# Programación Avanzada de entornos de escritorio

**2021-2022**

**Escuela Politécnica Superior**

**Universidad de Alicante**

# Patrones UI

- Hasta ahora hemos mezclado lógica de negocio y visualización

```
this.labelCPU.Content = lectorRecursosSistema.getCPU();
this.labelDiscoLectura.Content = lectorRecursosSistema.getDatosDisco(DiskData.Read).ToString();
this.labelDiscoEscritura.Content = lectorRecursosSistema.getDatosDisco(DiskData.Write).ToString();
```

- Esta arquitectura acoplada puede generar muchos problemas (véanse siguientes diapositivas)

# Patrones UI

- Hasta ahora hemos mezclado lógica de negocio y visualización
- Taller monitor sistema:

```
this.labelCPU.Content = lectorRecursosSistema.getCPU();
this.labelDiscoLectura.Content = lectorRecursosSistema.getDatosDisco(DiskData.Read).ToString();
this.labelDiscoEscritura.Content = lectorRecursosSistema.getDatosDisco(DiskData.Write).ToString()
```

# Patrones UI

- Hasta ahora hemos mezclado lógica de negocio y visualización
- Taller Calculadora

```
private void BtnCalcular_Click(object sender, EventArgs e)
{
    operando2 = Convert.ToInt32(labelDisplay.Text);
    Trace.TraceInformation("Operación " + operando1 + operacion.ToString() + operando2);
    switch (operacion) {
        case Operaciones.Suma:
            resultado = operando1 + operando2;
            break;
        case Operaciones.Resta:
            resultado = operando1 - operando2;
            break;
        case Operaciones.Division:
            if (operando2 == 0)
            {
```

OFICIAL Máster Web

# Patrones UI: problemas acoplamiento

- Cambios de librería gráfica implican cambios en todo el código
- No permite hacer pruebas unitarias de la lógica de negocio … ni de la vista
- No es escalable
- No aprovechamos código para la versión Web, Móvil, Escritorio
- Mantenimiento difícil
- Comportamiento gráfico mezclado con reglas de negocio

# Patrones UI: problemas acoplamiento

- Solución, separar:
  - Vista (elementos interfaz gráfica)
  - Interacción
  - Modelo (lógica de negocio)

  (siguiente diapositiva)

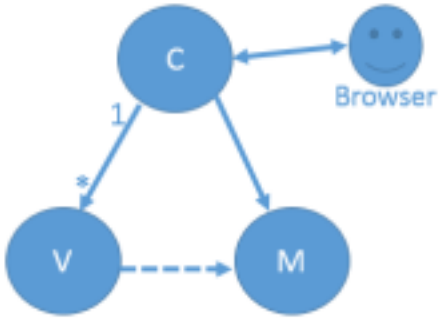
# Patrones UI: modelo, vista, controlador

- Modelo: contiene la lógica de negocio (reglas como la aplicación de una oferta en el cálculo del importe total de una factura o cuándo debe morir un personaje en un juego)
- Vista: visualización gráfica
- Controller / Presenter / View Model: intermediarios entre el modelo y la vista

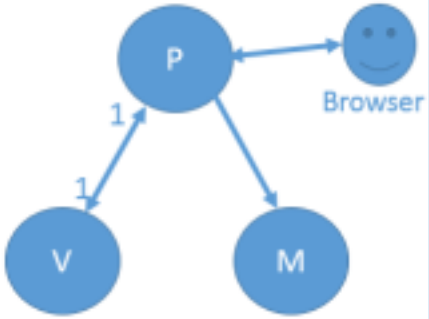
# Patrones UI: problemas acoplamiento

- Modelos desacoplados
  - model view controller (MVC)
  - model view presenter (MVP)
  - model view-view model (MVVM)

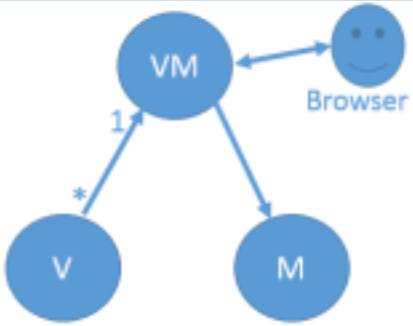
# MVC – MVP - MVVM




- Controller is the entry point to the application
- One to Many relationship between Controller and View
- View does not have reference to the Controller
- View is very well aware of the Model
- Smalltalk, ASP.Net MVC

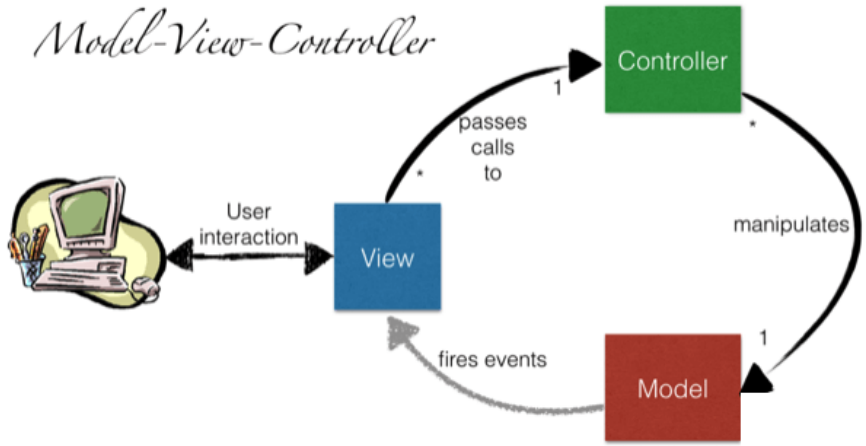


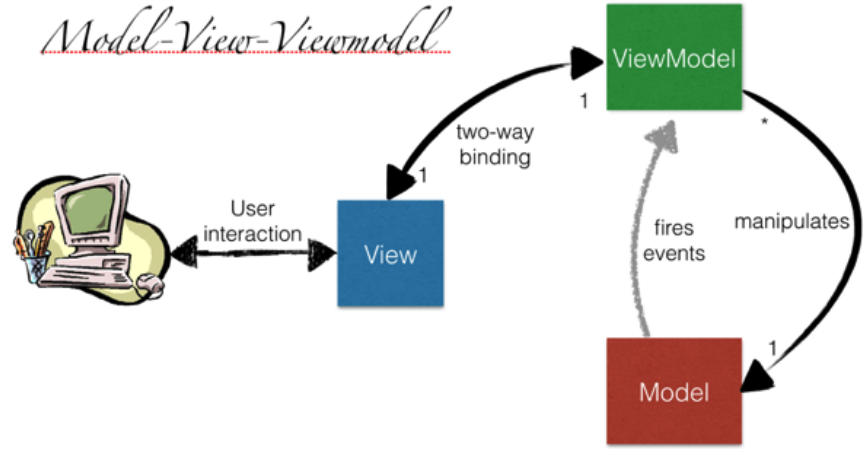

- View is the entry point to the application
- One to One mapping between View and Presenter
- View have the reference to the Presenter
- View is not aware of the Model
- Windows forms

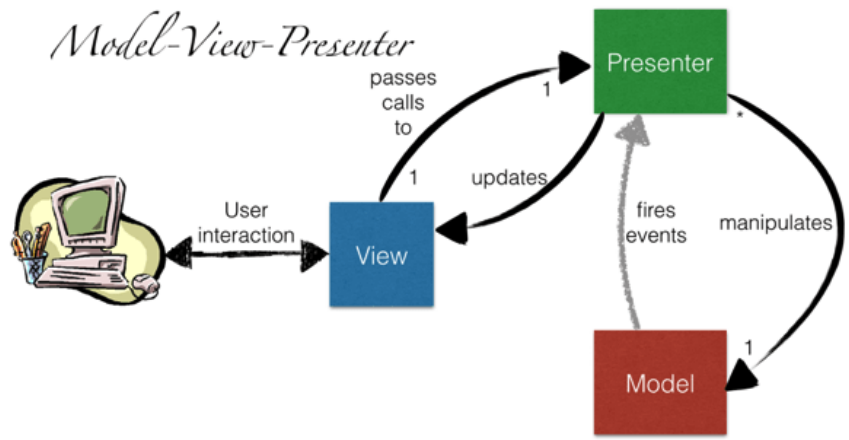



- View is the entry point to the application
- One to Many relationship between View and ViewModel
- View have the reference to the View Model
- View is not aware of the Model
- Silverlight, WPF, HTML5 with Knockout/AngularJS

http://www.tellmehow.co/wp-content/uploads/2017/10/difference-between-mvc-mvp-mvvm.png




© 2015 by Stephan Rauh, http://www.beyondjava.net

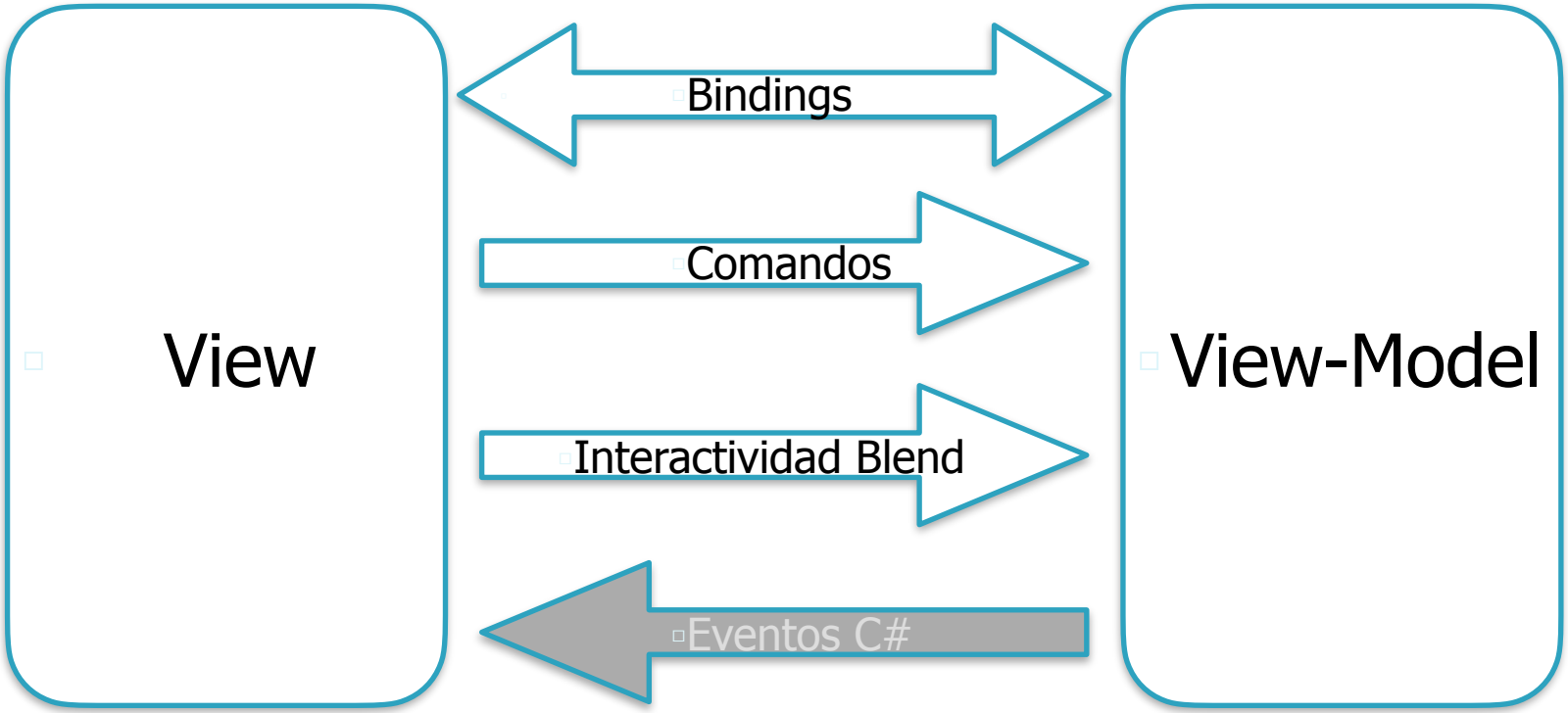
Computer image by Max Lee,
licensed under a Creative Commons 2.0 license ((CC BY-SA 2.0)
(https://www.flickr.com/photos/keetsa/524715635)
Graphic inspired by
http://joel.inpointform.net/software-development/mvvm-vs-mvp-vs-mvc-the-differences-explained/

# Model-View-ViewModel (MVVM)

- ViewModel:
  - Intermediario entre modelo y vista
  - Adapta los datos del modelo para representarlos / modificarlos en la vista
    - p.ej. si estoy mostrando una mapa, transformamos las coordenadas geométricas en coordenadas de la pantalla x,y
  - Controla las interacciones de la vista
- El modelo no conoce ni a la vista ni al ViewModel
- La vista sí conoce al ViewModel

# Model-View-ViewModel (MVVM)

- ¿Cómo interactúan la vista y el view-model?



# Model-View-ViewModel (MVVM)

- Bindings WPF:
  - cada binding conecta dos propiedades (posiblemente en dos objetos distintos) tal que si una cambia la otra también (en una o ambas direcciones)
  - podemos asociarle disparadores (triggers) que nos notifiquen de cambios
- WPF Data Templates:
  - Convierte datos no visuales (ViewModel) en su representación visual (View)
- Comandos WPF (o Microsoft Blend SDK comportamientos de interactividad) sirven para pasar los eventos de las vistas a los ViewModels
- Aún podemos notificar cambios a la vista con eventos de C#, pero es una práctica antigua porque hay que registrar el manejado del evento en la vista

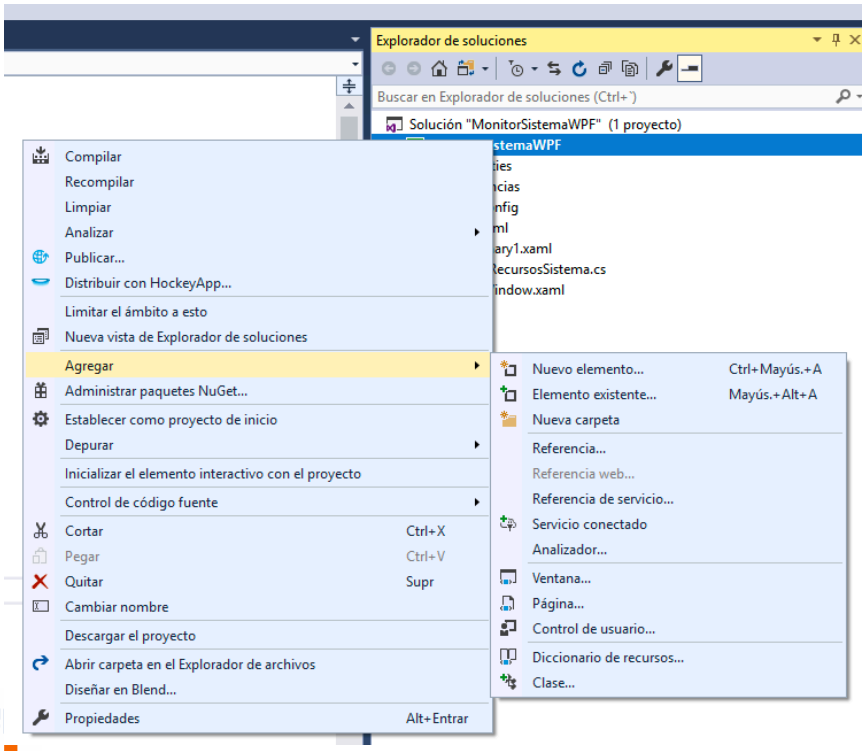
# Proyecto ejemplo

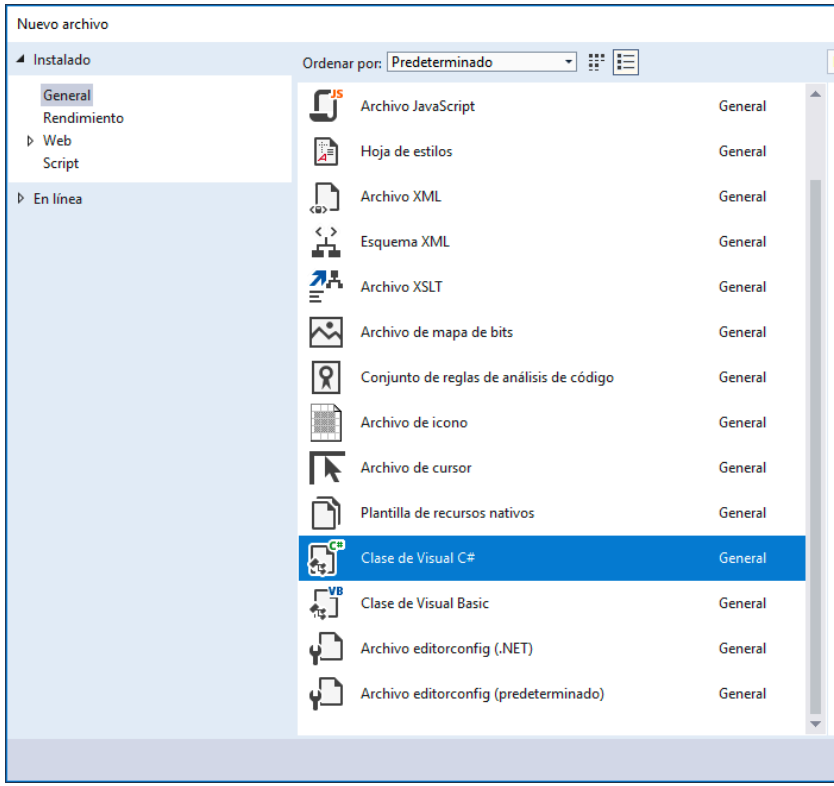
- Vamos a cambiar el monitor de actividad de la sesión 2 (sesion4-inicio.zip en Moodle)
  - Generalizemos las medidas en una clase “Medida”
  - Pondremos todas las medidas (CPU, Memoria…) en una clase RecursosSistema
  - Ambas clases preparadas para usar binding y con ello MVVM
  - Y desacoplaremos el modelo de la vista

# ACTIVIDAD 1

# Bindings WPF

- Creamos la clase Medida que contiene una lectura del estado actual de un recurso del sistema y RecursosSistema que contiene el estado de todos los recursos a medir



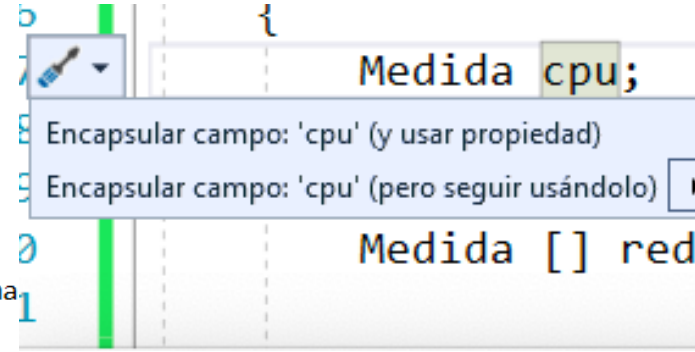

Botón derecho sobre paquete

OFICIAL Máste Web

```
using System;
namespace MonitorSistema
{
    public class Medida
    {
        string titulo = null;

        public Medida(string titulo)
        {
            this.titulo = titulo;
        }

        public string Titulo
        {
            get
            {
                return titulo;
            }
            set
            {
                titulo = value;
            }
        }

        double valor;
        public double Valor
        {
            get
            {
                return valor;
            }

            set
            {
                valor = value;
            }
        }
    }
}
```

```
namespace MonitorSistema
{
    public class RecursosSistema
    {
        Medida cpu;
        Medida[] disco;
        Medida[] memoria;
        Medida[] red;
        String fabricante;
        string modelo;
        string usuario;
        string procesador;

        public RecursosSistema
        {
            cpu = new Medida("CPU");
            disco = new Medida[]
            {
                new Medida("Lectura"),
                new Medida("Escritura"),
                new Medida("Lectura/Escritura")
            };
            memoria = new Medida[] {
                new Medida("Física"),
                new Medida("Virtual")
            };
            red = new Medida[] {
                new Medida("Paquetes enviados"),
                new Medida("Paquetes recibidos"),
                new Medida("Paquetes recibidos/recibidos")
            };
        }

        public Medida Cpu { get => cpu; set => cpu = value; }
        public Medida[] Disco { get => disco; set => disco = value; }
        public Medida[] Memoria { get => memoria; set => memoria = valu
        public Medida[] Red { get => red; set => red = value; }
        public Medida DiscoLectura { get => disco[0]; }
```

Generamos getters y setters con ctr+. (punto incluido) sobre el campo




OFICIAL

# RecursosSistema como ViewModel

- Esta clase tiene preparados los datos para pintar
- Hacemos que esta clase sea la encargada de preguntar al Modelo sobre los cambios (en este caso LectorRecursosSistema)
- La vista ya no tendrá acceso directo al modelo
- El modelo (lectorRecursosSistema) no decidirá cómo se "pintan" los datos (sig. diapositiva)

```
class RecursosSistema
{
    Medida cpu;
    Medida[] disco;
    Medida[] memoria;
    String fabricante;

    LectorRecursosSistema lectorRecursosSistema;
    System.Windows.Threading.DispatcherTimer dispatcherTimer;

    public RecursosSistema()
    {
        lectorRecursosSistema = new LectorRecursosSistema();
        dispatcherTimer = new System.Windows.Threading.DispatcherTimer();
        dispatcherTimer.Tick += new EventHandler(manejadorDispatcherTimer);
```

```
private void manejadorDispatcherTimer(object sender, EventArgs e)
{
    this.cpu.Valor = lectorRecursosSistema.getCPU();
    this.fabricante = lectorRecursosSistema.getOrdenadorFabricante();
    this.memoria[0].Valor = lectorRecursosSistema.getMemoriaFisica();
    this.memoria[1].Valor = lectorRecursosSistema.getMemoriaVirtual();
```

# Refactorizar responsabilidades

- Encapsulamos el timer en el ViewModel RecursosSistema, leemos y preparamos (damos formato) los datos del VM

View Model

Model = lectorRecursosSistema

```
class RecursosSistema
{
    Medida cpu;
    Medida[] disco;
    Medida[] memoria;
    String fabricante;

    LectorRecursosSistema lectorRecursosSistema;
    System.Windows.Threading.DispatcherTimer dispatcherTimer;

    public RecursosSistema()...

    public void iniciar()
    {
        dispatcherTimer.Interval = new TimeSpan(0, 0, 1); // se ac
        dispatcherTimer.Start();
    }

    public void parar()
    {
        dispatcherTimer.Stop();
    }

    public bool ejecutandose()
    {
        return dispatcherTimer.IsEnabled;
    }
```

Vista

(code-behind Form - siguientes diapositivas)

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    if (recursosSistema.ejecutandose())
    {
        recursosSistema.parar();
        btnActualizar.Content = "Iniciar";
    } else
    {
        recursosSistema.iniciar();
        btnActualizar.Content = "Parar";
    }
    // esto está mal aquí, sólo se actualiza cuando pulsamos el botón
    this.labelCPU.Content = recursosSistema.Cpu.Valor + "%";
```

(véase el comentario en verde arriba)

Montábamos la cadena en el modelo

Antes

```
public string getCPU()
{
    double d = GetCounterValue(cpu, "Processor", "% Processor Time", "_Total");
    return compacto ? (int)d + "%" : d.ToString("F") + "%";
}
```

Ahora

```
public double getCPU()
{
    return GetCounterValue(cpu, "Processor", "% Processor Time", "_Total");
}
```

De esta forma quitamos la responsabilidad de formatear el porcentaje disponible del modelo (lectorRecursosSistema)

El objeto recursos es nuestro "ViewModel"

# View (formulario)

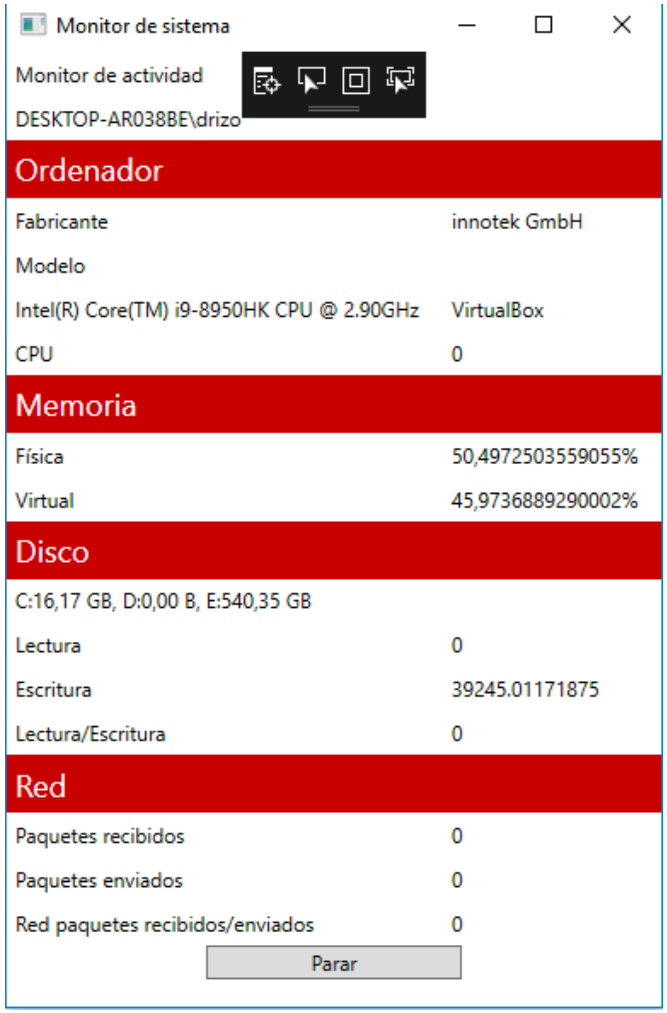
```
public partial class MainWindow : Window
{
    //LectorRecursosSistema lectorRecursosSistema = new LectorRecursosSistema
    //System.Windows.Threading.DispatcherTimer dispatcherTimer = new System.W
    RecursosSistema recursosSistema = new RecursosSistema();

    public MainWindow()
    {
        InitializeComponent();
        //dispatcherTimer.Tick += new EventHandler(manejadorDispatcherTimer);
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        //if (dispatcherTimer.IsEnabled)
        if (recursosSistema.ejecutandose())
        {
            //dispatcherTimer.Stop();
            recursosSistema.parar();
            btnActualizar.Content = "Iniciar";
        }
        else
        {
            //dispatcherTimer.Interval = new TimeSpan(0, 0, 1); // se actuali
            //dispatcherTimer.Start();
            recursosSistema.iniciar();
            btnActualizar.Content = "Parar";
        }
    }
```

# Aclaración solución temporal

- En la solución actual (y las parciales subidas a Moodle), RecursosSistema no está como un MVVM estricto pues es LectorRecursosSistema quien rellena los datos

# Actividad 1

- Llegar al estado actual del sistema
- No es necesario de momento implementar todas las medidas
- Podemos copiar / pegar del ejemplo en Moodle
  - (Solución en Moodle: sesion4-solucion-actividad1.zip)




- ¡La vista no se actualiza!
  - (Solución en siguientes diapositivas…)

# ACTIVIDAD 2

# Binding

- Conforme lo tenemos, la vista sólo recoge los datos cuando le damos al botón
- En MVVM la vista no accede al modelo, por lo que la vista no sabe cuándo el modelo se actualiza

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    if (recursosSistema.ejecutandose())
    {
        recursosSistema.parar();
        btnActualizar.Content = "Iniciar";
    } else
    {
        recursosSistema.iniciar();
        btnActualizar.Content = "Parar";
    }
    // esto está mal aquí, sólo se actualiza cuando pulsamos el botón
    this.labelCPU.Content = recursosSistema.Cpu.Valor + "%";
```

# Binding

- Hacemos que el contenido de la vista (xaml) se obtenga automáticamente del ViewModel (recursosSistema) mediante binding

Antes

MainWindow.xaml

```
<Label x:Name="labelUsuario" Content="ValorUsuario" Grid.Row="1"/>
```

Ahora

```
<Label x:Name="labelUsuario" Grid.Row="1" Content="{Binding Path=Usuario}"/>
```

¿Cómo se sabe en qué objeto está la propiedad Usuario? (sig. diap.)

# Binding

- Establecemos la fuente de datos con DataContext:

MainWindow.xaml.cs

```
public MainWindow()
{
    InitializeComponent();
    recursosSistema = new RecursosSistema(); // el View-Model
    this.DataContext = recursosSistema;
}
```

# Resumen Binding etiqueta Usuario

MainWindow.xaml.cs (método click())

```
this.labelCPU.Content = recursosSistema.Cpu.Valor + "%";
```

Dato enlazado se saca de recursosSistema.Usuario

MainWindow.xaml

```
<Label x:Name="labelUsuario" Grid.Row="1" Content="{Binding Path=Usuario}"/>
```

MainWindow.cs

```
public MainWindow()
{
    InitializeComponent();
    recursosSistema = new RecursosSistema(); // el View-Model
    this.DataContext = recursosSistema;
}
```

# Binding

- El problema con el esquema actual es que si recursosSistema actualiza el dato la vista no se entera del cambio y por tanto no lo actualiza

- En este ejemplo, el nombre de usuario siempre es la misma, pero … ¿y las medidas de memoria ocupada, disco….?

  - Debemos hacer que la propiedad enlazada en el binding sea “observable”
  - Lo hacemos con Medida que es lo que vamos a enlazar y cambia

```
<Label x:Name="labelMemoriaFisica" Grid.Row="8" Grid.Column="1" Content="{Binding Path=MemoriaFisicaOcupada.Valor}"/>
```

```
namespace MonitorSistema
{
    public class Medida : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
```

```
public double Valor
{
    get
    {
        return valor;
    }

    set
    {
        valor = value;
        OnPropertyChanged("Valor");
    }
}

void OnPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Esto notificará a quien esté “escuchando” que el valor ha cambiado

# Actividad clase 2 + ampliación

- Hacer que funcionen bindings de al menos dos elementos (p.ej. CPU y Memoria)
  - (Solución en Moodle: sesion4-solucion-actividad2.zip)
  - En la anterior solución (actividad1.zip) no estaba:
    - enlazado en el viewmodel el timer.tick += manejador
  - En la solución actividad2 falta:
    - quitar la asignación del valor labelCPU en la vista
    - Comprobar el constructor de modelo (ver si es conveniente dejar la actualización fabricante donde está o moverla al constructor)

- Ampliación:
  - Visualizar los % de ocupación de memoria y disco en lugar del valor absoluto respetando la arquitectura
  - Ahora muestra en ocasiones valores 0 porque la librería de acceso a la carga del sistema devuelve esos valores en ocasiones. Deberíamos mostrar el último valor no 0

# ACTIVIDAD 3

# Comandos

- Para aprender cómo funcionan los comandos desde la vista vamos a añadir primero una columna más a nuestra vista donde vamos a guardar los valores máximos obtenidos en los porcentajes de uso de memoria (el valor máximo lo calcularemos en el método manejadorTimer en RecursosSistema)

```
<Label Grid.Column="2" Grid.Row="8" Content="{Binding MemoriaFisicaPorcentajeMaximo.Valor}"/>
<Label Grid.Column="2" Grid.Row="9" Content="{Binding MemoriaVirtualPorcentajeMaximo.Valor}"/>
```

# Actividad clase 3

- Añade esas dos etiquetas con sus correspondientes valores en el ViewModel (recursosSistema) para las medidas que hayas implementado

```
memoria = new Medida[] {
    new Medida("Física disponible"),
    new Medida("Física ocupada"),
    new Medida("Virtual disponible"),
    new Medida("Virtual ocupada"),
    new Medida("Física porcentaje"),
    new Medida("Virtual porcentaje"),
    new Medida("Física porcentaje máximo"),
    new Medida("Virtual porcentaje máximo")
```

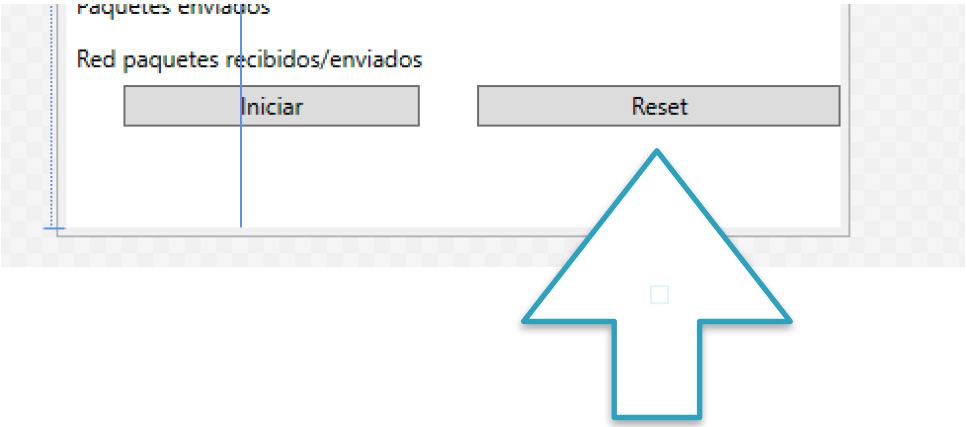
```
public Medida MemoriaFisicaDisponible { get => memoria[0]; }
public Medida MemoriaFisicaOcupada { get => memoria[1]; }
public Medida MemoriaVirtualDisponible { get => memoria[2]; }
public Medida MemoriaVirtualOcupada { get => memoria[3]; }
public Medida MemoriaFisicaPorcentaje { get => memoria[4]; }
public Medida MemoriaVirtualPorcentaje { get => memoria[5]; }
public Medida MemoriaFisicaPorcentajeMaximo { get => memoria[6]; }
public Medida MemoriaVirtualPorcentajeMaximo { get => memoria[7]; }
```

(Solución en Moodle: sesion4-solucion-actividad3.zip)

# ACTIVIDAD 4

# Comandos

- También podemos hacer binding de acciones. P.ej. cuando se pulsa un botón que se modifiquen los contenidos del ViewModel




```
<Button Content="Reset" Command="{Binding ComandoReset}" Grid.Row="19" Grid.Column="
```

# Comandos

- Creamos un comando que será llamado por el botón
- La responsabilidad de cambiar los datos es del ViewModel,
  no de la Vista

```
public class RecursosSistema
{
    Medida cpu;
    Medida[] disco;
    Medida[] memoria;
    Medida[] red;
    String fabricante;
    string modelo;
    string usuario;
    string procesador;
    ICommand comandoReset;
```

```
comandoReset = new ComandoReset(this);
```

```
public class ComandoReset : ICommand
{
    RecursosSistema destinatario;

    public event EventHandler CanExecuteChanged;

    public ComandoReset(RecursosSistema r)
    {
        this.destinatario = r;
    }

    public bool CanExecute(object parameter)
    {
        return true;
    }

    public void Execute(object parameter)
    {
        destinatario.MemoriaFisicaPorcentajeMaximo.Valor = 0;
        destinatario.MemoriaVirtualPorcentajeMaximo.Valor = 0;
    }
}

public class RecursosSistema
```

```
public ICommand ComandoReset { get => comandoReset; }
```

Máster
OFICIAL

# Actividad clase 4

- Incluye este comportamiento en tu proyecto
  - (Solución en Moodle:
  - sesion4-solucion-actividad4.zip)

# ACTIVIDAD 5

# Comandos

- Siguiendo el mismo principio, la actualización de datos del modelo no es responsabilidad del sistema (como está ahora), sino del ViewModel (RecursosSistema)
- Para ello creamos dos comandos nuevos:
  - Iniciar
  - Parar
  - ➡ (no usamos el mismo botón para Iniciar y parar para poder ilustrar el uso de ICommand.CanExecute)
- Aprovechamos para generalizar los comandos que usa RecursosSistema (ver diapositiva siguiente)

```
internal abstract class ComandosRecursosSistema : ICommand
{
    protected RecursosSistema destinatario;

    public event EventHandler CanExecuteChanged;

    public ComandosRecursosSistema(RecursosSistema destinatario)
    {
        this.destinatario = destinatario;
    }

    public abstract bool CanExecute(object parameter);
    public abstract void Execute(object parameter);
}

internal class ComandoReset : ComandosRecursosSistema
{
    public ComandoReset(RecursosSistema r) : base(r)
    {
    }

    public override bool CanExecute(object parameter)
    {
        return true;
    }

    public override void Execute(object parameter)
    {
        destinatario.MemoriaFisicaPorcentajeMaximo.Valor = 0;
        destinatario.MemoriaVirtualPorcentajeMaximo.Valor = 0;
    }
}
```

- Añadimos los comandos de inicio y parada a RecursosSistema (recordad, es el ViewModel)

```
public class RecursosSistema
{
    LectorRecursosSistema lectorRecursosSistema;
    System.Windows.Threading.DispatcherTimer dispatcherTimer;

    Medida cpu;
    Medida[] disco;
    Medida[] memoria;
    Medida[] red;
    string fabricante = "N/A";
    string modelo;
    string usuario;
    string procesador;
    string discosLogicos;

    ICommand comandoReset;
    ICommand comandoIniciar;
    ICommand comandoParar;

    public RecursosSistema()
    {
        dispatcherTimer = new System.Windows.Threading.DispatcherTimer();
        dispatcherTimer.Tick += manejadorDispatcherTimer;
        dispatcherTimer.Interval = new TimeSpan(0, 0, 1);
        lectorRecursosSistema = new LectorRecursosSistema();
        comandoReset = new ComandoReset(this);
        comandoIniciar = new ComandoIniciar(this);
        comandoParar = new ComandoParar(this);

        cpu = new Medida("CPU");
```

# RecursosSistema inicia / para a través de un comando

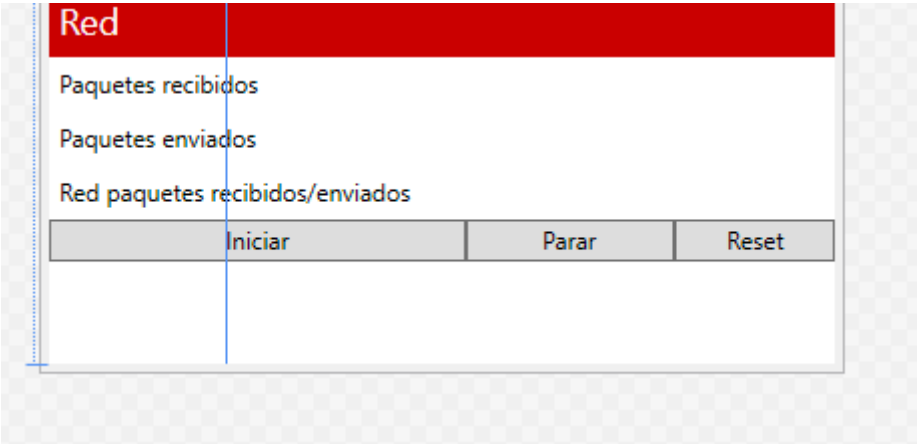
```
internal class ComandoIniciar : ComandosRecursosSistema
{
    public ComandoIniciar(RecursosSistema r) : base(r)
    {
    }

    public override bool CanExecute(object parameter)
    {
        return !destinatario.ejecutandose();
    }

    public override void Execute(object parameter)
    {
        destinatario.iniciar();
    }
}

internal class ComandoParar : ComandosRecursosSistema
{
    public ComandoParar(RecursosSistema r) : base(r)
    {
    }

    public override bool CanExecute(object parameter)
    {
        return destinatario.ejecutandose();
    }

    public override void Execute(object parameter)
    {
        destinatario.parar();
    }
}
```

```
public void iniciar()
{
    dispatcherTimer.Interval = new TimeSpan(0, 0, 1); //
    dispatcherTimer.Start();
}

ic void parar()

    dispatcherTimer.Stop();
}

public bool ejecutandose()
{
    return dispatcherTimer.IsEnabled;
}
```

```
    public ICommand ComandoReset { get =
    public ICommand ComandoIniciar { get
    public ICommand ComandoParar { get =

}
```

- RecursosSistema inicia / para a través de un comando

En MainWindow.xaml queda:




```
<Label Grid.Column="2" Grid.Row="[illegible] Content="{Binding MemoriaVirtualPorcentajeMaximo.Valor
<Button Content="Iniciar" Grid.Row="19" Command="{Binding ComandoIniciar}" />
<Button Content="Parar" Grid.Row="19" Command="{Binding ComandoParar}" Grid.Column="1" />
<Button Content="Reset" Command="{Binding ComandoReset}" Grid.Row="19" Grid.Column="2"/>
```

- RecursosSistema inicia / para a través de un comando

En MainWindow.xaml.cs queda:

```
public partial class MainWindow : Window
{
    RecursosSistema recursosSistema;

    public MainWindow()
    {
        InitializeComponent();
        recursosSistema = new RecursosSistema(); // el View-Model
        this.DataContext = recursosSistema;
    }

    /* Ya no se usa - lo hemos reemplazado por comandos
    private void Button_Click(object sender, RoutedEventArgs e)
    {
        if (recursosSistema.ejecutandose())
        {
            recursosSistema.parar();
        } else
        {
            recursosSistema.iniciar();
        }
     }
*/
}
}
```

# ICommand.canExecute

- Sin embargo, esto no funciona (el botón de parar no se activa) si el ViewModel no notifica a los comandos del cambio de estado

# ICommand.canExecute

- Para que funcione debemos notificar de los cambios (véase EventoCambioEstado)

En RecursosSistema:

```
public class RecursosSistema : INotifyPropertyChanged
{
    ComandosRecursosSistema comandoIniciar;
    ComandosRecursosSistema  comandoParar;

    public void iniciar()
    {
        dispatcherTimer.Interval = new TimeSpan(0, 0, 1); // se actualiza cada s
        dispatcherTimer.Start();
        EventoCambioEstado("Ejecutando");
        comandoIniciar.NotifyCanExecuteChanged();
        comandoParar.NotifyCanExecuteChanged();
    }

    void EventoCambioEstado(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }

    public void parar()
    {
        dispatcherTimer.Stop();
        EventoCambioEstado("Ejecutando");
        comandoIniciar.NotifyCanExecuteChanged();
        comandoParar.NotifyCanExecuteChanged();
    }
```

Indicamos que la clase puede notificar cambios

Cambiamos clase comandoIniciar y parar para que tengan los métodos propios necesarios

Notificamos de los cambios

Notificamos de los cambios

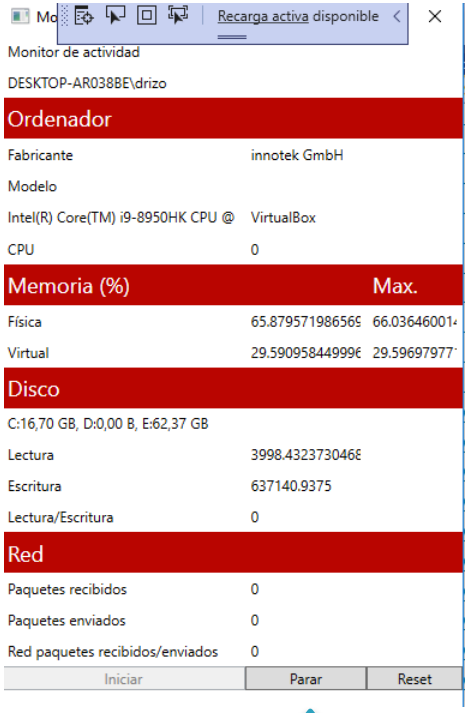
# ICommand.canExecute

- Para que funcione debemos notificar de los cambios (véase EventoCambioEstado)

En RecursosSistema:

```
internal abstract class ComandosRecursosSistema : ICommand
{
    protected RecursosSistema destinatario;

    public event EventHandler CanExecuteChanged;

    public ComandosRecursosSistema(RecursosSistema destinatario)
    {
        this.destinatario = destinatario;
    }

    public void NotifyCanExecuteChanged()
    {
        if (CanExecuteChanged != null)
        {
            CanExecuteChanged(this, EventArgs.Empty);
        }
    }
```

Es notificado de que ha habido un cambio y que debe revisar si se puede ejecutar



# Ventaja adicional ICommand

- Usando ICommand nos podemos permitir enlazar varios controles (p.ej. un botón, un item de una barra de menú y un menú contextual) al mismo comando de forma que todos se activen / desactiven según el estado del comando (CanExecute)

# Actividad clase 5

- Incluye este comportamiento en tu proyecto
  - (Solución en Moodle:
  - sesion4-solucion-actividad5.zip)
  -

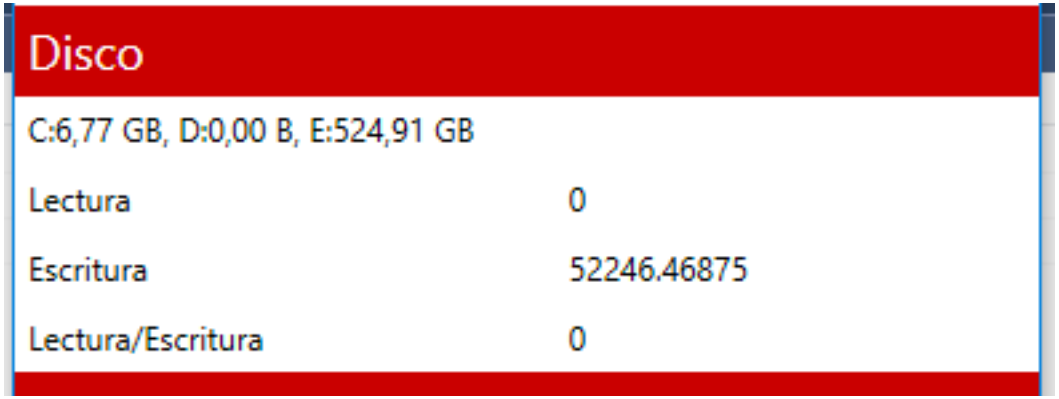
# ACTIVIDAD 6 (OPCIONAL)

# Data Templates

- En la solución actual estamos usando el "anti-pattern" copiar/pegar para añadir todos los controles de visualización de las mediciones del sistema
- Cada medición de sistema usa una instancia de la clase Medida.
- Vamos a crear una vista que sepa visualizar Medidas
- Vamos a reemplazar las copias manuales de medidas por una lista de vistas de Medida

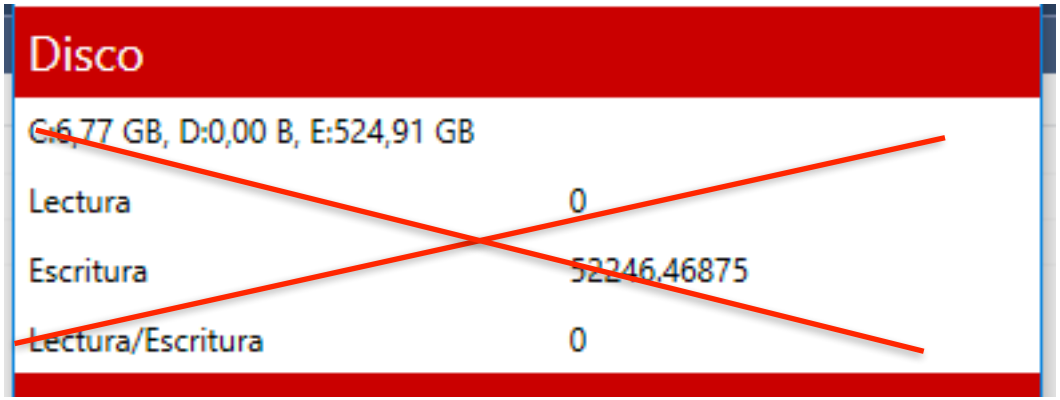
# Data Templates

- Como tenemos las medidas por secciones vamos a cambiar sólo las medidas de la sección
  - Podríamos cambiarlo todo añadiendo un concepto "Categoría" a la clase Medida



# Data Templates

- De momento sólo borramos las Label con información de Disco



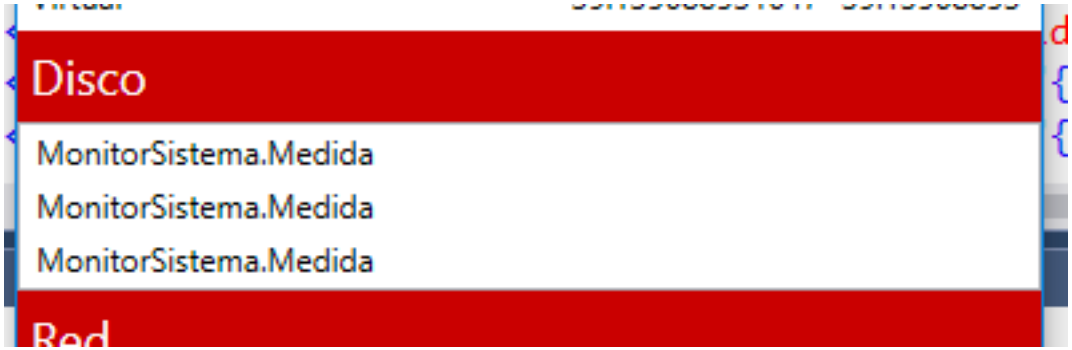

```
<Label x:Name="labelDiscosLogicos" Grid.Row="11"  Content=
<Label x:Name="labelDiscoLectura" Grid.Row="12" Grid.Colum
<Label x:Name="labelDiscoEscritura" Grid.Row="13"  Grid.Co
<Label x:Name="labelDiscoLecturaEscritura" Grid.Row="14"
```

# Data Templates

- Y las sustituimos por un ListBox que usa el array Medida [] disco que ya teníamos creado

```
<ListBox Grid.Row="11" Grid.ColumnSpan="3" Grid.RowSpan="3" ItemsSource="{Binding Path=Disco}"/>
```

- … que actualmente muestra en ejecución:




- En ItemsSource se obtienen los datos,
- que son una colección de Medida sin ToString

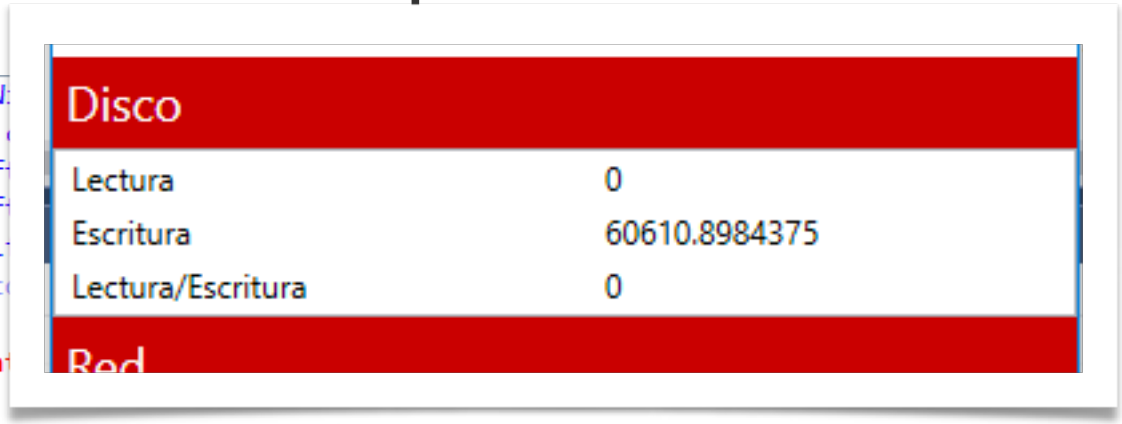
# Data Templates

- Podríamos implementar el método ToString
  - pero se limita a formatear una cadena
  - … y no se actualiza por defecto

```
public class Medida : INotifyPropertyChanged
{
    string titulo;
    double valor;

public override string ToString()
{
    return String.Format("{0,18:00000000.000}", Valor);
}
```

# Data Templates

- Implementamos una Vista para la Medida




```
<Window x:Class="MonitorSistemaWPF.MainW
        xmlns="http://schemas.microsoft.
        xmlns:x="http://schemas.microsof
        xmlns:d="http://schemas.microsof
        xmlns:mc="http://schemas.openxml
        xmlns:local="clr-namespace:Monit
        mc:Ignorable="d"
        Title="Monitor de sistema" Heigh

    <Window.Resources>
        <DataTemplate x:Key="medidaTemplate">
            <StackPanel Orientation="Horizontal">
                <TextBlock Width="200" Text="{Binding Path=Titulo}" />
                <TextBlock Width="100" Text="{Binding Path=Valor}"/>
            </StackPanel>
        </DataTemplate>
    </Window.Resources>
```

...

```
<ListBox Grid.Row="11" Grid.ColumnSpan="3" Grid.RowSp="3" ItemsSource="{Binding Path=Disco}"
         ItemTemplate="{StaticResource medidaTemplate}"/>
```

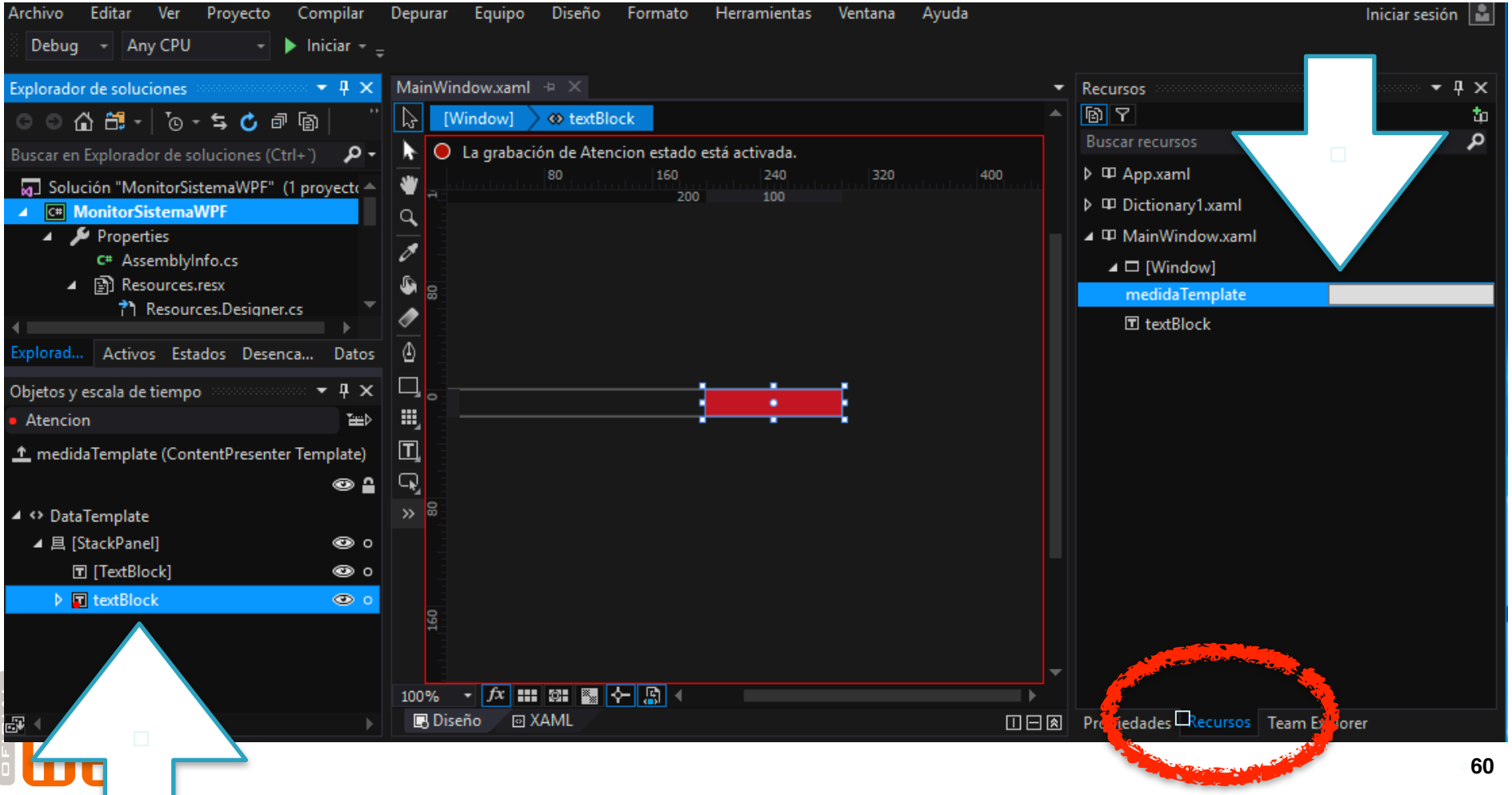
# Actividad clase 6

- Cambiar los Label por DataTemplates en tu solución actual
  - (Solución de partida en Moodle con DataTemplate para
  - Información sobre discos: sesion4-solucion-actividad6.zip)
- Ampliación:
  - Añadir el campo “máximo valor” (como el usado en la memoria) en Medida y emplearlo para todas las Medidas

# ACTIVIDAD 7 (OPCIONAL)

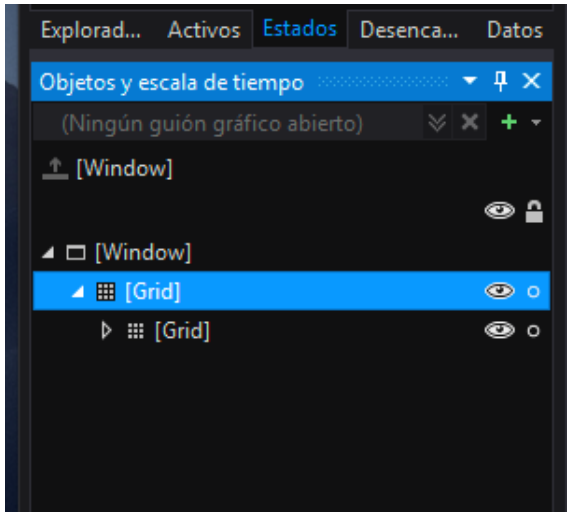
# DataTemplate en Blend

- Podemos personalizar visualmente el DataTemplate en Blend



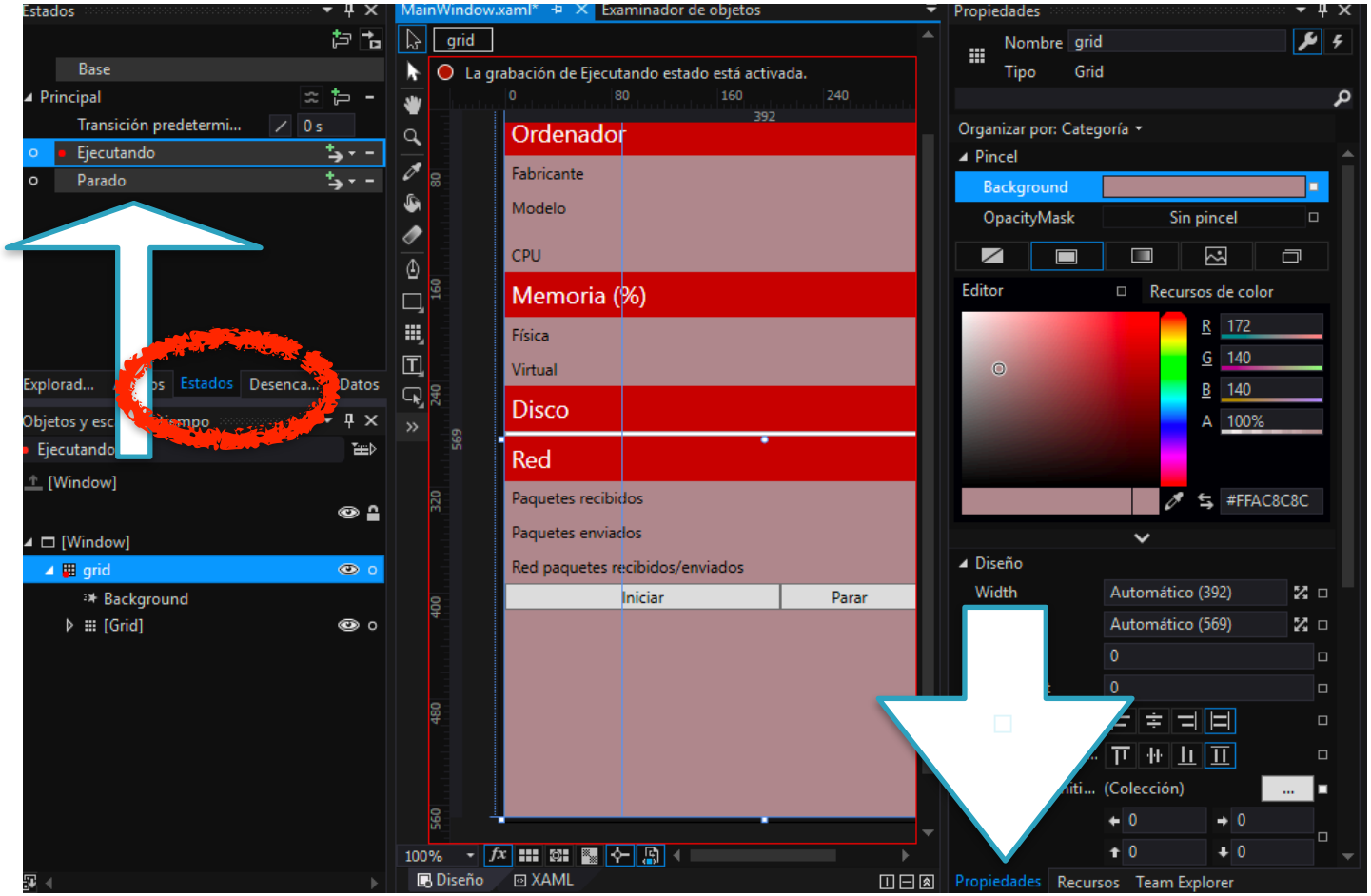
# Estados

- Podemos diseñar el estado del UI usando una máquina de estados incorporada por el sistema
  - Se asocia al elemento activo
  - En este caso vamos a activar el grid principal



# Estados

- Creamos dos estados (podríamos crear también transiciones)
  - Se graban los valores que visualmente cambiemos en la pestaña de propiedades



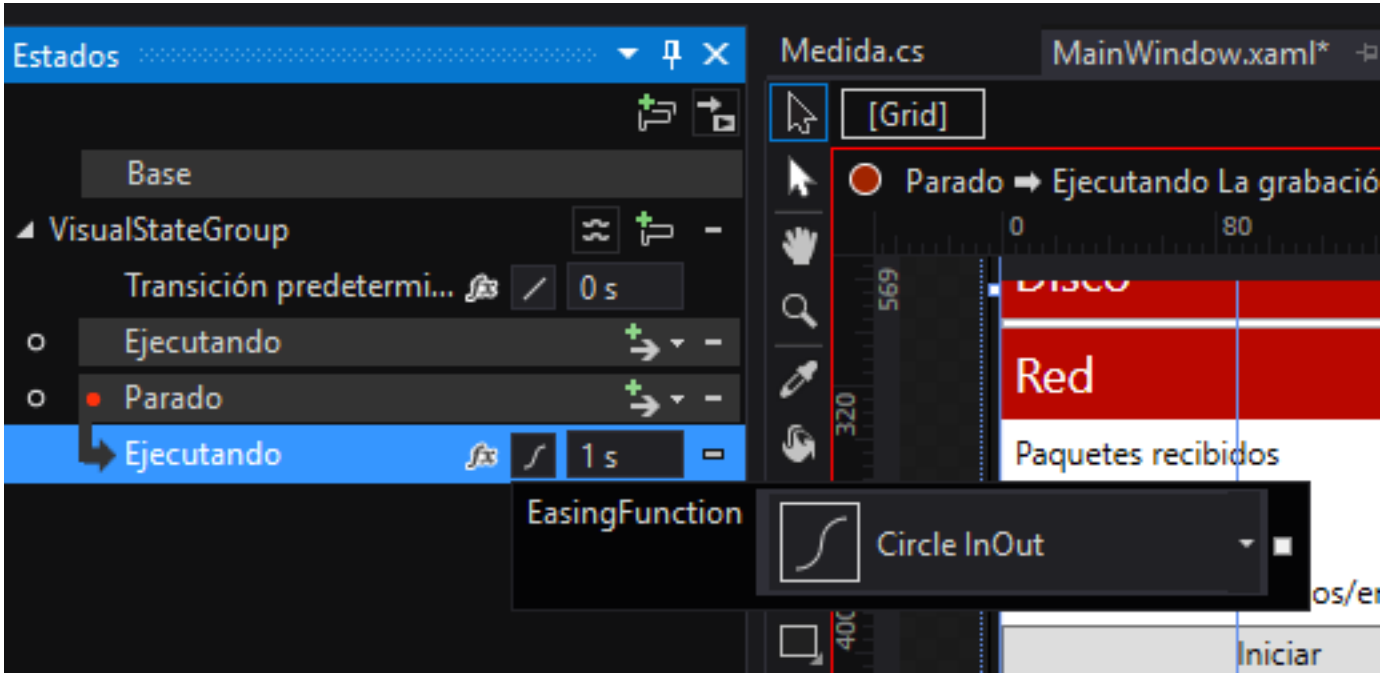
# Estados

- Internamente crea en el XAML:

```
<Window.Resources>
    <DataTemplate x:Key="medidaTemplate">
        <StackPanel Orientation="Horizontal">
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup x:Name="EstadosMedida">
                    <VisualState x:Name="Normal"/>
                    <VisualState x:Name="Advertencia">
                        <Storyboard>
                            <ObjectAnimationUsingKeyFrames Storyboard.TargetProper
                                <DiscreteObjectKeyFrame KeyTime="0">
                                    <DiscreteObjectKeyFrame.Value>
                                        <SolidColorBrush Color="#FFCDDE1E"/>
                                    </DiscreteObjectKeyFrame.Value>
                                </DiscreteObjectKeyFrame>
                            </ObjectAnimationUsingKeyFrames>
                        </Storyboard>
                    </VisualState>
```

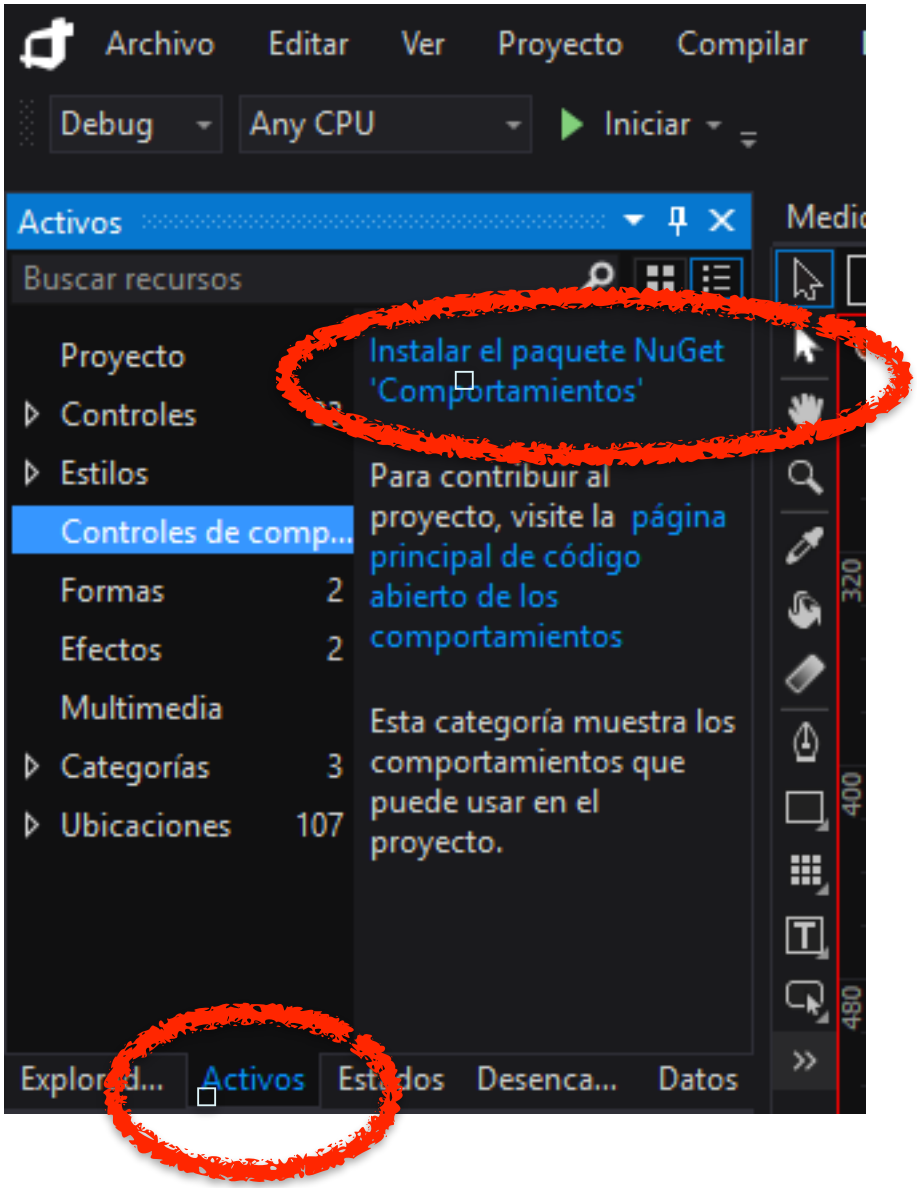
# Estados

- Transiciones: podemos decir cómo se pasa de un estado a otro (tiempo, función de interpolación)

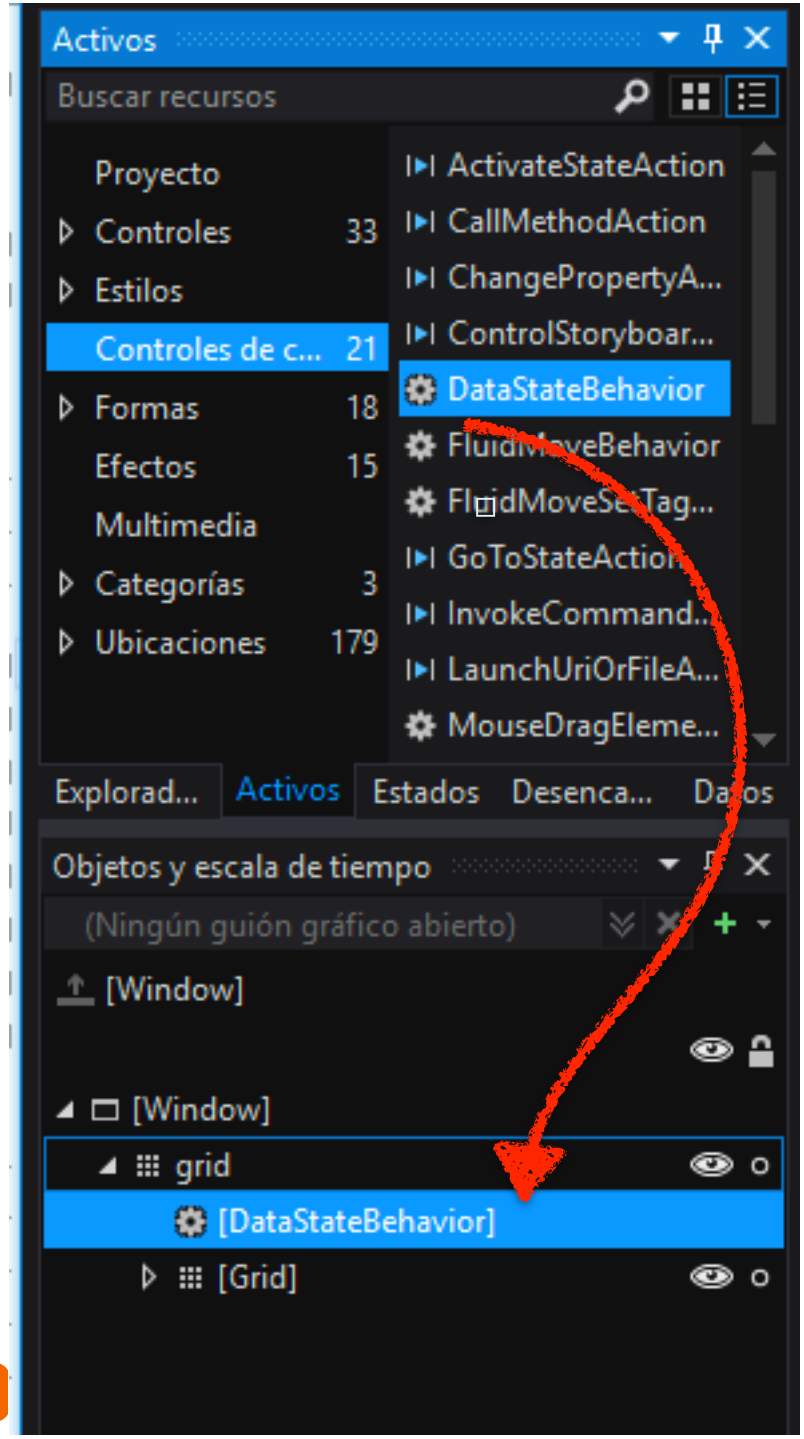


# Vamos a cambiar el estado

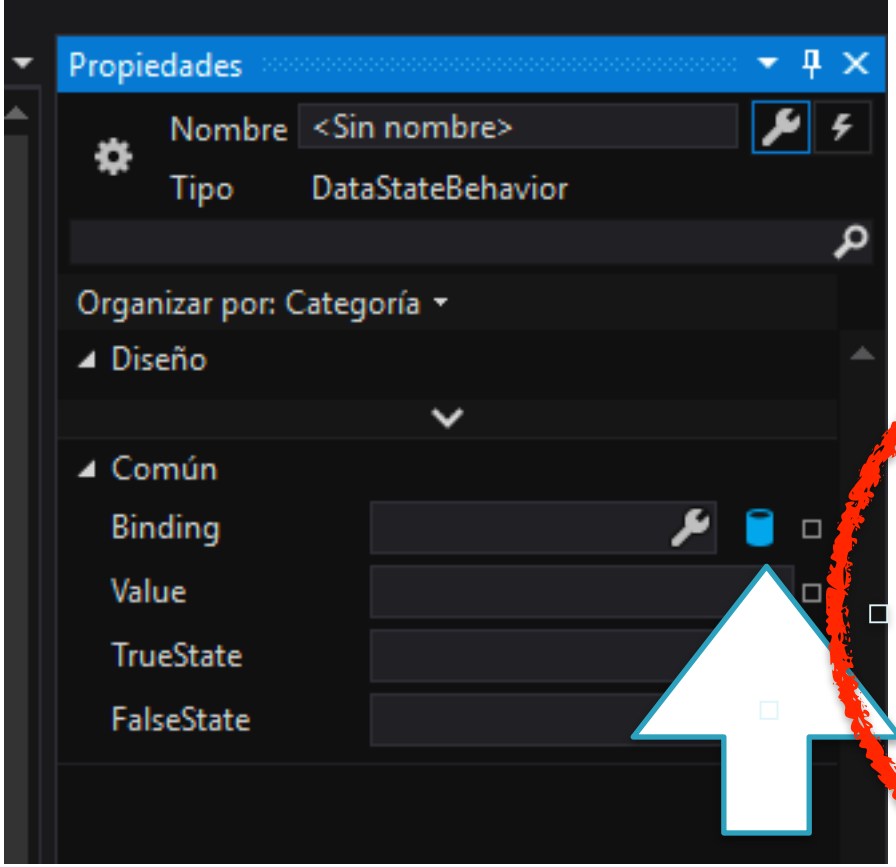
- En C#, podemos usar el método VisualStateManager.goToState, pero éste requiere conocer el control en el code behind
- Usamos Blend para ello a través de los "Controles de Comportamiento"

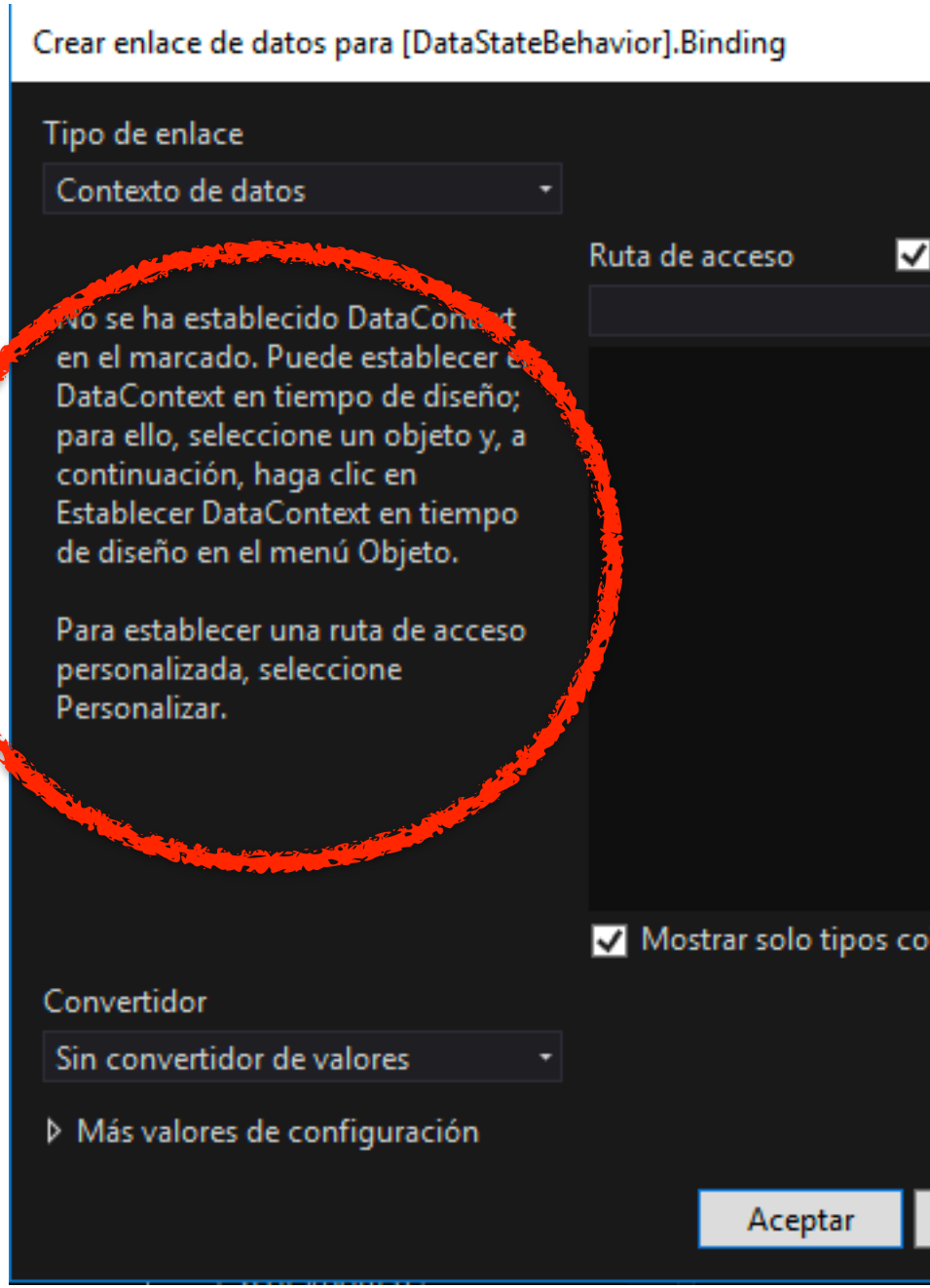



- Por defecto no están instalados: instalamos el SDK




- Añadimos este comportamiento al grid principal




Queremos enlazar el estado al ViewModel

... pero....

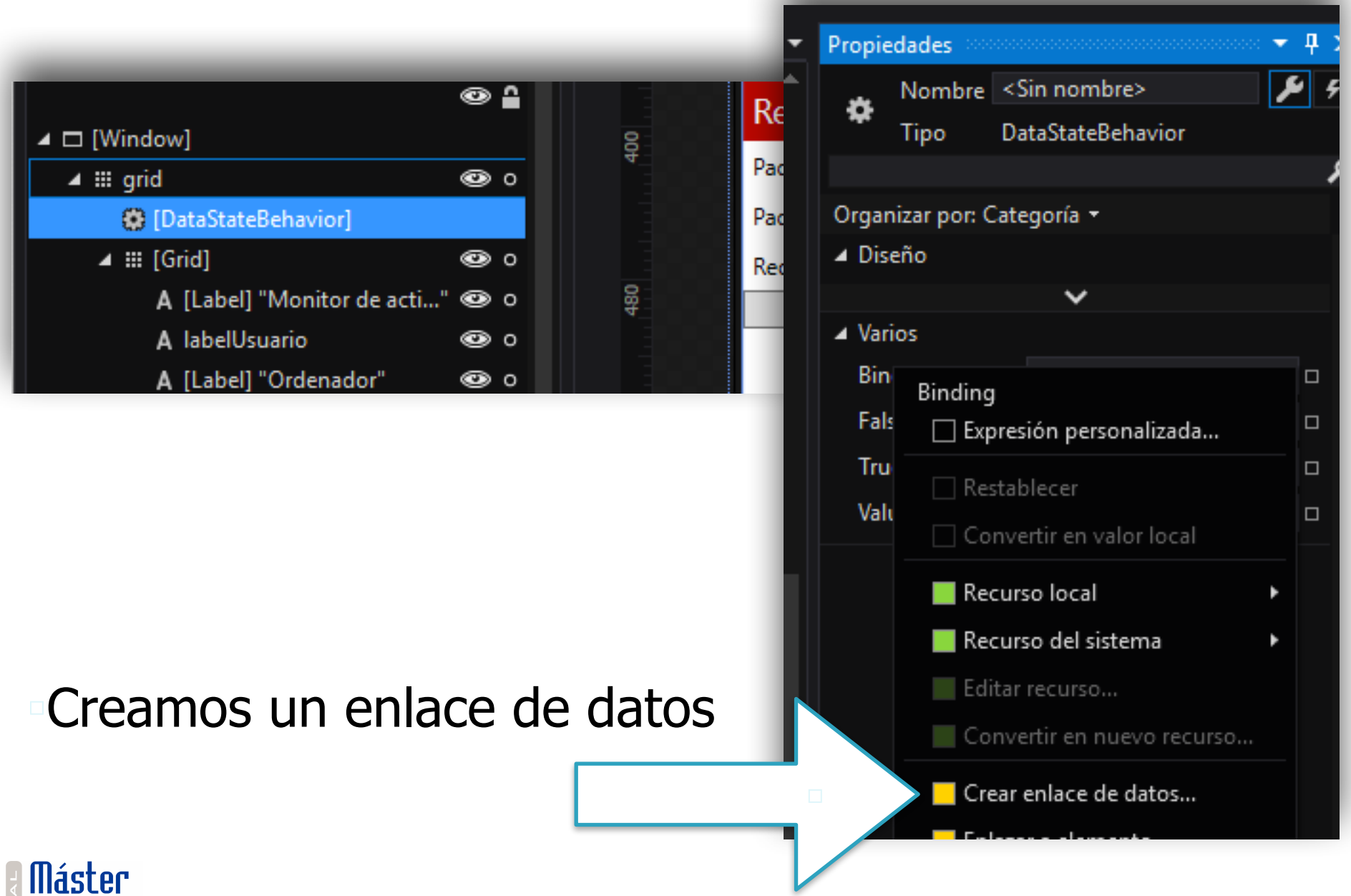
# En lugar de declarar el DataContext en CS lo hacemos en el XAML

```
/// </summary>
public partial class MainWindow : Window
{
    // RecursosSistema recursosSistema;

    public MainWindow()
    {
        // recursosSistema = new RecursosSistema();
        // Lo ponemos en el XAML this.DataContext = recursosSistema;
        InitializeComponent();
```

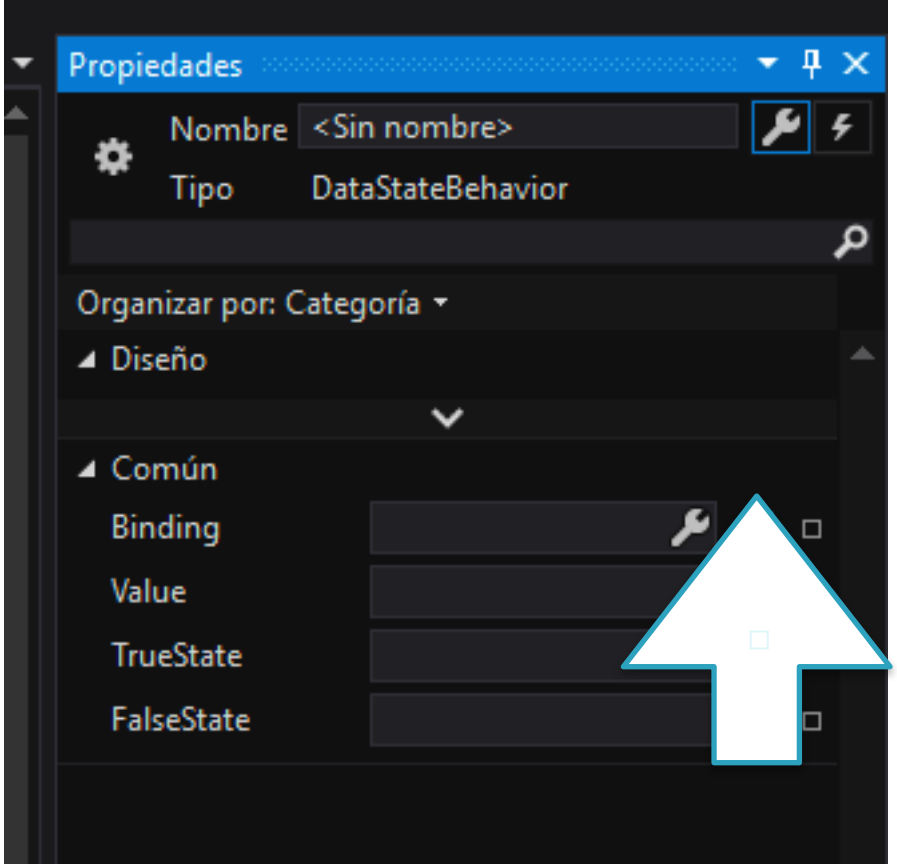
```
xmlns:mc= http://schemas.openxmlformats.org/markup-compatibility/2006
xmlns:local="clr-namespace:MonitorSistema" x:Class="MonitorSistemaWPF.MainWindow"
mc:Ignorable="d"
Title="Monitor de Sistema" Height="600" Width="400">

<Window.DataContext>
    <local:RecursosSistema />
</Window.DataContext>
```

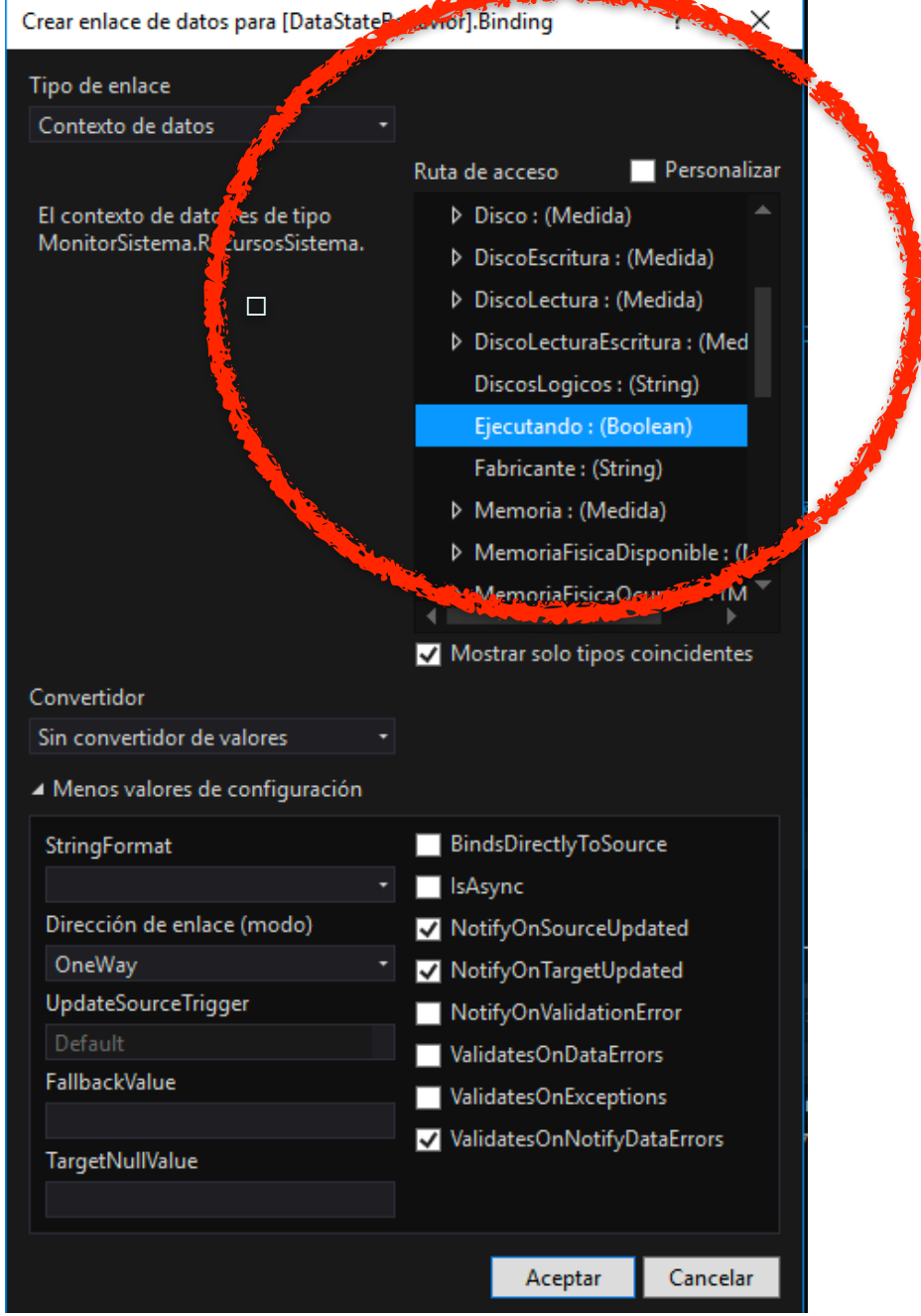
# Creamos una **propiedad** “ejecutando” en el ViewModel

```
public class RecursosSistema : INotifyPropertyChanged
{
    LectorRecursosSistema lectorRecursosSistema;
    System.Windows.Threading.DispatcherTimer dispatcherTimer;

    Medida cpu;
    Medida[] disco;
    Medida[] memoria;
    Medida[] red;
    string fabricante = "N/A";
    string modelo;
    string usuario;
    string procesador;
    string discosLogicos;

    ICommand comandoReset;
    ComandosRecursosSistema comandoIniciar;
    ComandosRecursosSistema  comandoParar;

    public event PropertyChangedEventHandler PropertyChanged;

    public Boolean Ejecutando { get => ejecutandose(); }
```




- Creamos un enlace de datos



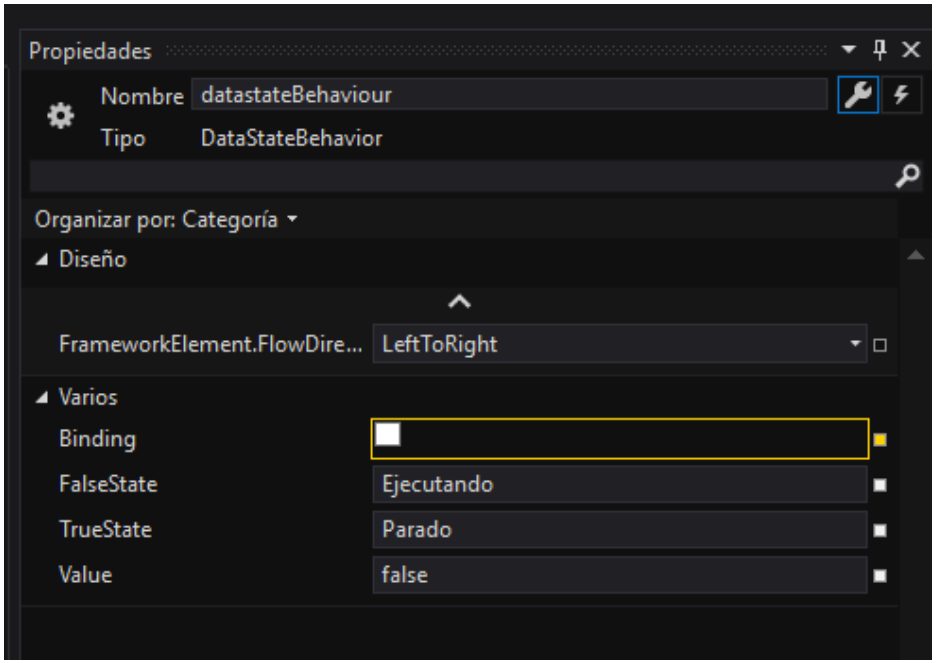




Queremos enlazar el estado al ViewModel

... pero....

NOTA: si no sale la propiedad, debemos compilar

# Cambiamos las condiciones de cambio de estado




```
<Grid x:Name="grid">
    <i:Interaction.Behaviors>
        <ei:DataStateBehavior Binding="{Binding Ejecutando}" TrueState="Parado" FalseState="Ejecutando" Value="false"/>
    </i:Interaction.Behaviors>
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="Principal">
            <VisualState x:Name="Ejecutando">
```

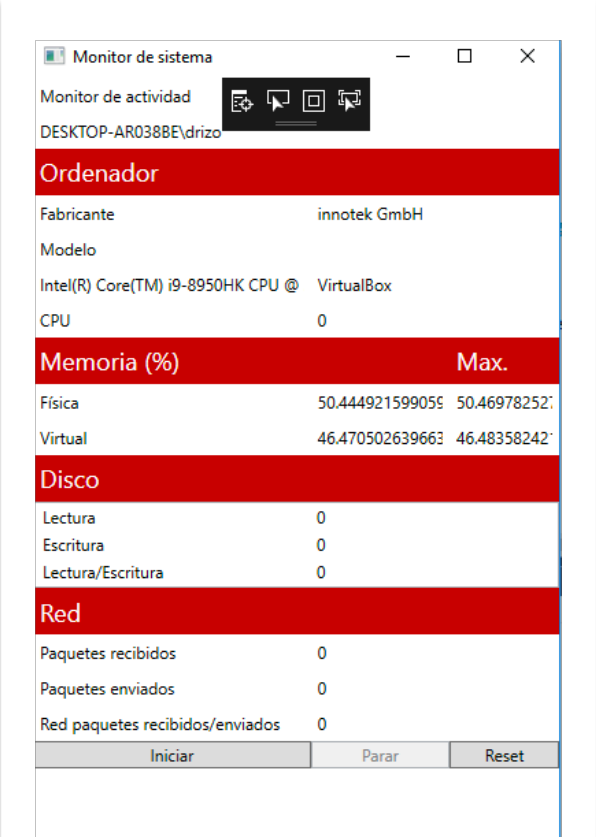
- Si no funciona es porque es una propiedad "no observable"
- La hacemos "observable" como hicimos con Medida

```
public class RecursosSistema : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged; // para el cambio de estado
```

```
void EventoCambioEstado(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}

public void parar()
{
    dispatcherTimer.Stop();
    EventoCambioEstado("Ejecutando");
    comandoIniciar.NotifyCanExecuteChanged();
    comandoParar.NotifyCanExecuteChanged();
}
```

# Actividad clase

- Introduce estados en tu solución actual
  - (Solución en Moodle: sesion4-solucion-actividad7.zip)



# Actividad clase final: ampliación

- Crear una pequeña aplicación de escritorio WPF con MVVM entre éstas
  - Lectura de valores de un tensiómetro
    - El sistema no será asíncrono, sino que irá leyendo en un bucle los valores mientras no reciba un valor que indique que no hay más valores
    - Se mostrará gráficamente el estado Leyendo / Leído con Estados
    - Se usará un DataTemplate para mostrar los valores de tensión sistólica (alta) y tensión diastólica (baja)