



PRÁCTICA 06 – MVVM (2)

Repaso del patrón MVVM

DESARROLLO RÁPIDO DE APLICACIONES

Por Juan José Camacho Hidalgo
UNIVERSIDAD DE ALMERÍA

Contenido

| | | |
|------|---|---|
| 1. | Introducción | 2 |
| 2. | Patrón Modelo-Vista-Modelo de la Vista (MVVM) | 2 |
| 2.1. | MVVMBasicoWpfApp..... | 2 |

1. Introducción

Este documento tiene como objetivo repasar el uso del patrón Modelo-Vista-Modelo de la Vista, mediante ejemplos programados en C# y XAML.

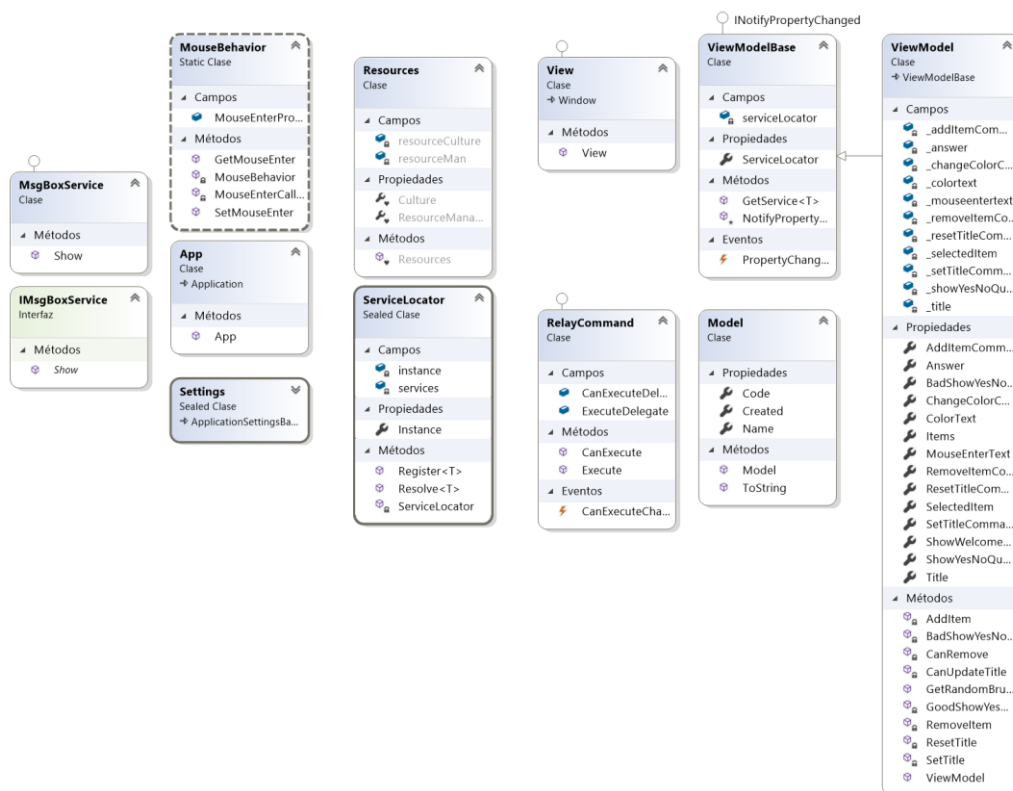
2. Patrón Modelo-Vista-Modelo de la Vista (MVVM)

RECORDATORIO:

El patrón Modelo-Vista-Modelo de la Vista (MVVM) se trata de un tipo de patrón derivado del patrón Modelo-Vista-Presentador (MVP). Su objetivo es separar la visualización de la interfaz de usuario, es decir, las ventanas, de la lógica de visualización (conlleva todo el código que interactúa con el usuario), y del modelo.

2.1. MVVMBasicoWpfApp

Este es el ejemplo que vamos a comentar, de la solución MVVMBasicoWpfApp:

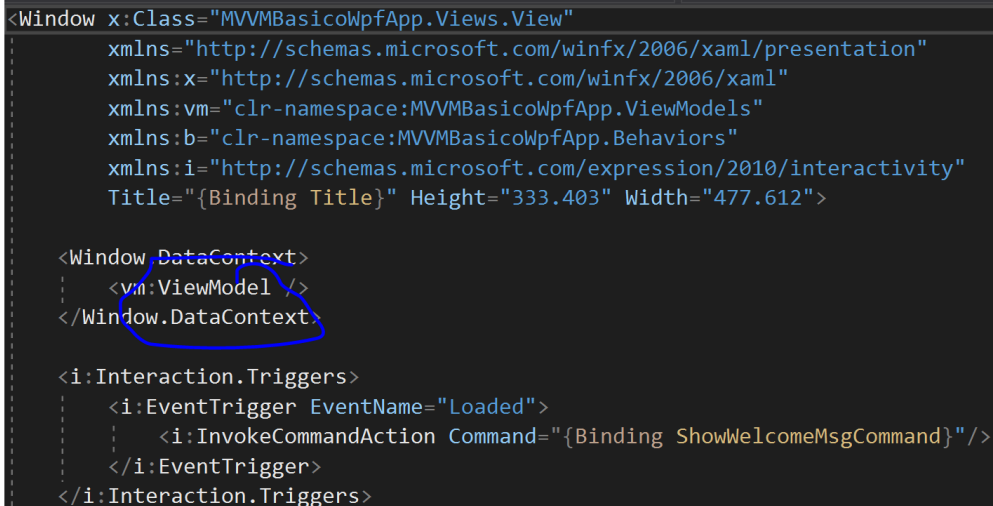


Básicamente este ejemplo tiene como objetivo implementar un ejemplo básico del uso del patrón MVVM. Para ello, se desarrolla una aplicación que tiene como objetivo realizar funciones simples y comunes acerca del uso de las ventanas. En este caso son recibir un mensaje de bienvenida a la app, cambiar el título de la ventana, establecerlo a uno por

defecto, mostrar una pregunta y tomar la respuesta para mostrarla, añadir y eliminar ítems, así como eventos de control sobre el movimiento del ratón.

No se hace uso de eventos directamente, sino que hacemos uso de los comandos asociando una a una cada acción con su control concreto. Como WPF no tiene propiedades para enlazar todos los eventos, se implementa el patrón Attached Behavior y para ello nos proporciona las clases Windows mediante System.Windows.Interactivity cuando esto ocurre.

Atendiendo a la estructura del patrón, como vemos, la vista **View** dispone de un DataContext donde se define el modelo de la vista: la clase **ViewModel**.



```
<Window x:Class="MVVMBasicWpfApp.Views.View"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:vm="clr-namespace:MVVMBasicWpfApp.ViewModels"
        xmlns:b="clr-namespace:MVVMBasicWpfApp.Behaviors"
        xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
        Title="{Binding Title}" Height="333.403" Width="477.612">

    <Window.DataContext>
        <vm:ViewModel />
    </Window.DataContext>

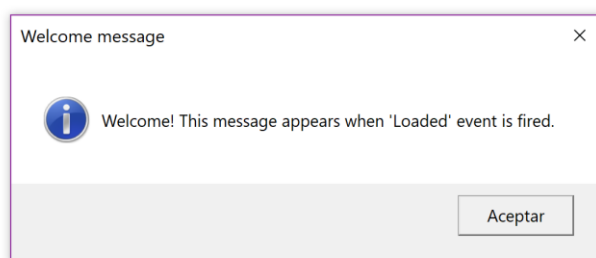
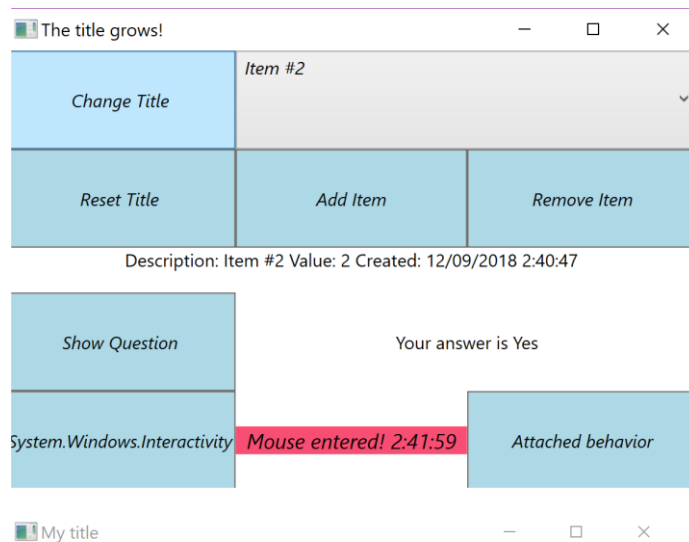
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="Loaded">
            <i:InvokeCommandAction Command="{Binding ShowWelcomeMsgCommand}" />
        </i:EventTrigger>
    </i:Interaction.Triggers>
```

Para realizar todas las operaciones de la interfaz tenemos en primer lugar la clase **RelayCommand** (implementa ICommand) que, mediante delegados, define un método llamado en la invocación del comando y otro para determinar si el comando se puede ejecutar o no.

Cuando se produce un cambio en alguna propiedad, ViewModel mediante la interfaz que implementa, INotifyPropertyChanged, notifica los cambios. Para ello hace uso de un evento.

ViewModelBase es una clase de la que heredan todas las ViewModels, en la cual se incorpora el evento PropertyChanged, ya que es común a todas.

Para crear la funcionalidad de la aplicación ViewModel dispone de los métodos CanUpdateTitle y SetTitle, que se encargan de gestionar la propiedad Title:



Realizan un cambio al título, comprobando previamente que tenga menos de 50 caracteres.

El combobox de la aplicación se realiza mediante SelectedItem de la ViewModel y para eliminarlos y añadirlos se sigue el mismo procedimiento añadiendo métodos que hace uso de este método.

El mensaje para mostrar la cuestión y mostrar la respuesta, se hace mediante un button y un textblock. Las propiedades de estos elementos se encargan de asociarse con ShowYesNoQuestionCommand, uno de los comandos y una propiedad Answer de ViewModel.

Para mostrar esta respuesta se hace uso de las clases **ServiceLocator** y **MsgBoxService** (esta última implementa **IMsgBoxService**). La clase MsgBoxService se encarga de mostrar el mensaje mediante parámetros, mientras que ServiceLocator se trata de un patrón específico de Inversión of Control. Esto permite hacer uso de un servicio para mostrar el mensaje y no incluirlo en la ViewModel. El objeto del mensaje obtenido de dicho servicio, lo obtenemos mediante GetService en la ViewModel.

Para registrar el servicio, el constructor de la clase App crea una instancia:

```
/// </summary>
4 referencias
public partial class App : Application
{
    1 referencia
    public App()
    {
        ServiceLocator.Instance.Register<IMsgBoxService>(new MsgBoxService());
    }
}
```

La clase **MouseBehavior** implementa el patrón Attached Behavior según el cual se definen propiedades asociadas en una clase estática, creando así dependencia. De esta forma enlazamos el modelo de la vista con la vista sin necesidad de comandos. Esto permite obtener la funcionalidad mediante los métodos que se muestran en el código, del cambio de color sobre el movimiento del ratón.

The screenshot shows a WPF application window titled "My title". The window contains a grid of buttons and a status bar. The buttons are labeled "Change Title", "Reset Title", "Add Item", "Remove Item", "Show Question", and "Attached behavior". The status bar displays "Description: Item #2 Value: 2 Created: 12/09/2018 2:40:47". Below the status bar, there is a text area showing "Your answer is Yes" and a green bar with the text "Mouse entered! 2:40:53".

| | | |
|---|------------------------|-------------------|
| Change Title | Item #2 | |
| Reset Title | Add Item | Remove Item |
| Description: Item #2 Value: 2 Created: 12/09/2018 2:40:47 | | |
| Show Question | Your answer is Yes | |
| System.Windows.Interactivity | Mouse entered! 2:40:53 | Attached behavior |

Por último, la clase **Model** representa el modelo de la aplicación:

```
namespace MVVMBasicoWpfApp.Models
{
    7 referencias
    public class Model
    {
        4 referencias
        public int Code { get; set; }
        2 referencias
        public string Name { get; set; }
        2 referencias
        public DateTime Created { get; private set; }

        1 referencia
        public Model()
        {
            Created = DateTime.Now;
        }

        0 referencias
        public override string ToString()
        {
            return String.Format("Description: {0} Value: {1} Created: {2}", Name, Code, Created);
        }
    }
}
```

El modelo nos proporciona un esquema donde define las propiedades Code, Name, y Created, siendo esta última la fecha en tiempo real del sistema.