LISTBOX UND DATAGRID

WPF-Trick für MVVM-Apps

Wie Sie Mehrfachauswahlen in WPF-MVVM-Anwendungen handhaben.

Ein Klassiker in den FAQ zum Model-View-ViewModel-Pattern (MVVM) in der Windows Presentation Foundation (WPF) ist, wie man mit Auflistungen umgeht, die eine Mehrfachauswahl zulassen. Steuerelemente wie ListBox oder DataGrid unterstützen das zwar im User Interface (UI), bieten aber keine geeignete Schnittstelle an, um über eine einfache Datenbindung an die Auswahl zu gelangen. Dieser Beitrag zeigt eine mögliche Vorgehensweise.

Steuerelemente für Auflistungen in der WPF wie beispielsweise DataGrid oder ListBox verfügen über eine Eigenschaft *SelectedItem*, die auf das aktuell ausgewählte Element verweist. Diese Eigenschaft lässt sich an eine Property im ViewModel binden, sodass deren Setter immer dann aufgerufen wird, wenn sich die Auswahl geändert hat. Das geht auch in der Gegenrichtung: Durch Setzen der Eigenschaft *SelectedItem* kann man dem Steuerelement vorgeben, welcher Eintrag zu markieren ist. Wie der Name *SelectedItem* schon vermuten lässt, geht es aber immer nur um genau ein Element. Wird bei den Steuerelementen die Mehrfachauswahl zugelassen (Eigenschaft *SelectionMode*), dann hilft *SelectedItem* nicht mehr weiter.

Nun verfügen die betreffenden Steuerelemente zwar auch über eine Eigenschaft namens *SelectedItems* und diese verweist auch tatsächlich auf eine Auflistung der ausgewählten Elemente, doch ist diese leider schreibgeschützt und kann somit auch nicht gebunden werden. Auch um Elemente per C#-Code im ViewModel auswählen zu können und diese Auswahl von der Oberfläche reflektieren zu lassen, wäre das der falsche Ansatz. Was also tun?

Zaubertrick ItemContainerStyle

Die typischen WPF-Steuerelemente, die Auflistungen repräsentieren können, sind direkt oder indirekt abgeleitet von der Klasse *ItemsControl*. Bindet man eine Auflistung an die

Listing 1: Auswahl und Verfügbarkeit verknüpfen

```
// Listenelement für die Bindung an ListBox,
// DataGrid etc.
public class SelectableItem : NotificationObject
{
  // Wird ausgelöst, wenn IsEnabled geändert wurde
  public event EventHandler IsEnabledChanged;

  // Wird ausgelöst, wenn IsSelected geändert wurde
  public event EventHandler IsSelectedChanged;

  private object data;

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

  private bool isEnabled = true;

  // Gibt an, ob das Element verfügbar ist
  public bool IsEnabled
  {
    get { return isEnabled; }
    set
    {
      if (isEnabled == value) return;
      isEnabled = value;
      OnPropertyChanged();
      IsEnabledChanged?.Invoke(this, EventArgs.Empty);
    }
  }

  private bool isSelected;

  // Gibt an, ob das Element ausgewählt wurde
  public bool IsSelected
  {
    get { return isSelected; }
    set
    {
      if (IsSelected == value) return;
      isSelected = value;
      OnPropertyChanged();
      IsSelectedChanged?.Invoke(this, EventArgs.Empty);
    }
  }
}
```

Eigenschaft *ItemsSource*, dann legt das Control für jedes Element der Auflistung ein Container-Element an und setzt dessen *Content*-Eigenschaft auf das jeweilige Datenobjekt. Von welchem Typ diese Container-Elemente sind, hängt vom jeweiligen Steuerelement ab. Die Basisklasse *ItemsControl* verwendet den Typ *ContentPresenter*, eine ListBox erzeugt Container vom Typ *ListBoxItem* und ein DataGrid *DataGridRow*-Objekte. Diese Container-Objekte haben eine Reihe von Eigenschaften gemein. So verfügen sie zum Beispiel über die Property *IsSelected*, um eine Auswahl zu repräsentieren, oder über die Property *IsEnabled*, um zwischen verfügbaren und gesperrten Elementen unterscheiden zu können.

Doch wie kommt man an die Eigenschaften der Container, wenn diese ja automatisch erstellt werden? Der C#-Code des ViewModels soll ja keinen Zugriff auf die Objekte des Visual Tree erhalten.

Die Lösung liefert die Eigenschaft *ItemContainerStyle* [1] der Klasse *ItemsControl*. Ihr weist man einen Style zu, der auf jedes der automatisch generierten Container-Elemente angewendet wird. Über die Setter des Styles hat man den Zugriff auf alle Properties des betreffenden Container-Typs.

Das wiederum erfordert aber etwas Vorarbeit auf der Seite des ViewModels. Hier kann man die ursprüngliche Datenliste nicht mehr wie gewohnt durchreichen, sondern muss eine Hilfskonstruktion schaffen.

Der erste Schritt besteht aus der Definition einer Hilfsklasse, welche die Eigenschaften *IsSelected* beziehungsweise *IsEnabled* sowie den Verweis auf das Datenobjekt aufweist, vergleiche Listing 1.

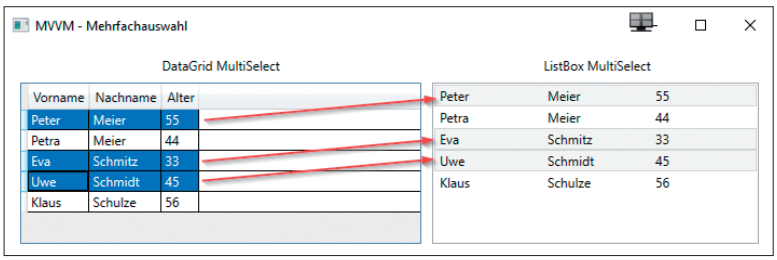
In Listing 2 sehen Sie die Definitionen der *ItemContainerStyle*-Eigenschaften für ein DataGrid und eine ListBox. Die Setter der Styles stellen jeweils die Verbindung zwischen den Eigenschaften der Container-Objekte und denjenigen der Hilfsklassen-Instanzen her. Beachten Sie bitte, dass diese Eigenschaften in der WPF typischerweise *OneWay*-Bindungen zum Standard haben, sodass die Eigenschaft *Mode* des Bindungsausdrucks explizit auf *TwoWay* gesetzt werden muss, wenn, wie im Fall von *IsSelected*, die Auswahländerung vom UI zum Datenobjekt zurückübertragen werden soll.

Die Verknüpfung von *IsSelected* funktioniert dann in beide Richtungen. Setzen Sie im Hilfsobjekt die Eigenschaft auf *true*, dann zeigen die Steuerelemente den Eintrag als aus- ▶

Listing 2: IsSelected und IsEnabled binden

```
<Window x:Class="WpfMvvmMultiselect.MainWindow"
  ...
  Title="MVVM - Mehrfachauswahl" Height="450"
    Width="800">
  <Window.Resources>
    <DataTemplate x:Key="PersonTemplate">
      <StackPanel Orientation="Horizontal"
        DataContext="{Binding Data}">
        <TextBlock Text="{Binding Vorname}"
          Width="100"/>
        <TextBlock Text="{Binding Nachname}"
          Width="100"/>
        <TextBlock Text="{Binding Alter}" Width="30"/>
      </StackPanel>
    </DataTemplate>
  </Window.Resources>
  ...
  <DataGrid ItemsSource="{Binding Liste}"
AutoGenerateColumns="False" IsReadOnly="False"
   Margin="5" CanUserAddRows="False" Grid.Row="1" >
    <!--IsSelected und IsEnabled der automatisch
      generierten DataGridRow-Objekte mit Objekt
      vom Typ SelectableItem binden-->
    <DataGrid.ItemContainerStyle>
      <Style TargetType="DataGridRow">
        <Setter Property="IsSelected" Value=
          "{Binding IsSelected, Mode=TwoWay}" />
        <Setter Property="IsEnabled" Value=
          "{Binding IsEnabled}" />
      </Style>
    </DataGrid.ItemContainerStyle>
    <DataGrid.Columns>
      <DataGridTextColumn Header="Vorname"
        Binding="{Binding Data.Vorname}"/>
      <DataGridTextColumn Header="Nachname"
        Binding="{Binding Data.Nachname}"/>
      <DataGridTextColumn Header="Alter"
        Binding="{Binding Data.Alter}"/>
    </DataGrid.Columns>
  </DataGrid>

  <ListBox ItemsSource="{Binding Liste}"
    SelectionMode="Extended" Grid.Column="1"
    Margin="5" ItemTemplate=
    "{StaticResource PersonTemplate}" Grid.Row="1">
    <!--IsSelected und IsEnabled der automatisch
      generierten ListBoxItem-Objekte mit Objekt
      vom Typ SelectableItem binden-->
    <ListBox.ItemContainerStyle>
      <Style TargetType="ListBoxItem">
        <Setter Property="IsSelected" Value=
          "{Binding IsSelected, Mode=TwoWay}" />
        <Setter Property="IsEnabled" Value=
          "{Binding IsEnabled}" />
      </Style>
    </ListBox.ItemContainerStyle>
  </ListBox>
  ...
</Window>
```

gewählt an. Ändern Sie die Auswahl in der Oberfläche, dann wird diese Eigenschaft von *SelectableItem* neu gesetzt. Die Implementierung von *INotifyPropertyChanged* für *IsSelected* sorgt dafür, dass Änderungen an andere Controls weitergegeben werden. Im Beispiel sind das DataGrid und die ListBox an dieselbe Auflistung gebunden. Änderungen im einen Control werden sofort im anderen nachgeführt – leider nur in leichtem Grau, da das jeweils andere Control nicht den Fokus hat.



Der erste Ansatz steht: Die Mehrfachauswahl in DataGrid und ListBox wird mit der Datenstruktur verbunden (Bild 1)

Wird *IsEnabled* auf *false* gesetzt, dann wird der jeweilige Eintrag ausgegraut und kann weder bearbeitet noch selektiert werden. Die Verknüpfung zu jeweils einer CheckBox, um auch *IsEnabled* einstellen zu können, werden wir später noch vorsehen.

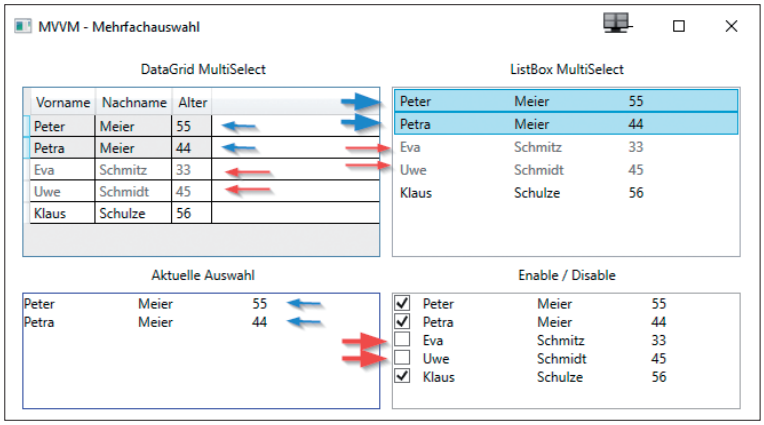
Etwas mehr Automatismus

Damit man es bei der Implementierung des ViewModels etwas bequemer hat und sich nicht um die Details der *SelectedItem*-Hilfsobjekte kümmern muss, bietet sich der Einsatz einer weiteren Hilfsklasse für die passende Auflistung an (Listing 3). *SelectableItemList* ist von *ObservableCollection* abgeleitet und erhält eine Factory-Methode, der eine beliebige Auflistung übergeben werden kann. Für jedes Element dieser Auflistung wird eine Instanz von *SelectableItem* angelegt und hinzugefügt. In der Überschreibung von *InsertItem* wird das *IsSelectedChanged*-Event verknüpft, um zentral auf Auswahländerungen reagieren zu können. In *RemoveItem* wird dieser Handler wieder entfernt. Eine Read-only-Property *SelectedItems* kann ferner bereitgestellt werden, um eine Liste der ausgewählten Objekte bekannt zu machen. Bei Bedarf könnte man hier auch die Liste der betreffenden Datenobjekte durchreichen. Die Implementierung lässt sich frei gestalten und den jeweiligen Anforderungen anpassen.

```
class Person {
  public string Vorname { get; set; }
  public string Nachname { get; set; }
```

Listing 3: SelectableItemList

```
// Vereinfacht den Umgang mit bestehenden Auflistungs-
// klassen und Änderungen der Auswahlliste von
// SelectableItem. Hilfsklasse zum Erstellen der
// Auflistung und zum Nachverfolgen der
// ausgewählten Elemente

public class SelectableItemList : ObservableCollection
<SelectableItem>
{
  // Factory-Methode zum Erstellen der Auflistung
  // <param name="items">Liste der Datenobjekte
  // </param>
  // <returns>Liste der SelectableItems</returns>
  public static SelectableItemList FromItems(
    IEnumerable items)
  {
    var list = new SelectableItemList();
    foreach (var item in items)
    {
      list.Add(new SelectableItem { Data = item });
    }
    return list;
  }

  private  void Si_IsSelectedChanged(
    object sender, EventArgs e)
  {
    base.OnPropertyChanged(
      new PropertyChangedEventArgs(
      nameof(SelectedItems)));
  }

  protected override void InsertItem(
    int index, SelectableItem item)
  {
    base.InsertItem(index, item);
    // Auf Änderungen von IsSelected reagieren
    item.IsSelectedChanged += Si_IsSelectedChanged;
  }

  protected override void RemoveItem(int index)
  {
    // Handler wieder entfernen
    this[index].IsSelectedChanged +=
      Si_IsSelectedChanged;
    base.RemoveItem(index);
  }

   // Ausgewählte Elemente
   public IEnumerable<SelectableItem> SelectedItems =>
     this.Where(i => i.IsSelected).ToList();
}
```



Separate Liste mit ausgewählten Objekten sowie Steuerung der IsEnabled-Eigenschaften der Listenobjekte (Bild 2)

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

class Personenliste : List<Person> {
  public Personenliste() {
    this.Add(new Person { Nachname = "Meier",
      Vorname = "Peter", Alter = 55 });
    this.Add(new Person { Nachname = "Meier",
      Vorname = "Petra", Alter = 44 });
    ...
  }
}
```

Diese Code-Zeilen zeigen die Implementierung der Beispieldaten für eine Klasse *Person*. Der Aufbau des ViewModels beschränkt sich dann auf das Instanzieren der Demo-Daten und das Anlegen der Hilfsstruktur. Das Ergebnis sehen Sie in Bild 1.

```
public class ViewModel : NotificationObject {
  // Liste auswählbarer Objekte
  public ObservableCollection<SelectableItem> Liste {
    get; set; }
  ...
  // ctor
  public ViewModel() {
    // Liste aus Demodaten erstellen
    Liste = SelectableItemList.FromItems(
      new Personenliste());
    ...
  }
  ...
}
```

Nachdem die Basis für den Umgang mit Mehrfachauswahlmöglichkeiten geschaffen wurde, können nun weitere Funktionalitäten in der Oberfläche genutzt werden. In Listing 4 werden zwei weitere Steuerelemente hinzugefügt (Typ *ItemsControl*).

Das erste ist an die oben beschriebene Eigenschaft *SelectedItems* der Hilfsklasse *SelectableItemList* gebunden und zeigt die Liste aller ausgewählten Personen. Das zweite ist an die gesamte Liste gebunden und zeigt für jedes Listenelement zusätzlich eine CheckBox an, deren *IsChecked*-Eigenschaft mit der *IsEnabled*-Eigenschaft der Hilfsklasse *SelectableItem* verknüpft ist. Änderungen in den CheckBoxen wirken sich sofort im DataGrid und in der ListBox aus (ist *IsEnabled false*, dann werden die Listeneinträge ausgegraut, siehe Bild 2).

Setzen von IsSelected im ViewModel

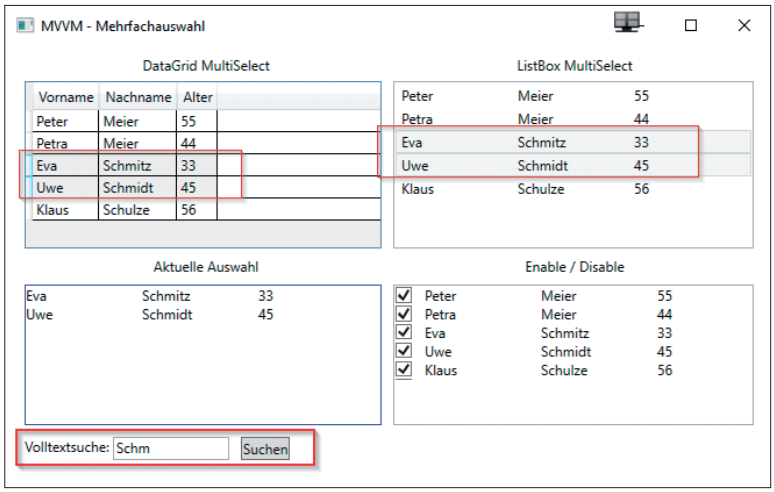
Auch das ist ein denkbares Szenario: In Bild 3 ist das Beispielprogramm um eine TextBox zur Volltextsuche erweitert worden. Im imperativen C#-Code wird bei Eingabe eines Suchtextes die gesamte Liste durchlaufen und je nach Prüfung der Texte die *IsSelected*-Eigenschaft der *SelectableItem*-Objekte gesetzt. Das Ergebnis wird sofort in der Oberfläche sichtbar. DataGrid und ListBox zeigen die Einträge als markiert an, die den vorgegebenen Suchtext enthalten. Listing 5 zeigt den XAML-Code für die Eingabe des Suchtextes und den ViewModel-Code für die Auswahl der Elemente, die dem Suchkriterium entsprechen. ▶

Listing 4: Ausgewählte Elemente anzeigen

```
<Window x:Class="WpfMvvmMultiselect.MainWindow"
  ...>
  <Grid Margin="10">
    ...
    <ItemsControl Grid.Row="3" Margin="5"
      ItemsSource="{Binding Liste.SelectedItems}"
      ItemTemplate="{StaticResource PersonTemplate}"
      BorderBrush="blue" BorderThickness="1" />

    <ItemsControl Grid.Row="3" Grid.Column="1"
      Margin="5" ItemsSource="{Binding Liste}"
      BorderBrush="DarkGray" BorderThickness="1">
      <ItemsControl.ItemTemplate>
        <DataTemplate>
          <StackPanel Orientation="Horizontal">
            <CheckBox IsChecked="{Binding IsEnabled,
              Mode=TwoWay}" Margin="0,0,10,0"/>
            <ContentControl Content="{Binding }"
              ContentTemplate=
              "{StaticResource PersonTemplate}" />
          </StackPanel>
        </DataTemplate>
      </ItemsControl.ItemTemplate>
    </ItemsControl>
    ...
  </Grid>
</Window>
```

Listing 5: XAML- und ViewModel-Code für eine Volltextsuche

```
<Window x:Class="WpfMvvmMultiselect.MainWindow"
  ...
  <StackPanel Orientation="Horizontal"
    Grid.Row="4" Margin="5">
    <TextBlock Text="Volltextsuche: "/>
    <TextBox Text="{Binding Suchfeld,
      UpdateSourceTrigger=PropertyChanged}"
      Width="100"/>
    <Button Content="Suchen" Command=
      "{Binding VolltextsucheCommand}"
      Margin="10,0"/>
  </StackPanel>
  ...
</Window>
...
public class ViewModel : NotificationObject {
  ...
  // Kommando Volltextsuche
  public ActionCommand VolltextsucheCommand {
    get; set; }
  // Eingabe für die Volltextsuche
  public string Suchfeld { get; set; }

  // ctor
  public ViewModel()
  {
    ...
    // Command anlegen
    VolltextsucheCommand =
      new ActionCommand(Volltextsuche);
    Suchfeld = "Schm";
  }

  private void Volltextsuche() {
    // IsSelected gemäß Suchergebnis setzen
    foreach (var item in Liste)
    {
      var person = (Person)item.Data;
      string text =
        $"{person.Vorname}_{person.Nachname}_
        {person.Alter}";
      item.IsSelected = text.Contains(Suchfeld);
    }
  }
}
```



Bei der Volltextsuche in diesem Beispiel wird die IsSelected-Eigenschaft der Hilfsobjekte im C#-Code gesetzt (Bild 3)

Beide Richtungen sind also möglich. Man kann im ViewModel auf im UI geänderte Auswahlen reagieren, und man kann im ViewModel die Auswahl vorgeben und so Einfluss auf die Darstellung nehmen.

Fazit

Der Zaubertrick mit *ItemContainerStyle* eröffnet viele Möglichkeiten, die Eigenschaften der automatisch generierten Container mit Eigenschaften von Hilfsobjekten zu binden und so auf Änderungen im UI reagieren zu können und andererseits die Oberfläche über Änderungen der Datenstrukturen (Auswahl, Verfügbarkeit et cetera) zu informieren. Mit ein wenig Infrastruktur ist es dann vergleichsweise einfach, ohne Eingriff in den Visual Tree, nur über POCO-Strukturen, die Bezüge zwischen ViewModel und Auflistungs-Controls herzustellen. Die Beispielimplementierung soll die Zusammenhänge verdeutlichen. Nehmen Sie den Code als Anregung für Ihre eigenen Implementierungen. Der Sourcecode des Beispiels steht unter [2] zur Verfügung. Für das Beispiel wurde .NET 5 eingesetzt. Die Vorgehensweise funktioniert aber gleichermaßen in .NET 4.x. ■

[1] ItemsControl.ItemContainerStyle, www.dotnetpro.de/SL2107DataGridMultiselect1
[2] Beispielcode, www.dotnetpro.de/SL2107DataGridMultiselect2

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 A2107DataGridMultiselect