WPF DataBinding - Workshop

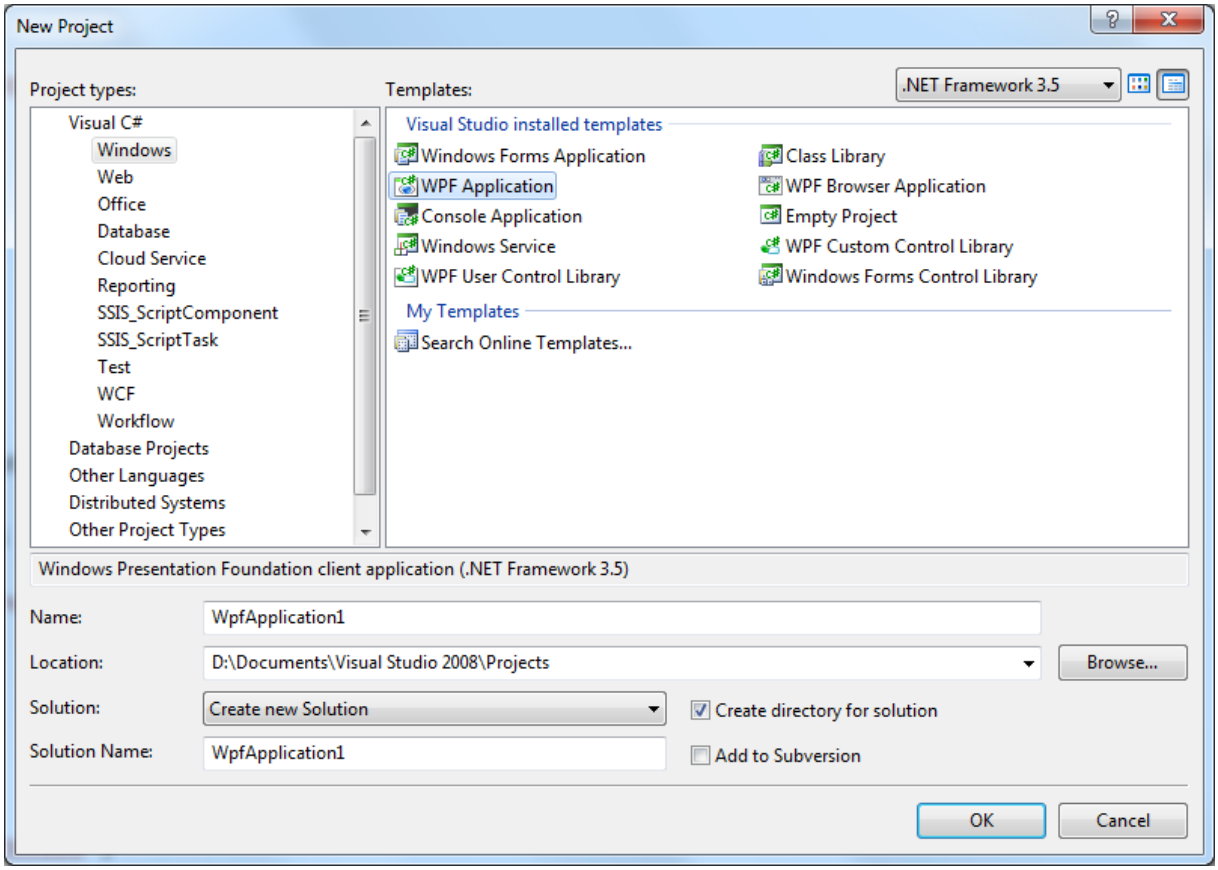
Aufgabe 1 – Grundlagen

Diese Aufgabe soll die Grundlagen von WPF DataBinding vermitteln. Sollten Sie schon etwas Erfahrung gemacht haben, trauen Sie sich ruhig an eine der Aufgaben weiter unten!

Die in dieser Aufgabe verwendeten Codesnippets sind nur beispielhafte Implementierungen. Ihrer Kreativität sind keine Grenzen gesetzt, nennen Sie die Objekte so wie Sie wollen!

Aufgabe 1.1

Erstellen Sie ein neues „WPF Projekt“ in Visual Studio.



Fügen Sie eine neue Klasse hinzu mit einem **string** und einem **int** Property.

```
public class Person
{
    public string Name { get; set; }

    public int WrittenLinesOfCode { get; set; }
}
```

Referenzieren Sie den Projektnamespace im XAML des Windows.

```
<Window ...
     xmlns:local="clr-namespace:Projektnamespace" ... />
```

Fügen Sie eine neue Instanz des Objekts den **Resources** des **Windows** hinzu.

```
<Window.Resources>
     <local:Person x:Key="myPerson"
                   Name="Jan Molnar"
                   WrittenLinesOfCode="128309" />
</Window.Resources>
```

Das Objekt kann nun als **Source** in DataBindings mittels

```
{StaticResource myPerson}
```

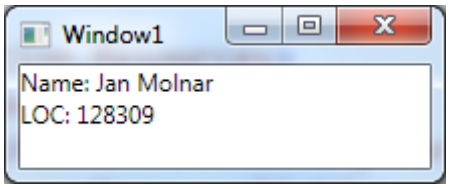
referenziert werden.

Aufgabe 1.2

Binden Sie einen **TextBlock** an das **string** Property und einen an das **int** Property.

```
<TextBlock Text="{Binding PropertyName, Source=...}" />
```

Führen Sie das Programm aus.



Aufgabe 1.3

Binden Sie eine **TextBox** mit einem TwoWay-Binding an das string Property.

Binden Sie einen **Slider** mit einem TwoWay-Binding an das int Property.

Binden Sie eine **TextBox** mit einem TwoWay-Binding an das **Value** Property des **Sliders**.

Aufgabe 1.4

Schreiben Sie eine ValidationRule für die TextBox, die nur gültige Werte in einem bestimmten Bereich zulässt.

```
public class MinMaxValidationRule : ValidationRule
{
     public double Minimum { get; set; }

     public double Maximum { get; set; }

     public override ValidationResult Validate(
          object value,
          System.Globalization.CultureInfo cultureInfo)
     {
          // Validation Logic
          return new ValidationResult(
          false, "Error Message");
     }
}
```

Um die Validation Rule verwenden zu können, muss im Xaml für das Binding die Property-Element-Syntax verwendet werden, d.h.

```
<TextBox>
     <TextBox.Text>
          <Binding ElementName="ELEMENT"
               Path="PATH"
               Mode="TwoWay"
               UpdateSourceTrigger="PropertyChanged">
               <Binding.ValidationRules>
                    <local:MinMaxValidationRule
                         Minimum="YourMin"
                         Maximum="YourMax"/>
               </Binding.ValidationRules>
          </Binding>
     </TextBox.Text>
</TextBox>
```

Aufgabe 1.5

Erstellen Sie eine neue Class, die ein Property vom Typ **List** hat.

```
public class HardCoders
{
     public HardCoders () {
          // Opfer initialisieren...
     studentPartners = new List<Person>();
     studentPartners.Add(new Person() {
          Name = "Tobias Grass",
          WrittenLinesOfCode = 21798 });
     studentPartners.Add(new Person() {
          Name = "Dennis Zielke",
          WrittenLinesOfCode = 9218312 });
     studentPartners.Add(new Person() {
          Name = "Alex Bierhaus",
          WrittenLinesOfCode = 81293 });
     studentPartners.Add(new Person() {
          Name = "Felix Leitner",
          WrittenLinesOfCode = 81293 });
     ...
     }

     private List<Person> coders;

     public List<Person> Coders {
             get { return coders; }
        }
}
```

Erstellen Sie im Projekt ein neues **Window** und setzten in der App.xaml das **StatupItem** auf dieses Window.

Fügen Sie dem **Window** die sobenen erstellete Klasse als **Resource** sowie eine **ListBox** hinzu und binden die **ItemsSource** Eigenschaft der ListBox an die Resource.

Definieren Sie ein **DataTemplate** für **Person** in den Resources des Windows.

```
<DataTemplate DataType="{x:Type local:Person}"> ...
```

Starten Sie die Anwendung.

Aufgabe 1.6

Nun soll das **DataTempalte** erweitert werden. Wählen Sie dazu ein Element im Template das durch einen Trigger verändert werden soll und geben Sie ihm einen Namen durch:

```
<ElementName x:Name="Name" ... />
```

Sie können nun einen **DataTrigger** definieren, der abhäng von den Daten eine Eigenschaft ändert.

```
<DataTemplate ...>
     <DataTemplate.Triggers>
          <DataTrigger
               Binding="{Binding PropertyName}"
               Value="WERT">
               <Setter  TargetName="Name"
                        Property="Foreground"
                        Value="Red" />
          </DataTrigger>
     </DataTemplate.Triggers>

</DataTemplate>
```

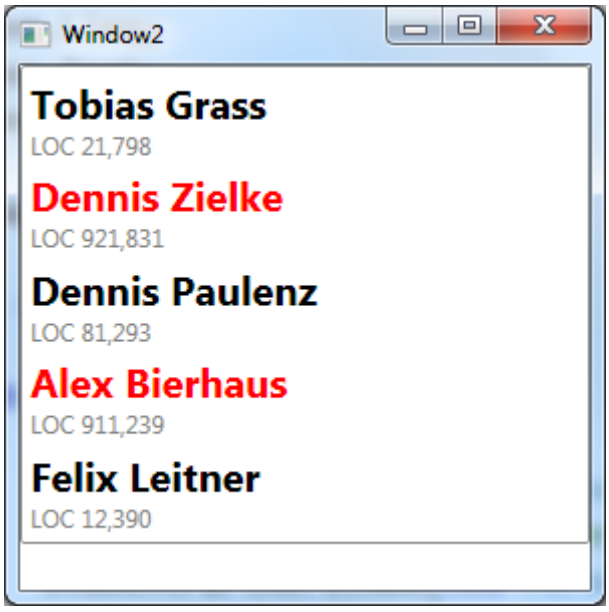


Abbildung 1 - Coder mit >100.000 LOC werden durch einen DataTrigger rot markiert

Aufgabe 1.7

Implementieren Sie einen Converter. Diesen Converter können Sie auch im DataBinding des Triggers verwenden (um z.b. True zu erzeugen, falls ein Wert eine Grenze überschreitet)

```
public class YourConvert : IValueConverter
{
     public object Convert(object value,
          Type targetType, object parameter,
          CultureInfo culture) {

          // Conversion
     }

     public object ConvertBack(object value,
          Type targetType, object parameter,
          CultureInfo culture)
     {
          // Backwards Conversion
     }
}
```

Der **Converter** muss anschließend den **Resources** des Windows hinzugefügt werden. Er kann dann in DataBindings als **StaticResource** verwendet werden.

```
{Binding PropertyName, Converter=...}
```

Aufgabe 1.8

Fügen Sie Ihrem Window ein oder meehrere Buttons hinzu. Fügen Sie im Click Handler der Collection neue Items hinzu bzw. entfernen Sie diese.

Zugriff auf Resourcen erhalten Sie im Code mittels

```
HardCoders coders = (HardCoders)this.Resources["coders"];
```

Warum passiert nichts?

Ersetzen Sie nun die **List** durch eine **ObservableCollecion** und versuchen es erneut.

Aufgabe 1.9

Fügen Sie Ihrem Projekt die SimpleStyles.xaml aus dem Resources Ordner hinzu. Editieren Sie nun die App.xaml wie folgt.

Fügen Sie einige WPF Elemente hinzu und experimentieren Sie mit dem Style. Sie können das Aussehen ändern, indem Sie die Definitionen in SimpleStyles.xaml ändern.

Aufgabe 2 – Wpf Produktgallerie

Öffnen Sie das Projekt WpfHtc als Ausgangspunkt.

Aufgabe 2.1

Die Daten für dieses Projekt liegen befinden sich **data/catalog.xml**. Machen Sie sich kurz mit der Struktur des Xml Dokuments vertraut.

Aufgabe 2.2

Fügen Sie einen **XmlDataProvider** hinzu, dessen Source auf das Xml Dokument zeigt.

```
<XmlDataProvider x:Key="catalog" Source="..."
                 XPath="..." />
```

Wir benötigen die <device /> Einträge, die XPath Query hiefür lautet **catalog/device**.

Die Daten des XmlDataProviders sollen nun in einer **ListBox** angezeigt werden. Verwenden Sie dazu DataBinding und setzten die **Source** des Bindings auf die statische Resource **catalog.**

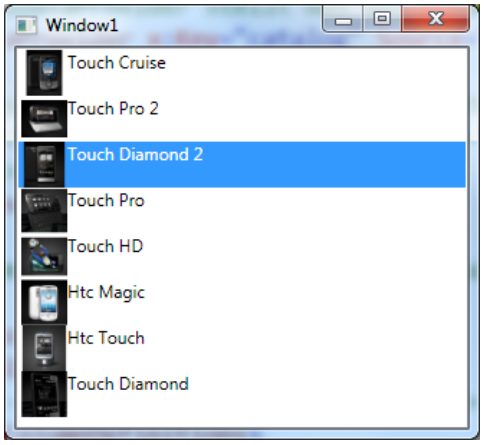
```
<ListBox ItemsSource="{Binding Source=...}" />
```

Um die Daten wie gewünscht anzuzeigen ist ein **DataTemplate** notwendig. Definieren Sie nun wie die Daten in der Liste angezeigt werden sollen.

```
<ListBox ...>
    <ListBox.ItemTemplate>
        <DataTemplate>
            ...
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Auf ein Attribut greifen Sie zu mit XPath=@Attributname. Auf ein Element mit XPath=Elementname.

```
<TextBlock Text="{Binding XPath=Diameter}" />
```



Aufgabe 2.3

Nun soll ein Master-Detail-Szenario implementiert werden. Dazu werden die gerade erstellte Liste und eine Detailansicht benötigt. Als Container für diese Aufgabe bietet sich das **Grid** an, für die Detailansicht ein **ContentPresenter**.

```
<Grid>
     <Grid.ColumnDefinitions>
          <ColumnDefinition  />
          <ColumnDefinition Width="400" />
     </Grid.ColumnDefinitions>

     <ListBox .../>

     <ContentPresenter Grid.Column="1" ... />
</Grid>
```

Der Contentpresenter wird wie die ListBox an den XmlDataProviderGebunden, jedoch mit Path=CurrentItem.

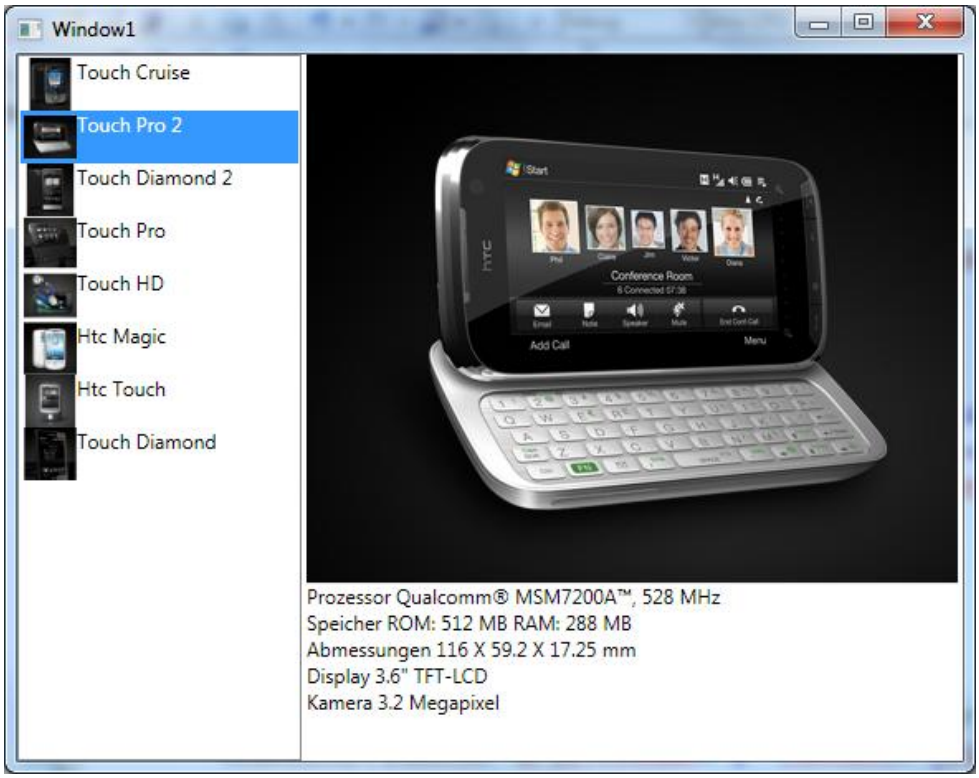
```
Content="{Binding CurrentItem, Source=...}"
```

In der ListBox muss die Eigenschaft **IsSynchronizedWithCurrentItem="True"** gesetzt werden. Warum?

Auch hier muss ein DataTemplate für den ContentPresenter definiert werden.

```
<ContentPresenter>
     <ContentPresenter.ContentTemplate>
          <DataTemplate>
               ...
          </DataTemplate>
     </ContentPresenter.ContentTemplate>
</ContentPresenter>
```

Um alle <features /> innerhalb des ContentPresenters aufzulisten kann ein ItemsControl verwendet werden.



Um eine Tabellenähnliche Auflistung zu erreichen kann im **ItemTempalte** des **ItemsControl** ein **Grid** verwendet werden mit **SharedSizeGroup** Property (http://msdn.microsoft.com/de-de/library/system.windows.controls.definitionbase.sharedsizegroup.aspx).

Aufgabe 2.3

Verwenden Sie Expression Blend, um das Look&Feel der Anwendung zu verbessern.

Aufgabe 2.4 (schwer, eher für zu Hause)

Schreibe ein Carousel-Control, um die Bilder in einer 3d-Ansicht darzustellen.

Aufgabe 3 - Linq und Wpf

Öffnen Sie das Projekt WpfLinq2Objects als Ausgangspunkt.

Die Klasse **NetworkDataProvider** stellt die Eigenschaft **Status** sowie die Methode **GetNetworkInterfaces**() zur Verüfung.

Status gibt den Text „Online“ bzw. „Offline“ zurück, je nachdem ob eine Internetverbindung vorhanden ist.

GetNetworkInterfaces() gibt eine Liste mit Objekten folgender Form zurück

Status (Netzwerkstatus)	**Values** (Geräte)
Online	• Netwerkgerät #1 • Netwerkgerät #3
Offline	• Netwerkgerät #2

Aufgabe 3.1

Fügen Sie den **NetworkDataProvider** als globale Resource dem Hauptfenster hinzu

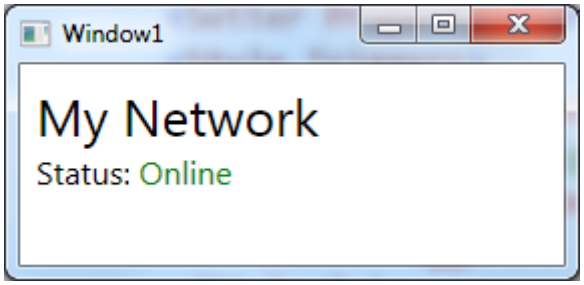
```
<Window.Resources>
     <local:NetworkDataProvider x:Key="ndp" />

     ...
</Window.Resources>
```

Erstellen Sie nun eine Oberfläche die den Netzwerkstatus mit einem **TextBlock** anzeigt. Binden Sie das **Text** Property des TextBlocks an das **Status** Property des **ndp**.

Definieren Sie einen **Style** für den **TextBlock** der mittels eines **DataTriggers** die Textfarbe vom **Status** abhängig macht (bspw. grün für online und rot für offline)

```
<Style x:Key="StatusText" TargetType="{x:Type TextBlock}">
    <Setter Property="..." Value="..." />
    ...
    <Style.Triggers>
        <DataTrigger Binding="..." Value="...">
            <Setter ... />
        </DataTrigger>
    </Style.Triggers>
</Style>
```



Aufgabe 3.2

Fügen Sie nun den Resourcen einen **ObjectDataProvider** hinzu, der die Netzwerkdaten aus der Methode **GetNetworkInterfaces**() des **NetworkDataProvider** beziehen soll.

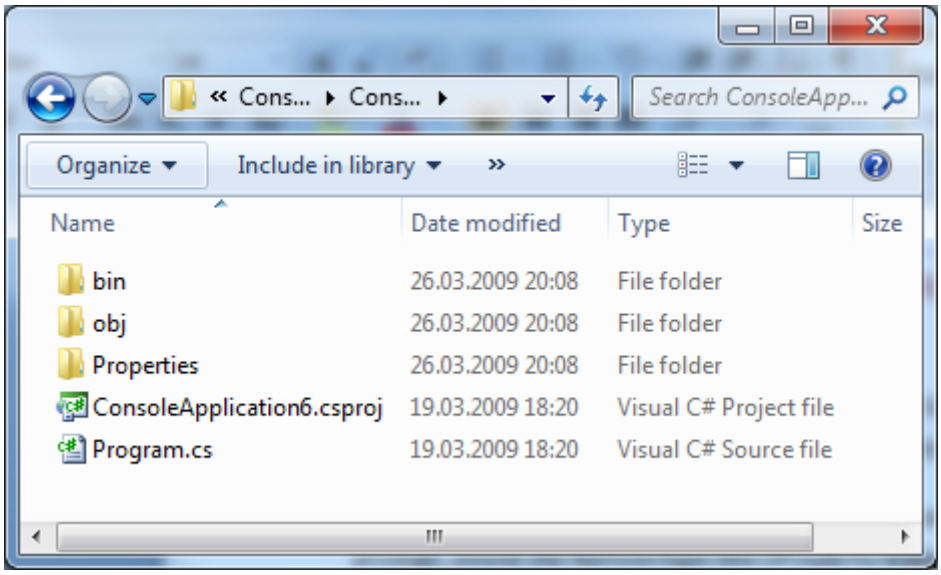
```
<ObjectDataProvider x:Key="..."
    ObjectInstance="{StaticResource ...}"
    MethodName="..." />
```

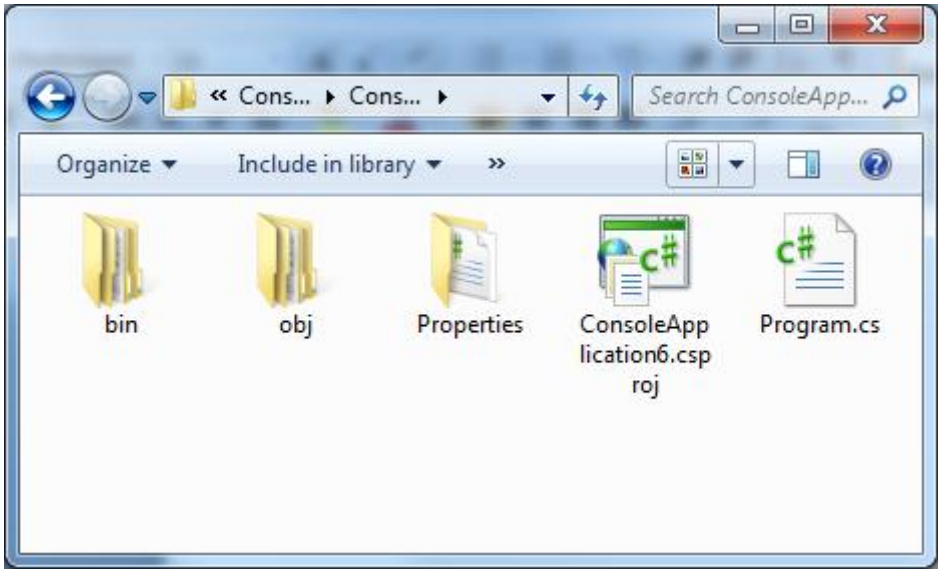
Fügen Sie nun ein **ItemsControl** ein und binden Sie dessen **ItemsSource** Eigenschaft an den **ObjectDataProvider**. (Warum binden wir nicht direkt an den **NetworkDataProvider**?)

Definieren Sie ein **DataTemplate** das den Netzwerkstatus (Property **Status**) in einem **TextBlock** anzeigt, sowie die Netwerkgeräte (Property **Values**) in einem **ListView**.

```
<ItemsControl ...>
    <ItemsControl.ItemTemplate>
        <DataTemplate>
            <TextBlock .../>
            <ListView .../>
        </DataTemplate>
    </ItemsControl.ItemTemplate>
</ItemsControl>
```

Das ListView bietet ähnlich wie der Windows Explorer mehrere “Views” auf Daten.





Wir wollen für die Netzwerkgeräte ein **GridView** wählen. Für das GridView lassen sich Spalten, **GridViewColumns**, definieren. Binden Sie die Properties **Name**, **Speed**, **Description** und **Type**.

```
<ListView ...>
    <ListView.View>
        <GridView>
            <GridViewColumn
                DisplayMemberBinding="{Binding ...}"
                Header="..." />
            ...
        </GridView>
    </ListView.View>
</ListView>
```

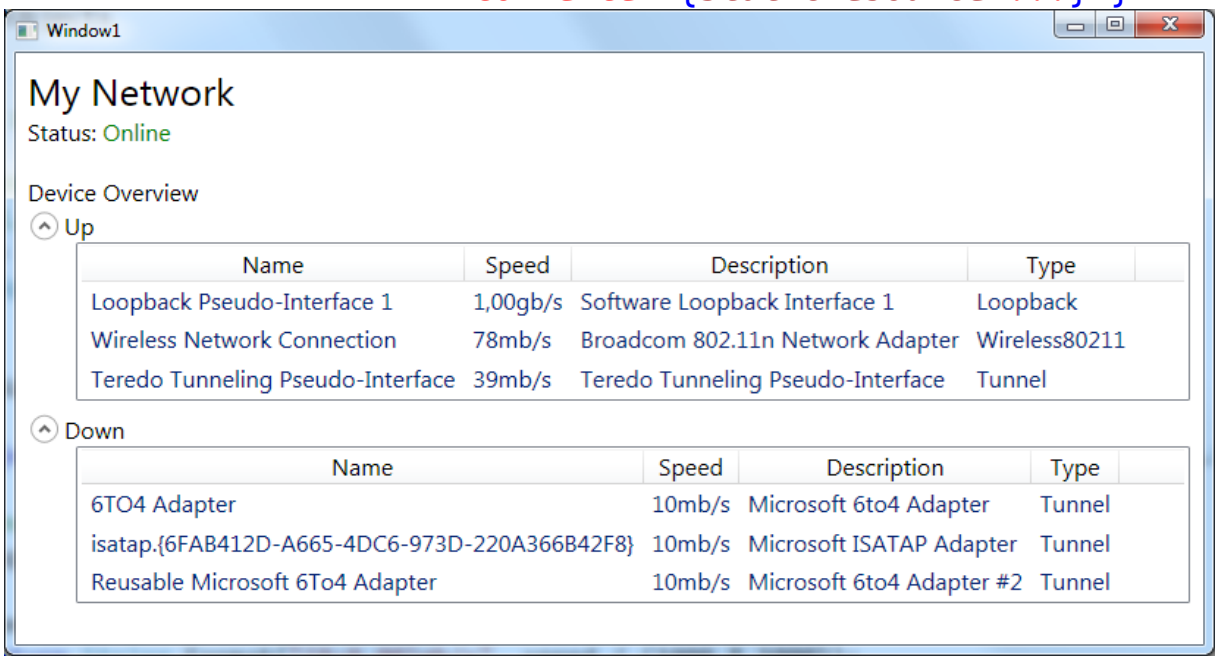
Aufgabe 3.4

Definieren Sie für das **Speed** Property einen **Converter**, der die Geschwindigkeit von Bit/s in eine passende Größe (MB/s, GB/s) umwandelt.

```
public class SpeedConverter : IValueConverter
```

Fügen Sie den Converter den Resourcen hinzu und ändern Sie das Binding entsprechend.

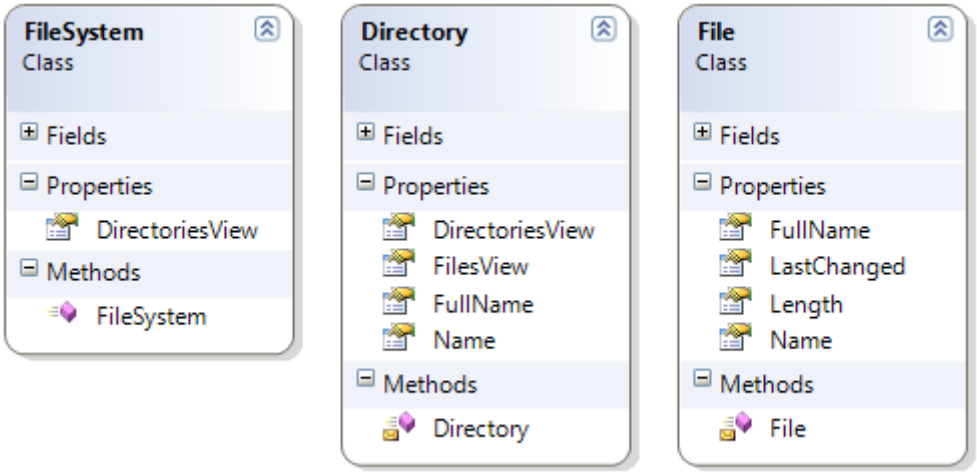
```
DisplayMemberBinding="{Binding ...,
                       Converter={StaticResource ...} }"
```



Aufgabe 4 – WPF Explorer

Öffnen Sie das Projekt WpfExplorer als Ausgangspunkt. Gegeben ist eine Implementierung FileSystem, die bei Instanzierung eine hirarchische Liste von Objekten im Dateisystem erstellt.

```
FileSystem fs = new FileSystem("D:\\Documents");
```



FileSystem enthält eine Liste von **Directory** Objekten des Wurzelverzeichnisses. **Directory** enthält eine Liste von **File** Objekten im Verzeichnis, sowie eine Liste von **Directory** Objekten als Unterverzeichnisse.

Aufgabe 4.1

Entwerfen Sie eine Oberfläche in der klassischen Exploreransicht, d.h. TreeView für den Verzeichnisbaun, ListView für die Dateien, Statusbar.

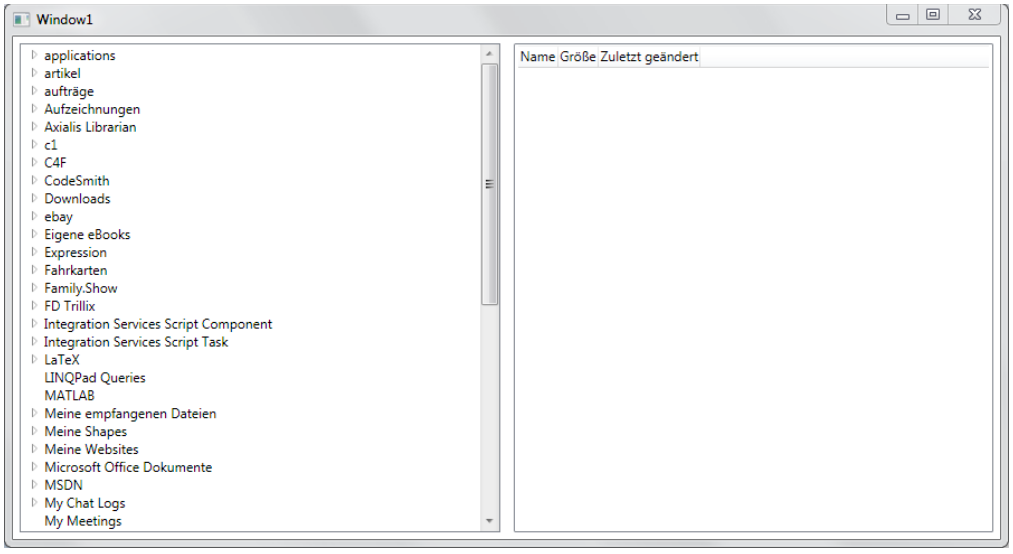
Binden Sie das TreeView an das **DirectoriesView** Property des **FileSystem** Objekt und das ListView an das **FilesView** Property des **SelectedItem** des TreeViews.

Verwenden Sie für die Darstellung des Directory ein **HirarchialDataTemplate**.

```
<HierarchicalDataTemplate
     ItemsSource="{Binding Path=DirectoriesView}"
     DataType="{x:Type local:Directory}">
     <TextBlock Text="{Binding Path=Name}" />
</HierarchicalDataTemplate>
```

Das ListView bietet verschiedene Views für die Daten. Verwenden Sie das GridView um die Spalten Name (Binding Name), Größe (Binding Length), Zuletzt geändert (Binding LastChanged) zu erzeugen.

```
<ListView ...>
     <ListView.View>
          <GridView>
               <GridViewColumn
               DisplayMemberBinding="{Binding PROPERTY}"
               Header="NAME" />
          </GridView>
     </ListView.View>
</ListView>
```



Aufgabe 4.2

Erweitern Sie das File Objekt um das Property **Icon**. Verwenden Sie den **IconProvider**, um bei Initialisierung des Objekts das zum Dateityp zugehörige Icon zu erhalten.

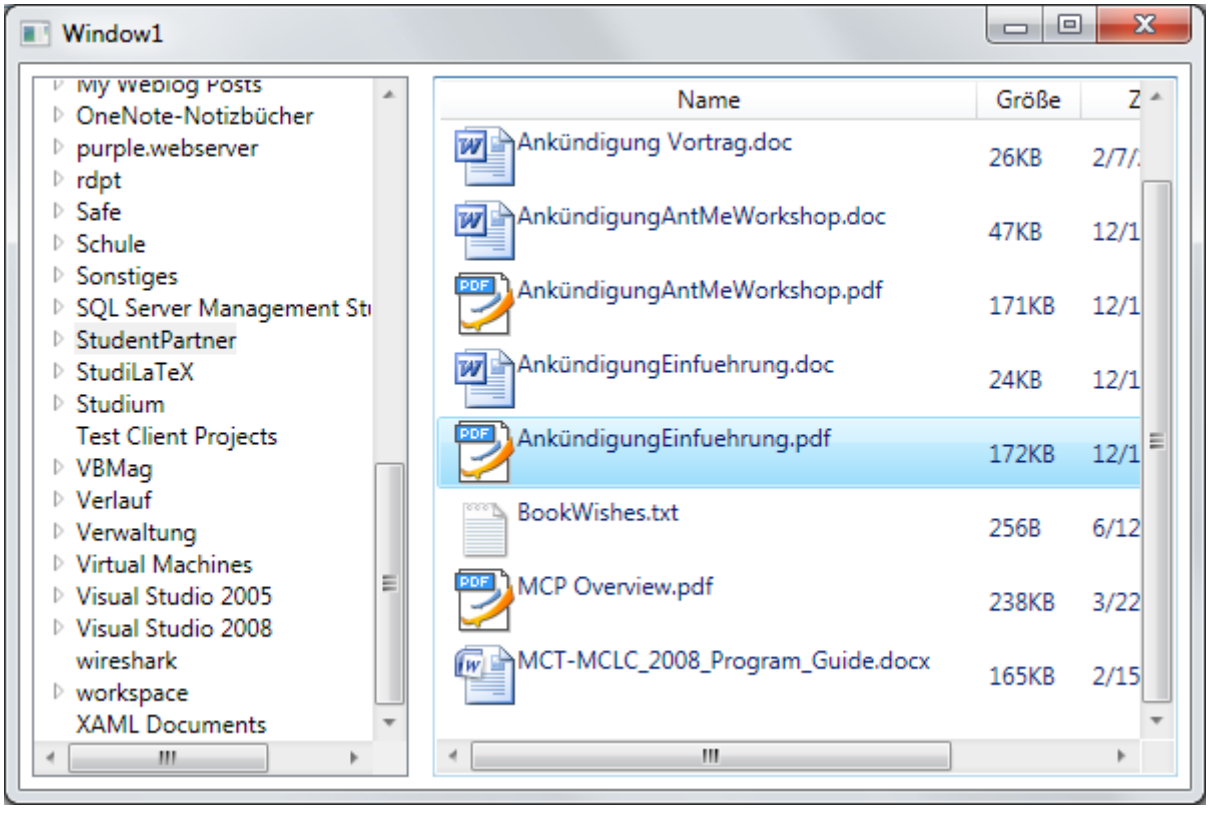
Erweitern Sie die GridViewColumn im ListView für das Property **Name**, indem Sie anstatt des **DisplayMemberBinding** Properties ein **DataTempalte** definieren, das sowohl den Namen als auch das Icon enthält.

```
<GridViewColumn Header="Name">
     <GridViewColumn.CellTemplate>
          <DataTemplate>
               ...
          </DataTemplate>
     </GridViewColumn.CellTemplate>
</GridViewColumn>
```

Schreiben Sie einen Converter FileSizeConverter, der die Größe in Byte in die größtmögliche Form GB, MB, KB, etc ohne führende Null umwandelt.

```
public class FileSizeConverter : IValueConverter
```

Fügen Sie in das DataTempalte für **Directory** einen **DataTrigger** ein, der die Schrift grau färbt, falls das Verzeichnis leer ist.



Aufgabe 4.3

Verwenden Sie das **GridViewColumnHeader.Click** Event des GridViews um die Spalten zu sortieren.

```
<ListView GridViewColumnHeader.Click="HANDLER_METH" ... />
```

Zugriff auf den Header und das View erhalten Sie durch

```
private void HANDLER_METH(object sender, RoutedEventArgs
e)
{
     ListView lv = sender as ListView;
      GridViewColumnHeader header =
(GridViewColumnHeader)e.OriginalSource;

     ICollectionView view =
(ICollectionView)lv.ItemsSource;

      ...
}
```

Erstellen Sie nun **SortDescriptions** für die angeklickte Spalte. Berücksichtigen Sie dabei auch, dass bei mehrmaligem Klicken der selben Spalte die Sortierreihenfolge umgekehrt werden soll.

Aufgabe 4.4

Fügen Sie ein Suchfeld in die Maske ein und verfahren Sie analog zur Aufgabe 3.3, um die Ansicht zu filtern.

Aufgabe 4.5 (Schwer)

Erweitern Sie das **FileSystem** Objekt um einen **FileSystemWatcher**, der das Dateisystem auf Änderungen überwacht und diese an die **Directory** Objekte propagiert.

Öffnen Sie nun den Wpf Explorer, löschen und erstellen Sie einige Dateien in den überwachten Verzeichnissen. Was fällt auf?