MVVM – Tutorial 2: Mehrere Fenster / Datagrid

Übersicht

Um bei MVVM mit mehreren Windows zu arbeiten, existieren unterschiedliche Ansätze. Wesentlich ist dabei, dass die ViewModels die einzelnen Views nicht kennen sollen. Also diese auch nicht direkt erzeugen sollen.

Nachfolgend wird ein sehr einfach gehaltener Ansatz vorgestellt, der diesen Grundsatz jedoch erfüllt!

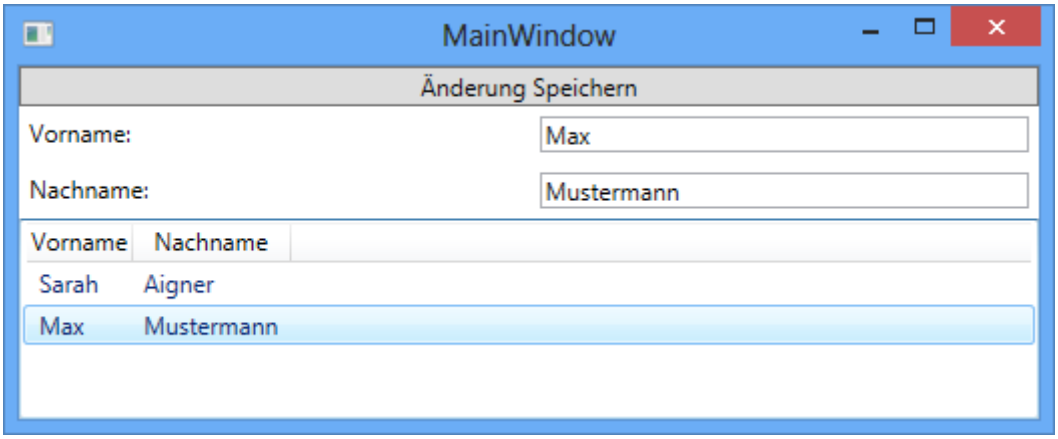
Dabei übernimmt ein eigener Controller die Erstellung und Anzeige der jeweiligen Views (Windows). Der Controller wird im WPF-Assembly (könnte auch eigene Controller-Assembly sein) implementiert.

Eine Referenz auf den Controller wird jedem ViewModel über seinen Konstruktor übergeben. Dadurch kann das ViewModel das Öffnen eines neuen Fensters anstoßen. (Details dazu im nachfolgenden Beispiel.)

Damit von der Assembly der ViewModels keine Referenz auf die View-Assembly (WPF) erforderlich ist, wird im ViewModel-Assembly ein Interface IController implementiert, welches nur eine Methode ShowWindow(BaseViewModel viewModel) enthält. Der Controller implementiert dieses Interface.

Beispielanwendung

Wir wollen den ActivityManger aus dem letzten Teil nun mit der Verwaltung der Aktvititäten erweitern.



Schritt 1: IController Interface

Im ViewModel-Assembly erstellen wir ein Interface IController:

```
namespace ActivityReport.ViewModel
{
    public interface IController
    {
        void ShowWindow(BaseViewModel viewModel);
    }
}
```

Das Controller-Interface definiert nur eine Methode ShowWindow. Dieser Methode wird eine ViewModel Instance übergeben. Anhand der jeweiligen View-Model Instanz wird dann vom Controller entschieden welches Window erstellt werden soll und das übergebenen ViewModel als Datacontext gesetzt!

Hinweis: Würde das ViewModel als Auswahlkriterium für das zu öffnende Fenster nicht ausreichen, müssten für ShowWindow weitere Parameter definiert werden. Für unsere einfach gehaltene Demonstration wird das ViewModel als Parameter ausreichen.

Schritt 2: BaseViewModel erweitern

Die Basisklasse aller ViewModels erweitern wir so, daß der Konstruktor einen IController als Parameter entgegennimmt und ein (protected) Feld setzt:

```
public class BaseViewModel : INotifyPropertyChanged
{
    protected IController _controller;

    public event PropertyChangedEventHandler PropertyChanged;

    0 references
    public BaseViewModel(IController controller)
    {
        this._controller = controller;
    }

    4 references
    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Schritt 3: Konstruktor im ViewModel anpassen

Nun müssen natürlich auch alle vom BaseViewModel abgeleiteten ViewModels den IController im Konstruktor entgegennehmen und an den Basiskonstruktor weiterleiten:

```
public EmployeeViewModel(IController controller):base(controller)
{
    LoadEmployees();
}
```

Schritt 4: Controller implementieren

Wir implementieren einen Controller nun direkt im WPF Projekt. Als Bezeichnung wählen wir MainController. Der Controller implementiert IController. Je nach ViewModel erzeugt er das Fenster und setzt den DataContext. Anschließend wird das Fenster (modal) angezeigt.

```
public class MainController : IController
{
  public void ShowWindow(BaseViewModel viewModel)
  {
    Window window = viewModel switch
    {
      // Wenn viewModel null ist -> ArgumentNullException
      null => throw new ArgumentNullException(nameof(viewModel)),

      // Wenn viewModel vom Type EmployeeViewModel ist -> neues MainWindow instanzieren
      EmployeeViewModel _ => new MainWindow(),

      // default -> InvalidOperationException
      _ => throw new InvalidOperationException($"Unknown ViewModel of type '{viewModel}'"),
    };

    window.DataContext = viewModel;
    window.ShowDialog();
  }
}
```

Schritt 5: Startfenster anzeigen

Wir **entfernen** zunächst aus dem App.xaml die Startup-Uri damit nicht automatisch das MainWindow angezeigt wird.

```
<Application x:Class="ActivityReport.Wpf.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

Stattdessen ergänzen wir ein Startup-Event:

```
<Application x:Class="ActivityReport.Wpf.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:ActivityReport.Wpf"
             Startup="Application_Startup">
```

In der Behandlungsroutine des Startup-Events wird die Controller-Instanz erstellt und diesem eine EmployeeViewModel-Instanz übergeben. Dadurch wird vom Controller die zum ViewModel entsprechende View angezeigt.

Es darf dabei nicht vergessen werden, dass dem ViewModel auch eine Referenz auf den Controller übergeben werden muss, damit das ViewModel dann selbst das Öffnen weiterer Views anstoßen kann.

```
public partial class App : Application
{
    1 reference
    private void Application_Startup(object sender, StartupEventArgs e)
    {
        IController controller = new MainController();
        controller.ShowWindow(new EmployeeViewModel(controller));
    }
}
```

Der Kontroller wird nun bereits das MainWindow erstellen! Da der Kontroller auch den DataContext setzt, können wir das Setzen des DataContext aus der CodeBehind-Datei des **Main-Windows** löschen:

```
namespace ActivityReport.Wpf
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

Nun ist das Framework fertig! Es können nun einfach weitere Fenster ergänzt werden!

Aktivitäten bearbeiten

Zunächst erstellen wir ein neues ViewModel für die Aktivitäten. Da auf dem Aktivitätenformular der volle Name des Mitarbeiters und seine Tätigkeiten angezeigt werden sollen, werden diese Properties im ViewModel vorgesehen. In den Konstruktor wird zusätzlich der gewählte Mitarbeiter übergeben und seine Tätigkeiten werden in eine ObservableCollection geladen!

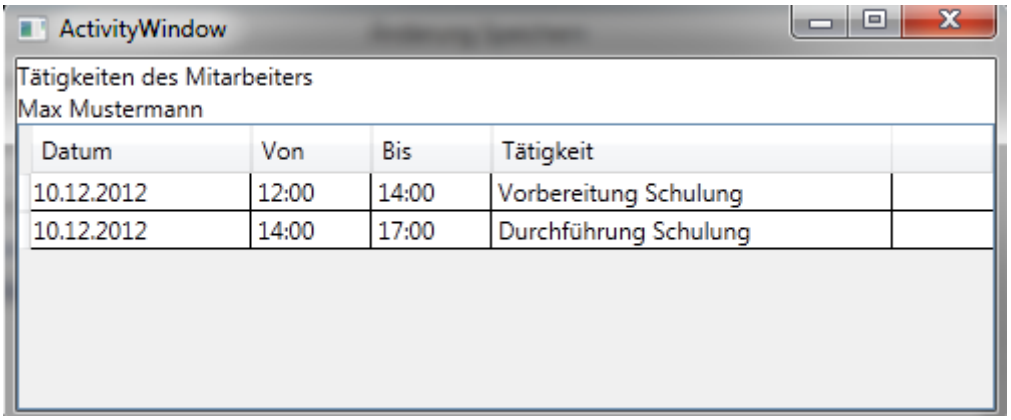
```
public class ActivityViewModel : BaseViewModel
{
  private Employee _employee;
  private ObservableCollection<Activity> _activities;

  public ObservableCollection<Activity> Activities
  {
    get => _activities;
    set
    {
      _activities = value;
      OnPropertyChanged(nameof(Activities));
    }
  }

  public string FullName => $"{_employee.FirstName} {_employee.LastName}";

  public ActivityViewModel(IController controller, Employee employee) : base(controller)
  {
    _employee = employee;
    using IUnitOfWork uow = new UnitOfWork();
    Activities = new ObservableCollection<Activity>(uow.ActivityRepository.Get(
      filter: x => x.Employee_Id == employee.Id,
      orderBy: coll => coll.OrderBy(activity => activity.Date).ThenBy(activity => activity.StartTime)));
  }
}
```

Die neue View (ActivityWindow) erstellen und entsprechend über Binding verknüpfen:

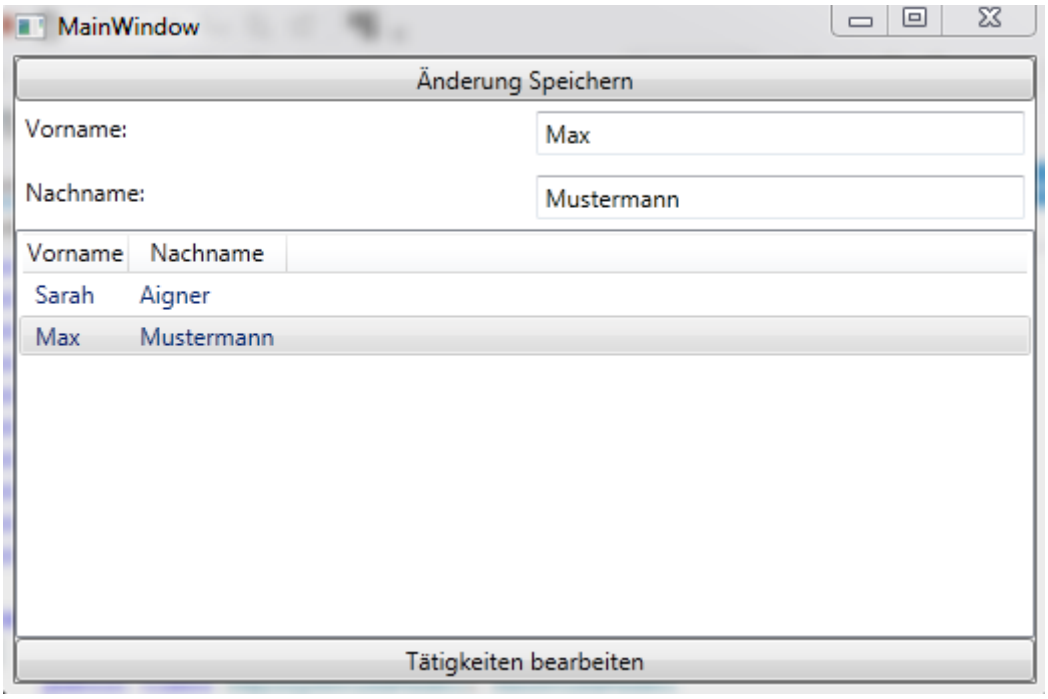


Code:

```
<Window x:Class="ActivityReport.Wpf.ActivityWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Tätigkeiten" Height="300" Width="600">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <StackPanel Grid.Row="0" Orientation="Vertical">
            <TextBlock>Tätigkeiten des Mitarbeiters</TextBlock>
            <TextBlock Foreground="DarkGreen" Text="{Binding FullName}"></TextBlock>
        </StackPanel>
        <DataGrid Grid.Row="1" ItemsSource="{Binding Activities}" AutoGenerateColumns="False">
            <DataGrid.Columns>
                <DataGridTemplateColumn Header="Datum" MinWidth="110">
                    <DataGridTemplateColumn.CellEditingTemplate>
                        <DataTemplate>
                            <DatePicker SelectedDate="{Binding Path=Date, UpdateSourceTrigger=PropertyChanged}"
                                        SelectedDateFormat="Short" />
                        </DataTemplate>
                    </DataGridTemplateColumn.CellEditingTemplate>
                    <DataGridTemplateColumn.CellTemplate>
                        <DataTemplate>
                            <TextBlock Text="{Binding Path=Date, StringFormat=dd.MM.yyyy}"></TextBlock>
                        </DataTemplate>
                    </DataGridTemplateColumn.CellTemplate>
                </DataGridTemplateColumn>
                <DataGridTextColumn Header="Von" MinWidth="60" Binding="{Binding Path=StartTime,
                    StringFormat=HH:mm, UpdateSourceTrigger=LostFocus}"/>
                <DataGridTextColumn Header="Bis" MinWidth="60" Binding="{Binding Path=EndTime,
                    StringFormat=HH:mm, UpdateSourceTrigger=LostFocus}"></DataGridTextColumn>
                <DataGridTextColumn Header="Tätigkeit" MinWidth="200" Binding="{Binding Path=ActivityText,
                    UpdateSourceTrigger=LostFocus}"></DataGridTextColumn>
            </DataGrid.Columns>
        </DataGrid>
    </Grid>
</Window>
```

Nun ergänzen wir einen Button im MainWindow und verbinden diesen mit einem neuen Command im EmployeeViewModel.

```
<Button Grid.Row="3" Command="{Binding CmdEditActivities}">Tätigkeiten bearbeiten</Button>
```



Dieses Command im EmplyoeeViewModel soll eine Instanz des ActivityViewModel erstellen und das Activity Window über den Controller öffnen! Wichtig dabei ist es, dass der gewählte Mitarbeiter im ActivityViewModel gesetzt wird.

```
private ICommand _cmdEditActivities;

public ICommand CmdEditActivities
{
  get
  {
    if (_cmdEditActivities == null)
    {
      _cmdEditActivities = new RelayCommand(
        execute: _ => _controller.ShowWindow(new ActivityViewModel(_controller, SelectedEmployee)),
        canExecute: _ => SelectedEmployee != null);
    }

    return _cmdEditActivities;
  }
}
```

D.h. im Controller ergänzen wir das Öffnen des neuen Formulars:

```
public class MainController : IController
{
  public void ShowWindow(BaseViewModel viewModel)
  {
    Window window = viewModel switch
    {
      // Wenn viewModel null ist -> ArgumentNullException
      null => throw new ArgumentNullException(nameof(viewModel)),

      // Das dem Type des ViewModel entsprechende Window instanzieren
      EmployeeViewModel _ => new MainWindow(),
      ActivityViewModel _ => new ActivityWindow(),

      // default -> InvalidOperationException
      _ => throw new InvalidOperationException($"Unknown ViewModel of type '{viewModel}'"),
    };

    window.DataContext = viewModel;
    window.ShowDialog();
  }
}
```

Jetzt kann das Programm bereits gestartet werden und die Anzeige sollte funktionieren. Auf Änderungen im Datagrid wird jedoch noch nicht reagiert. Dazu ist folgendes erforderlich:

- Wie bereits vor MVVM müssen wir zum einen auf gelöschte Datensätze im Grid reagieren. Hier wird wiederum CollectionChanged der ObservableCollection abonniert. Da diese nun im ViewModel deklariert ist, können wir korrekterweise dieses Event direkt im ViewModel abonnieren (gleich im Konstruktor):

```
private void LoadActivities()
{
    using (UnitOfWork uow = new UnitOfWork())
    {
        Activities = new ObservableCollection<Activity>(
            uow.ActivityRepository.Get(filter: p => p.Employee_Id == _actEmp.Id,
            orderBy: coll => coll.OrderBy(act => act.Date).ThenBy(act => act.StartTime)));
    }
    Activities.CollectionChanged += Activities_CollectionChanged;
}

1 reference
private void Activities_CollectionChanged(object sender,
    System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
{
    if (e.Action == System.Collections.Specialized.NotifyCollectionChangedAction.Remove)
    {
        using (UnitOfWork uow = new UnitOfWork())
        {
            foreach (var item in e.OldItems)
            {
                uow.ActivityRepository.Delete((item as Activity).Id);
            }
            uow.Save();
        }
    }
}
```

- Um auf Änderung und Neuanlegen reagieren zu können, könnte wiederum eine Datenbindung mit SelectedItem des DataGrid erfolgen. In den Properties dazu kann dann darauf reagiert werden. Dies ist jedoch aufwändiger, da auf einige Sonderfälle speziell reagiert werden sollte (Bearbeitung abbrechen…). RowEditEnding des Datagrid, erledigt viele dieser Dinge. -> Siehe dazu Anhang B (Selbststudium).

  In unserem Fall werden wir jedoch in der View einfach Buttons anbringen für Edit / New / Delete und das DataGrid per Property IsReadOnly sperren.

  Im Fall von Edit/New öffnen wir ein neues Formular, in welchem die Detaildaten eingegeben werden können und „Save“ bzw. „Cancel“ gewählt werden kann.

  Dies soll zur Übung selbstständig implementiert werden!

  Hinweis:

  Das DataGrid hat eine IsReadOnly-Eigenschaft um das gesamte Grid zu sperren. Falls Delete weiterhin über Tastatur funktionieren soll, müssten jedoch die einzelnen Spalten gesperrt werden (ebenfalls IsReadOnly je Spalte) und zusätzlich muss CanUserAddRows des Grids auf false gesetzt werden!

  (Nun ist Delete noch erlaubt, Spalten sind gesperrt, Neuanlegen funktioniert auch nur über Buttons…

  Alternativ: ListView verwenden!

Anhang A:

IController und dessen Implementierung soll so erweitert werden, dass sein ViewModel auch das Schließen eines Fensters anstoßen kann. (Ansonsten könnte ein Window nur durch den Close Button in der Ecke rechts oben geschlossen werden und nicht vom Programm heraus.).

Dazu wird im Interface IController eine zusätzliche Methode CloseWindow vorgesehen:

```
4 references
public interface IController
{
    1 reference
    void ShowWindow(BaseViewModel viewModel);
    1 reference
    void CloseWindow(BaseViewModel viewModel);
}
```

Im Controller selbst werden alle geöffneten Fenster mit dem jeweiligen ViewModel in einem Dictionary gespeichert.

Wenn dann die Methode CloseWindow aufgerufen wird, kann mittels dem übergebenen ViewModel das zugehörige Fenster aus dem Dictionary gewonnen werden und das Fenster kann geschlossen werden.

```
public class MainController : IController
{
    // Speicherung der Zuordnung zw. ViewModel und Window
    private Dictionary<BaseViewModel, Window> _windows;

    public MainController()
    {
        _windows = new Dictionary<BaseViewModel, Window>();
    }

    public void ShowWindow(BaseViewModel viewModel)
    {
        Window window = viewModel switch
        {
            // Wenn viewModel null ist -> ArgumentNullException
            null => throw new ArgumentNullException(nameof(viewModel)),

            // Das dem Type des ViewModel entsprechende Window instanzieren
            EmployeeViewModel _ => new MainWindow(),
            ActivityViewModel _ => new ActivityWindow(),
            NewActivityViewModel _ => new NewActivityWindow(),
            EditActivityViewModel _ => new EditActivityWindow(),

            // default -> InvalidOperationException
            _ => throw new InvalidOperationException($"Unknown ViewModel of type '{viewModel}'"),
        };

        _windows[viewModel] = window;
        window.DataContext = viewModel;
        window.ShowDialog();
    }

    public void CloseWindow(BaseViewModel viewModel)
    {
        if (_windows.ContainsKey(viewModel))
        {
            _windows[viewModel].Close();
            _windows.Remove(viewModel);
        }
    }
}
```

Anhang B:

Um bei Änderung eines einzelnen Feldes von Activity (z.B. ActivityText) von seitens des ViewModels auch immer ein korrektes Aktualisieren in der View zu gewährleisten, muß Activity ebenfalls INotifyPropertyChange implementieren.

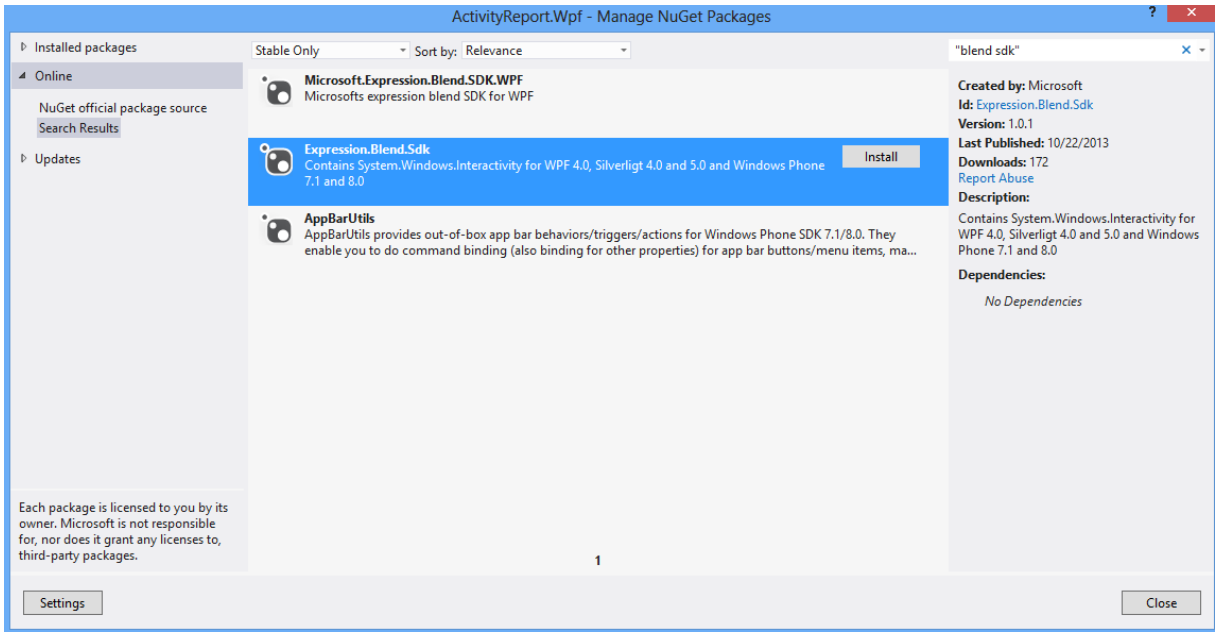
In der ObservableCollection wird aber leider trotz INotifyPropertyChange des Activity-Datensatzes kein Event ausgelöst, wenn ein Datensatz geändert wird!

Es gibt auch die Möglichkeit ein Event dazu zu veranlassen ein Command auszulösen (z.B. das RowEditEnding Event soll ein Command auslösen).

Wie kann jetzt im ViewModel auf ein Event der View reagiert werden?

Um Events mit Command zu verbinden verwenden wir Interaction Triggers:

Zunächst müssen über NuGet Expression.Blend.Sdk installieren:



Nun muss in Xaml folgender Namespace eingebunden werden:

```
xmlns:i="clr-namespace:System.Windows.Interactivity;assembly=System.Windows.Interactivity"
```

Nun kann ein Event ein Command auslösen (leider ist die Übergabe der EventArgs etwas umständlich… hier würde sich dann ein vorgefertigtes Framework anbieten)

```
<DataGrid  Grid.Row="1" ItemsSource="{Binding Activities}" AutoGenerateColumns="False">
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="RowEditEnding">
            <i:InvokeCommandAction  Command="{Binding CmdRowEditEnding}"/>
        </i:EventTrigger>
    </i:Interaction.Triggers>
    <DataGrid.Columns>
```