";
stackPanel.Children.Add(button);
```

Como se ve, la versión en XAML es más corta y clara, de ahí que en WPF se prefiera para realizar muchas tareas frente al código.

# Entorno de trabajo

WPF es una combinación de **XAML** (para el marcado) y **C#/VB.NET** o cualquier lenguaje de la plataforma .NET. Todos ellos pueden ser editados en cualquier editor de texto y compilados desde la línea de comandos.

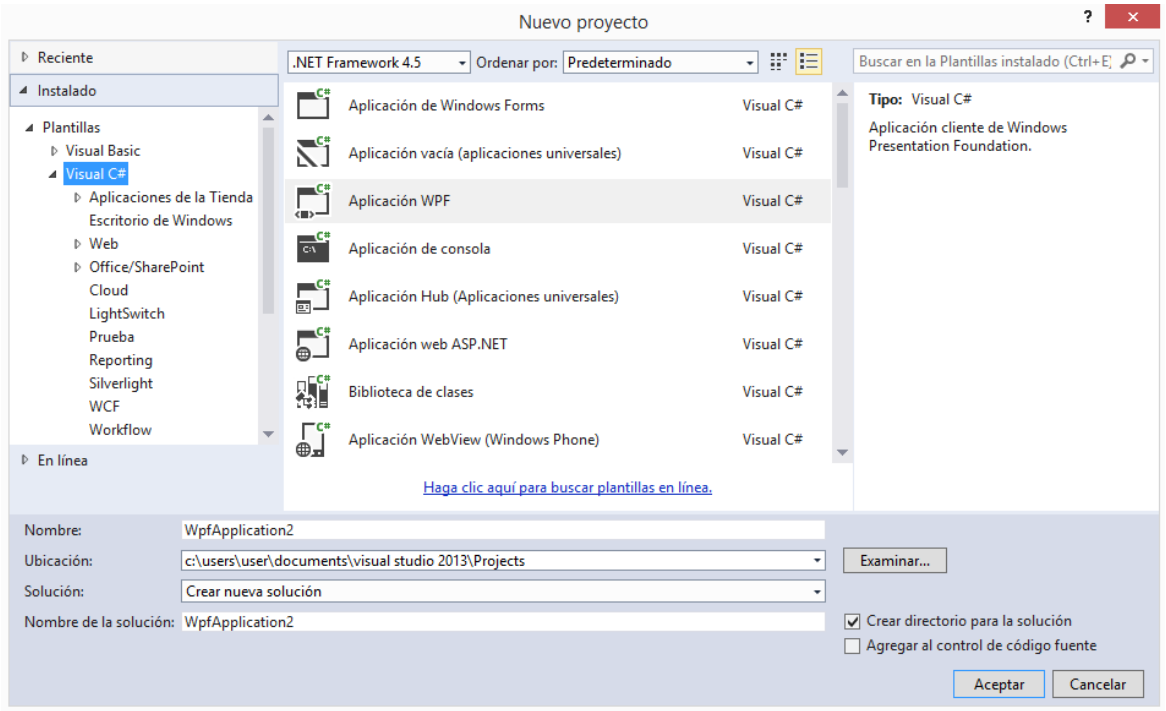
Sin embargo, la mayoría de desarrolladores prefieren utilizar un **IDE** (*Integrated Development Enviroment*) que facilita gran parte de las tareas: Escritura y autocomplección de código, diseño de la interfaz, compilado y depuración, etc.

El IDE más recomendado para NET/WPF es **Visual Studi**o, cuya licencia no es gratuita. Por suerte, Microsoft facilita a los desarrolladores que comienzan a probar .NET y WPF proporcionando una versión gratuita llamada **Visual Studio Express**. Esta versión tiene menos funcionalidad que el Visual Studio real, pero contiene todo lo necesario para comenzar a aprender WPF y crear aplicaciones reales. A fecha de hoy, la versión disponible es Visual Studio Express 2015 que puede descargar desde la Página Web de Microsoft.

## Primer proyecto

Para comprobar que todo funciona correctamente, vamos a crear un primer proyecto en WPF. Para ello iremos al menú ***ARCHIVO > Nuevo Proyecto***.

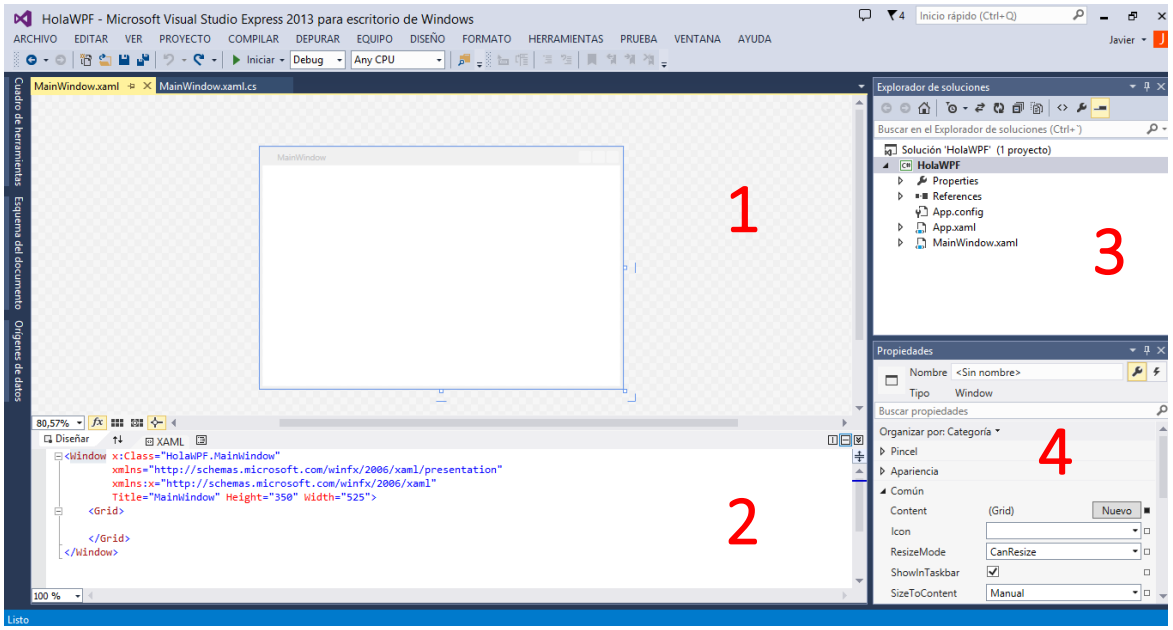
Podremos escoger entre varias plantillas. En principio, podremos programar en los tres lenguajes recomendados para la plataforma .NET (Visual Basic, C# y C++). Escogeremos C#, por ser el que tiene unas características bastante similares a Java. Dentro del grupo **Visual C#**, escogemos la plantilla *Aplicación WPF*.




El nombre del proyecto será *HolaWPF.* El proyecto se guardará en la carpeta de documentos del usuario, dentro de la subcarpeta *visual studio 2013\Projects*

Dentro de Visual Studio tenemos opción de **agrupar varios proyectos en una solución,** que simplemente es un contenedor de un nivel superior al proyecto. Por defecto, el nombre de la solución es el mismo que el del proyecto.

Veremos que aparece una ventana como la siguiente, dividida en varias zonas:




- La **vista de diseño** (1) nos permite visualizar de forma gráfica el aspecto de la interfaz que estamos programando.
- La **vista de código** (2) nos permite visualizar el código XAML correspondiente a la interfaz que estamos elaborando.
- El **explorador de soluciones** (3) nos permite comprobar los archivos que forman parte de nuestro proyecto o nuestras soluciones.
- La **vista de propiedades** (4) nos permite comprobar y editar las propiedades de los componentes que vamos creando y añadiendo a la interfaz.

Si nos centramos en el explorador de soluciones, podemos comprobar que nuestro proyecto tiene entre otros los siguientes archivos (se verán en el tema siguiente con algo más de detalle):

- **`App.xaml.`**
  Este archivo XAML define una aplicación WPF y los recursos que forman parte de la misma. También se utiliza para especificar la interfaz de usuario que se muestra automáticamente cuando se inicia la aplicación, en este caso **`MainWindow.xaml`**
  Si pulsamos sobre el archivo la vista de código se maximiza y veremos el código asociado:

```
<Application x:Class="HolaWPF.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

- **`MainWindow.xaml`**
  Este archivo de código XAML es la ventana principal de la aplicación:

```
<Window x:Class="HolaWPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Como vemos, es un objeto de clase **`Window`** dentro del cual hay otro objeto de clase **`Grid`**.

Asociados a los archivos XAML vistos tenemos dos archivos de extensión **.cs** que nos permitirán escribir el código en C# correspondiente a la lógica de interacción. A este tipo de archivos se le llaman *Code-Behind*, y de manera cooperativa con XAML permiten construir cada una de las ventanas:

- **`App.xaml.cs`** permite implementar la lógica para la aplicación:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;

namespace HolaWPF
{
    /// <summary>
    /// Lógica de interacción para App.xaml
    /// </summary>
    public partial class App : Application
    {
    }
}
```

- **`MainWindow.xaml.cs`** permite implementar el código para controlar los eventos declarados en `MainWindow.xaml` mediante una clase parcial[1]

```
using System;
using System.Collections.Generic;
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 HolaWPF
{
    /// <summary>
    /// Lógica de interacción para MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

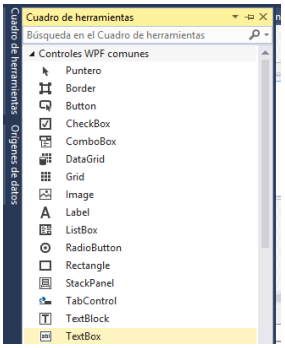
[1] El empleo de una clase parcial simplemente indica que se pueden implementar otras partes de la clase en otros archivos (como ocurre con la vista, cuyo código se genera a partir del XAML).

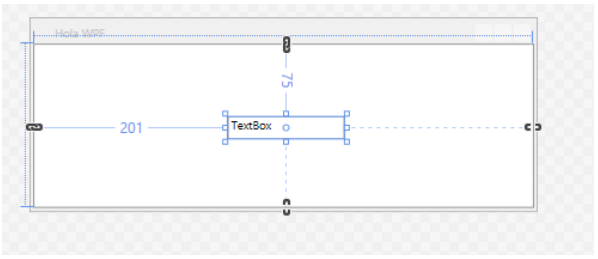
Vamos a añadir un bloque de texto al panel. Para ello, con el archivo XAML seleccionado, tenemos dos alternativas:

- **Añadir el código XAML a mano:**
  Como vemos, el objeto que añadimos es una etiqueta XAML cuyos atributos se corresponden con propiedades del objeto. También hemos aprovechado para redefinir el título de la ventana y las dimensiones de la misma.
  A medida que escribimos el código, podemos ayudarnos de la autocomplección de código para escribir de manera automática el mismo.

```
<Window x:Class="HolaWPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Hola WPF" Height="200" Width="525">
    <Grid>
        <TextBox HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="72">
            Hola WPF!
        </TextBox>
    </Grid>
</Window>
```

- Utilizar el ***Cuadro de Herramientas***
  Accedemos al mismo desde parte izquierda de la pantalla (por defecto está oculto). Una vez que seleccionamos un control pulsando sobre el mismo, debemos pulsar en el sitio de la ventana donde queremos añadirlo:



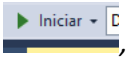




Como vemos, el código XAML se genera automáticamente, si bien se generan algunos atributos que podemos no necesitar:

```
<TextBox HorizontalAlignment="Left" Height="23" Margin="201,75,0,0" TextWrapping="Wrap"
         Text="TextBox" VerticalAlignment="Top" Width="120"/>
```

Como vemos, el texto del componente puede determinarse a partir del contenido de la etiqueta (como se hizo en el caso anterior) o mediante el atributo **`Text`**.

Podemos editar el componente a través del propio código, como hicimos anteriormente, o a través del inspector de propiedades. En este caso modificaremos las propiedades **`HorizontalAligment`**, **`VerticalAligment`**, **`FontSize`** y **`Text`**. Además, si queremos un comportamiento equivalente al especificado en el ejemplo anterior, tendremos que borrar las propiedades que especifican márgenes y dimensiones.

Se puede lanzar la aplicación pulsando el botón ***Iniciar*** 

, o mediante el menú **DEPURAR > Iniciar Depuración** o pulsando la tecla **F5**. En cualquier caso debería mostrarse la siguiente ventana: