PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

UT2.3. Interacción del usuario con aplicaciones móviles multiplataforma

Departamento de Informática y Comunicaciones

CFGS Desarrollo de Aplicaciones Multiplataforma

Segundo Curso

IES Virrey Morcillo

Villarrobledo (Albacete)

Índice

1. Eventos

Un **evento** o **event** es un mensaje que envía un objeto cuando ocurre una acción. La acción podría ser debida a la interacción del usuario, como hacer clic en un botón, o podría proceder de cualquier otra lógica del programa, como el cambio del valor de una propiedad. El objeto que provoca el evento se conoce como emisor del evento. El emisor del evento no sabe qué objeto o método recibirá o controlará los eventos que genera.

En Xamarin.Forms se pueden manejar los eventos a través de las siguientes maneras:

1. Mediante XAML.
2. Mediante código usando la sintaxis estándar.
3. Mediante código usando las expresiones lambda.

1.1. Manejo de eventos desde XAML

En esta forma de manejar los eventos declararemos el nombre del manejador del evento directamente en el XAML y en el código subyacente se creará el evento donde pondremos el código de respuesta. Por ejemplo, tenemos una página XAML y tenemos un botón el cual declararíamos de la siguiente manera:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ButtonDemos.BasicButtonClickPage"
             Title="Basic Button Click">
    <StackLayout>

        <Label x:Name="label"
               Text="Click the Button below"
               FontSize="Large"
               VerticalOptions="CenterAndExpand"
               HorizontalOptions="Center" />

        <Button Text="Click to Rotate Text!"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center"
                Clicked="OnButtonClicked" />

    </StackLayout>
</ContentPage>
```

En el código subyacente quedaría como sigue:

```
public partial class BasicButtonClickPage : ContentPage
{
    public BasicButtonClickPage ()
    {
        InitializeComponent ();
    }

    async void OnButtonClicked(object sender, EventArgs args)
    {
        await label.RelRotateTo(360, 1000);
    }
}
```

1.2. Manejo de eventos mediante código usando la sintaxis estándar

Si sabes programar en el lenguaje C# podemos utilizar la sintaxis tradicional para el manejo de eventos, esto se puede utilizar para todos los elementos visuales y cualquier otro objeto. Esta técnica está basada en el operador de sobrecarga += y es una técnica muy recomendada ya que podemos desasignar un evento a través del uso del operador sobrecargado +=, optimizando el uso de memoria del dispositivo.

En el siguiente ejemplo de código se muestra el uso de la sintaxis estándar para manejar el evento clic en un botón.

```
public class CodeButtonClickPage : ContentPage
{
    public CodeButtonClickPage ()
    {
        Title = "Code Button Click";

        Label label = new Label
        {
            Text = "Click the Button below",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            VerticalOptions = LayoutOptions.CenterAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };

        Button button = new Button
        {
            Text = "Click to Rotate Text!",
            VerticalOptions = LayoutOptions.CenterAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };
        button.Clicked += async (sender, args) => await label.RelRotateTo(360, 1000);

        Content = new StackLayout
        {
            Children =
            {
                label,
                button
            }
        };
    }
}
```

1.3. Manejo de eventos mediante código usando las expresiones lambda

Las expresiones lambda son un tipo de método anónimo, los cuales no requieren la declaración del cuerpo del método por separado, todo se hace en un solo lado, cumpliendo con la signatura del delegado en el que se basa el evento. El compilador infiere los tipos de los argumentos por lo que tampoco es necesario que se especifique de manera explícita.

```
button.Click += (sender, args) => {
    clickCount += 1;    // access variable in surrounding code
    button.Text = string.Format ("Clicked {0} times.", clickCount);
};
```

2. Enlaces de datos

Una aplicación de Xamarin.Forms consta de una o varias páginas, cada una de los cuales está basada en un determinado diseño y generalmente contienen varios objetos de interfaz de usuario denominados vistas.

Una de las tareas principales de la aplicación es para mantener estas vistas sincronizadas y realizar un seguimiento de los distintos valores o selecciones que representan. A menudo las vistas representan valores de un origen de datos subyacente y el usuario manipula estas vistas para cambiar esos datos. Cuando se cambia la vista, los datos subyacentes deben reflejar ese cambio, y de forma similar, cuando cambian los datos subyacentes, ese cambio debe reflejarse en la vista.

Para controlar este trabajo correctamente, la aplicación debe recibir una notificación de cambios en estas vistas o en los datos subyacentes. La solución habitual consiste en definir los eventos que indican cuando se produce un cambio. A continuación, se instala un controlador de eventos que se le informa de estos cambios. Responde mediante la transferencia de datos de un objeto a otro. Sin embargo, cuando existen muchas vistas, también debe haber muchos controladores de eventos y se genera una gran cantidad de código.

La solución a este problema está en los **enlaces de datos** o **data bindings**. Los enlaces de datos automatizan este trabajo y representan a los controladores de eventos innecesarios. Los enlaces de datos pueden implementarse en el código o en XAML, pero son más frecuentes en XAML, ya que ayudan a reducir el tamaño del archivo de código subyacente. Al reemplazar el código de procedimientos en los controladores de eventos con código declarativo o de marcado, la aplicación es más sencilla de entender y se simplifica.

Uno de los dos objetos implicados en un enlace de datos casi siempre es un elemento que se deriva de View y forma parte de la interfaz visual de una página, el otro objeto es:

- Otro View derivado, normalmente en la misma página.
- Un objeto en un archivo de código.

De esta manera, los enlaces de datos permiten que las propiedades de dos objetos se vinculen para que un cambio en una de ellas provoque un cambio en la otra. Este es un mecanismo muy importante que se puede definir completamente en código XAML.

Los enlaces de datos vinculan propiedades de dos objetos, denominados origen y destino. Atendiendo al código fuente es necesario seguir los siguientes dos pasos:

1. La propiedad BindingContext del objeto destino debe ser inicializada al objeto origen.
2. El método SetBinding, del objeto destino debe ser llamado para vincular una propiedad de ese objeto con una propiedad del objeto origen.

La propiedad destino debe ser una propiedad enlazable, esto quiere decir que el objeto destino debe heredar de la clase BindableObject. Por ejemplo, la propiedad Text de un objeto Label se asocia con la propiedad enlazable TextProperty.

2.1. Enlaces básicos

En el código de marcado XAML, se deben también llevar a cabo los mismos dos pasos anteriores, a no ser que el elemento Binding tome el lugar de la llamada SetBinding y la clase Binding.

Sin embargo, cuando se declaran enlaces de datos en XAML existen varias alternativas para establecer el BindingContext del objeto destino. En algunas ocasiones se establece desde el archivo del código subyacente, en otras como el contenido de etiquetas de elementos de propiedad.

Se pueden definir enlaces de datos para enlazar propiedades de dos elementos de tipo Views en la misma página. En este caso, se debe inicializar el atributo BindingContext del objeto destino utilizando la extensión de marcado x:Reference.

Consideremos el siguiente ejemplo donde se define el enlace de datos completamente mediante código XAML:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataBindingDemos.BasicXamlBindingPage"
             Title="Basic XAML Binding">
    <StackLayout Padding="10, 0">
        <Label Text="TEXT"
               FontSize="80"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand"
               BindingContext="{x:Reference Name=slider}"
               Rotation="{Binding Path=Value}" />

        <Slider x:Name="slider"
                Maximum="360"
                VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

El enlace de datos se establece en el objeto destino, en este caso la Label. Se consideran dos extensiones de marcado XAML fácilmente reconocibles por estar delimitadas por llaves { }:

- La extensión de marcado **x:Reference** se necesita para referenciar el objeto origen, en este caso el **Slider** llamado **slider**.
- La extensión de marcado **Binding** vincula la propiedad **Rotation** del objeto **Label** con la propiedad **Value** del objeto **Slider**.

Las extensiones de marcado de XAML como x:Reference y Binding pueden tener definiciones de atributos de tipo content property, esto quiere decir que el nombre de la propiedad no es necesario. La propiedad Name es el content property de x:Reference y la propiedad Path es el content property de Binding, con lo cual es posible simplificar el código XAML del diseño del objeto Label de la siguiente manera:

```
<Label Text="TEXT"
       FontSize="80"
       HorizontalOptions="Center"
       VerticalOptions="CenterAndExpand"
       BindingContext="{x:Reference slider}"
       Rotation="{Binding Value}" />
```

2.2. Enlaces sin contexto de enlace

La propiedad BindingContext es un componente importante en el contexto del enlace de datos, pero no siempre es necesario. El objeto origen se puede especificar en la llamada SetBinding o en la extensión de marcado Binding.

El archivo de código XAML es similar al ejemplo anterior excepto que el objeto Slider se define para controlar la propiedad Scale del objeto Label. Por este motivo, el Slider se inicializa para el rango de –2 a 2:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataBindingDemos.AlternativeXamlBindingPage"
             Title="Alternative XAML Binding">
    <StackLayout Padding="10, 0">
        <Label Text="TEXT"
               FontSize="40"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand"
               Scale="{Binding Source={x:Reference slider},
                               Path=Value}" />

        <Slider x:Name="slider"
                Minimum="-2"
                Maximum="2"
                VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

Ahora la extensión de marcado Binding inicializa dos propiedades, Source y Path, separadas por una coma.

Aunque las extensiones de marcado de delimitan habitualmente por llaves { }, pueden también ser representadas como elementos de objeto:

```
<Label Text="TEXT"
       FontSize="40"
       HorizontalOptions="Center"
       VerticalOptions="CenterAndExpand">
    <Label.Scale>
        <Binding Source="{x:Reference slider}"
                 Path="Value" />
    </Label.Scale>
</Label>
```

Ahora las propiedades Source y Path son atributos normales en XAML, es decir, los valores aparecen sin comillas y los atributos no están separados por una coma. La extensión de marcado x:Reference puede convertirse en un elemento de objeto:

```
<Label Text="TEXT"
       FontSize="40"
       HorizontalOptions="Center"
       VerticalOptions="CenterAndExpand">
    <Label.Scale>
        <Binding Path="Value">
            <Binding.Source>
                <x:Reference Name="slider" />
            </Binding.Source>
        </Binding>
    </Label.Scale>
</Label>
```

Esta sintaxis no es demasiado común, pero a veces es necesaria cuando se involucran objetos más complejos.

2.3. Herencia en los enlaces de datos

La configuración de la propiedad BindingContext se hereda a través del árbol visual de elementos. Consideramos el siguiente fragmento de código:

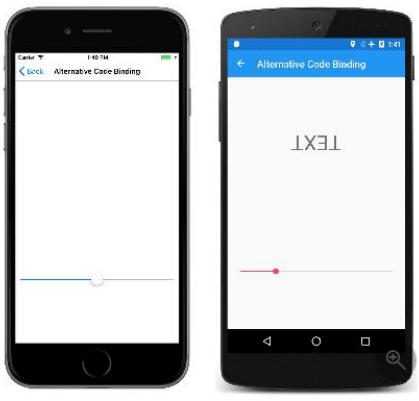
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataBindingDemos.BindingContextInheritancePage"
             Title="BindingContext Inheritance">
    <StackLayout Padding="10">

        <StackLayout VerticalOptions="FillAndExpand"
                     BindingContext="{x:Reference slider}">

            <Label Text="TEXT"
                   FontSize="80"
                   HorizontalOptions="Center"
                   VerticalOptions="EndAndExpand"
                   Rotation="{Binding Value}" />

            <BoxView Color="#800000FF"
                     WidthRequest="180"
                     HeightRequest="40"
                     HorizontalOptions="Center"
                     VerticalOptions="StartAndExpand"
                     Rotation="{Binding Value}" />
        </StackLayout>

        <Slider x:Name="slider"
                Maximum="360" />

    </StackLayout>
</ContentPage>
```

La propiedad BindingContext del diseño StackLayout se inicializa al objeto Slider. Este contexto de enlace es heredado por los objetos View y BoxView, ambos objetos tienen sus propiedades Rotation inicializadas al valor de la propiedad del Slider.

2.4. Modos de enlace

En los ejemplos anteriores se ha tenido en cuenta un objeto de tipo Label donde su propiedad Scale toma un valor de la propiedad Value de un objeto de tipo Slider.

Debido a que el valor inicial del Slider es 0, esto provocó que la propiedad Scale de la Label se inicializara en 0 en lugar de 1, y la Label desapareció.



El siguiente ejemplo es similar a los anteriores, excepto que el enlace de datos se define en el Slider en lugar de en la Label:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataBindingDemos.ReverseBindingPage"
             Title="Reverse Binding">
    <StackLayout Padding="10, 0">

        <Label x:Name="label"
               Text="TEXT"
               FontSize="80"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

        <Slider x:Name="slider"
                VerticalOptions="CenterAndExpand"
                Value="{Binding Source={x:Reference label},
                                Path=Opacity}" />
    </StackLayout>
</ContentPage>
```

A primera vista, esto puede parecer un retroceso, ahora la Label es el origen del enlace de datos, y el Slider es el destino. El enlace hace referencia a la propiedad Opacity de la Label, que tiene un valor predeterminado de 1.

Como es de esperar, el Slider se inicializa con el valor 1 del valor de inicial de la propiedad Opacity del objeto Label.

Pero sorprendentemente el Slider sigue funcionando. Esto parece insinuar que el enlace de datos funciona mejor cuando el Slider es el destino de enlace en lugar de la Label, ya que la inicialización funciona como podríamos esperar.

La explicación de este misterio está en el modo del enlace.

Por otro lado, un único elemento de tipo View puede tener enlace de datos en varias de sus propiedades. Sin embargo, cada elemento de tipo View puede tener sólo un BindingContext, de esta manera el enlace de datos en ese elemento de tipo View debe vincular propiedades del mismo objeto.

La solución a este y otros problemas se encuentra en la propiedad Mode, la cual se inicializa a un miembro de la enumeración BindingMode que puede tener los siguientes valores:

- Default.
- OneWay. Los valores se transfieren desde el objeto origen al destino
- OneWayToSource. Los valores se transfieren desde el objeto destino al origen
- TwoWay. Los valores se transfieren en ambos sentidos entre los objetos origen y destino.
- OneTime. Los valores se transfieren desde el origen al destino, pero únicamente cuando cambia el BindingContext.

Cada propiedad enlazable tiene un modo de enlace por defecto que se establece cuando dicha propiedad se crea, el cual está disponible desde la propiedad DefaultBindingMode del objeto BindableProperty. Este modo de enlace por defecto indica el modo efectivo de cuando esa propiedad es un destino de enlace de datos.

El modo de enlace por defecto para la mayoría de las propiedades tales como Rotation, Scale y Opacity es OneWay. Cuando estas propiedades son destinos de enlaces de datos, la propiedad destino se inicializa desde el origen.

Sin embargo, el modo de enlace de datos por defecto para la propiedad Value de un objeto Slider, por ejemplo, es TwoWay. Esto significa que cuando el valor de la propiedad Value es un destino de enlace de datos, el destino se inicializa desde el origen, como es lo normal, pero el origen se inicializa también desde el destino. Esto permite que el objeto Slider sea inicializado desde el valor inicial de Opacity.

2.5. Formato de las cadenas

A veces es conveniente usar enlaces de datos para mostrar la representación de cadena de un objeto o valor. Por ejemplo, es posible que se quiera utilizar una Label para mostrar el valor actual de un Slider. En este enlace de datos, el Slider es el origen y el destino es la propiedad Text de la Label.

Esta facilidad se transfiere a los enlaces de datos estableciendo la propiedad StringFormat de Binding (o la propiedad StringFormat de la extensión de marcado de Binding) a una cadena de formato estándar .NET con un marcador de posición:

```
<Slider x:Name="slider" />
<Label Text="{Binding Source={x:Reference slider},
                      Path=Value,
                      StringFormat='The slider value is {0:F2}'}" />
```

Hay que tener en cuenta que la cadena de formato está delimitada por caracteres de comilla simple (apóstrofe) para ayudar al analizador XAML a evitar que se traten las llaves como otra extensión de marcado XAML. De lo contrario, esa cadena sin el carácter de comillas simples es la misma que usaría para mostrar un valor de punto flotante en una llamada a String.Format. Una especificación de formato de F2 hace que el valor se muestre con dos decimales.

La propiedad StringFormat sólo tiene sentido cuando la propiedad de destino es de tipo string, y el modo de enlace es OneWay o TwoWay. Para los enlaces TwoWay, el StringFormat solo es aplicable para los valores que pasan del origen al destino.

3. Gestos

En las aplicaciones móviles la pantalla táctil permite una interacción profunda del usuario con la misma. Gracias a la pantalla táctil el usuario puede interactuar utilizando gran variedad de gestos. Los **gestos**, en definitiva, son las interacciones que el usuario hace con la pantalla táctil utilizando uno o dos dedos.

Los gestos permitidos en la mayoría de los dispositivos móviles son los siguientes:

- **Tap**. Equivale a un toque.
- **Doble Tap**. Toque doble.
- **Pan**. Sirve para mover algo en horizontal o vertical, manteniéndolo apretado.
- **Swipe**. Similar a pan pero más rápido se usa, por ejemplo, para pasar una página.
- **Press**. Cuando pulsamos y mantenemos apretado.
- **Pinch**. Con dos dedos, los separamos o juntamos para, por ejemplo, hacer zoom.
- **Rotate**. Con dos dedos, sirve para rotar algo es una u otra dirección.

En Xamarin.Forms, los diferentes objetos de tipo **View** reconocen los gestos **tap**, **pinch**, **pan** y **swipe** por medio de la clase base **GestureRecognizer**.

3.1. Tap

Para crear un elemento de la interfaz de usuario en el que puedas hacer clic mediante el gesto tap, hay que crear una instancia de la clase **TapGestureRecognizer**, gestionar el evento **Tapped** y añadir el nuevo gesto reconocido a la colección **GestureRecognizers** del elemento de la interfaz de usuario.

Se puede añadir un reconocedor de gestos a un elemento en XAML utilizando las propiedades asociadas. Una **propiedad asociada** (attached property) es un tipo especial de propiedad vinculada, definida en una clase pero asociada a otro objeto, y reconocido en XAML como un atributo que contiene una clase y un nombre de propiedad separado por un punto. Las propiedades asociadas permiten a un objeto asignar un valor a una propiedad que no está definida en su propia clase.

```
<Image Source="tapped.jpg">
    <Image.GestureRecognizers>
        <TapGestureRecognizer
                Tapped="OnTapGestureRecognizerTapped"
                NumberOfTapsRequired="2" />
  </Image.GestureRecognizers>
</Image>
```

```
void OnTapGestureRecognizerTapped(object sender, EventArgs args)
{
    tapCount++;
    var imageSender = (Image)sender;
    // watch the monkey go from color to black&white!
    if (tapCount % 2 == 0) {
        imageSender.Source = "tapped.jpg";
    } else {
        imageSender.Source = "tapped_bw.jpg";
    }
}
```

3.2. Pinch

El gesto pinch se utiliza para realizar un zoom interactivo y se implementa utilizando la clase **PinchGestureRecognizer**. Un escenario común para el gesto pinch suele ser la realización de acciones de aumentar y disminuir sobre una imagen.

Para crear un elemento de la interfaz de usuario en el que puedas hacer zoom mediante el gesto pinch, se debe crear una instancia de la clase **PinchGestureRecognizer**, gestionar el evento **PinchUpdated**, añadir el nuevo gesto reconocido a la colección **GestureRecognizers** del elemento de la interfaz de usuario.

```
<Image Source="waterfront.jpg">
  <Image.GestureRecognizers>
    <PinchGestureRecognizer PinchUpdated="OnPinchUpdated" />
  </Image.GestureRecognizers>
</Image>
```

```
void OnPinchUpdated (object sender, PinchGestureUpdatedEventArgs e)
{
  // Handle the pinch
}
```

3.3. Pan

El gesto pan se utiliza para detectar el movimiento de dedos alrededor de la pantalla y aplicar ese movimiento al contenido. Se implementa utilizando la clase **PanGestureRecognizer**. Un escenario común para el gesto pan puede ser el desplazamiento horizontal y vertical de una imagen.

Para crear un elemento de la interfaz de usuario en el que puedas desplazar mediante el gesto pan, se debe crear una instancia de la clase **PanGestureRecognizer**, gestionar el evento **PanUpdated** y añadir el nuevo gesto reconocido a la colección **GestureRecognizers** del elemento de la interfaz de usuario.

```
<Image Source="MonoMonkey.jpg">
  <Image.GestureRecognizers>
    <PanGestureRecognizer PanUpdated="OnPanUpdated" />
  </Image.GestureRecognizers>
</Image>
```

```
void OnPanUpdated (object sender, PanUpdatedEventArgs e)
{
  // Handle the pan
}
```

3.4. Swipe

Un gesto swipe se produce cuando un dedo se mueve a través de la pantalla en dirección horizontal o vertical y, a menudo, se usa para iniciar la navegación a través de contenido.

Para crear un elemento de la interfaz de usuario que pueda reconocer este gesto, debes crear una instancia de **SwipeGestureRecognizer**, inicializar la propiedad **Direction** a uno de los posibles valores de la enumeración **SwipeDirection** (Left, Right, Up o Down), opcionalmente inicializar la propiedad **Threshold**, gestionar el evento **Swiped** y añadir el nuevo gesto reconocido a la colección **GestureRecognizers** del elemento de la interfaz de usuario.

```
<BoxView Color="Teal" ...>
    <BoxView.GestureRecognizers>
        <SwipeGestureRecognizer Direction="Left" Swiped="OnSwiped"/>
    </BoxView.GestureRecognizers>
</BoxView>
```

```
var boxView = new BoxView { Color = Color.Teal, ... };
var leftSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Left };
leftSwipeGesture.Swiped += OnSwiped;

boxView.GestureRecognizers.Add(leftSwipeGesture);
```

4. Comportamientos

Los **comportamientos** o **behaviors** permiten agregar funcionalidades a los controles de la interfaz de usuario sin necesidad de agregarles subclases. Estos comportamientos se escriben en código y se agregan a los controles en XAML o en código subyacente.

Los comportamientos permiten implementar el código que normalmente se tendría que escribir como código subyacente, ya que interactúa directamente con la API del control de la interfaz de usuario, de tal manera que puede ser adjuntado al control y empaquetado para su reutilización en más de una aplicación. Pueden ser utilizados para proporcionar un amplio abanico de funcionalidades a los controles, tales como:

- Agregar un validador de correo electrónico a un Entry.

- Creación de un control de clasificación mediante un reconocedor de movimiento de tap.
- Controlar una animación.
- Agregar un efecto a un control.

Los comportamientos también permiten escenarios más avanzados. En el contexto de comandos, los comportamientos son un enfoque útil para conectar un control a un comando. Además, pueden usarse para asociar los comandos a los controles que no se han sido diseñados para interactuar con los comandos. Por ejemplo, pueden usarse para invocar un comando como respuesta a una activación del evento.

Xamarin.Forms es compatible con dos estilos diferentes de comportamientos: comportamientos de Xamarin.Forms y comportamientos asociados.

4.1. Comportamientos de Xamarin.Forms

El proceso de creación de un comportamiento de Xamarin.Forms es como sigue:

1. Crear una clase que hereda de la clase Behavior o Behavior<T>, donde T es el tipo del control al que debe aplicarse el comportamiento.
2. Sobreescribir el método OnAttachedTo para realizar cualquier operación.
3. Sobreescribir el método OnDetachingFrom para realizar cualquier limpieza.
4. Implementar la funcionalidad básica del comportamiento.

```
public class CustomBehavior : Behavior<View>
{
    protected override void OnAttachedTo (View bindable)
    {
        base.OnAttachedTo (bindable);
        // Perform setup
    }

    protected override void OnDetachingFrom (View bindable)
    {
        base.OnDetachingFrom (bindable);
        // Perform clean up
    }

    // Behavior implementation
}
```

4.2. Comportamientos asociados

Los comportamientos asociados o vinculados son clases estáticas con una o más propiedades vinculadas. Una propiedad vinculada (bindable property) es un tipo especial de propiedad. Este tipo de propiedades se definen en una clase, pero se asocian a otros objetos y se definen en XAML como atributos que contienen el nombre de una clase separado por un punto del nombre de una de sus propiedades.

Una propiedad vinculada se puede definir mediante el delegado llamado propertyChanged que puede ser ejecutado cuando el valor de la propiedad cambia, entonces es cuando la propiedad de un control o vista se inicializa. Cuando el delegado propertyChanged se ejecuta, éste pasa una referencia al control sobre el cual está siendo vinculado y a los parámetros que contienen el viejo y el nuevo valor de la propiedad. Este delegado se puede utilizar para añadir nuevas funcionalidades al control que se vincula a la propiedad para manipular la referencia que se le ha pasado del siguiente modo:

1. El delegado propertyChanged convierte la referencia del control, que se recibe como un BindableObject, al tipo de control que el comportamiento está diseñado a mejorar.
2. El delegado propertyChanged modifica las propiedades del control, llama a métodos del control o registra manejadores de eventos expuestos por el control, para implementar la funcionalidad del comportamiento básico.

Un problema con los comportamientos asociados es que están definidos en una clase estática, con propiedades y métodos estáticos. Esto dificulta la creación de comportamientos asociados que tienen estado. Por ello, Xamarin.Forms ha sustituido los comportamientos asociados como la aproximación preferida para la construcción de comportamientos.

5. Disparadores

Los **disparadores** o **triggers** permiten expresar acciones mediante declaraciones en XAML que cambian la apariencia de controles basados en eventos o en cambios de propiedades.

Se puede asignar un disparador directamente a un control o agregarlo al diccionario de recursos de nivel de aplicación o de nivel de página para ser aplicado a varios controles. Existen cuatro tipos de disparadores:

- Disparadores de propiedades (Property Trigger). Ocurren cuando una propiedad en un control se inicializa a un valor concreto.
- Disparadores de datos (Data Trigger). Utilizan la vinculación de datos para el disparo a partir de las propiedades de otro control.
- Disparadores de eventos (Event Trigger). Ocurren cuando sucede un evento en un control.
- Multidisparadores (Multi Trigger). Permiten múltiples condiciones de disparo para ser inicializadas antes de que suceda una acción.

Un disparador simple se puede expresar puramente en XAML, agregando un elemento disparador a la colección de disparadores de un control. Este ejemplo muestra un disparador que cambia el color de fondo de una entrada cuando recibe el foco:

```
<Entry Placeholder="enter name">
    <Entry.Triggers>
        <Trigger TargetType="Entry"
             Property="IsFocused" Value="True">
            <Setter Property="BackgroundColor" Value="Yellow" />
        </Trigger>
    </Entry.Triggers>
</Entry>
```

Las partes más importantes en la declaración de los disparadores son las siguientes:

- TargetType: el tipo de control al que se aplica el disparador.
- Property: la propiedad en el control que se supervisa.
- Value: el valor, cuando se produce para la propiedad supervisada, que hace que se active el disparador.
- Setter: una colección de elementos Setter, cuándo se cumple la condición de activación, se debe especificar la Property y el Value.
- EnterActions y ExitActions: están escritas en código y se pueden usar además de o en lugar de elementos Setter.