WPF-TREEVIEW UND MVVM IN EINKLANG BRINGEN, TEIL 2

Drücken, ziehen, loslassen

Ereignisbasierte Aktionen, antiquierte Windows-Techniken und MVVM kombinieren.

Der erste Teil [1] dieser Artikelserie zeigte Ihnen, wie Sie mit einigen trickreichen WPF-Techniken und etwas C#-Infrastruktur auch TreeView-Controls sehr elegant in das MVVM-Konzept einbinden können. Geht es jedoch um Techniken wie Drag-and-drop, dann wird es etwas kniffliger. Denn hier kommt man um die Implementierung zahlreicher Eventhandler nicht herum. In Bezug auf MVVM ist das kein Problem, solange man den Zugriffscode kapseln kann und im ViewModel keine Eventhandler oder Ähnliches auswerten muss. Dazu bieten sich mehrere Vorgehensweisen an. Man könnte ein Behavior (Ableitung von *System.Windows.Forms.Design.Behavior<T>*) schreiben, das man im XAML-Code einer gewöhnlichen TreeView zuordnet und das dann mit den betreffenden Eventhandlern umgehen kann. Alternativ nutzt man den Vererbungsmechanismus und leitet eine neue Klasse von *TreeView* ab. Oder man verwendet Aggregation und setzt ein neues Steuerelement aus einer TreeView und weiteren erforderlichen Komponenten zusammen. Und auch hier gibt es verschiedene Möglichkeiten, das zu realisieren.

Für die Umsetzung in diesem Artikel fiel die Wahl auf Aggregation in Form eines UserControls. Die Vorbereitungen hierzu wurden bereits im vorangegangenen Artikel getroffen. Der Vorteil des UserControls ist in diesem Fall, dass es den Entwicklungsfluss in Visual Studio besser unterstützt. Experimente, Tests und Fehlersuche gestalten sich damit einfacher, und man kann sich auf das Wesentliche beschränken. Später lässt sich der Code mit etwas Fleißarbeit in eine der anderen genannten Varianten transformieren.

Drag-and-drop

Die Technik ist den meisten Anwendern geläufig: Man zieht mit der Maus oder bei Touch-Bedienung mit dem Finger ein

Listing 1: Events und Properties verknüpfen

```
<UserControl x:Class=
    "MVVM_Utilities.ExtendedTreeView" ...>
  <Grid >
    <!--AllowDrop muss für TreeView eingeschaltet
      sein, sonst geht Drag-and-drop nicht -->
    <!--Implementieren von DragOver blockiert ein
      Drop auf die TreeView selbst-->
    <TreeView Margin="5,5,5,5"  AllowDrop="True"
      ScrollViewer.CanContentScroll="True"
      ScrollViewer.HorizontalScrollBarVisibility=
      "Auto" ScrollViewer.VerticalScrollBarVisibility=
      "Auto" Name="_tv_"  DragOver="TV_DragOver" >

      <TreeView.ItemContainerStyle>
        <Style TargetType="TreeViewItem">
          <!--Verknüpfung der Eigenschaften und Events
            eines TreeViewItems-->
          <EventSetter Event="PreviewMouseDown"
            Handler="TVI_PreviewMouseDown"/>
          <EventSetter Event="MouseMove" Handler=
            "TVI_MouseMove"/>
          <EventSetter Event="GiveFeedback" Handler=
            "TVI_GiveFeedback"/>
          <EventSetter Event="DragEnter" Handler=
            "TVI_DragEnter"/>
          <EventSetter Event="PreviewDragEnter"
            Handler="TVI_PreviewDragEnter"/>
          <EventSetter Event="Drop" Handler=
            "TVI_Drop"/>
          <EventSetter Event="DragOver" Handler=
            "TVI_DragOver"/>
          <EventSetter Event="DragLeave" Handler=
            "TVI_DragLeave"/>
          <EventSetter Event="Selected" Handler=
            "TVI_Selected"/>
          <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>
    ...
  </Grid>
</UserControl>
```

Element von einem Punkt zu einem anderen und will damit eine bestimmte Aktion anstoßen, zum Beispiel Verschieben oder Kopieren. Während der Aktion erwartet der Benutzer ein visuelles Feedback. Das beginnt mit dem Markieren des zu ziehenden Elements, einer visuellen Repräsentation während des Ziehens (Mauszeigersymbole, Abbild des Ausgangsobjekts), dem Anzeigen von erlaubten oder auch verbotenen Drop-Möglichkeiten und endet eventuell mit einer grafischen Rückmeldung beim Abschluss der Aktion, also beim Fallenlassen.

Das klingt recht kompliziert und nach sehr viel Aufwand. Auch wenn die Technik teilweise im Betriebssystem inte- ▶

Listing 2: Die Basis für Drag-and-drop im UserControl

```
public partial class ExtendedTreeView : UserControl {
  private bool isDragging = false;
  private Point startposition;
  private TreeViewItem treeViewItemToDrag;
  ...
  // MouseDown auf das zu verschiebende TreeViewItem
  private void TVI_PreviewMouseDown(
      object sender, MouseButtonEventArgs e)
  {
    // Informationen für Drag-Operation speichern
    startposition = e.GetPosition(this);
    treeViewItemToDrag = sender as TreeViewItem;
    isDragging = false;
  }

  // MouseMove über einem TreeViewItem
  private void TVI_MouseMove(
      object sender, MouseEventArgs e)
  {
    if (DragDropController?.CanDrag == null) return;
    TreeViewItem tvi = sender as TreeViewItem;
    // Ist die Maus auf ein anderes Item gerutscht?
    if (tvi != treeViewItemToDrag) return;

    isDragging = e.GetPosition(this) != startposition;
    var ti = tvi.DataContext as TreeItem;

    // Rahmenbedingungen für DragDrop-Start prüfen
    if (e.LeftButton == MouseButtonState.Pressed &&
        isDragging && DragDropController.CanDrag(ti))
    {
      // DoDragDrop kehrt erst nach Abschluss der
      // Aktion wieder zurück
      DragDrop.DoDragDrop(
        tvi, ti, DragDropEffects.All);
    }
  }

  // DragOver einem TreeViewItem, das
  // Drop-Target sein kann
  private void TVI_DragOver(object sender,
      DragEventArgs e)
  {
    if (DragDropController?.CanDrop == null) return;
    var tvi = sender as TreeViewItem;
    var ti = tvi.DataContext as TreeItem;
    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;
    // ViewModel nach Gültigkeit für Drop fragen
    e.Effects = DragDropController.CanDrop(
      draggedTreeItem, ti, yPercentage);
    ...
    e.Handled = true;
  }

  // Kein Drop auf TreeView selbst zuzulassen
  private void TV_DragOver(object sender,
      DragEventArgs e)
  {
    e.Effects = DragDropEffects.None;
    e.Handled = true;
  }
  // Drag-and-drop-Operation abgeschlossen
  private void TVI_Drop(object sender,
      DragEventArgs e)
  {
    if (DragDropController?.Drop == null) return;
    var tvi = sender as TreeViewItem;
    var ti = tvi.DataContext as TreeItem;
    // Relative Position in y-Richtung ermitteln
    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));
    // ViewModel über Abschluss benachrichtigen
    DragDropController.Drop(draggedTreeItem, ti,
      mousePos.Y * 100 / heightHeader);
    e.Handled = true;
  }
  ...
}
```

griert ist, lautet die Antwort in diesem Fall leider: „Ja, das ist es auch." Eine Reihe von Events sind zu berücksichtigen, und die visuellen Feedbacks bekommt man in WPF leider auch nicht geschenkt, zumindest nicht von den mitgelieferten Steuerelementen. Zunächst gilt es die Events zu betrachten, um die es geht, wenn man innerhalb der TreeView einen Knoten (*TreeViewItem*) verschieben möchte.

Große Ereignisse ...

Bereits der Start einer Drag-and-drop-Aktion muss vom Code der Anwendung selbst initiiert werden. In [2] hat Microsoft ein Beispiel bereitgestellt, das zur Orientierung dienen kann. Allerdings werden dort nicht alle Umstände berücksichtigt, sodass einige zusätzliche Events zu behandeln sind. Im Wesentlichen sind dies (für ein *TreeViewItem*):

- *PreviewMouseDown*, um mitzubekommen, wenn der Benutzer möglicherweise einen Drag-and-drop-Vorgang beginnen möchte.
- *MouseMove*, um auszuwerten, ob die Geste zu Drag-and-drop führen soll und erlaubt ist.
- *GiveFeedback*, für visuelles Feedback während des Ziehvorgangs.
- *PreviewDragEnter*, *DragEnter*, um festzustellen, ob ein Drop erlaubt wäre, und Feedback hierzu.
- *DragOver*, Feedback zu den Folgen eines Fallenlassens (zum Beispiel Darstellung, an welche Position das Element verschoben würde).
- *DragLeave*, Entfernen zuvor gegebener visueller Feedbacks.
- *Drop*, Ausführen der gewünschten Aktion

Die genannten Events sind für jedes von der TreeView automatisch generierte *TreeViewItem* zu berücksichtigen. Wie schon im ersten Teil am Beispiel der Eigenschaften *IsSelected* oder *IsExpanded* erreicht man auch diese Events über einen Style, den man der Eigenschaft *ItemContainerStyle* der TreeView zuweist (Listing 1). Statt der *Setter*-Elemente werden für die Bindung der Eventhandler Elemente vom Typ *EventSetter* benötigt. Das Implementieren der Handler erfolgt im Code-behind des UserControls und soll im Folgenden schrittweise besprochen werden.

Erster Ansatz

In Listing 2 ist die Implementierung der wichtigsten Eventhandler zu sehen. Bereits bei *PreviewMouseDown* (der *MouseDown*-Event wird leider nicht auf dem *TreeViewItem* gefeuert) sollten Mausposition sowie das Element unter dem Mauszeiger für die weitere Auswertung gespeichert werden. In *MouseMove* sind dann einige Bedingungen zu prüfen, bevor der Drag-Vorgang eingeleitet werden kann. Ein Rückrufmechanismus (*DragDropController*, siehe unten) stellt die Verbindung zum ViewModel her, das aufgrund einer anwendungsspezifischen Logik entscheiden muss, ob ein Drag-Vorgang zulässig wäre. Die Geste, die für den Start der Drag-and-drop-Operation notwendig ist, setzt voraus, dass sich gegenüber *PreviewMouseDown* einerseits die Position des Mauszeigers verändert hat, andererseits aber immer noch

Listing 3: Typdefinition DragDropController

```
// Ermitteln, ob ein TreeItem gezogen werden darf
// dragSource ist das zu ziehende TreeItem
// <returns>true, wenn ja</returns>
public delegate bool CanDragHandler(
  TreeItem dragSource);

// Ermitteln, ob ein Objekt auf einem TreeItem
// abgelegt werden darf. draggedItem ist das
// gezogene Objekt. dropTarged ist das TreeItem
// unter der Maus. yPosition: Prozentwert der
// Mausposition in Bezug auf das TreeItem in
// Parameter 2. 0: oben, 100: unten.
// <returns>Erlaubte Drag&Drop-Effekte</returns>
public delegate DragDropEffects CanDropHandler(
  object draggedItem, TreeItem dropTarged,
  double yPosition);

// <returns>Aktion war erfolgreich</returns>
public delegate bool DropHandler(object draggedItem,
  TreeItem dropTarget, double yPosition);

// Hilfsstruktur zur Bindung von Rückrufmethoden
public class DragDropController {
  // Prüfen, ob ein Element gezogen werden darf
  public CanDragHandler CanDrag { get; set; }
  // Prüfen, ob ein Element abgelegt werden darf
  public CanDropHandler CanDrop { get; set; }
  // Schlussbehandlung der Drag-and-drop-Aktion
  public Func<object, TreeItem, double, bool> Drop
    { get; set; }
}
```

dasselbe Element unter dem Mauszeiger ist, sodass nicht versehentlich ein benachbartes Objekt gezogen wird.

Sind die Voraussetzungen erfüllt, kann der Vorgang durch den Aufruf der statischen Methode *DragDrop.DoDragDrop* initiiert werden. Die Methode erwartet als Parameter das zu ziehende Element (hier das *TreeViewItem*), ein beliebiges Datenobjekt (hier das zugehörige *TreeItem*) sowie die Angabe aller erlaubten Effekte (Enumeration *DragDropEffects*). *DoDragDrop* kehrt erst nach Abschluss der Operation zurück.

Während des Ziehens funktionieren die meisten Mausereignisse (*MouseXXX*) nicht mehr. Jetzt kommen nur noch die *Drag*-Events zum Tragen. So sollte man im *DragOver*-Event ermitteln, ob ein Loslassen (*Drop*) an der aktuellen Stelle zulässig ist und welche Konsequenzen es hätte. Im vorliegenden Beispiel soll es die Möglichkeit geben, das gezogene Element vor oder hinter einem anderen zu platzieren oder es dem anderen als Unterelement zuzuordnen. Wesentlich für das Ergebnis ist daher auch die Mausposition relativ zum jeweiligen Drop-Kandidaten. Ist die Maus im oberen Bereich, soll das Element davor angeordnet werden, ist sie im hinteren

Listing 4: DragDropController-Methoden im ViewModel

```
public class MainViewModel : NotificationObject {
  ...
  public DragDropController DragDropControllerTV1
    { get; set; }
  public MainViewModel() {
    ...
    DragDropControllerTV1 = new DragDropController() {
      CanDrag = CanDragTV1, CanDrop=CanDropTV1,
      Drop=DropTV1 };
  }

  private bool CanDragTV1(TreeItem itemToDrag) {
    var data = itemToDrag.Data as DataObjectBase;
    if (data == null) return false;

    // anwendungsspezifische Kriterien prüfen,
    // Rückgabe von true, wenn Drag erlaubt ist
    var startsWithPoint =
      data.Caption.StartsWith(".");
    return !startsWithPoint;
  }

  private DragDropEffects CanDropTV1(
      object draggedObject, TreeItem dropTarget,
      double verticalPercentage)
  {
    // Sicherstellen, dass ein Item nicht auf sich
    // selbst oder ein Unterelement gezogen wird
    if (dropTarget.IsAncestorOfOrEqualTo(
      draggedObject)) return DragDropEffects.None;

    // Geht es um eines unserer Datenobjekte?
    var data = dropTarget.Data as DataObjectBase;
    if (data == null) return DragDropEffects.None;
    if (!(draggedObject is TreeItem))
      return DragDropEffects.None;

    // anwendungsspezifische Kriterien prüfen,
    // Rückgabe der erlaubten Effekte
    return data.Caption.StartsWith(".") ?
    DragDropEffects.None : DragDropEffects.Move;
  }
  private bool DropTV1(object draggedObject,
      TreeItem dropTarget, double verticalPercentage)
  {
    var source = draggedObject as TreeItem;
    if (source == null) return false;

    // Sicherstellen, dass das gezogene Item nicht auf
    // sich selbst oder ein Unterelement gezogen wird
    if (dropTarget.IsAncestorOfOrEqualTo(
      draggedObject)) return false;

    // Anwendungsspezifische Aktionen
    source.Parent.Items.Remove(source);
    if (verticalPercentage < 20) {
      // davor einfügen
      var parentList = dropTarget.Parent.Items;
      var pos = parentList.IndexOf(dropTarget);
      parentList.Insert(pos, source);
    }
    else if (verticalPercentage > 80) {
      // dahinter einfügen
      var parentList = dropTarget.Parent.Items;
      var pos = parentList.IndexOf(dropTarget);
      parentList.Insert(pos + 1, source);
    } else {
      // als Unterelement einfügen
      dropTarget.Items.Add(source);
    }
    return true;
  }
  ...
}
```

Bereich, soll das Element dahinter eingeordnet werden oder, falls die Maus im mittleren Bereich steht, soll das Element in die Liste der Kind-Elemente übernommen werden.

Die Entscheidung, was später tatsächlich als Aktion geschehen soll, liegt beim ViewModel. Hier muss man aber unbedingt berücksichtigen, dass das ViewModel mit Mauspositionen nicht viel anfangen kann, da es ja keine Kenntnis der Benutzeroberfläche hat. Daher werden im Eventhandler die Koordinaten in Relation zu Größe und Position des TreeViewItems, speziell des Teils mit dem Namen *PART_Header*, berechnet. Für die Beispielimplementierung ist nur der y-Anteil relevant. Er wird auf den Wertebereich von 0 (oben) bis 100 (unten) skaliert. Der Wert wird neben der Angabe, was gezogen wird und welches potenzielle Ziel sich unter der Maus befindet, über den *DragDropController* an das ViewModel übergeben. Dieses gibt die gemäß der Anwendungslogik erlaubten Effekte zurück, die dann direkt für das Feedback verwendet werden.

An dieser Stelle ist noch ein Umstand zu berücksichtigen. Drag-and-drop funktioniert für die TreeViewItems nur, wenn in der TreeView *AllowDrop* auf *true* gesetzt wird (vergleiche Listing 1). Dadurch wird die gesamte TreeView zum potenziellen Ziel von Drop-Aktionen. Will man das verhindern und nur TreeViewItems als Ziel zulassen, muss man zusätzlich den *DragOver*-Event der TreeView implementieren und dort *e.Effects* auf *DragDropEffects.None* setzen, siehe Listing 2.

Der *Drop*-Event wird gefeuert, wenn der Benutzer das gezogene Objekt auf einem (erlaubten) Ziel abgelegt hat. ▶

Listing 5: Knoten automatisch aufklappen

```
public partial class ExtendedTreeView : UserControl {
  ...
  private DispatcherTimer delayTimer =
    new DispatcherTimer();
  private TreeItem treeItemToExpandAfterDelay;

  public ExtendedTreeView() {
    delayTimer.Interval =
      TimeSpan.FromMilliseconds(500);
    delayTimer.Tick += DelayTimer_Tick;
    InitializeComponent();
  }

  // Nach Ablauf der eingestellten Zeit soll das
  // TreeViewItem unter dem Mauszeiger automatisch
  // aufgeklappt werden
  private void DelayTimer_Tick(object sender,
       EventArgs e)
  {
    delayTimer.Stop();
    if (treeItemToExpandAfterDelay != null)
      treeItemToExpandAfterDelay.IsExpanded = true;
  }
  private void TVI_PreviewDragEnter(object sender,
      DragEventArgs e)
  {
    var tvi = sender as TreeViewItem;
    var ti = tvi.DataContext as TreeItem;
    // vorbereiten auf automatisches Aufklappen
    if (ti != treeItemToExpandAfterDelay)
    {
      delayTimer.Stop();
      treeItemToExpandAfterDelay = ti;
      delayTimer.Start();
    }
  }
  ...
}
```

Hier werden noch einmal, wie bei *DragOver*, die benötigten Parameter ermittelt (Quellobjekt, Zielobjekt, y-Position) und via *DragDropController.Drop* an das ViewModel übergeben. Dieses ist dann für die anwendungsspezifische Umsetzung der Aktion zuständig.

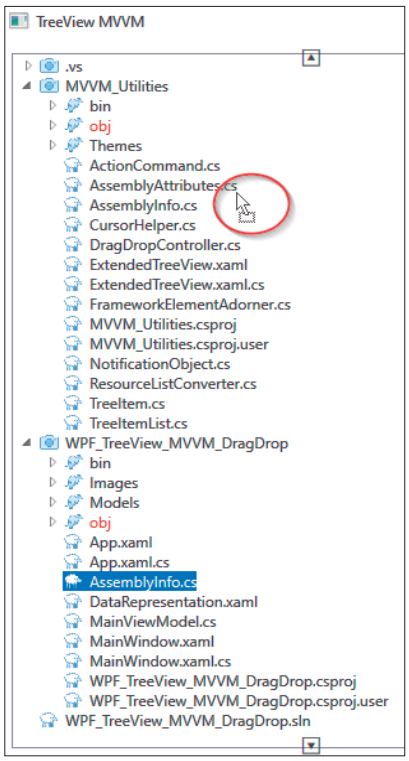
DragDropController (Listing 3) ist lediglich eine kleine Hilfsstruktur, die drei Eigenschaften aufweist (*CanDrag*, *CanDrop* und *Drop*). Alle drei sind Delegat-Typen. Die *delegate*-Deklarationen wurden explizit vorgenommen, damit die Parameter besser beschrieben werden können. Alternativ könnte man stattdessen auch den Typ *Func<...>* einsetzen.

Im ViewModel wird *DragDropController* instanziert, die Delegates werden mit lokalen Methoden verknüpft und das Objekt wird über eine Property nach außen bereitgestellt (Listing 4). Das Einbinden in den XAML-Code erfolgt in dieser Form:

```
<mvvm:ExtendedTreeView Items=
  "{Binding Tree1}"
  DragDropController=
  "{Binding DragDropControllerTV1}"
/>
```

Das ViewModel wertet hier zur Demo aus, ob der Text des zu ziehenden Elements mit einem Punkt beginnt, und lehnt in diesem Fall das Ziehen ab. Analog dazu wird in *CanDropTV1* das Ablegen abgelehnt, wenn der Text mit einem Punkt beginnt oder falsche Datenobjekte gezogen werden. Zudem muss in diesem Beispiel verhindert werden, dass ein Element auf sich selbst oder auf einen seiner Nachfahren gezogen wird. Eine solche Aktion würde hier keinen Sinn ergeben. Die Prüfung übernimmt die Methode *IsAncestorOfOrEqualTo* der Hilfsklasse *TreeItem*:

```
// Hilfsmethode für TreeItem
public class TreeItem :
    NotificationObject
{
  // Prüfen, ob ein übergebenes
  // Objekt mit dem aktuellen
  // TreeItem identisch oder ein
  // Nachfolger davon ist
  public bool IsAncestorOfOrEqualTo(
    object other)
  { // Identität?
    if (other == this) return true;
    // Parent vorhanden?
    if (this.Parent == null)
      return false;
    // Rekursiv nach oben prüfen
    return this.Parent.
      IsAncestorOfOrEqualTo(other);
  }
  ...
}
```



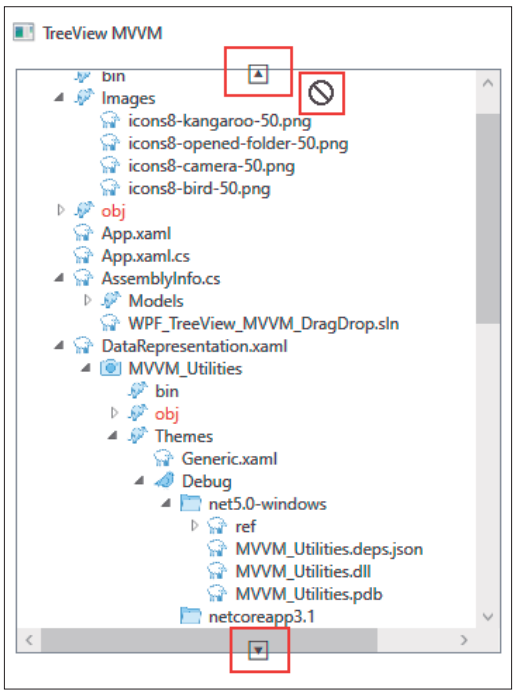
Basisausstattung für Drag-and-drop mit Standard-Mauszeiger (Bild 1)

Die Abfrage wurde absichtlich ins ViewModel verlegt, da die Aktion in einem

anderen Kontext vielleicht sinnvoll sein könnte und daher nicht grundsätzlich im Eventhandler der TreeViewItems bereits ausgeschlossen werden sollte.

Dass *CanDrop* einen Enumerationswert vom Typ *DragDropEffects* zurückgibt, ist hier ein pragmatischer Ansatz. Eigentlich passt das nicht zum MVVM-Pattern, da der Typ zum UI gehört. Die Alternative wäre, hier eine eigene Enumeration zu verwenden und diese an anderer Stelle in den Typ *DragDropEffects* umzuwandeln. Da die Enums aber keinerlei Eigenleben haben, tut der Kompromiss hier nicht weh. Der eine oder andere Leser wird dem aber vielleicht nicht zustimmen und auch hier eine saubere Trennung vorsehen wollen. *DropTV1* führt nach Plausibilitätsprüfungen die eigentliche Aktion aus. Hier wird die zuvor berechnete relative Mausposition ausgewertet, um das Quellobjekt vor oder nach dem Zielobjekt einzuordnen oder es in die Liste von dessen Unterelementen zu verschieben. Aus der ursprünglichen Liste wird es zuvor entfernt.

Nun, da der erste Ansatz steht, ist es an der Zeit, das Beispiel auszuprobieren. Das Ergebnis zeigt Bild 1. Ein Element kann via Drag-and-drop verschoben werden. Während des Ziehens wird der entsprechende Mauszeiger dargestellt, sodass ersichtlich ist, ob ein Fallenlassen zulässig ist oder nicht.



Schaltflächen als Hilfe, um in einer TreeView den vertikalen Bildlauf während des Drag-and-drop-Vorgangs auslösen zu können (Bild 2)

Klapp mich auf!

Dass die visuellen Feedbacks noch nicht dem entsprechen, was man von modernen Windows-Anwendungen erwartet, liegt auf der Hand. Hierzu wurde ja noch nichts implementiert. Aber es gibt noch ein anderes Bedienungsproblem: Wenn der Benutzer das Ziehen eines Knotens begonnen hat, diesen aber auf einen Knoten ablegen will, der noch gar nicht geöffnet ist, kommt schnell Frust auf. Er muss den Vorgang beenden, die gewünschten Zweige öffnen und dann die Aktion erneut durchführen. Wünschenswert wäre es daher, dass ein Knoten automatisch aufklappt, sobald der Mauszeiger eine bestimmte Zeit darüber verweilt.

Listing 5 zeigt eine mögliche Umsetzung. Im Ereignis *PreviewDragEnter* wird gespeichert, über welchem Element der Mauszeiger steht, und ein Timer gestartet. Nach Ablauf des Timers wird dieses Element gegebenenfalls aufgeklappt und der Timer wieder gestoppt. Die Verzögerungszeit wurde hier auf eine halbe Sekunde eingestellt.

Scroll me up, Scotty

Sind viele Zweige einer größeren Struktur aufgeklappt, stellt sich das nächste Problem ein: Wie soll man ein Element auf ein anderes ziehen, das sich gerade nicht im sichtbaren Bereich der TreeView befindet? Unter Umständen ist dieser Bereich auch zu klein, um Quell- und Zielelement gleichzeitig darstellen zu können. Eine automatische Scroll-Funktion wäre schön.

Jetzt beginnt es allmählich, kompliziert zu werden. Denn nun muss man festlegen, wann gescrollt werden soll. Jeder kennt bestimmt Situationen, in denen bei einer komplexeren Ansicht auf einmal ein Automatismus erwacht und den gerade betrachteten Inhalt aus dem Sichtfeld verschiebt – egal ob man das wollte oder nicht.

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 (Bild 2). Als Steuerflächen werden hierfür einfach *ContentControls* angelegt und in einem Grid über der TreeView positioniert. *DragOver* ist der Event, der hier zu berücksichtigen ist.

Vor der Einleitung und nach Abschluss des Drag-Vorgangs wird die Sichtbarkeit der Steuerflächen umgeschaltet. 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 belässt, wird der Event kontinuierlich ausgelöst. Die erlaubten Drag-and-Drop-Effekte werden auf *None* gestellt, damit die Flächen nicht versehentlich zum Drop-Ziel werden.

Alle Details sowie den Code zu den Scroll-Hilfen gibt's in der nächsten Folge dieser Serie. Außerdem wird geklärt, wo genau das Drag-Item einzufügen ist. ■

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

[2] Drag-and-drop WPF, www.dotnetpro.de/SL2108TreeViewMVVM1

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 A2108TreeViewMVVM