Comandos

Contenidos

1. Introducción

En capítulos anteriores hemos hablado acerca del modo de tratar eventos, por ejemplo cuando el usuario pulsa un botón. En una interfaz de usuario moderna es normal que una función sea accesible desde varios sitios, siendo por tanto invocada por distintas acciones.

Por ejemplo, en una típica interfaz con un menú principal y una serie de barras de herramientas, una acción como *Nuevo* o *Abrir* puede estar disponible tanto en el menú como en la barra de herramientas o un menú contextual (por ejemplo, al pulsar con el botón derecho en el área principal de la aplicación), además de un atajo de teclado como Control+N o Control+A.

Cada una de estas acciones habitualmente ejecuta el mismo trozo de código. En una aplicación de Windows Forms, tendríamos que definir un evento para cada uno de ellos y luego llamar a una función común. En el ejemplo anterior, esto implica al menos 3 manejadores de eventos, además del código correspondiente para manejar el evento de teclado.

Con WPF, Microsoft trata de mejorar esta situación a través de un concepto denominado comando. Un comando nos permite definir acciones en un único lugar y posteriormente referirnos a las mismas desde los controles de la interfaz de usuario como los que se acaban de mencionar. WPF escucha además eventos de teclado y los traduce al comando apropiado si existe.

Por otro lado, con Windows Forms el programador es el responsable de escribir código que deshabilita los elementos de la interfaz cuando éstos no están disponibles. Por ejemplo, si la aplicación permite usar un comando de portapapeles como *Cortar* pero sólo cuando tenemos texto seleccionado, tendríamos que habilitar y deshabilitar manualmente el comando cada vez que cambia la selección de texto.

Con los comandos WPF esto se realiza de manera centralizada. Con un único método decidimos si queremos que el comando se ejecute o no, tras lo cual WPF habilita o deshabilita a los controles suscritos de manera automática.

1.1. Comandos predefinidos

Si bien podemos implementar nuestros propios comandos, el equipo de WPF ha definido más de 100 comandos de uso habitual divididos en 5 categorías:

- ApplicationCommands
- NavigationCommands
- MediaCommands
- EditingCommands
- ComponentCommands

Por ejemplo, los primeros contienen una gran variedad de acciones comúnmente usadas como *Nuevo, Abrir, Guardar* o *Cortar, Copiar, Pegar*.

2. Uso de comandos WPF

2.1. Command Bindings

Los comandos por sí mismos no hacen nada realmente. En esencia, consisten en una implementación de la interfaz **`ICommand`**, que únicamente define dos métodos:

- **`Execute()`** Ejecuta la acción correspondiente.
- **`CanExecute()`** Determina si la acción está disponible

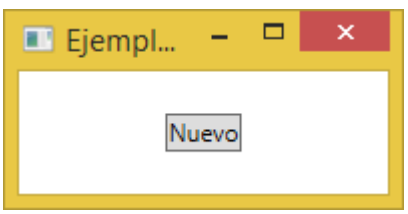
Para llevar a cabo la acción correspondiente, necesitamos un enlace entre el comando y nuestro código, que es donde entra el **`CommandBinding`**.

Un **`CommandBinding`** normalmente se define en una ventana o un control de usuario, y mantiene una referencia al comando correspondiente, además de los manejadores de eventos que permiten tratar los eventos **`Execute()`** y **`CanExecute()`** del comando.

Para ilustrar el uso de comandos, como siempre empezaremos con un ejemplo simple:

```
<Window x:Class="Comandos.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Ejemplo de Comandos" Height="100" Width="200">
    <Window.CommandBindings>
        <CommandBinding Command="ApplicationCommands.New"
              Executed="New_Executed" CanExecute="New_CanExecute" />
    </Window.CommandBindings>

    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
        <Button Command="ApplicationCommands.New">Nuevo</Button>
    </StackPanel>
</Window>
```

```
    namespace
     Comandos
{

    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void New_Executed(object sender, ExecutedRoutedEventArgs e)
        {
            MessageBox.Show("Comando nuevo invocado");
        }

        private void New_CanExecute(object sender, CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
        }
    }
}
```



Definimos el **`CommandBinding`** en la ventana, añadiéndolo a la colección **`CommandBindings`**. Especificamos el **`Command`** que queremos usar (en este caso el comando New de la colección **`ApplicationCommands`**), además de dos manejadores de eventos. La interfaz consiste en un único botón que vinculamos al comando usando la propiedad **`Command`**.

El manejador **`Executed`** simplemente muestra un mensaje indicando que el comando ha sido invocado. Si ejecutamos el ejemplo y pulsamos el botón deberíamos ver dicho mensaje.

Una ventaja de usar un comando predefinido es que tiene un atajo de teclado asociado. Podemos pulsar Control+N en lugar de pulsar el botón –el resultado es el mismo-.

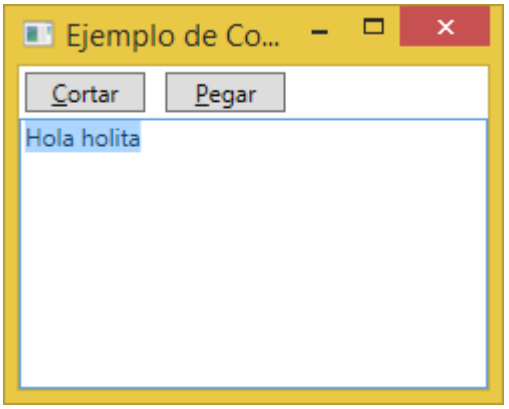
2.2. Uso del método CanExecute

En el primer ejemplo implementamos un manejador de evento para CanExecute que simplemente devolvió verdadero, de modo que el botón está disponible todo el tiempo. Sin embargo, esto no es cierto para todos los botones, pues dependiendo del caso podríamos querer habilitar o deshabilitar el botón en función del estado de la aplicación.

Un ejemplo muy común es la alternancia de botones al usar el portapapeles de Windows, donde queremos que los botones *Cortar* y *Copiar* se habiliten solo si hay texto seleccionado, y el botón *Paste* se habilite únicamente si hay texto en el portapapeles. Esto es precisamente lo que vamos a hacer en este ejemplo:

```
<Window x:Class="Comandos.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Ejemplo de Comandos" Height="200" Width="250">
    <Window.CommandBindings>
        <CommandBinding Command="ApplicationCommands.Cut"
              Executed="Cut_Executed" CanExecute="Cut_CanExecute" />
        <CommandBinding Command="ApplicationCommands.Paste"
              Executed="Paste_Executed" CanExecute="Paste_CanExecute" />
    </Window.CommandBindings>

    <DockPanel>
        <WrapPanel DockPanel.Dock="Top" Margin="3">
            <Button Command="ApplicationCommands.Cut" Width="60">_Cortar</Button>
            <Button Command="ApplicationCommands.Paste"
              Width="60" Margin="10,0">_Pegar</Button>
        </WrapPanel>
        <TextBox AcceptsReturn="True" Name="txtEditor" />
    </DockPanel>
</Window>
```

```
     namespace
      Comandos
{

    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Cut_Executed(object sender, ExecutedRoutedEventArgs e)
        {
            txtEditor.Cut();
        }

        private void Cut_CanExecute(object sender, CanExecuteRoutedEventArgs e)
        {
         if((txtEditor!=null)&&(txtEditor.SelectionLength>0))
             e.CanExecute=true;
        }

        private void Paste_Executed(object sender, ExecutedRoutedEventArgs e)
        {
            txtEditor.Paste();
        }

        private void Paste_CanExecute(object sender, CanExecuteRoutedEventArgs e)
        {
            if (Clipboard.ContainsText())
                e.CanExecute = true;
        }
    }
}
```



Se ha implementado una interfaz simple con un par de botones y un control **TextBox**. El primer botón copia al portapapeles y el segundo pega a partir del contenido del mismo.

En el *Code-Behind* tenemos dos eventos para cada botón:

- El que realiza la acción (evento **Executed**). Para cortar hacemos uso del método **Cut()** correspondiente del **TextBox**, mientras que para pegar usamos el método estático **Paste()** de la clase **Clipboard**. Como se ve no se realiza ninguna comprobación
- El que determina si se realiza la acción (evento **CanExecute**). En este caso se realizan las comprobaciones oportunas, a partir del contenido del **TextBox** en el primer caso y el **Clipboard** en el segundo caso. Si la acción puede ejecutarse devolvemos el valor true.

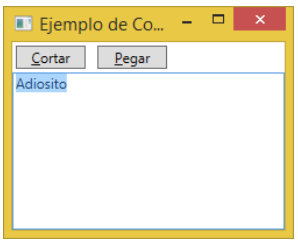
La mejor parte consiste en que no tenemos que llamar a estos métodos para actualizar los botones: WPF lo hace automáticamente cuando la aplicación lo permite, haciendo que la interfaz permanezca actualizada en todo momento.

2.3. Comportamiento de los comandos por defecto

En el ejemplo anterior podríamos haber evitado el código del *Code-Behind*, pues un TextBox puede manejar automáticamente comandos como *Cut, Copy*, *Paste*, *Undo* y *Redo.*

WPF lleva a cabo estas operaciones manejando los eventos **Executed** y **CanExecute** de manera automática cuando un evento de texto como el **TextBox** tiene el foco. Podemos sobrescribir estos eventos (que es lo que hemos hecho en el ejemplo anterior), pero si queremos el comportamiento por defecto podemos dejar que WPF conecte los comandos con el control y haga el trabajo por nosotros:

```
<Window x:Class="Comandos.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Ejemplo de Comandos" Height="200" Width="250">

    <DockPanel>
        <WrapPanel DockPanel.Dock="Top" Margin="3">
            <Button Command="ApplicationCommands.Cut"
              CommandTarget="{Binding ElementName=txtEditor}" Width="60">_Cortar</Button>
            <Button Command="ApplicationCommands.Paste"
              CommandTarget="{Binding ElementName=txtEditor}"
              Width="60" Margin="10,0">_Pegar</Button>
        </WrapPanel>
        <TextBox AcceptsReturn="True" Name="txtEditor" />
    </DockPanel>
</Window>
```



Este ejemplo no lleva *Code-Behind*, pues WPF trata todos los eventos por nosotros. A través de la propiedad **CommandTarget** del botón vinculamos su comando asociado al TextBox. Nótese que en este caso no es necesario definir la lista de comandos desde la ventana como hacíamos en el ejemplo anterior.

3. Implementación de comandos a medida

Hasta ahora hemos visto varias maneras de usar comandos predefinidos en WPF, pero como ya se comentó podemos definir nuestros propios comandos.

La manera más sencilla de implementar nuestros propios comandos es tener una clase estática que los contenga. Cada comando es añadido a dicha clase como cambios estáticos, lo que nos permite usarlos en nuestra aplicación. Dado que WPF no implementa un comando de Salir (*Exit*), vamos a implementarlo en nuestros propios comandos a medida:

```
<Window x:Class="Comandos.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
        " xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Comandos"
        Title="Ejemplo de Comandos" Height="100" Width="300">

    <Window.CommandBindings>
        <CommandBinding Command="local:ComandosMedida.Salir"
                CanExecute="Salir_CanExecute" Executed="Salir_Executed" />
    </Window.CommandBindings>
    <Grid>
        <Button
                VerticalAlignment="Center" HorizontalAlignment="Center"
                 Command="local:ComandosMedida.Salir">_Salir</Button>
    </Grid>
</Window>
```

```
    namespace
     Comandos
{

    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Salir_CanExecute(object sender, CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
        }
        private void Salir_Executed(object sender, ExecutedRoutedEventArgs e)
        {
            Application.Current.Shutdown();
        }
    }
    public static class ComandosMedida
    {
         public static readonly RoutedUICommand Salir = new RoutedUICommand
                ("Salir", "Salir", typeof(ComandosMedida));
    }
}
```

Únicamente se ha definido un botón que utiliza nuestro comando a medida **`Salir`**. El comando está definido en el *Code-Behind*, en la clase **`ComandosMedida`**, y luego referenciado en la colección CommandBindings de la ventana, donde asignamos los eventos que debería usar para ejecutar y comprobar si puede ser ejecutado.

Todo ocurre igual que en los primeros ejemplos del capítulo, con la excepción de estamos referenciando un comando de nuestro propio código (usando el espacio de nombres local que hemos definido en la parte superior) en lugar de un comando predefinido.

El evento que trata el comando simplemente llama a **`Shutdown`** que finaliza nuestra aplicación.

Como ya se explicó previamente, implementamos nuestro comando Salir como un campo en la clase estática ComandosMedida.