WPF-TREEVIEW UND MVVM IN EINKLANG BRINGEN, TEIL 3

Scroll-Hilfen und Adorner

Ein wenig Komfort für das Drag-and-drop von TreeViewItems.

In den vorangegangenen beiden Teilen dieser Serie [1][2] wurde die Basis für Drag-and-drop-Funktionen in Verbindung mit TreeViews im MVVM-Stil gelegt. Zuletzt ging es darum, wie man erreicht, dass die TreeView, über die der Anwender ein Item mit der Maus zieht, automatisch den für das Ablegen des gezogenen Items anvisierten Zielbereich aufklappt und anzeigt. Der Lösungsansatz im Demoprojekt besteht darin, während des Zieh-Vorgangs Steuerflächen im oberen beziehungsweise unteren Bereich der TreeView einzublenden, die beim Überfahren mit der Maus die gewünschte Bildlauf-Aktion auslösen. Als Steuerflächen werden hier einfach *ContentControls* angelegt und in einem Grid über der TreeView positioniert (erster Teil von Listing 1). *DragOver* ist der Event, der hier zu berücksichtigen ist.

Vor der Einleitung und nach Abschluss des Drag-Vorgangs wird die Sichtbarkeit der Steuerflächen umgeschaltet (unterer Teil in Listing 1). In den jeweiligen Handlern der *DragOver*-Events wird dann das *ScrollViewer*-Objekt der TreeView aufgefordert, eine Zeile nach oben oder unten zu scrollen. Solange der Anwender den Mauszeiger auf einer Schaltfläche be-

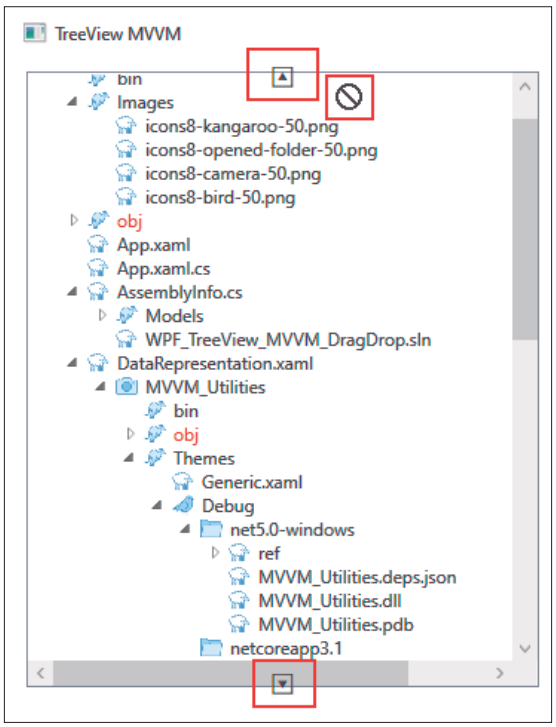
Listing 1: Buttons als Scroll-Hilfe

```
<UserControl x:Class=
    "MVVM_Utilities.ExtendedTreeView"...>
  <Grid >
    ...
    <!--Buttons für Scrolling während Drag-and-drop-->
    <ContentControl Name="ScrollUpBtn"
      Content="??" AllowDrop="True"
      DragOver="ScrollUpBtn_DragOver"
      Visibility="Collapsed" .../>
    <ContentControl Name="ScrollDownBtn"
      Content="??" AllowDrop="True"
      DragOver="ScrollDownBtn_DragOver"
      Visibility="Collapsed" .../>
    </Grid>
</UserControl>

public partial class ExtendedTreeView : UserControl
{
  ...
  private void TVI_MouseMove(object sender,
      MouseEventArgs e)
  {
    ...
    // Rahmenbedingungen für DragDrop-Start prüfen
    if (e.LeftButton == MouseButtonState.Pressed &&
        isDragging && DragDropController.CanDrag(ti))
    {
      ScrollUpBtn.Visibility = Visibility.Visible;
      ScrollDownBtn.Visibility = Visibility.Visible;

      // DoDragDrop kehrt erst nach Abschluss
      // der Aktion wieder zurück
      DragDrop.DoDragDrop(tvi, tvi.DataContext,
        DragDropEffects.All);

      // fertig
      ScrollUpBtn.Visibility = Visibility.Collapsed;
      ScrollDownBtn.Visibility = Visibility.Collapsed;
    }
  }
  // ScrollViewer aus Template des TreeViews ermitteln
  private ScrollViewer TVScrollViewer =>
    _tv_.Template.FindName(
    "_tv_scrollviewer_", _tv_) as ScrollViewer;

  // Hilfs-Buttons für Scrolling während Drag-and-drop
  private void ScrollUpBtn_DragOver(object sender,
      DragEventArgs e)
  {
    TVScrollViewer.LineUp();
    // Kein Drop zulassen
    e.Effects = DragDropEffects.None;
  }

  private void ScrollDownBtn_DragOver(
      object sender, DragEventArgs e)
  {
    TVScrollViewer.LineDown();
    // Kein Drop zulassen
    e.Effects = DragDropEffects.None;
  }
}
```

lässt, wird der Event kontinuierlich ausgelöst. Die erlaubten DragDrop-Effekte werden auf *None* gestellt, damit die Flächen nicht versehentlich zum Drop-Ziel werden.

Der Pferdefuß an der Sache ist allerdings, dass die TreeView von sich aus keine Scroll-Anforderungen von außen unterstützt. Der Ansatz hier ist daher etwas gebastelt und funktioniert auch nur mit dem Original-Template. Das beinhaltet nämlich einen ScrollViewer mit dem Namen *_tv_scrollviewer_*. Die Namensgebung lässt schon darauf schließen, dass es eher kein öffentlicher Name sein soll. Sonst wäre die Konvention *PART_einName*. Wer sichergehen will, ersetzt das Template durch eine Kopie und passt den Namen entsprechend an.

Führt man nun die Maus über die in Bild 1 gezeigten Steuerflächen, wird die TreeView die Ansicht kontinuierlich scrollen, bis das jeweilige Ende erreicht ist. Nach dem Drag-and-drop-Vorgang verschwinden die Schaltflächen wieder.



Schaltflächen als Hilfe, um in einer TreeView den vertikalen Bildlauf während des Drag-and-drop-Vorgangs auslösen zu können (Bild 1)

Was verschiebe ich denn da gerade?

Bedienungen mit der Maus können recht fummelig sein. Arbeitsplätze mit großen Bildschirmen mit hoher Auflösung haben schnell den Nebeneffekt, dass Mausbewegungen sehr unpräzise werden. Umso mehr ist es für den Anwender hilfreich, wenn er sehen kann, welchen Knoten er getroffen hat und soeben durch die Gegend schiebt.

Klassisch lässt sich das über das Symbol des Mauszeigers realisieren. Man könnte in WPF zwar auch versuchen, über Adorner eine Kopie des Quellobjekts in der Oberfläche zu positionieren, aber lassen Sie uns mal das aus alten Zeiten bekannte Vorgehen umsetzen. Das bedeutet, wir müssen das zu verschiebende Element erfassen, seine Darstellung kopieren und in einen Cursor umwandeln. Auch das gestaltet sich ungefähr genauso kompliziert, wie es klingt.

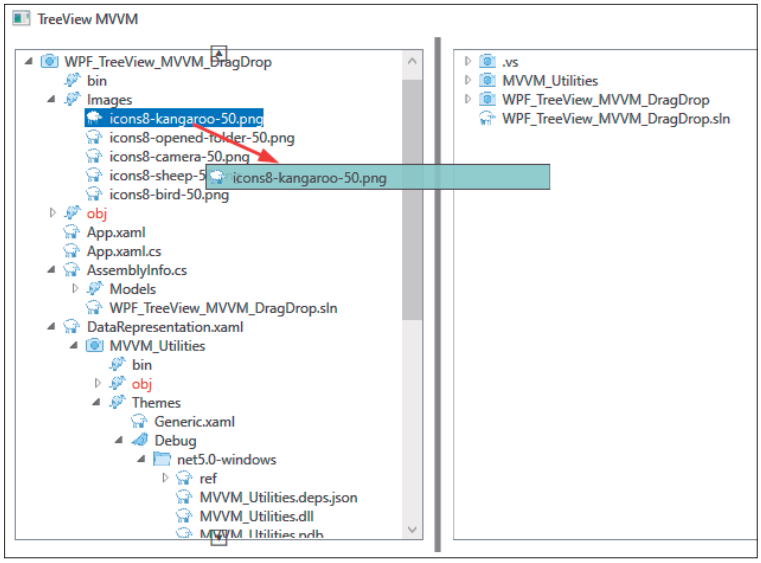
Einen Ansatz finden Sie zum Beispiel unter [3]. Der funktioniert allerdings in .NET 5 nicht mehr so ganz und muss etwas abgewandelt werden.

Listing 2 zeigt einen Möglichkeit, wie Sie ausgehend von einem WPF-Objekt einen Cursor generieren können.

Zunächst wird ein *ContentControl* angelegt, dessen *Content* auf das mit dem *TreeItem* verbundene Datenobjekt (das war im Beispiel eine Instanz von *DirectoryDataObject* beziehungsweise *FileDataObject*) gesetzt wird. Die Darstellung dieses ContentControls ergibt sich dann aus dem *DataTemplate* für den jeweiligen Datentyp. Die Werte für Breite und Höhe werden dem gezogenen *TreeViewItem* entnommen.

Hier kommt jetzt viel Fleißarbeit ins Spiel, will man die Darstellung ansprechend gestalten. Vielleicht eine Grafik in Form einer Sprechblase mit einem Pfeil, der auf die Mausposition verweist? Nun, beschränken wir uns hier auf die wesentliche Technik und überlassen den Designern die weitere Arbeit. Das *ContentControl* wird in einem *Border*-Control gekapselt, um die Darstellung etwas besser hervorheben zu können. Ein paar Farbeinstellungen, Rahmendicke, Rand und Skalierung sollen vorerst reichen. Über *border.Measure* und *border.Arrange* werden Position und Größe für die anschließende Umwandlung in ein Bitmap ermittelt. Auf ein paar Umwegen, die [3] näher beschreibt, wird dann letztlich das benötigte Cursor-Objekt generiert. Damit alle hierfür erforderlichen Datentypen verfügbar sind, muss noch das Paket *System.Drawing.Common* eingebunden werden.

Den Cursor muss man dann noch ein- und ausschalten, wie es aus Listing 3 hervorgeht. WPF schleift hierfür den *GiveFeedback*-Event durch. Das sollte keinerlei Einfluss auf die eigentliche Aktion nehmen, sondern lediglich dem Anwender ein visuelles Feedback geben. Dieser Event lässt sich ▶



Während des Ziehens wird eine visuelle Kopie des gezogenen Elements als Cursor angezeigt (Bild 2)

Listing 2: Mauszeiger während des Ziehens

```
public class CursorHelper
{
  ...
  // Erstellen eines Cursors für die Darstellung
  // während eines Drag-Vorgangs
  // <param name="dragElement"></param>
  public static Cursor CreateCursor(
      Control dragElement)
  {
    // Zur Demo: Border-Control um Kopie des Inhalts
    // des TreeViewItems
    ContentControl cc = new ContentControl();
    cc.Content = (
      dragElement.DataContext as TreeItem)?.Data;
    cc.Height = dragElement.ActualHeight;
    cc.Width = dragElement.ActualWidth;

    // Einstellungen für Rahmen
    Border border = new Border();
    border.Child = cc;
    border.BorderBrush = Brushes.Black;
    border.BorderThickness = new Thickness(1);
    border.Padding = new Thickness(2);
    border.Background = new SolidColorBrush(
      Color.FromArgb(230, 190, 230, 230));

    // Größe anpassen
    border.LayoutTransform =
      new ScaleTransform(0.8, 0.8);
    border.Visibility = Visibility.Visible;
    border.Measure(new Size(
      double.PositiveInfinity,
      double.PositiveInfinity));

    // Offset zum Mauszeiger, falls gewünscht
    var offset = new Point(0, 0);
    border.Arrange(new Rect(offset,
      border.DesiredSize));

    RenderTargetBitmap rtb = new RenderTargetBitmap(
      (int)border.DesiredSize.Width+(int)offset.X,
      (int)border.DesiredSize.Height+(int)offset.Y,
      96, 96, PixelFormats.Pbgra32);
    rtb.Render(border);

    var encoder = new PngBitmapEncoder();
    encoder.Frames.Add(BitmapFrame.Create(rtb));

    using (var ms = new MemoryStream()) {
      encoder.Save(ms);
      using (var bmp =
          new System.Drawing.Bitmap(ms))
      {
        return InternalCreateCursor(bmp);
      }
    }
  }
}
```

gut benutzen, um den Cursor umzuschalten. Nach Abschluss der Drag-and-drop-Operation wird der Cursor wieder zurückgesetzt. Während des Ziehens ergibt sich eine Darstellung wie in Bild 2 zu sehen. Wie bereits gesagt – hier kann man viel Fleißarbeit investieren, um das schöner zu machen.

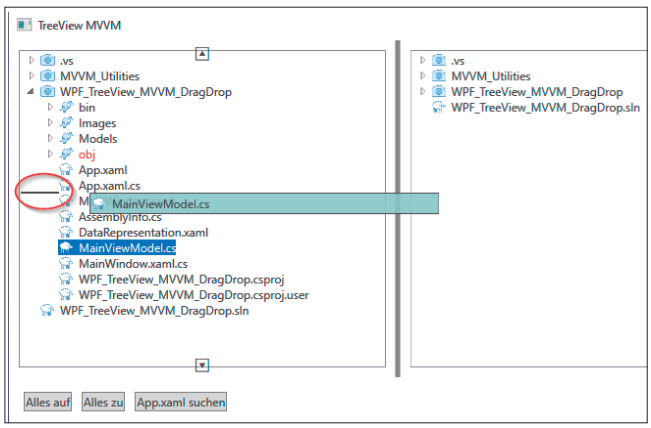
Davor, dahinter oder mittendrauf?

Die oben beschriebene Implementierung fügt das verschobene Element anhand der ermittelten relativen Position des Mauszeigers zum Zielobjekt ein. Für den Benutzer ist es aber schwierig zu erkennen, welche das sein wird. Noch ein Punkt also, bei dem ein visuelles Feedback hilfreich wäre.

Auch hierzu findet man im Internet passende Ansätze. Meist wird dafür ein Adorner eingesetzt, wie zum Beispiel in [4]. Adorner sind nichts anderes als WPF-Controls, die über einem anderen Control in einem spezifischen Layer platziert werden und in z-Richtung immer ganz oben liegen, also auf jeden Fall über dem zugehörigen Control sichtbar sind.

Diese Idee soll hier auch aufgegriffen werden. Wie in Bild 3 zu sehen ist, wird neben dem Zielknoten ein schwarzer Strich angezeigt, der die Stelle markiert, an der das Objekt eingefügt würde. Steht der Mauszeiger über der Mitte eines Knotens, wird stattdessen eine rote Ellipse angezeigt. Beides bietet wieder viel Verbesserungspotenzial bezüglich des Designs.

In Listing 4 ist die Implementierung der Klasse *Adorner* zu sehen. Die y-Koordinate wird bereits im Konstruktor durchgereicht, kann aber auch über die Methode *Update* ange-



Über Adorner lässt sich anzeigen, wo das gezogene Objekt abgelegt würde (Bild 3)

passt werden. Für die Zuordnung eines Adorners zu einem UI-Element wird ein *AdornerLayer* benötigt, der über die statische Methode *AdornerLayer.GetAdornerLayer* abgerufen wird. *SetPosition* gibt Prozentwerte für die drei Bereiche vor. So lässt sich in der Methode *OnRender* das gewünschte UI (Strich oder Ellipse oder doch etwas Schöneres) einfacher berechnen und erzeugen.

Remove entfernt über den betreffenden *AdornerLayer*, der beim Anlegen gespeichert wurde, das *Adorner*-Objekt. Alternativ könnte man auch für alle TreeViewItems, die mit der Maus überfahren wurden, einen Adorner anlegen und nur dessen Sichtbarkeit steuern, aber dann müsste man diese an anderer Stelle wieder löschen.

Das Ein- und Ausschalten der Adorner in den Eventhandlern zeigt Listing 5. In *TVI_DragEnter* wird ein Adorner angelegt, in *TVI_DragOver* dann kontinuierlich angepasst. Bei *TVI_DragLeave* verlässt der Mauszeiger das betreffende Objekt und der Adorner wird wieder entfernt. Ebenso muss er nach Abschluss der Drag-and-drop-Aktion entfernt werden.

Die beschriebene Umsetzung lässt sich aber lediglich als Ansatz verwerten und hat noch einen Haken: Führt man den Mauszeiger über den Adorner, wird der *DragLeave*-Event aufgerufen und der Code entfernt den Adorner daraufhin wieder. Im nächsten Moment wird dann, da der Cursor ja nun wieder über dem TreeViewItem steht, der DragEnter-Event erneut ausgelöst, und das Spiel beginnt von Neuem. Sie sollten sich also vor der Umsetzung einer solchen Visualisierung Gedanken machen, wie diese in das Event-Geschehen eingreift, und entweder den Adorner passend gestalten oder einen anderen Weg wählen, um die voraussichtliche Drop-Aktion zu visualisieren.

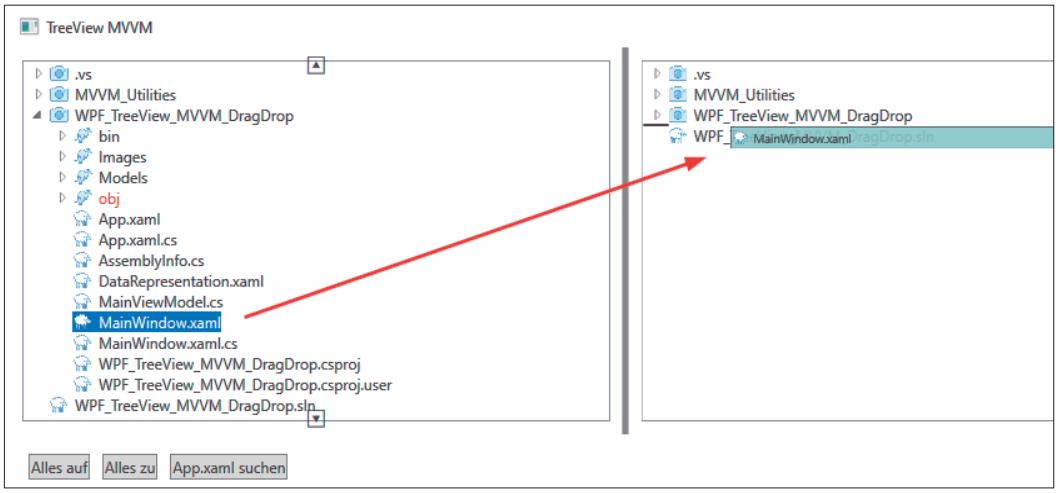
Drag-and-drop zwischen zwei Controls

Wie man einen Knoten innerhalb einer TreeView an eine andere Position schieben kann, haben Sie jetzt gesehen. Aber funktioniert das auch von einer TreeView zu einer anderen TreeView oder von beziehungsweise zu einem ganz anderen Steuerelement?

Wenn eine zweite TreeView ebenfalls über die beschriebenen Datenstrukturen versorgt wird, dann sollten auch die Drag-and-drop-Aktionen zwischen beiden bereits funktionieren (Bild 4). Beide Seiten verwenden dieselben Events ▶

Listing 3: Cursor-Steuerung

```
public partial class ExtendedTreeView : UserControl
{
  private Cursor customCursor = null;
  ...
  // sender ist das TreeViewItem,
  // das verschoben wird
  private void TVI_GiveFeedback(
      object sender, GiveFeedbackEventArgs e)
  {
    var tvi = sender as TreeViewItem;
    var ti = tvi.DataContext as TreeItem;
    if (e.Effects.HasFlag(DragDropEffects.Move))
    {
      if (customCursor == null)
        customCursor = CursorHelper.CreateCursor(
        e.Source as Control);
      if (customCursor != null)
      {
        e.UseDefaultCursors = false;
        Mouse.SetCursor(customCursor);
      }
    } else {
      e.UseDefaultCursors = true;
    }
    e.Handled = true;
  }
  private void TVI_Drop(object sender,
      DragEventArgs e)
  { ...
    customCursor = null;
  }
}
```



Auch das Ziehen von Einträgen zwischen zwei TreeViews ist möglich (Bild 4)

Listing 4: Adorner für visuelles Feedback

```
class TreeItemDragOverAdorner : Adorner
{
  private AdornerLayer adornerLayer;
  private double yPercentage = 0;
  public TreeItemDragOverAdorner(
      UIElement adornedElement,
      double yPercentage) : base(adornedElement)
  {
    SetPosition(yPercentage);
    this.adornerLayer =
      AdornerLayer.GetAdornerLayer(
      this.AdornedElement);
    this.adornerLayer.Add(this);
  }

  internal void Update(double yPercentage)
  {
    // Adorner neu positionieren und zeichnen
    SetPosition(yPercentage);
    Debug.WriteLine($"Adorner update {yPercentage}%");
    this.adornerLayer.Update(this.AdornedElement);
    this.Visibility =
      System.Windows.Visibility.Visible;
  }

  private void SetPosition(double yPercentage)
  {
    // y-Position einschränken auf 0%, 50% oder 100%
    if (yPercentage < 20)
      this.yPercentage = 0;
    else if (yPercentage > 80)
      this.yPercentage = 100;
    else this.yPercentage = 50;
  }

  public void Remove() {
    adornerLayer.Remove(this);
  }

  // Veranschaulichen, was bei einem Drop
  // passieren würde
  protected override void OnRender(
    DrawingContext drawingContext) {
    Rect adornedElementRect = new Rect(
      this.AdornedElement.DesiredSize);
    // Farbgebung, Größe...
    SolidColorBrush renderBrush =
      new SolidColorBrush(Colors.Red);
    renderBrush.Opacity = 0.5;
    Pen renderPen = new Pen(
      new SolidColorBrush(Colors.White), 1.5);
    double renderRadius = 5.0;
    // Position
    double y =
      adornedElementRect.Height * yPercentage / 100;
    if (yPercentage == 50)
    {
      // Gezogenes Element würde Unterelement werden
      drawingContext.DrawEllipse(renderBrush,
        renderPen, new Point(0,y), renderRadius,
        renderRadius);
    } else {
      // Gezogenes Element würde davor oder dahinter
      // geschoben werden
      Pen pen = new Pen(Brushes.Black, 2);
      Point p1 = new Point(-50, y);
      Point p2 = new Point(0, y);
      drawingContext.DrawLine(pen, p1, p2);
    }
  }
}
```

und dieselben Hilfsklassen wie *TreeItem, DragDropController* et cetera. Im ViewModel lässt sich das gewünschte Verhalten leicht implementieren.

Wird eine Interaktion mit anderen Controls erforderlich, dann muss der Code an einigen Stellen angepasst werden. Für das gezogene Objekt wurde aus diesem Grund bei den Delegates des *DragDropControllers* der Typ *Object* verwendet. Man könnte also prinzipiell etwas Beliebiges auf die TreeView-Knoten ziehen, müsste das aber im ViewModel passend handhaben. Auch das Ziehen eines TreeViewItems auf ein anderes Control lässt sich umsetzen. Einige der Events werden dann von diesem Control empfangen und müssen dort bearbeitet werden. Nach Abschluss der Aktion muss die Darstellung in der TreeView gegebenenfalls angepasst werden. Gleiches gilt auch, wenn Quelle oder Ziel der Drag-and-drop-Operation außerhalb der Anwendung liegen. Also zum Beispiel, wenn ein markierter Text aus einem Word-Dokument auf einen TreeView-Knoten gezogen wird oder ein TreeView-Knoten auf ein Browser-Fenster. In diesen Fällen kommt erschwerend hinzu, dass man auf die spezifischen Datenstrukturen Rücksicht nehmen muss, die in der anderen Anwendung zum Einsatz kommen.

Fazit

Die verwendeten Techniken, um Drag-and-drop für eine TreeView MVVM-konform zu implementieren, sind überschaubar. Wie der Artikel gezeigt hat, muss man sich allerdings mit vielen, meist historisch bedingten Kuriositäten und Umwegen auseinandersetzen. Und spätestens für ein ansprechendes, benutzerfreundliches visuelles Feedback ist sehr

Listing 5: Adorner anzeigen und entfernen

```
public partial class ExtendedTreeView : UserControl
{
  private TreeItemDragOverAdorner adorner;
  ...
  private void TVI_DragEnter(object sender,
      DragEventArgs e)
  {
    if (DragDropController?.CanDrop == null) return;
    var tvi = sender as TreeViewItem;
    var ti = tvi.DataContext as TreeItem;
    if (this.adorner == null)
    {
      var mousePos = e.GetPosition(tvi);
      var header = tvi.Template.FindName(
        "PART_Header", tvi) as FrameworkElement;
      var heightHeader =
        header?.ActualHeight ?? tvi.ActualHeight;
      var draggedTreeItem =
        e.Data.GetData(typeof(TreeItem));
      var yPercentage =
        mousePos.Y * 100 / heightHeader;
      this.adorner = new TreeItemDragOverAdorner(
        header, yPercentage);
    }
    ...
  }

  private void TVI_Drop(object sender,
      DragEventArgs e)
  {
    ...
    this.adorner?.Remove();
    adorner = null;
    ...
  }

  private void TVI_DragOver(object sender,
      DragEventArgs e)
  {
    ...
    if (this.adorner != null)
      this.adorner.Update(yPercentage);
    e.Handled = true;
  }

  private void TVI_DragLeave(object sender,
      DragEventArgs e)
  {
    this.adorner?.Remove();
    adorner = null;
    e.Handled = true;
  }
}
```

viel Fleißarbeit vonnöten. Wer die nicht investieren will, kann aber auf Alternativen zurückgreifen. Viele Drittthersteller haben bereits komplexe und anspruchsvolle Steuerelemente für WPF im Angebot. Die Gedanken zur MVVM-Anbindung aus dem ersten Teil des Artikels lassen sich oft auch auf solche Controls übertragen. Visuelle Feedbacks für Drag-and-drop bringen die Toolboxen teilweise bereits mit, sodass einem diese Arbeit zumindest teilweise erspart bleibt.

Wichtig ist aber vor allem, dass man sehr genau plant, wie Drag-and-drop-Operationen eingesetzt werden sollen, was von wo nach wo gezogen werden kann und welche Auswirkungen das im Einzelfall haben soll. Hierfür gibt es keine allgemeingültige Festlegung. Das Verhalten ist immer anwendungsspezifisch und führt somit auch zu einem weiteren Punkt, der berücksichtigt werden muss: Drag-and-drop ist nicht intuitiv! Zwar kennen die meisten Anwender die Vorgehensweise, aber dass sich irgendein Element der Oberfläche auf ein anderes ziehen lässt, ist erst einmal nicht ersichtlich. Das muss in geeigneter Form dokumentiert und dem Anwender vermittelt werden. Werden Drag-and-drop-Gesten noch um optionale Tastendrücke ergänzt (STRG, ALT und so weiter), um das Verhalten während des Zieh-Vorgangs beeinflussen zu können, wird es noch wichtiger, dem Anwender die Verfügbarkeit dieser Optionen bewusst zu machen. Abgesehen davon sollte es zumindest noch eine andere Vorgehensweise neben Drag-and-drop geben, um die gewünschte Aktion auszulösen. Nicht jeder Benutzer kann sich mit mauslastigen Bedienungen anfreunden. ■

[1] Joachim Fuchs, WPF-TreeView und MVVM in Einklang bringen, Teil 1 , dotnetpro 7/2021, Seite 8 ff., www.dotnetpro.de/A2107TreeViewMVVM

[2] Joachim Fuchs, WPF-TreeView und MVVM in Einklang bringen, Teil 2 , dotnetpro 8/2021, Seite 68 ff., www.dotnetpro.de/A2108TreeViewMVVM

[3] Cursor für Drag-and-drop in WPF, www.dotnetpro.de/SL2108TreeViewMVVM2

[4] Adorner für Drag-and-drop, www.dotnetpro.de/SL2108TreeViewMVVM3

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 A2109TreeViewMVVM