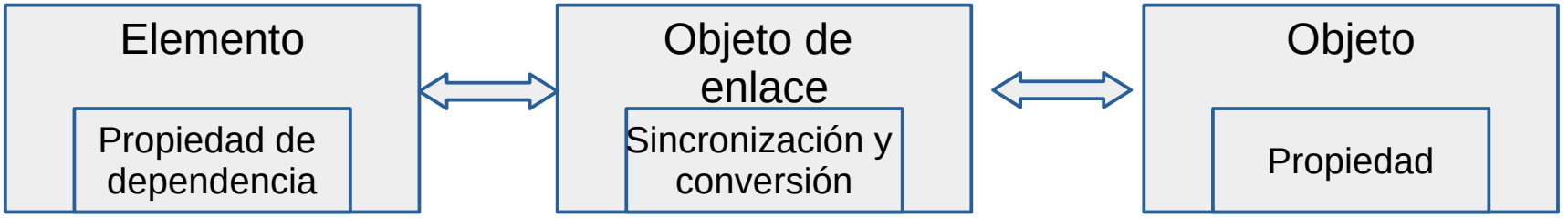
UD4 Enlace a datos WPF

Ciclo: Desarrollo de Aplicaciones Multiplataforma
Módulo: Desarrollo de interfaces.

Introducción

- WPF facilita el diseño de interfaces de usuario sólidas y que también proporciona unas capacidades muy eficaces de enlace de datos.
- Con WPF, podemos realizar la manipulación de datos mediante código de un lenguaje .Net, por ejemplo C#, XAML o una combinación de ambos, y podemos también realizar el enlace de datos a controles, a propiedades públicas, a XML y a objetos, convirtiendo las operaciones de enlace de datos en tareas rápidas, flexibles y realmente fáciles.

- El enlace de datos es un proceso que establece una conexión entre la interfaz gráfica del usuario (IGU) de la aplicación y la lógica de negocio, para que cuando los datos cambien su valor, los elementos de la IGU que estén enlazados a ellos reflejen los cambios automáticamente.
- Para utilizar el enlace de datos de WPF, debe disponer siempre de un destino y un origen.
- El objeto enlace de datos es, en esencia, un puente entre el destino del enlace (elemento) y el origen del mismo (objeto).



- El destino del enlace puede ser cualquier elemento o propiedad accesible que se derive de DependencyProperty, un ejemplo es la propiedad Text del control TextBox.
- El origen del enlace puede ser cualquier propiedad public, incluidas propiedades de otros controles, objetos del CLR, elementos XAML, DataSets de ADO.NET, fragmentos XML, etc.
- Para facilitar la conexión con estas fuentes de datos, WPF proporciona dos proveedores especiales: **XmlDataProvider**, y **ObjectDataProvider**.
- El flujo de datos de un enlace puede ir desde el destino al origen (por ejemplo, el valor del origen cambia cuando un usuario modifica el valor del TextBox enlazado), desde el origen al destino si el origen del enlace proporciona las notificaciones correspondientes (por ejemplo, el contenido del TextBox se actualiza con los cambios en el origen del enlace), o en ambas direcciones.

- Un enlace de datos en WPF viene definido por un objeto de la clase Binding.
- La ruta de acceso a los datos se especifica por medio de su propiedad Path (o XPath si los datos están en XML), la dirección del flujo de datos queda especificada por su propiedad Mode, UpdateSourceTrigger especifica cuándo ocurre la actualización del origen (por ejemplo, un valor PropertyChanged para esta pro piedad indica que el origen del enlace se actualizará en cuanto cambie la propie dad de destino),
- ElementName indica el nombre del origen de datos cuando éste es otro elemento de la interfaz y Source especifica, de forma explícita, el objeto que se va a utilizar como origen de los datos, invalidando el contexto de datos predeterminado, que es definido implícitamente por la propiedad DataContext del elemento seleccionado de la IGU.
-

- El aspecto más importante de un contexto de datos es hacer referencia al origen de datos que se utiliza para el enlace y permitir a los elementos secundarios del árbol de elementos que define la interfaz gráfica heredar esa información cuando haya sido definida en un elemento primario.
- Por ejemplo, en el código siguiente, el contexto de datos del Grid hace referencia al objeto origen de datos, objGrados, que podrá ser utilizado por todos sus elementos secundarios.

```
<Window ...
            xmlns:local="clr-namespace:ConverTemps.Clases">
      <Window. Resources>
          <local:CGrados x: Key="objGrados" GradosC="0.00" GradosF="32.00" />
   </Window. Resources>
   <Grid Name="gridVentPpal".
            DataContext="{StaticResource objGrados}">
       ...
   </Grid>
</Window>
```

- Y este otro código enlaza la propiedad Text de una caja de texto con la propiedad GradosF del objeto origen de datos objGrados, y la propiedad Foreground de esa caja de texto con la propiedad Foreground de otro elemento denominado ctGradosC de la interfaz gráfica de usuario.

```
<TextBox Name="ctGradosF"..
   Text="{Binding Path=GradosF,...}"
    Foreground="{Binding Path=Foreground, ElementName=ctGradosC}"
…../>
```

- Para que el elemento de la IGU vinculado con el objeto origen de datos pueda actualizarse automáticamente cuando éste cambie, dicho objeto origen de datos debe implementar la interfaz INotifyPropertyChanged, para notificar los cambios cuando éstos sucedan.
- Precisamente, el objeto de enlace (Binding) está siempre a la escucha de estas notificaciones, para comunicar al destino cuándo debe actualizarse.

ENLACE A COLECCIONES DE OBJETOS

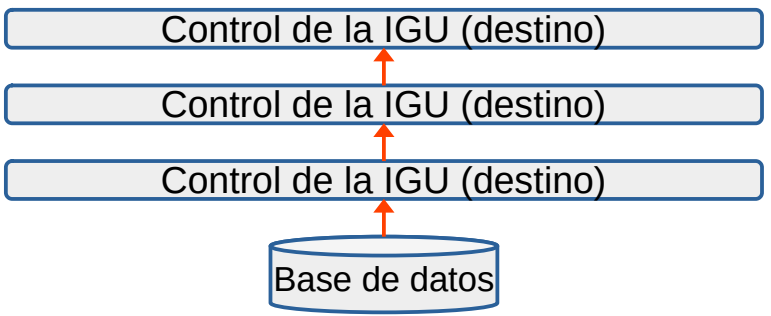
- El origen de un enlace puede ser un objeto único, cuyas propiedades contienen los datos, o una colección de objetos resultado, por ejemplo, de una consulta a una base de datos.
- Es bastante habitual utilizar un ItemsControl, como por ejemplo ListBox, ListView o TreeView, para mostrar una colección de objetos.
- Para enlazar un ItemsControl a una colección de objetos, la propiedad que se utiliza es ItemsSource. Esto es, se puede considerar la propiedad ItemsSource como el contenido del ItemsControl (esta propiedad admite el enlace OneWay, desde el origen al destino, de forma prede terminada).

Cómo implementar colecciones

- Para configurar enlaces dinámicos que actualicen la interfaz gráfica del usuario de forma automática cuando en la colección origen del enlace se realicen cambios, dicha colección debe implementar la interfaz INotifyCollectionChanged.
- La clase ObservableCollection<T> de WPF proporciona una colección que implementa esta interfaz. Cada objeto de la colección que publica propiedades enlazables debe implementar también esa interfaz. Otras clases de colección existentes, que también podemos utilizar, son List<T>, Collection<T> y BindingList<T>, entre otras.

Vistas de colección

- Una vez que un ItemsControl está enlazado a una colección de objetos, podemos navegar, filtrar, ordenar o agrupar los datos. Para ello, se utilizan vistas de colección, que son clases que implementan la interfaz ICollectionView.
- Una vista de colección es un objeto situado un nivel por encima de la colección proporcionada por el origen del enlace. Dicha vista permite navegar y mostrar la colección en función de las consultas de ordenación, filtrado y agrupación, sin tener que cambiar la propia colección subyacente en el origen.



- Los cambios en la colección se propagarán a las vistas si la colección implementa la interfaz INotifyCollectionChanged, pero las vistas no cambian la colección subyacente; por eso, cada colección proporcionada por el origen puede tener varias vistas asociadas.
- Esto quiere decir que el uso de vistas permitirá mostrar los mismos datos de formas diferentes.
- Cada colección proporcionada por el origen del enlace tiene una vista predeterminada. ¿Cómo se obtiene dicha vista? Para crear la vista, se necesita una referencia a la colección subyacente en el origen, que se puede obtener directamente, a través del contexto de datos, de una propiedad del origen u obteniendo una pro piedad del enlace.
- Por ejemplo, supongamos que hemos establecido en el contexto de datos de un StackPanel una referencia a un proveedor de datos, *datos Alumnos*, que es una colección ObservableCollection<T> de objetos alumno:

```
<StackPanel.DataContext Name="SpAlumnos" )
    <Binding Source="{StaticResource datosAlumnos}"/>
</StackPanel.DataContext>
```

- En este caso, para obtener la vista de la colección subyacente en el origen utilizaremos la propiedad DataContext del control StackPanel así:

```
ICollectionView vista =
    CollectionViewSource.GetDefaultView(spAlumnos.DataContext);
```

- La clase CollectionView representa una vista para navegar, filtrar, ordenar y agrupar datos en una colección y CollectionViewSource es un proxy (interfaz de comunicación) para una clase que implemente ICollectionView, como ocurre con CollectionView, que permite al código comunicarse con la vista.
- CollectionViewSource tiene una propiedad View que hace referencia a la vista real y una propiedad Source que hace referencia a la colección proporcionada por el origen.
- Desde el punto de vista de código XAML, la clase CollectionViewSource es la representación en XAML de la clase CollectionView. Las operaciones que realicemos sobre la vista se reflejarán automáticamente en la ventana.

- Por ejemplo, el código siguiente obtiene la vista de una colección. Si suponemos que esta colección está enlazada a un ListBox, el siguiente código, actuando sobre la vista, desplazaría el cursor al siguiente elemento cada vez que se ejecute:

```
IcollectionView vista = ObtenerVista();
vista.MoveCurrentToNext();
```

- Este desplazamiento se reflejaría automáticamente sobre el ListBox.

- Si además deseamos que la lista se desplace para mostrar el elemento ac tualmente seleccionado, tendríamos que añadir una línea como la siguiente:

```
lista.ScrollIntoView(vista.CurrentItem)
```

- Este otro ejemplo es un método que obtiene la vista de la colección de objetos proporcionada por el proveedor de datos XML xmlAlumnos:

```
private ICollectionView ObtenerVista ()
{
    DataSourceProvider proveedor = (DataSourceProvider)
        this.FindResource("xml Alumnos");
   return CollectionViewSource.GetDefaultView(proveedor.Data);
}
```

- El tipo de vista utilizada dependerá del tipo del objeto origen de los datos. Todas las vistas derivan de Collection View y concretamente, estas dos implementaciones especializadas: ListCollectionView y BindingListCollectionView. ¿Cuándo se crea una u otra?
  - Si el origen de datos implementa la interfaz IBindingList, se crea una colec ción BindingListCollectionView; esto sucede cuando el enlace es con un DataTable de ADO.NET.
  - Si el origen de datos no implementa IBindingList pero implementa la interfaz IList, entonces se crea una colección ListCollectionView. Esto sucede cuando el enlace es con un ObservableCollection. En los dos ejemplos anteriores se devuelve una colección ListCollectionView.
  - Y si el origen de datos no implementa ni IBindingList ni IList pero imple menta la interfaz IEnumerable, entonces se crea una colección CollectionView.
  -

PLANTILLAS DE DATOS

- El modelo de plantillas de datos de WPF proporciona gran flexibilidad para definir la presentación de los datos.
- Un objeto DataTemplate permite especificar cómo se presentarán los objetos de datos, proceso que resulta particularmente útil al enlazar un control ItemsControl, por ejemplo un ListBox, a una colección completa de objetos.

- Como ejemplo, vamos a diseñar una aplicación que muestre una ventana (un diálogo) que presente un objeto ListBox enlazado a una lista de objetos alumno almacenados en un fichero XML.
- El fichero alumnos.xml podría ser el siguiente:

```
<?xml version="1.0" encoding="utf-8" ?>
<alumnos>
   <alumno>
      <foto>Imagenes/foto.jpg</foto>
      <nombre>Alumno 01</nombre>
     <direccion>Dirección 01</direccion>
     <estudios>Estudios 01</estudios>
     <beca>True</beca>
     ….
  </alumno>
</alumnos>
```

- Suponiendo que ya ha creado una nueva aplicación, por ejemplo EnlaceDeDatosXML, con una barra de menús con las opciones *Archivo, Diálogos y Ayuda*, queremos añadir a la sección de recursos del objeto Application un proveedor XML que haga referencia al fichero anterior:

```
<Application. Resources>
  <XmlDataProvider x:Key="xmlAlumnos"
                   Source="datos/alumnos.xml" XPath="alumnos/alumno"/>
</Application.Resources>
```

El valor alumnos/alumno del atributo XPath indica que se seleccionarán to dos los elementos alumno que son hijos de alumnos.

- A continuación añadimos una nueva ventana DlgDataTemplate que muestra un ListBox, según el diseño especificado a continuación, y un elemento al menú Diálogos de la ventana principal que permite visualizar esta ventana

```
<Window x:Class="EnlaceDeDatosXML.DigDataTemplate" ...
        Title="D1gDataTemplate" height="300" width="400">
        <Grid>
                <StackPanel)
                        <TextBlock Name="tbTitulo" FontSize="20"
                                Text="Lista de alumnos:"/>
                        <ListBox Height="260" width="400" Margin="10"
                                Items Source="{Binding Source={StaticResource xmlAlumnos }}">
                        </ListBox>
                </StackPanel>
        </Grid>
</Window>
```

- La propiedad ItemsSource del ListBox está enlazada con la fuente de datos proporcionada por el proveedor XML. Esta fuente de datos se corresponde con todos los objetos alumno de alumnos, según especifica el atributo XPath del proveedor de datos

- En este caso, si lo ejecutamos, la lista no muestra el resultado esperado:
- Un control ListBox cuyos elementos sean objetos de una colección, sin instrucciones concretas acerca de cómo se deben presentar esos elementos, muestra la representación de cadena de los objetos (como estamos trabajando con elementos XML, la cadena se corresponde con toda la información que incluye un alumno; con clases de objetos CLR la información sería la dada por el método ToString de la clase de esos objetos).
- ¿Cuál es la solución? Pues utilizar un objeto DataTemplate para definir cómo deben mostrarse los objetos de datos; esto es, el contenido de DataTemplate se convierte en la estructura visual de los objetos de datos.

Definir una plantilla de datos

- Una forma de definir una plantilla para un control lista es definir un objeto DataTemplate y asignarlo a la propiedad Item Template del objeto ListBox.
- El objeto DataTemplate definirá los controles de datos adecuados al tipo de los datos a mostrar; según esto, para mostrar los datos nombre, dirección y estudios podemos utilizar elementos TextBlock y para el dato beca, un elemento CheckBox, que podemos colocar dentro de un objeto StackPanel.
- Después, enlazamos cada uno de ellos con el elemento correspondiente del origen de datos…….(transparencia siguiente...)

```
<ListBox Height="210" width="350" Margin="10"
        Items Source="{Binding Source={StaticResource xml Alumnos}}">
    <ListBox.ItemTemplate>
        <DataTemplate>
        <StackPanel>
            <TextBlock Text="{Binding XPath=nombre}" />
            <TextBlock Text="{Binding XPath=direccion}"/>
            <TextBlock Text="{Binding XPath=estudios}"/>
            <CheckBox IsChecked="{Binding Xpath=beca)"/>
        </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

- De esta forma la lista tendrá el aspecto esperado con alumno, dirección, estudios y un check.En general, el objeto DataTemplate se aplica a cada uno de los elementos ListBoxItem generados en la lista.

- La plantilla puede definirse también como un recurso de la ventana y asignar después ese recurso a la propiedad ItemTemplate del ListBox. La ventaja de esto es que podríamos utilizar este recurso en otros controles:

```
<Window x:Class="EnlaceDeDatos XML.DigDataTemplate" ...
            Title="DlgDataTemplate" height="300" width="400">
      <Window. Resources>
            <DataTemplate x:Key="plantillaAlumno">
              <StackPanel>
                  <TextBlock Text="{Binding XPath=nombre}" />
                  <TextBlock Text="{Binding XPath=direccion}"/>
                  <TextBlock Text="{Binding XPath=estudios}"/>
                  <CheckBox IsChecked="{Binding Xpath=beca)"/>
            </StackPanel>
          </DataTemplate>
      </Window. Resources>
      <Grid>
      <StackPanel>
            <TextBlock Name="tbTitulo" FontSize="20"
                        Text="Lista de alumnos:"/>
                  <ListBox Height="190" width="350" Margin="10"
                        Items Source="{Binding Source={StaticResource xml Alumnos }}”
                        ItemTemplate=" (StaticResource plantillaAlumno)">
                  </ListBox>
      </StackPanel>
      </Grid>
</Window>
```

Mejorar la presentación

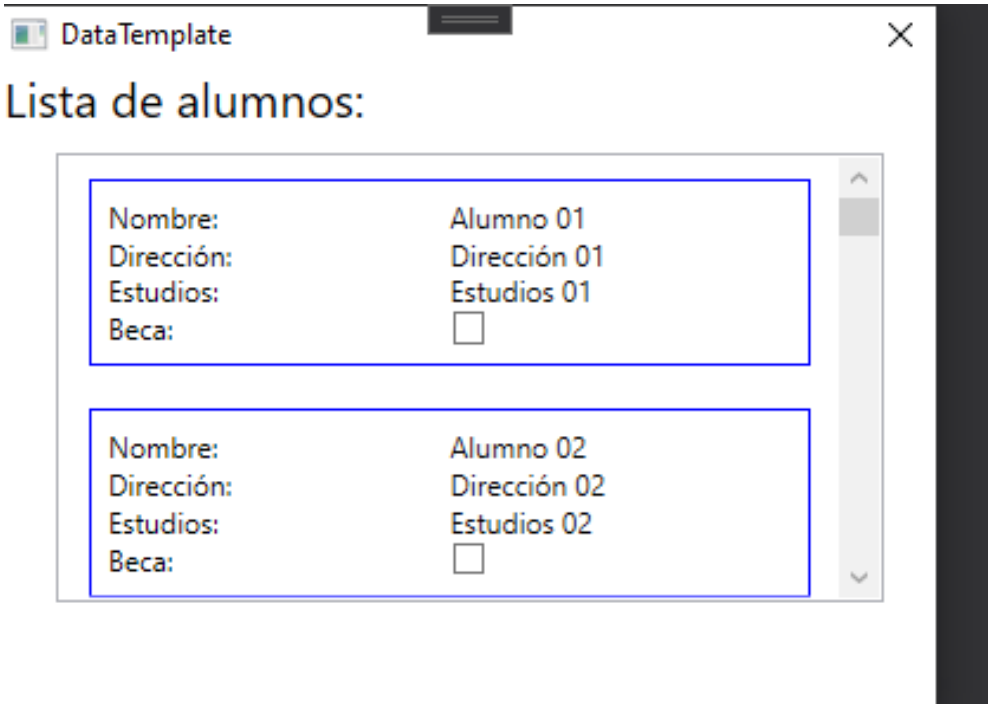
- Podemos mejorar la presentación agregando un Border, elementos TextBlock que describan los datos que se muestran y, en lugar del elemento StackPanel, un Grid con cuatro filas (tantas como datos mostrados) y dos columnas: la primera para las descripciones y la segunda para los datos…...

```
<DataTemplate x:Key="plantillaAlumno">
  <Border Name="bordeAlumno" Border Brush="Blue" BorderThickness="1"
                    Padding="7" Margin="7">
      <Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions)
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <TextBlock Grid.Row="0" Grid.Column="0" Text="Nombre: "/>
        <TextBlock Grid.Row="0" Grid.Column="1"
                    Text="{Binding XPath=nombre}" />
        <TextBlock Grid.Row="1" Grid.Column="0" Text="Dirección: " />
        <TextBlock Grid.Row="1" Grid.Column="1"
                    Text="{Binding XPath=direccion}"/>
        <TextBlock Grid.Row="2" Grid.Column="0" Text="Estudios: " />
        <TextBlock Grid.Row="2" Grid.Column="1"
                    Text="{Binding XPath=estudios}"/>
        <TextBlock Grid.Row="3" Grid.Column="0" Text="Beca: " />
        <CheckBox Grid.Row="3" Grid.Column="1"
                    IsChecked="{Binding XPath=beca}" />
      </Grid>
  </Border>
</DataTemplate>
```

- Para que los elementos de la lista ocupen todo el ancho del ListBox, se puede asignar a la propiedad HorizontalContentAlignment del ListBox el valor Stretch:

```
<ListBox Height="190" width="350" Margin="10"
    Items Source="{Binding Source={StaticResource xmlAlumnos}}"
    ItemTemplate="{StaticResource plantillaAlumno}".
    HorizontalContentAlignment="Stretch">
</ListBox>
```

- Ahora, el control ListBox mostraría los datos así:



- Y podríamos pintar el borde de los alumnos con beca en otro color utilizando DataTrigger.

Utilizar desencadenadores para aplicar valores de propiedad

- En el código siguiente, el objeto DataTrigger de la colección Triggers del DataTemplate establece el valor de la propiedad BorderBrush del elemento deno minado borde Alumno en LightGreen si el atributo beca del elemento alumno del origen de datos mostrado vale True.

```
<DataTemplate x:Key="plantilla Alumno">
    <Border Name="bordeAlumno" BorderBrush="Blue" Border Thickness="1"
    ...
    </Border>
    <DataTemplate. Triggers)
        <DataTrigger Binding="{Binding XPath=beca}" Value="True">
            <Setter TargetName="bordeAlumno"
Property="BorderBrush" Value="LightGreen"/> </DataTrigger) </DataTemplate.Triggers)
</DataTemplate)
```

- Otra forma de hacer esto mismo es utilizando un conversor. En el ejemplo anterior, el objeto Setter del desencadenador establece el valor de una propiedad de un elemento (Border en nuestro caso) que se encuentra dentro del DataTemplate.
- Sin embargo, si las propiedades afectadas por Setter no corresponden a elementos que estén dentro del DataTemplate actual, puede que sea más conveniente establecer las propiedades mediante un objeto Style para la clase del elemento al que se desea aplicar el estilo (ListBoxItem en nuestro caso). Por ejemplo, si deseamos animar el valor Opacity del elemento ListBoxItem cuando el ratón señale a un elemento, puede definir en los recursos de la ventana el siguiente estilo:

```
<Style Target Type="ListBoxItem">
   <Style. Triggers>
     <Event Trigger RoutedEvent="ListBoxItem.MouseEnter">
          <BeginStoryboard>
               <Storyboard>
                    <DoubleAnimation
                       Storyboard.Target Property="(ListBoxItem.Opacity)"
                       From="1" To="0.1" Duration="0:0:3" Auto Reverse="True" />
               </Storyboard>
          </BeginStoryboard>
     </EventTrigger>
   </Style.Triggers>
</Style>
```

- El objeto Storyboard define una escala de tiempo para realizar una animación y DoubleAnimation anima el valor de una propiedad Double entre dos valores usando la interpolación lineal durante 3 segundos, propiedad Duration especificada; AutoReverse indica si la escala de tiempo retrocede después de completar una iteración de avance.

XML COMO FUENTE DE DATOS

- El proveedor XmlDataProvider proporciona una forma rápida y sencilla de ex traer datos XML de un fichero, de una dirección web o de los recursos de la aplicación y ponerlo a disposición de los elementos de una aplicación. Este proveedor ha sido diseñado para ser de sólo lectura y la forma de utilizarlo es muy sencilla:

```
<XmlDataProvider x:Key="xml Alumnos"
        Source="datos/alumnos.xml" XPath="/alumnos/alumno"/>
```

- Source es una propiedad de tipo Uri (identificador de recursos uniforme); esta propiedad permite obtener o establecer el objeto Uri correspondiente al nombre del fichero de datos XML que se va a usar como origen del enlace, y XPath es una propiedad de tipo string que hace referencia a la ruta de acceso a los datos sobre la que se realizará la consulta para obtener el conjunto de datos especificado. Esta propiedad Source también se puede establecer desde el código subyacente, por ejemplo, si aún no conocemos el fichero XML que necesitamos usar.

- El proveedor XmlDataProvider carga el contenido XML asíncronamente, excepto si asignamos a su propiedad IsAsynchronous el valor false.
- Anteriormente, tomando como fuente de datos un fichero XML, hemos podido comprobar que .Net proporciona una amplia biblioteca para trabajar con este tipo de información. Así que cuando necesitemos la capacidad de modificar XML o convertir los datos XML en objetos para realizar una programación orientada a objetos, es mejor olvidarnos del proveedor de datos y utilizar el soporte que proporciona .Net.
- Como ejemplo, volviendo a la aplicación anterior, en la ventana DigDataTemplate, podemos añadir a los elementos mostrados por la lista, la información restante, es decir, las asignaturas de las que está matriculado cada alumno.

```
<?xml version="1.0" encoding="utf-8" ?>
<alumnos>
	<alumno>
		<foto> Imagenes/foto.jpg</foto>
			<nombre>Alumno 01</nombre>
			<direccion>Dirección 01</direccion>
			<estudios >Estudios 01</estudios>
			<beca>True</beca>
			<asignaturas tipo="Obligatorias">
				<asignatura>
					<nombre>Asignatura ob01</nombre>
					<nota>8,5</nota>
				</asignatura>
				<asignatura>
					<nombre>Asignatura ob02</nombre>
					<nota>5,5</nota>
				</asignatura>
				<asignatura>
					<nombre>Asignatura ob03</nombre>
					<nota>7,0</nota>
				</asignatura>
			</asignaturas>
			<asignaturas tipo="Optativas">
				<asignatura>
					<nombre>Asignatura op01</nombre>
					<nota>6,0</nota>
				</asignatura>
				<asignatura>
					<nombre>Asignatura op02</nombre>
					<nota>4,5</nota>
				</asignatura>
			</asignaturas>
	</alumno>
...
</alumnos
```

La colección de datos presentada está organizada jerárquicamente, por lo tanto, para mostrarla lo más fácil será utilizar controles que admitan este tipo de organización……..

Datos jerárquicos

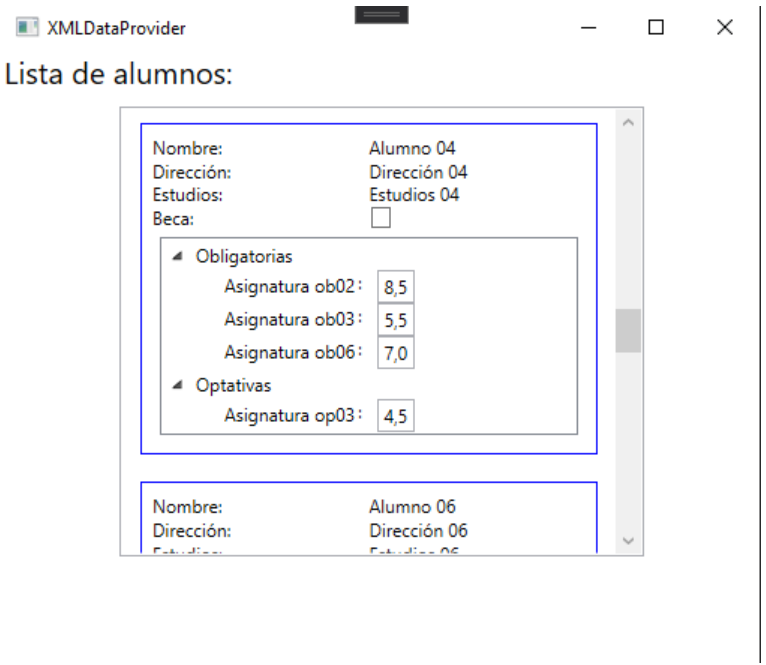
- En ocasiones tendremos colecciones que contengan otras colecciones. Por ejemplo, en la fuente de datos que acabamos de ver en la transparencia anterior, un alumno tiene una colección de asignaturas.
- El tratamiento de datos jerárquicos puede resultar sencillo si se utiliza la plantilla HierarchicalDataTemplate, que representa un DataTemplate que admite controles HeaderedItemsControl, como TreeViewItem o MenuItem.

- Para añadir a cada elemento de la lista las asignaturas de las que está matriculado cada alumno, utilizariamos un TreeView.
- Para esto modificamos la plantilla de la lista, plantillaAlumno, para añadir este nuevo elemento a continuación de los existentes….

```
<DataTemplate x:Key="plantillaAlumno">
    <Border Name="borde Alumno" ...)
        <Grid>
        ….
            <TreeView Grid.Row="4" Grid.ColumnSpan="2"
                Name="trvAlumnos" Margin="5"
            Items Source="{Binding Path=asignaturas}"
            ItemTemplate="{StaticResource kAsignaturas}">
            </TreeView)
        </Grid>
    </Border>
    ...
</DataTemplate>
```

- El origen de datos para TreeView, propiedad ItemsSource, es proporcionado por un enlace que indica por medio de su atributo XPath dónde se encuentra la colección que representa el nivel siguiente en la jerarquía de datos (en el ejemplo, se encuentra en asignaturas), y la plantilla que indica cómo se va a mostrar cada elemento de esa colección queda especificada por la propiedad Item Template (en el ejemplo, se mostrarán según indique el recurso kAsignaturas).

- El resultado que obtendríamos, después de implementar el recurso kAsignaturas sería parecido al siguiente…..



- El recurso kAsignaturas también es de tipo HierarchicalDataTemplate, ya que un elemento asignaturas contiene elementos asignatura. Cada uno de estos elementos irá encabezado por el TextBlock “Obligatorias” u “Optativas”. Este va lor lo proporciona el atributo tipo del elemento asignaturas.

```
<!-- PLANTILLA ASIGNATURAS -->
<HierarchicalDataTemplate x:Key="kAsignaturas"
                    Items Source="{Binding XPath=asignatura}"
                    Item Template="{StaticResource kAsignatura}">
      <TextBlock Text="{Binding XPath=@tipo}" padding="2" />
</HierarchicalDataTemplate>
```

- Respecto a la sintaxis de Xpath, cuando el identificador del dato asociado es un atributo de un elemento XML, por ejemplo XPath=@tipo va precedido por @, no sucediendo esto cuando el identificador del dato se corresponde con un elemento XML, por ejemplo XPath=asignatura.

- Ahora, el origen de datos para esta plantilla es *asignatura*, elemento donde se encuentran los datos para el siguiente nivel, y la plantilla que indica cómo se van mostrar estará definida por el recurso *kAsignatura*.
- El recurso *kAsignatura* será de tipo DataTemplate (no jerárquico) por tratarse de nodos terminales del árbol, es decir, un elemento asignatura contiene elementos *nombre y nota*, que son datos, no colecciones de datos.

```
<!-- PLANTILLA ASIGNATURA -->
<DataTemplate x:Key="kAsignatura">
    <Wrap Panel>
        <TextBlock Text="{Binding XPath=nombre}" Padding="2" />
        <TextBlock Text=":      " />
        <TextBox Text="{Binding XPath=nota}" Padding="2" />
    </Wrap Panel>
</DataTemplate>
```

- En este caso, la plantilla define los controles utilizados para mostrar los datos. La propiedad que se utilice de estos controles para mostrar el dato irá enlazada al elemento XML correspondiente.

- Otras expresiones XPath que son muy comunes:
  - /alumno. Selecciona el elemento raíz alumno.
  - alumno. Selecciona todos los nodos hijo del elemento alumno.
  - . (punto). Selecciona el nodo actual.
  - .. (dos puntos). Selecciona el padre del nodo actual.
  - @. Selecciona atributos de un elemento.
  - asignaturas/asignatura. Selecciona todos los elementos asignatura que son hijos de asignaturas.
  - /asignaturas/*. Selecciona todos los nodos hijo del elemento asignaturas.
  - asignaturas[1]. Selecciona todos los nodos hijo del primer elemento asignaturas.
  - asignaturas[last()]. Selecciona todos los nodos hijo del último elemento asignaturas.

https://www.w3schools.com/xml/xpath_intro.asp

Islas de datos

- Una isla de datos XML es como un objeto secundario de proveedor de datos XML que permite insertar un documento XML directamente en el código XAML de la ventana.

```
<Window x:Class="EnlaceDeDatosXML.DlgDataTemplate"
        ...
            Title="DlgDataTemplate" height="370" width="400">
        <Window. Resources>
            <XmlDataProvider x:Key="xmlAlumnos" XPath="/alumnos/alumno")
                <x:XData)
                    <alumnos xmlns="">
                        <alumno>
                        ...
                        </alumno>
                        ….
                        </alumnos>
                    </x:Xdata>
            </XmlDataProvider>
            ...
        </Window.Resources>
        <Grid>
        ...
        </Grid>
</Window)
```

En este caso, x:XData se utiliza como un objeto secundario de XmlDataProvider.

Soporte .Net para trabajar con XML

- Además de la clase XmlDataProvider, la biblioteca .Net proporciona el espacio de nombres System.Xml que contiene clases diseñadas para trabajar con datos XML.