WPF-TREEVIEW UND MVVM IN EINKLANG BRINGEN (TEIL 1)

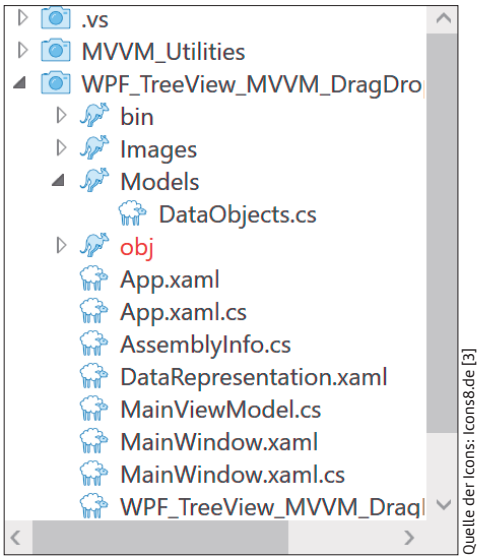
Verzweigte Strukturen

Mit einigen WPF-Tricks View-Model und TreeView sauber trennen.

So manche Steuerelemente in der Windows Presentation Foundation (WPF) können zu Kopfschmerzen führen, wenn man versucht, sie unter Einhalten des Model-View-ViewModel-Patterns einzusetzen. Das klassische TreeView-Control ist eines davon. Doch es gibt Lösungen.

Als die WPF entwickelt wurde, hatte niemand einen Plan, wie die Programmierer mit der neuen Technik umgehen sollten. Vieles war an frühere Vorgehensweisen angelehnt, wie man sie aus Visual Basic oder Windows Forms kannte. Strukturiertere Ansätze folgten erst im Lauf der Zeit, zum Beispiel in Form des Model-View-ViewModel-Patterns, kurz MVVM. Dieses Pattern kam nicht von Microsoft, sondern hatte sich durch Diskussionen in diversen Communities entwickelt. Es ist auch eher als Denkmuster denn als Implementierungsvorschrift zu verstehen, und so finden sich viele unterschiedliche Umsetzungen von MVVM. Nicht nur in WPF findet man dieses Muster. Auch bei Single-Page-Webanwendungen kommt es zum Einsatz (beispielsweise bei Angular), wenngleich auch mit anderen Programmiersprachen und anderer Syntax (JavaScript, HTML et cetera).

Heutzutage setzen die meisten Entwicklungsteams auf dieses Pattern, wenn sie Desktop-Anwendungen mit der WPF erstellen. Es bietet eine saubere Trennung der Gestaltung der Benutzeroberfläche von deren Logik. Da das Pattern aber erst später kam, sind viele der WPF-Steuerelemente nicht gut darauf vorbereitet. MVVM setzt auf Datenbindungen zwischen .NET-Properties. Für Methodenaufrufe oder Bindungen an Eventhandler müsste im C#- oder VB-Code auf die Instanzen der Steuerelemente zugegriffen werden, was die gewünschte Trennung wieder zunichtemachen würde.



Quelle der Icons: Icons8.de [3]

Darstellung einer Verzeichnisstruktur im TreeView-Control (Bild 1)

Bei einigen der Standard-Steuerelemente muss man daher tiefer in die Trickkiste greifen, um das MVVM-Pattern sinnvoll einsetzen zu können. Das trifft auch auf das TreeView-Control zu, das hier näher betrachtet werden soll.

Im ersten Teil dieser zweiteiligen Serie werden die grundlegenden Konzepte für die Anwendung von TreeView-Controls in Verbindung mit MVVM beschrieben. In einem zweiten Teil geht es dann um Drag-and-drop, was eigentlich simpel klingt, in der Umsetzung aber doch recht mühsam und aufwendig ist.

Das Fundament schaffen

Das Thema ist nicht neu. Bereits vor zehn Jahren veröffentlichte die dotnetpro einen Beitrag zum Umgang mit einer TreeView in WPF und MVVM [1]. An den Konzepten hat sich seitdem wenig geändert. Wichtig ist, dass man die verschiedenen Werkzeuge, welche die WPF zur Verfügung stellt, kennt und versteht. Dazu gehören beispielsweise Styles sowie ControlTemplates und DataTemplates. Anhand eines Beispiels sollen diese Techniken im Detail erläutert werden. Als Anwendungsfall dient ein Dateiverzeichnis, das in einer TreeView dargestellt werden soll, siehe Bild 1.

Zu sehen ist die Repräsentation der Verzeichnisstruktur in Form von hierarchisch angeordneten Knoten (hier vom Typ *TreeViewItem*). Ein Knoten besteht im Beispiel jeweils aus einem Miniaturbild und einem Text und kann ausgewählt oder auf- beziehungsweise zugeklappt werden sowie den Zustand *Verfügbar* oder *Nicht verfügbar* (Darstellung in Rot) aufweisen. In Umgebungen wie Windows-Forms würde man imperativ im Code Instanzen von *TreeViewItem* anlegen, Eigenschaften setzen und Eventhandler binden und hiermit das TreeView-Control füllen. Das würde aber das MVVM-Pattern verletzen.

Zunächst zur Beschreibung des Datenmodells. Im Beispiel könnten das Klassen sein, wie in Listing 1 zu sehen.

Listing 1: Modellklassen für das Beispielprojekt

```
public class DataObjectBase
{
  public string Caption { get; set; }
  public int Level { get; set; }
}

public class DirectoryDataObject : DataObjectBase {
}
public class FileDataObject : DataObjectBase { }
```

Die Basisklasse *DataObjectBase* hat zwei Eigenschaften (*Caption* und *Level*), die für die Repräsentation in der Baumstruktur genutzt werden können. Davon abgeleitet sind die Klassen *DirectoryDataObject* sowie *FileDataObject*, welche die Informationen zu einem Verzeichnis beziehungsweise einer Datei kapseln. Prinzipiell könnte man hier die 1:n-Beziehung zwischen einem Verzeichnis und seinen untergeordneten Verzeichnissen und Dateien direkt in Form von Auflistungen abbilden und diese Struktur auch direkt mit einer TreeView visualisieren (ein *HierarchicalDataTemplate* macht es möglich). Der Nachteil: Man hätte dann keinen Zugriff auf die Informationen der einzelnen *TreeViewItems*, also darüber, ob ein Knoten expandiert wurde oder nicht, ob er ausgewählt oder vielleicht nicht verfügbar ist. Daher empfiehlt sich die Definition von Hilfsklassen, die ihrerseits die benötigte Struktur für die Bindungen in der TreeView repräsentieren und andererseits allgemeingültig aufgebaut sind, sodass beliebige Daten angehängt werden können.

Die Klasse *TreeItem* (Listing 2) korreliert mit einem *TreeViewItem* im *TreeView*-Control. Sie stellt Properties wie *IsSelected*, *IsExpanded* oder *IsEnabled* zur Verfügung, die mit den entsprechenden Eigenschaften eines *TreeViewItem*-Objekts über Datenbindungen verknüpft werden sollen. Ein ähnliches Konzept wird im Artikel „WPF-Trick für MVVM-Apps“ [2] in dieser dotnetpro-Ausgabe vorgestellt, wenn es um den Umgang mit DataGrids und ListBoxen geht. Die eigentlichen Daten werden typneutral als *Object*-Referenz (Eigenschaft *Data*) angehängt. Eine Property *ToolTip* (auch vom Typ *Object*) kann benutzt werden, um Daten für ein im UI darzustellendes Tooltipp-Fenster bereitzustellen. Anstelle der *Object*-Referenzen könnte man auch Generics verwenden und Platzhalter für die Datentypen vorsehen, in der Praxis führt das aber meist zu zusätzlichem Aufwand und neuen Problemen, die den Nutzen durch die Typisierung übersteigen.

Für die Baumstruktur und den späteren Umgang damit sind zwei weitere Eigenschaften erforderlich: *Items* und *Parent*. *Parent* verweist auf den Elternknoten beziehungsweise hat den Wert *null*, falls keiner existiert. *Items* verweist auf die Auflistung der Kindelemente (Typ *TreeItemList*). Die Auflistung wird im Konstruktor von *TreeItem* instanziert, ist also auf jeden Fall vorhanden, zu Beginn aber leer.

TreeItemList besteht im Wesentlichen aus einer Ableitung von der Framework-Klasse *ObservableCollection*, stellt also über das implementierte Interface *INotifyCollectionChanged* ein Ereignis bereit, das im Code und im UI verwendet werden kann, um auf Änderungen zu reagieren.

```
public class TreeItemList :
    ObservableCollection<TreeItem>
{
  private readonly TreeItem owner;
  // ctor
```

▶

Listing 2: Hilfsklasse für die Datenbindung

```
public class TreeItem : NotificationObject
{
  public TreeItem() {
    Items = new TreeItemList(this);
  }

  private object data;

  // Angehängtes Model-Objekt
  public object Data {
    get { return data; }
    set { data = value; OnPropertyChanged(); }
  }

  // Daten für Anzeige eines Tooltipps
  public object ToolTip { get; set; }

  private bool isSelected;

  // Repräsentation der IsSelected-Eigenschaft
  // des TreeViewItems
  public bool IsSelected {
    get { return isSelected; }
    set { isSelected = value; OnPropertyChanged(); }
  }

   private bool isExpanded;

  // Repräsentation der IsExpanded-Eigenschaft
  // des TreeViewItems
  public bool IsExpanded {
    get { return isExpanded; }
    set { isExpanded = value; OnPropertyChanged(); }
  }

  private bool isEnabled = true;

  public bool IsEnabled {
    get { return isEnabled; }
    set { isEnabled = value; OnPropertyChanged(); }
  }

  // Navigation-Properties
  // Untergeordnete Items
  public TreeItemList Items { get; set; }

  // Parent dieses Items
  public TreeItem Parent { get; set; }
  ...
}
```

```
  // <param name="owner">Das TreeItem, zu dem
  // diese Liste gehört</param>
  public TreeItemList(TreeItem owner)
  {
    this.owner = owner;
  }

  // Beim Hinzufügen eines neuen Items dessen
  // Parent-Eigenschaft setzen
  // <param name="index"></param>
  // <param name="item"></param>
  protected override void InsertItem(
      int index, TreeItem item)
  {
    base.InsertItem(index, item);
    item.Parent = owner;
  }
  ...
}
```

Für den späteren Bedarf (Navigation innerhalb der Baumstruktur) wird die Liste noch um das Feld *owner* erweitert, das den übergeordneten Knoten referenziert. Das Feld wird im Konstruktor gesetzt (siehe auch Listing 2). Außerdem wird, wenn ein neues Element hinzugefügt wird, dessen Parent-Eigenschaft auf *owner* neu gesetzt (überschriebene Methode *InsertItem*).

Nachdem die benötigten Infrastrukturklassen definiert worden sind, gilt es nun, im ViewModel die Datenstrukturen für das Zusammenspiel mit der Oberfläche mit Leben zu füllen. Listing 3 zeigt den grundlegenden Aufbau des ViewModels. Im Konstruktor wird eine Instanz von *TreeItem* angelegt (Feld *rootTreeItem1*) und dessen Unterelemente durch den rekursiven Aufruf von *FillTree* mit den Informationen aus dem Dateisystem angelegt und gefüllt. Eine Read-only-Property (*Tree1*) stellt die Kindelemente des Root-Items für eine Datenbindung nach außen bereit. Nun ist es an der Zeit, den XAML-Teil näher zu betrachten

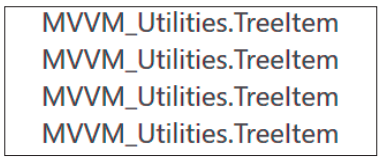
Griff in die WPF-Werkzeugkiste

Der erste Schritt besteht darin, die *ItemsSource*-Eigenschaft der TreeView über eine Datenbindung mit der Eigenschaft *Tree1* des ViewModels zu verknüpfen. Dadurch wird schon einmal die Liste der *TreeItems* der obersten Ebene in der TreeView sichtbar (Bild 2). Da die TreeView nichts mit dem Datentyp *TreeItem* anfangen kann, zeigt sie den aus *Object.ToString* gelieferten Text in der Oberfläche an, also den voll qualifizierten Typnamen *MVVM_Utilities.TreeItem*. Im zweiten Schritt müssen Sie der TreeView daher mitteilen, wie sie einen Knoten im Baum darstellen soll. Das geschieht mithilfe eines *DataTemplate* (Listing 4).

Zum Einsatz kommt hier der Typ *HierarchicalDataTemplate*, der sich bei Steuerelementen mit Baumstrukturen wie *TreeView* oder *Menu* anbietet. *HierarchicalDataTemplate* be-

Listing 3: Erzeugen der Datenstrukturen im ViewModel

```
public class MainViewModel : NotificationObject
{
  private TreeItem rootTreeItem1;
  public TreeItemList Tree1 {
    get { return rootTreeItem1.Items; }
  }
  ...
  public MainViewModel() {
    rootTree1 = new TreeItem();
    // Pfad bei Bedarf anpassen
    var path = Directory.GetParent(
      Environment.CurrentDirectory).Parent.
      Parent.Parent.FullName;
    FillTree(Tree1, path,0);
    ...
  }
  // Aufbau der Datenstruktur basierend auf dem
  // angegebenen Verzeichnis
  private void FillTree(TreeItemList tree,
      string directory, int level)
  {
    try {
      foreach (var dir in
        Directory.EnumerateDirectories(directory))
      {
        var ti = new TreeItem {
          Data = new DirectoryDataObject {
            Caption = Path.GetFileName(dir),
            Level=level }, ToolTip = dir, IsEnabled=
            Path.GetFileName(dir) != "obj" };
        tree.Add(ti);
        FillTree(ti.Items, dir, level+1);
      }
      foreach (var file in
          Directory.EnumerateFiles(directory))
      {
        var ti = new TreeItem {
          Data = new FileDataObject {
            Caption = Path.GetFileName(file),
            Level=level }, ToolTip = file };
        tree.Add(ti);
      }
    }
    catch (Exception) { }
  }
  ...
}
```



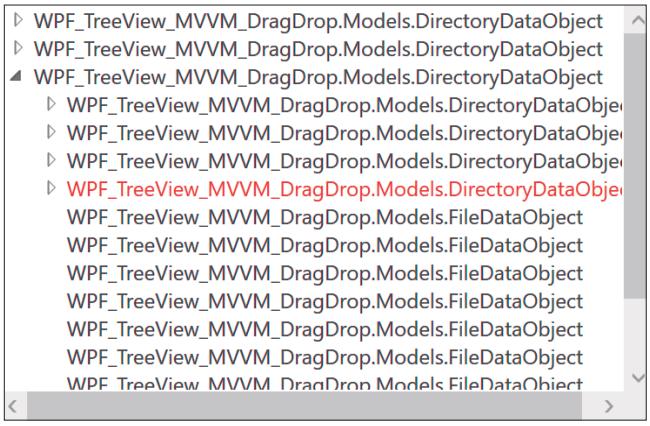
Der erste Ansatz: Die TreeView zeigt die erste Ebene der Hilfsstruktur (Bild 2)

steht im Inneren aus einem gewöhnlichen *DataTemplate*, über das die visuelle Repräsentation eines einzelnen Knotens (*TreeViewItem*) deklariert wird. Der Datenkontext für den Knoten ist jeweils ein *TreeItem*-Objekt aus der Hilfsstruktur. Um die Darstellung der Daten nicht einschränken zu müssen, wird ein *ContentPresenter*-Element an das durchgereichte Datenobjekt (Eigenschaft *TreeItem.Data*) gebunden. Das resultiert in der Ausgabe des für *DirectoryDataObject* beziehungsweise *FileDataObject* abgerufenen Textes von *ToString*. Noch nicht das, was wir wollen, aber die Datenstruktur ist schon mal erkennbar.

An dieser Stelle ist es auch möglich, auf die optische Gestaltung der Knoten Einfluss zu nehmen, bei denen *IsEnabled* auf *false* gesetzt ist. Innerhalb eines DataTemplates lässt sich das über einen *DataTrigger* einrichten. Dieser prüft den Wert von *IsEnabled* und führt im Fall von *false* einen Setter aus, der die Eigenschaft *Foreground* der TreeView auf *Red* umschaltet. Bei Bedarf können hier auch andere Eigenschaften beeinflusst werden.

Im Gegensatz zum klassischen *DataTemplate* hat das *HierarchicalDataTemplate* noch weitere Eigenschaften, insbesondere die hier verwendete Eigenschaft *ItemsSource*. Denn das Template beschreibt, wie die Zuordnung über *TreeView.ItemTemplate* erahnen lässt, nur die Visualisierung eines einzelnen Elements. Über Eigenschaft *ItemsSource* können Sie dem *TreeViewItem* nun mitteilen, woher es seine Kindelemente beziehen soll. Das ist im vorliegenden Beispiel die Eigenschaft *Items* der Hilfsklasse *TreeItem*. Und schon zeigt die TreeView die gesamte Baumstruktur an (Bild 3). Durch die Deklaration des *HierarchicalDataTemplate* legt die TreeView also für jedes *TreeItem* der Hilfsstruktur ein *TreeViewItem* für die Darstellung an und generiert somit den gleichen visuellen Aufbau des Baums, wie er im Hintergrund festgelegt wurde. Doch lassen sich nun die Verbindungen zwischen den Eigenschaften *IsSelected*, *IsExpanded* und *IsEnabled* in beiden Objektwelten herstellen? Ein *TreeViewItem* kann man im ViewModel ja nicht referenzieren, da es im UI erst zur Laufzeit automatisch generiert wird.

Die Lösung besteht in der Nutzung der Eigenschaft *ItemContainerStyle* der TreeView. Über diese Eigenschaft kann man einen Style vorgeben, der jedem generierten *TreeViewItem* zugewiesen wird.



Die Baumstruktur steht, aber die Inhalte fehlen noch (Bild 3)

Listing 4: HierarchicalDataTemplate

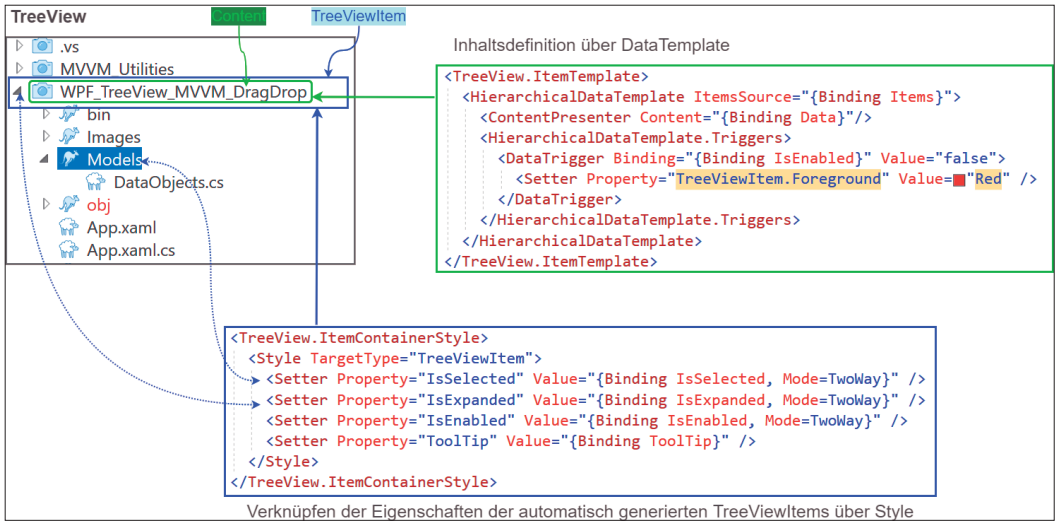
```
<TreeView ItemsSource={Binding Tree1} ...  >
  <TreeView.ItemTemplate>
    <HierarchicalDataTemplate ItemsSource=
        "{Binding Items}">
      <ContentPresenter Content="{Binding Data}"/>
      <HierarchicalDataTemplate.Triggers>
        <DataTrigger Binding="{Binding IsEnabled}"
            Value="false">
          <Setter Property="TreeViewItem.Foreground"
            Value="Red" />
        </DataTrigger>
      </HierarchicalDataTemplate.Triggers>
    </HierarchicalDataTemplate>
  </TreeView.ItemTemplate>
  ...
</TreeView>
```

```
<TreeView ...>
  <TreeView.ItemContainerStyle>
    <Style TargetType="TreeViewItem">
      <!--Verknüpfung der Eigenschaften eines
        TreeViewItems-->
      <Setter Property="IsSelected" Value=
        "{Binding IsSelected, Mode=TwoWay}" />
      <Setter Property="IsExpanded" Value=
        "{Binding IsExpanded, Mode=TwoWay}" />
      <Setter Property="IsEnabled" Value=
        "{Binding IsEnabled, Mode=TwoWay}" />
      <Setter Property="ToolTip" Value=
        "{Binding ToolTip}" />
    </Style>
  </TreeView.ItemContainerStyle>
  ...
</TreeView>
```

Der *TargetType* des Styles muss dem Objekttyp entsprechen, der automatisch generiert wird. Hier ist es der Typ *TreeViewItem*, bei einer ListBox wäre es *ListBoxItem* und so weiter.

Über die Setter kann im Style direkt auf die jeweiligen Properties zugegriffen werden und somit kann man ihnen, wie im Code zu sehen, auch einen Bindungsausdruck zuord- ▶

Komplexes Konstrukt: TreeViewItem, HierarchicalDataTemplate, ItemContainerStyle (Bild 4)



nen. Dieser stellt nun die Verbindung zum Beispiel zwischen der Eigenschaft *IsExpanded* auf der Seite des *TreeViewItem* und *IsExpanded* auf der Seite der Hilfsstruktur *TreeItem* her. Zu beachten ist hier, dass die Bindungen explizit auf *TwoWay* gesetzt werden müssen, da sonst Änderungen im UI (Selektion, Auf-/Zuklappen) nicht zu Änderungen in der Datenstruktur im Hintergrund führen.

Bild 4 veranschaulicht die Zusammenhänge noch einmal. Über *ItemTemplate/HierarchicalDataTemplate* wird der strukturelle Aufbau des Baums festgelegt. Die TreeView legt so für jede Instanz von *TreeItem* automatisch ein *TreeViewItem* an (blauer Rahmen). Das Template (grüner Rahmen) beschreibt die Repräsentation des Inhalts (Content). Es hat selbst aber keinen Zugriff auf die anderen Eigenschaften eines *TreeViewItem* – es definiert ja nur den Inhalt. Erst über *ItemContainerStyle* kommt man an die Eigenschaften eines so generierten *TreeViewItem* und kann sie via Setter-Definitionen setzen. Hier geht noch viel mehr. Über die Setter kann man beispielsweise auch jedem *TreeViewItem* ein Kontextmenü zuordnen (siehe auch [1]) oder farbliche Füllungen oder Schriftparameter setzen.

So weit steht die Infrastruktur, die sich allgemeingültig für beliebige Modellklassen benutzen lässt. Die nun noch fehlende sinnvolle Darstellung der Inhalte gelingt wieder auf dem

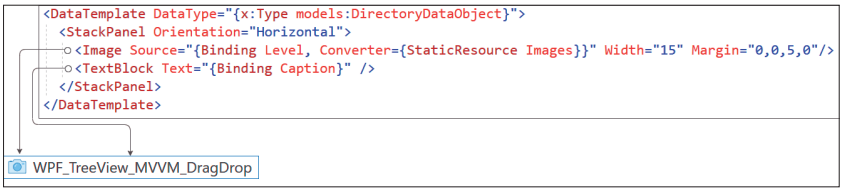
Listing 5: Vorlagen für die Model-Klassen

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/
    2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/
    2006/xaml"
  xmlns:models=
    "clr-namespace:WPF_TreeView_MVVM_DragDrop.Models"
  xmlns:mvvm="fuechse-online.de"
  xmlns:local=
    "clr-namespace:WPF_TreeView_MVVM_DragDrop">

  <mvvm:ResourceListConverter x:Key="Images">
    <mvvm:ResourceListConverter.Items>
      <BitmapImage UriSource=
        "/Images/icons8-camera-50.png"/>
      <BitmapImage UriSource=
        "/Images/icons8-kangaroo-50.png"/>
      <BitmapImage UriSource=
        "/Images/icons8-bird-50.png"/>
      <BitmapImage UriSource=
        "/Images/icons8-opened-folder-50.png"/>
      <BitmapImage UriSource=
        "/Images/icons8-sheep-50.png"/>
    </mvvm:ResourceListConverter.Items>
  </mvvm:ResourceListConverter>

  <!--Templates für die Darstellung der Model-Objekte
    (hier im Beispiel Directory/FileDataObject-->
  <DataTemplate DataType=
      "{x:Type models:DirectoryDataObject}">
    <StackPanel Orientation="Horizontal">
      <Image Source= "{Binding Level,
        Converter={StaticResource Images}}" Width="15"
        Margin="0,0,5,0" />
      <TextBlock Text="{Binding Caption}" />
    </StackPanel>
  </DataTemplate>
  <DataTemplate DataType=
      "{x:Type models:FileDataObject}">
    <StackPanel Orientation="Horizontal">
      <Image Source="/Images/icons8-sheep-50.png"
        Width="15" Margin="0,0,5,0"/>
      <TextBlock Text="{Binding Caption}" />
    </StackPanel>
  </DataTemplate>
</ResourceDictionary>
```

bekannten Weg über DataTemplates, hier ausgelagert in ein eigenes *ResourceDictionary* (Listing 5), das seinerseits wie folgt über die *App.xaml* eingebunden wurde:

```
<Application x:Class=
    "WPF_TreeView_MVVM_DragDrop.App"
    xmlns="http://schemas.microsoft.com/
           winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source=
          "/DataRepresentation.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

Für die Klassen *DirectoryDataObject* und *FileDataObject* wird hier jeweils eine allgemeingültige Vorlage beschrieben, die immer dann zum Einsatz kommt, wenn WPF Objekte des betreffenden Typs rendern soll und nicht explizit etwas anderes vorgegeben wird. Im vorliegenden Fall nutzen Templates also automatisch das im *ContentPresenter* (Listing 4) durchgeschleifte Datenobjekt. Der Kontext der Bindungen ist somit ein Datenobjekt, das durch die Ableitung von *DataObjectBase* (siehe Listing 1) die Eigenschaften *Caption* und *Level* aufweist. *Caption* wird in beiden Fällen direkt an die *Text*-Eigenschaft eines TextBlocks gebunden, *Level* im Fall von *DirectoryDataObject* an die *Source*-Eigenschaft eines Image-Controls. Bild 5 veranschaulicht noch einmal den Zusammenhang zwischen DataTemplate und der resultierenden Darstellung.



Repräsentation des Models (Bild 5)

Die Versuchung ist groß, Objekte wie Brushes oder Images, die für die Darstellung benötigt werden, vom ViewModel an die Oberfläche durchzureichen. Das ist jedoch aus verschiedenen Gründen ungünstig. Erstens würden WPF-spezifische Datentypen im C#-Code benötigt, was man ja eigentlich vermeiden wollte, und zweitens würde ein Aspekt der UI-Gestaltung in den C#-Code verlagert. Ein später hinzugerufener Designer wird Probleme haben, die Herkunft eines bestimmten Brushes oder eines Images oder vielleicht auch nur eines Ressourcen-Pfads wiederzufinden. Deswegen gehören solche Dinge immer in den XAML-Code. Es gibt verschiedene Wege, die Verbindung zwischen den Daten (hier die Eigenschaft *Level*) und den WPF-Objekten herzustellen.

Im Beispiel geschieht dies durch den Einsatz einer Converter-Klasse (*ResourceListConverter*, Listing 6). Der Converter definiert eine öffentliche Property namens *Items* vom Typ *List<object>*. Im XAML-Code können ihr beliebige Objekte zugewiesen werden, wie im Beispiel von Listing 5 die *BitmapImage*-Objekte. Der C#-Code muss bei dieser Konstruktion keinerlei Kenntnisse vom Typ dieser Objekte haben, sondern muss nur auf Basis des übergebenen Indexes ein Element aus dieser Liste auswählen. So bleibt der Algorithmus für die Auswahl Sache des C#-Codes und das Bereitstellen infrage kommender Ressourcen Aufgabe des XAML-Codes.

Statt die Zuordnung von Styles und Templates wie oben beschrieben vorzunehmen, kann alternativ auch der Weg über Selector-Klassen eingeschlagen werden. Je nach Control werden Eigenschaften wie *ContentTemplateSelector*, *ItemTemplateSelector* oder *ItemContainerStyleSelector* unterstützt. Ihnen weist man eine Referenz auf ein Objekt zu, dessen Typ von *TemplateSelector* beziehungsweise *StyleSelector* abgeleitet ist. In diesen abgeleiteten Klassen überschreibt man dann die Methode *SelectTemplate* beziehungsweise *SelectStyle* und kann so wiederum über C#-Code eine Auswahl treffen, die im UI zum Tragen kommt. Verschiedene Drittthersteller von WPF-Controls nutzen gerne diesen Weg. Aber auch hier sollte man – entgegen manchen Beispielen aus der Microsoft-Dokumentation und im Web – darauf achten, im C#-Code keine WPF-Objekte, auch keine Ressourcen- ▶

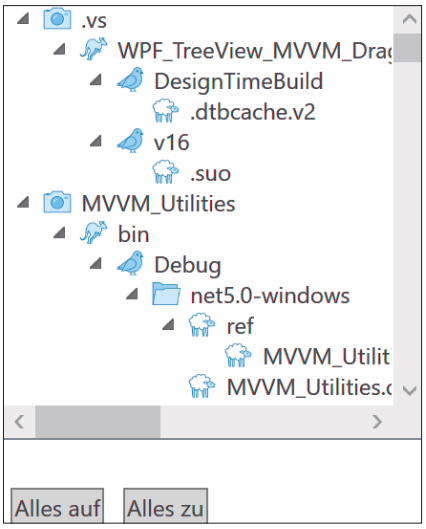
Listing 6: ResourceListConverter

```
// Brücke zwischen einer Auflistung im XAML-Code und
// einem über eine Datenbindung übergebenen Index
public class ResourceListConverter : IValueConverter
{
  // Liste der verfügbaren Objekte
  public List<object> Items { get; set; } =
    new List<object>();

  public object Convert(object value,
      Type targetType, object parameter,
      CultureInfo culture)
  {
    // Erwartet wird ein Index vom Typ Int32
    int index = (int)value;
    if (index >= Items.Count) index = 0;
    return Items[index];  // Bei Bedarf Fehler
  }                       // abfangen

  public object ConvertBack(object value,
      Type targetType, object parameter,
      CultureInfo culture)
  {
    throw new NotImplementedException();
  }
}
```

Keys zu generieren, die man später im XAML-Code vergeblich sucht.

Lohn der Mühe

Nach diesem Ausflug in die Welt der Zaubertricks von WPF zeigt das TreeView-Beispiel nun endlich die gewünschte Darstellung aus Bild 1. Auch wenn der Aufwand vergleichsweise groß erscheint, besteht ein erheblicher Teil des Codes aus wiederverwendbarer Infrastruktur. Letztere lässt sich noch um ein paar praktische Funktionen ergänzen. Eine davon wäre das vollständige Auf- oder Zusammenklappen des gesamten Baums oder eines Teilbaums. Hierzu muss man lediglich die *IsExpanded*-Eigenschaft eines vorgegebenen Knotens setzen und dasselbe anschließend für alle Kindelemente rekursiv durchführen.



Das Implementieren von „Alles auf“ und „Alles zu“ ist jetzt ein Klacks (Bild 6)

Eine Ergänzung von *TreeItemList* erfüllt die Aufgabe. *ExpandOrCollapseAll* nimmt einen booleschen Wert entgegen, der angibt, ob auf- oder zugeklappt werden soll.

```
public class TreeItemList : ObservableCollection<TreeItem>
{
  ...
  // Auf- oder Zuklappen aller Knoten dieser und
  // aller darunter liegenden Ebenen
  // <param name="expand">true: aufklappen,
  // false: zuklappen</param>
  public void ExpandOrCollapseAll(bool expand)
  {
    foreach (var item in this)
    {
      item.IsExpanded = expand;
      item.Items.ExpandOrCollapseAll(
          expand);
    }
  }
}
```

Für alle Kindelemente wird *IsExpanded* in einer Schleife gesetzt und über den rekursiven Aufruf derselben Methode der Zustand auch an die Kindelemente weitergegeben.

Im ViewModel kann man dann Commands bereitstellen, die für das Root-Element die Methode aufrufen und so dem Anwender ermöglichen, mit einem Klick den gesamten Baum aufzuklappen oder wieder auf die oberste Ebene einzuschränken:

```
public MainViewModel() {
  ...
  ExpandAllCommand = new ActionCommand(() =>
    Tree1.ExpandOrCollapseAll(true));
  CollapseAllCommand = new ActionCommand(() =>
    Tree1.ExpandOrCollapseAll(false));
  ...
}
```

Noch ein paar Buttons mit den Commands verknüpft, und schon sieht es aus wie in Bild 6.

Vielleicht möchten Sie ein bestimmtes TreeItem auswählen und alle Elternknoten aufklappen, sodass es zu sehen ist? Die Suche wäre Aufgabe des ViewModels (Listing 7), das Auswählen und Aufklappen kann *TreeItem* übernehmen:

```
public class TreeItem : NotificationObject {
  ...
  public void ExpandAncestors(
    bool expand)
```

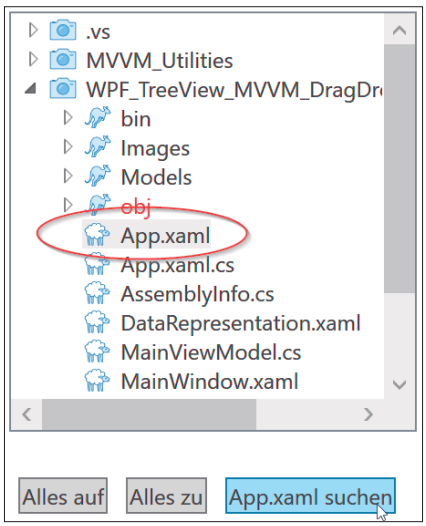
Listing 7: Suche nach einem Knoten

```
public class MainViewModel : NotificationObject {
  ...
  // Suche nach dem ersten Vorkommen von App.xaml
  private void OpenAppXaml() {
    var item = FindCaption("App.xaml", Tree1);
    if(item == null) return;

    item.IsSelected = true;      // TreeItem auswählen
    item.ExpandAncestors(true); // Alle Vorgänger-
  }                             // knoten aufklappen

  // Rekursive Suche nach einem Verzeichnis-
  // oder Dateinamen
  private TreeItem FindCaption(
      string caption, TreeItemList list)
  {
    foreach (var item in list) {
      if (((DataObjectBase)item.Data).Caption ==
        caption) return item;
      var subitem =
        FindCaption(caption, item.Items);
      if (subitem != null) return subitem;
    }
    return null;
  }
}
```

```
  {
    if (Parent != null) {
      Parent.IsExpanded = expand;
      Parent.ExpandAncestors(expand);
    }
  }
}
```

ExpandAncestors setzt die *IsExpanded*-Eigenschaft des Elternknotens, sofern dieser existiert, und ruft die Methode für diesen rekursiv auf. So werden vom ausgehenden *TreeItem* alle anderen oberhalb expandiert. Im Beispiel wird nach dem ersten Vorkommen des Dateinamens *App.xaml* gesucht. Das Ergebnis zeigt Bild 7. Allerdings soll an dieser Stelle ein Problem nicht verschwiegen werden: Wenn vor der Suche der gesamte Baum aufgeklappt war, wird der gewünschte Knoten zwar selektiert, aber unter Umständen liegt er außerhalb des Bereichs, der von der TreeView angezeigt wird. Um ihn sichtbar zu machen, müsste man entweder die TreeView anweisen, die passenden Scrollbar-Einstellungen vorzunehmen, oder alle anderen Knoten schließen. Ersteres könnte man über die Methode *FrameworkElement.BringIntoView* versuchen. Einige Experimente hierzu lieferten aber kein zufriedenstellendes Ergebnis. Vermutlich müsste man sich in diesem Fall doch einen genaueren Einblick in das Rendering der TreeView verschaffen.

TreeItem und *TreeItemList* sind keine Einbahnstraßen. Über ihre Eigenschaften kann man vom C#-Code aus das Verhalten in der Oberfläche beeinflussen. Andererseits werden die Setter der Properties ausgeführt, wenn der Anwender etwas in der TreeView auswählt oder aufklappt.



Selektion und Aufklappen eines bestimmten Knotens per Knopfdruck (Bild 7)

Zunutze machen kann man sich das, indem man in den Settern zusätzliche Events auslöst und diese nach außen bereitstellt. So lassen sich spezifische Ereignisse im ViewModel beobachten und auswerten. Die vorliegende Implementierung sieht bislang lediglich eine Behandlung via *INotifyPropertyChanged* vor. Aber die Infrastrukturklassen können ja beliebig erweitert werden. Um die aktuelle Auswahl nachverfolgen zu können, kann zum Beispiel in *TreeItem* ein Event deklariert werden:

```
public event EventHandler
  IsSelectedChanged;
```

Der Event wird im Setter von *IsSelected* gefeuert, sobald sich der Zustand geändert hat:

```
if (isSelected == value) return;
isSelected = value;
OnPropertyChanged();
IsSelectedChanged?.Invoke(this, EventArgs.Empty);
```

Der Event wird dann bei Änderung der Auswahl zweimal ausgelöst, einmal für das zuvor selektierte *TreeItem*, danach für das neu selektierte. Im ViewModel kann man den Event abonnieren und bleibt somit stets auf dem Laufenden, was die aktuelle Auswahl in der TreeView betrifft.

Noch ein bisschen aufräumen

Nicht zuletzt für den kommenden zweiten Teil, der durch den Einbau von Drag-and-drop sehr Event-lastig wird, sollten Sie den Code aufteilen und alle Infrastrukturklassen in eine eigenständige Klassenbibliothek packen. Die oben beschriebenen Klassen *TreeItem*, *TreeItemList* und *ResourceListConverter* und die hier nicht weiter dokumentierten Klassen *NotificationObject* und *ActionCommand* können einfach nach Anpassen der Namensräume in die Bibliothek verschoben werden.

Für die TreeView bietet sich eine Kapselung an, in der die beschriebenen Strukturen (*HierarchicalDataTemplate*, *ItemContainerStyle*) umgesetzt werden, damit man sie nicht bei jeder TreeView erneut explizit setzen muss. Eigentlich wäre das ein guter Einsatzfall für ein *CustomControl*, dessen *ControlTemplate* später ausgetauscht werden könnte. Allerdings ist, gerade im Hinblick auf das bevorstehende Eventhandling, damit leider auch ein sehr großer Aufwand verbunden. Daher soll hier der Weg unter Verwendung eines UserControls eingeschlagen werden. Der XAML-Code des UserControls kapselt zunächst nur die darzustellende TreeView mit ihren Definitionen für *ItemContainerStyle* und *ItemTemplate* (Listing 8). Zur Erinnerung: Das Template für die Darstellung der Model-Klassen gehört zum anwendungsspezifischen ▶

Listing 8: Kapseln in einem UserControl

```
<UserControl x:Class=
    "MVVM_Utilities.ExtendedTreeView" ... >
  <Grid>
    <TreeView Name="_tv_"  ...>
      <TreeView.ItemContainerStyle>
        <Style TargetType="TreeViewItem">
          ...
        </Style>
      </TreeView.ItemContainerStyle>
      <TreeView.ItemTemplate>
        <HierarchicalDataTemplate ItemsSource=
            "{Binding Items}">
          ...
        </HierarchicalDataTemplate>
      </TreeView.ItemTemplate>
    </TreeView>
   </Grid>
</UserControl>
```

Listing 9: CodeBehind des UserControls

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Threading;

namespace MVVM_Utilities
{
  // Interaction logic for ExtendedTreeView.xaml
  public partial class ExtendedTreeView : UserControl
  {
    public ExtendedTreeView() {
      InitializeComponent();
    }

    public TreeItemList Items {
      get {
        return (
          TreeItemList)GetValue(ItemsProperty);
      }
      set {
       SetValue(ItemsProperty, value);
      }
    }
    // Using a DependencyProperty as the
    // backing store for Items. This enables
    // animation, styling, binding, et cetera.
    public static readonly DependencyProperty
      ItemsProperty =  DependencyProperty.Register(
      "Items", typeof(TreeItemList),
      typeof(ExtendedTreeView),
      new FrameworkPropertyMetadata(OnItemsChanged));

    private static void OnItemsChanged(
        DependencyObject sender,
        DependencyPropertyChangedEventArgs e)
    {
      var uc = sender as ExtendedTreeView;
      // Datenstruktur an TreeView durchreichen
      uc._tv_.ItemsSource = uc.Items;
    }
}
```

Hauptprogramm. Listing 9 zeigt den Code-behind des UserControls. Die Verbindung zur Außenwelt erfolgt über dabei eine Dependency-Property namens *Items*. Sie ist vom Typ *TreeItemList*. Wird sie von außen gesetzt, resultiert das in einem Aufruf von *OnItemsChanged*. Dort wird die *ItemsSource*-Eigenschaft der TreeView gesetzt.

Um das Einbinden von Elementen einer Klassenbibliothek im XAML-Code zu vereinfachen, empfiehlt es sich, die verwendeten Namensräume über Assembly-Attribute bekannt zu machen:

```
using System.Windows.Markup;

[assembly: XmlnsDefinition(
  "fuechse-online.de", nameof(MVVM_Utilities))]
[assembly: XmlnsPrefix("fuechse-online.de","mvvm")]

/* Verwendung im XAML-Code:
  <Window xmlns:mvvm="fuechse-online.de"> ...
  <mvvm:ExtendedTreeView Items="{Binding ...}">
*/
```

Außerhalb der Bibliothek können dann die Aliasnamen für die Deklaration der Namespaces verwendet und auf die Angabe der Assembly verzichtet werden.

Fazit und Ausblick

Es wurde die Vorgehensweisen beschrieben, die mithilfe einiger Infrastrukturklassen und einiger WPF-Tricks eine saubere Trennung zwischen der Welt des View-Models und der Darstellung in einer TreeView ermöglichen. Über Templates, Styles und Datenbindungen lassen sich viele Anwendungsfälle abdecken. Änderungen der Datenstrukturen im View-Model führen zu Änderungen im UI und vice versa. Für die beschriebenen Konstruktionen müssen keine Eventhandler verknüpft und keine Methoden der Steuerelemente aufgerufen werden. Das wird sich im kommenden zweiten Teil ändern, wenn es darum geht, Drag-and-drop umzusetzen. Unter [4] finden Sie den Code der Beispielanwendung. ■

[1] Joachim Fuchs, Hierarchien bändigen, dotnetpro 1/2011, Seite 64ff., www.dotnetpro.de/A1101Hierarchic
[2] Joachim Fuchs, WPF-Trick für MVVM-Apps, dotnetpro 7/2021, Seite 26ff., www.dotnetpro.de/A2107DataGridMultiselect
[3] Quelle der Icons, https://icons8.de
[4] Sourcecode, www.dotnetpro.de/SL2107TreeViewMVVM1

Dr. Joachim Fuchs
ist begeisterter Anhänger von Microsofts .NET-Philosophie. Er arbeitet als Softwarearchitekt, Berater und Dozent im Expertennetzwerk www.it-visions.de. Seine Schwerpunkte liegen derzeit bei XAML- und Web-UI-Technologien.
dnp@fuechse-online.de

dnpCode A2107TreeViewMVVM