PROGRAMMIEREN MIT WINUI 3, TEIL 2

WinUI 3 im Praxistest

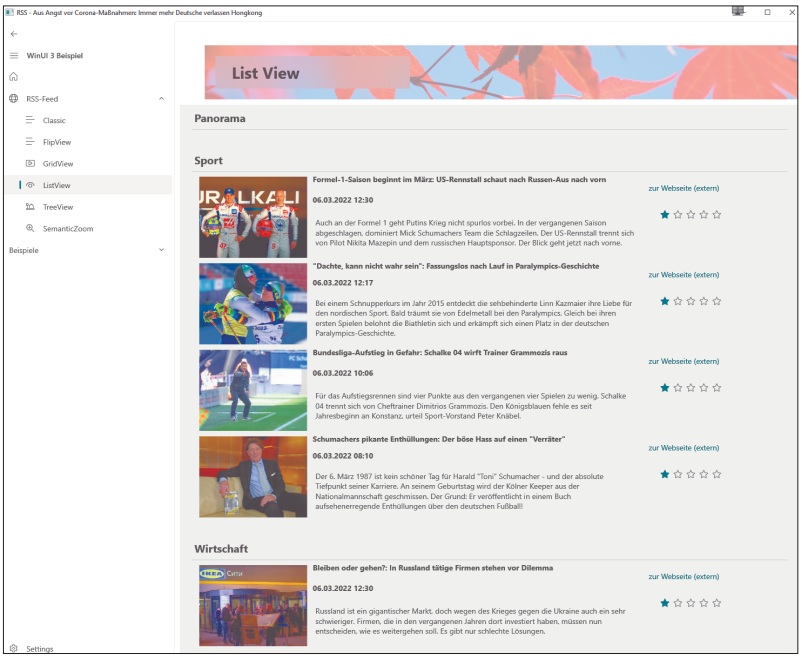
Wie lassen sich WinUI 3 und MVVM vereinen?

Die Grundideen von WinUI 3 hatten wir bereits in einem ersten Artikel vorgestellt [1]. Doch wie gut lässt sich das neue Framework in die heutige Programmierpraxis integrieren? Wie lassen sich die aus WPF und anderen Frameworks bekannten und beliebten Patterns wie Dependency Injection und Model-View-ViewModel umsetzen?

Auch bei WinUI 3 erinnert vieles wieder an die Anfangszeit von WPF (Windows Presentation Foundation). Microsoft liefert eine Toolbox mit vielen Komponenten, aber keine Anleitung, wie man ein sinnvolles Grundgerüst für eine Anwendung aufbauen könnte. Strukturierte Vorlagen, wie man sie aus der ASP.NET-Welt für Web APIs, MVC oder Razor Pages kennt, fehlen bislang leider. So ist es mal wieder die Aufgabe der Entwickler, sich selbst ein geeignetes Rahmenprogramm zu erstellen.

Im Zusammenhang mit WPF hat sich im Lauf der Zeit das Model-View-ViewModel-Pattern (MVVM) als praktisch und gut geeignet erwiesen. WinUI 3 ähnelt in der Architektur stark der von WPF. Lässt sich das Pattern auch hier anwenden? Und wie sieht es aus mit Dependency Injection? Kann man dieses Konzept auch für WinUI 3 sinnvoll nutzen?

Anhand eines Beispielprogramms mit verschiedenen Views und ViewModels sollen die Zusammenhänge und möglichen Umsetzungen erläutert werden. Wie in Bild 1 zu sehen, verwendet das Beispiel die aus dem ersten Teil bereits bekannte Komponente *NavigationView* für die Darstellung einer Menüstruktur, die Navigation sowie die Anzeige der ausgewählten Ansicht.



Beispielanwendung mit Navigationsstruktur und verschiedenen Ansichten (Bild 1)

Dependency Injection

Im Gegensatz zu den alten UWP-Anwendungen (Universal Windows Platform) bietet WinUI 3 den Vorteil, dass man fast alles aus der .NET-Welt benutzen kann. Neben dem .NET-6-Framework stehen viele NuGet-Pakete, die in .NET-Desktop- oder -Webanwendungen eingesetzt werden, auch hier zur Verfügung. So kann man für den Einsatz von Dependency Injection zwischen zahlreichen Frameworks wählen.

Für das hier verwendete Beispielprogramm fiel die Wahl auf das aus ASP.NET Core bekannte Dependency Injection Framework, das über das NuGet-Paket *Microsoft.Extensions.DependencyInjection* in die Anwendung integriert werden kann. Einen ersten Ansatz hierzu sehen Sie in Listing 1. Über eine Instanz der Klasse *ServiceCollection* werden die benötigten Komponenten registriert. Bei einer Desktop-Anwendung wird man hierfür entweder *AddSingleton* oder *AddTransient* aufrufen. Ersteres legt einmalig eine Instanz des angegebenen Typs an, sodass jeder weitere Abruf wieder dieselbe Referenz liefert, wohingegen *AddTransient* bei jedem Abruf eine neue Instanz generiert. Die Prinzipien der Dependency Injection wurden bereits in [2] erläutert.

Nach dem MVVM-Pattern sollte man möglichst auf Implementierungen im Code-behind der Fensterklassen verzichten. Insbesondere beim Hauptfenster wird das aber kaum vermeidbar sein, da einige Implementierungen, zum Beispiel zur Navigation oder zum Anzeigen von Dialogen, auf Eigenschaften oder Elemente der *MainWindow*-Klasse zurückgreifen müssen. Diese Implementierungen lassen sich jedoch sehr elegant über ein Interface (hier *IShell*) entkoppeln. Alle Methoden und Eigenschaften, auf die anderweitig im Code zugegriffen werden soll, werden in diesem Interface deklariert. Mittels Dependency Injection wird, wie in Listing 1 gezeigt, der Typ *IShell* als Singleton registriert und direkt mit der Instanz von *MainWindow* verknüpft. So kann an jeder Stelle im Programm die Funktionalität abgerufen werden, ohne dass der Typ oder die Implementierung von *MainWindow* bekannt sein müssen.

Der Aufruf von *BuildServiceProvider* schließt die Registrierung ab. Werden Instanzen eines Typs über Dependency Injection erzeugt, kön-

Listing 1: Aufbau der Dependency-Injection-Infrastruktur

```
protected override void OnLaunched(Microsoft.UI.Xaml
    .LaunchActivatedEventArgs args)
{
  m_window = new MainWindow();
  // Setup dependency injection
  services = ConfigureServices();
  m_window.Setup(services);
  // Setup ViewModel of main window
  ((MainWindow)m_window).ViewModel =
    Services.GetService<MainViewModel>();
  m_window.Activate();
}

// Setup dependency injection
private IServiceProvider ConfigureServices()
{
  var serviceCollection = new ServiceCollection();
  serviceCollection.AddSingleton(typeof(IShell),
    m_window); // add shell
  serviceCollection.AddSingleton<SettingsProvider>();
  // Add ViewModels
  services.AddSingleton<MainViewModel>();
  services.AddTransient<Sample1ViewModel>();
  services.AddTransient<Sample2ViewModel>();
  services.AddTransient<StartPageViewModel>();

  return serviceCollection.BuildServiceProvider();
}
```

nen Instanzen anderer registrierter Typen über Constructor Injection abgerufen werden. Alternativ kann durch den Aufruf von Methoden wie *GetService* oder *GetRequiredService* die betreffende Instanz auch imperativ angefordert werden (wie am Beispiel von *MainViewModel* im Listing zu sehen).

Kommen später viele ViewModels zum Einsatz, wird es etwas mühsam und auch fehleranfällig, jedes ViewModel einzeln bei der Dependency Injection registrieren zu müssen. Dann bietet es sich an, einen eigenen Automatismus zur Registrierung zu implementieren. Listing 2 zeigt einen solchen Ansatz. Ein eigenes Attribut (hier *ExportAttribute*) dient zur Kennzeichnung von Klassen, die registriert werden sollen. Die boolesche Property *AsSingleton* gibt hierbei an, ob dies via *AddSingleton* oder *AddTransient* erfolgen soll. Über *Inherited = true* wird ferner festgelegt, dass abgeleitete Klassen automatisch die Attributierung der Basisklasse erben. So muss im Fall der ViewModels lediglich das Attribut in der Basisklasse (*ViewModelBase*) gesetzt werden, dann werden alle abgeleiteten Klassen automatisch registriert.

Navigation und ViewModels

Je nach Anwendung werden die verschiedenen ViewModels mehr oder weniger Gemeinsamkeiten aufweisen. In vielen Fällen wird der Zugriff auf die über *IShell* abstrahierte Funktionalität des Hauptfensters notwendig sein. Auch möchte man vielleicht auf einen Titel für die ausgewählte Ansicht zurückgreifen können. Das lässt sich typischerweise in einer Basisklasse für alle ViewModels festlegen, wie in Listing 3 für *ViewModelBase* gezeigt. Die Klasse selbst ist hier als *abstract* definiert, damit sie nicht selbst durch den oben beschrie- ▶

Listing 2: Automatische Registrierung via Reflection

```
[AttributeUsage(AttributeTargets.Class,
  Inherited = true)]
public class ExportAttribute : Attribute
{
  public bool AsSingleton { get; set; }
}
private IServiceProvider ConfigureServices()
{
  var serviceCollection = new ServiceCollection();
  ...
  // add services with [Export] annotation for this
  // assembly
  AddServicesFromAssembly(serviceCollection,
    Assembly.GetExecutingAssembly());
  return serviceCollection.BuildServiceProvider();
}
// add services with [Export] annotation for given
// assembly
private void AddServicesFromAssembly(
    ServiceCollection services, Assembly assembly)
{
  foreach (var type in assembly.GetTypes())
  {
    var exportAttr = type.GetCustomAttribute
      <ExportAttribute>();
    if (exportAttr != null && !type.IsAbstract)
      if (exportAttr.AsSingleton)
        services.AddSingleton(type);
      else
        services.AddTransient(type);
  }
}
```

Listing 3: Basis für die ViewModels: ViewModelBase

```
[Export]
public abstract class ViewModelBase : NotificationObject
{
  public ViewModelBase(IShell shell)
  {
    this.shell = shell;
  }
  protected readonly IShell shell;
  private string title = "ohne Titel";
  // Title to be shown in UI
  public string Title
  {
    get { return title; }
    set { title = value; OnPropertyChanged(); }
  }
  // virtual methods that are connected to events and
  // may be overridden in derived classes
  public virtual void PageLoaded(object sender,
    RoutedEventArgs e) { }
  public virtual void PageUnloaded(object sender,
    RoutedEventArgs e) { }
  public virtual void OnNavigatingFrom(
    NavigatingCancelEventArgs e) { }
}
```

benen Automatismus in der Dependency Injection registriert und gegebenenfalls instanziert wird. Hinzu kommen später noch einige für die Navigation wichtige Methoden.

Zum Umschalten zwischen verschiedenen Ansichten soll hier das bereits im ersten Teil beschriebene Steuerelement *NavigationView* eingesetzt werden. Abweichend vom vorherigen Beispiel soll hier jedoch der Menüaufbau über die Bindung an eine Datenstruktur erfolgen. Listing 4 zeigt die XAML-Seite hierzu. Der Aufbau der rekursiven Datenstruktur zur Abbildung des Menübaums ist in Listing 5 zu sehen. *PageTypeTreeItem* ist hierbei nur eine Hilfsklasse, da der XAML-Compiler nicht mit generischen Datentypen umgehen kann. Initialisiert wird diese Struktur im Konstruktor von *MainViewModel* (Listing 6). Ein *TreeItem* verfügt einerseits über die für die Visualisierung wichtigen Eigenschaften wie *Title*, *Icon* und *IsSelected*, andererseits über die Eigenschaft *Data*, über welche das *Type*-Objekt des betreffenden View-Typs (nicht des ViewModels!) verknüpft wird. Untermenüs können über die Eigenschaft *Children* verknüpft werden. Dann sollte die *Data*-Eigenschaft den Wert null aufweisen.

Listing 4: Die NavigationView innerhalb des Hauptfensters

```
<Window
    x:Class="WinUI3MVVM.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/
      xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/
      xaml"
    xmlns:vh="using:WinUI3MVVM.ViewHelper">
  <Page>
    <Page.Resources>
      <DataTemplate x:Key="NVHT">
        ...
        <TextBlock Text="{Binding  Mode=OneWay}"
          VerticalAlignment="Center" MinWidth="300"
          Margin="30,0" FontSize="30"
          FontWeight="Bold" />
        ...
      </DataTemplate>
    </Page.Resources>

    <NavigationView MenuItemsSource=
        "{x:Bind ViewModel.MenuItems}"
      IsBackEnabled=
        "{x:Bind contentFrame.CanGoBack, Mode=OneWay}"
      BackRequested="BackRequested"
      HeaderTemplate="{StaticResource NVHT}"
      ItemInvoked="{x:Bind ViewModel.MenuItemInvoked}"
      Name="nvMain"
      Header="*** Header" PaneTitle="WinUI 3
        Beispiel">
      <NavigationView.MenuItemTemplate>
        <DataTemplate x:DataType=
            "vh:PageTypeTreeItem">
          <NavigationViewItem
            Content="{Binding Title}" Tag="{Binding}"
            MenuItemsSource="{Binding Children}"
            Icon="{Binding Icon}"/>
        </DataTemplate>
      </NavigationView.MenuItemTemplate>
      <Frame Name="contentFrame"
        Navigated="OnRootFrameNavigated"
        Navigating="OnRootFrameNavigating" Margin="10">
        ...
      </Frame>
    </NavigationView>
  </Page>
</Window>
```

Listing 5: Aufbau einer Datenstruktur zur Abbildung des Menübaums

```
/// <summary>
/// Helper class for use in XAML
/// </summary>
public class PageTypeTreeItem : TreeItem<Type>
{
}
  public class TreeItem<DataType> : NotificationObject
{
  /// Display text
  public string Title...
  // Display icon
  public IconElement Icon...
  // Connected data
  public DataType Data...
  // Child items
  public ObservableCollection<TreeItem<DataType>>
    Children...
  // Property to connect to IsSelected property of
  // control (e. g. via Style)
  public bool IsSelected...

}
```

Wie die eigentliche Navigation erfolgt, sehen Sie in Listing 7. Die innerhalb des *NavigationView*-Controls platzierte Komponente vom Typ *Frame* stellt die benötigte Methode *Navigate* bereit. Leider ist diese Methode, wie wir schon in [1] gesehen hatten, sehr unpraktisch umgesetzt worden. Sie nimmt als Parameter ein *Type*-Objekt entgegen und instanziert diesen angegebenen Typ selbstständig via Reflection. Damit vereitelt die Methode jeglichen Versuch, die Instanz über Dependency Injection zu erzeugen. Wir wollen jedoch, dass die zugehörigen ViewModel-Instanzen auf jeden Fall aus dem DI-Container bezogen werden. Das erzwingt leider zusätzlichen Code im Code-behind jeder View-Klasse. Hier muss über die App-Klasse auf den *ServiceProvider* zugegriffen werden, der die gewünschte Instanz liefern kann.

Da dies relativ oft benötigt wird, bietet sich die Implementierung einiger Hilfsmethoden in Form von Extension-Methoden an. Das verringert den in den Views erforderlichen Code deutlich. Listing 8 enthält die Methode *SetupViewModel* sowie noch zwei weitere Helferlein, während Listing 9 den dadurch vereinfachten Code am Beispiel einer View zeigt. *SetupViewModel* ermittelt über Dependency Injection die ViewModel-Instanz und gibt deren Referenz als Funktionswert ▶

Listing 6: Zusammensetzen der gewünschten Navigationsstruktur im MainViewModel

```
[Export(AsSingleton = true)]
public class MainViewModel : ViewModelBase
{
  // Menu tree for navigation
  public ObservableCollection<PageTypeTreeItem>
    MenuItems { get; set; }
  public MainViewModel(IShell shell) : base(shell)
  {
    // initialize the menu tree
    MenuItems = new()
    {
      new PageTypeTreeItem {
        Title = shell.GetResourceString("StartPage"),
        Icon = new SymbolIcon(Symbol.Home),
        Data = typeof(StartPageView) },
      new PageTypeTreeItem
      {
        Title = "RSS-Feed",
        Icon = new SymbolIcon(Symbol.Globe),
        Children = new()
        {
          new PageTypeTreeItem { Title = "Classic",
            Icon = new SymbolIcon(Symbol.List),
            Data = typeof(RssFeedReaderClassicView) },
          new PageTypeTreeItem { Title = "FlipView",
            Icon = new SymbolIcon(Symbol.List),
            Data = typeof(RssFeedReaderFlipView) },
          ...
        }
      },
      new PageTypeTreeItem
      {
        Title = "Beispiele",
        Children = new()
        {
          new PageTypeTreeItem { Title = "Beispiel 1",
            Icon = new SymbolIcon(Symbol.Scan),
            Data = typeof(Sample1View) },
          ...
        }
      }
    };
    // select start page
    shell.NavigateTo(typeof(StartPageView), "");
  }
}
```

Listing 7: Navigation über die Frame-Komponente des Hauptfensters

```
/// <summary>
/// Navigate to page
/// </summary>
/// <param name="pageType">Type of desired page
/// </param>
/// <param name="args"></param>
public void NavigateTo(Type pageType, object args)
{
  contentFrame.Navigate(pageType, args);
}
```

```
/// <summary>
/// Navigate to page
/// </summary>
/// <typeparam name="TView">Type of desired page
/// </typeparam>
public void NavigateTo<TView>() where TView : Page
{
  contentFrame.Navigate(typeof(TView),
    typeof(TView).Name);
}
```

zurück. Nebenbei wird auch die *DataContext*-Eigenschaft gesetzt, für den Fall, dass in der View mit klassischen Datenbindungen gearbeitet wird. Ferner werden noch zwei Events verknüpft, auf die weiter unten noch eingegangen wird.

In WPF kann man die Vorgehensweise umkehren, indem man über Dependency Injection zunächst die ViewModel-Instanz holt und dann über ein typisiertes DataTemplate die passende View zur Anzeige bringt (ViewModel-first-Ansatz). Diese Vorgehensweise lässt sich hier leider nicht umsetzen, da zum einen die *Frame.Navigate*-Methode sehr eingeschränkt ist und zum anderen in WinUI 3 bislang keine Möglichkeit besteht, ein typisiertes DataTemplate ohne Key als Ressource zu definieren.

Sofern innerhalb einer View die Datenbindung an das ViewModel über *{x:Bind}* erfolgen soll, muss die betreffende ViewModel-Eigenschaft auch in der View-Klasse definiert sein. Zwar ließe sich das an dieser Stelle über eine generische Basisklasse vereinfachen, dann hätte man aber zusätzlichen syntaktischen Aufwand im XAML-Code. Diese Kröte muss man wohl leider schlucken.

Betrachten wir noch einmal die Umsetzung im XAML-Code des Hauptfensters (Listing 4). Über *MenuItemsSource* wird die Datenstruktur für die Menüs mit dem *NavigationView*-Control verknüpft. Die eigentliche Darstellung der einzelnen Menüeinträge wird über ein DataTemplate vorgegeben. Möchte man *{x:Bind}* für die Bindung der *NavigationViewItem*-Eigenschaften an die Eigenschaften der Datenstruktur verwenden, muss man für das DataTemplate den betreffenden Typ mitgeben. Ein ähnliches Beispiel findet sich in der WinUI 3 Controls Gallery [3]. Allerdings scheint es hier

Listing 8: Erweiterungsmethoden für die Dependency Injection

```
public static class ExtensionMethods
{
  /// <summary>
  /// Get the DI service provider
  /// </summary>
  /// <param name="obj"></param>
  /// <returns></returns>
  public static IServiceProvider GetServiceProvider(
      this DependencyObject obj)
  {
    return ((App)Application.Current).Services;
  }
  /// <summary>
  /// Setup a ViewModel
  /// </summary>
  /// <typeparam name="TViewModel"></typeparam>
  /// <param name="page">the view</param>
  /// <returns>Initialized ViewModel</returns>
  public static TViewModel SetupViewModel<TViewModel>(
      this Page page) where TViewModel : ViewModelBase
  {
    var vm = ((App)Application.Current).Services
      .GetService<TViewModel>();
    page.Loaded += vm.PageLoaded;
    page.Unloaded += vm.PageUnloaded;
    page.DataContext = vm;
    return vm;
  }
  /// <summary>
  /// Get the shell
  /// </summary>
  /// <param name="obj">any kind of object</param>
  /// <returns></returns>
  public static IShell GetShell(this object obj)
  {
    return ((App)Application.Current).Services
      .GetService<IShell>();
  }

}
```

Listing 9: Abrufen der ViewModels im Code-behind der Page-Klasse

```
public sealed partial class Sample1View : Page
{
  public Sample1ViewModel ViewModel { get; set; }
  public Sample1View()
  {
    this.InitializeComponent();
    ViewModel =
      this.SetupViewModel<Sample1ViewModel>();
  }
}
```

einen Bug zu geben. Wenn nämlich in der obersten Ebene der Menüstruktur mehr als drei Views verlinkt werden, funktioniert die Anwendung nicht mehr. Ein Absturz (siehe Bild 2), falsche Darstellung oder defekte Navigation sind die Folge. Deshalb wurde im vorliegenden Beispiel auf die klassische Datenbindung zurückgegriffen.

Wollen Sie im MainWindow eine Ressource anlegen, wie am Beispiel des DataTemplates für den Header der NavigationView, dann kann das leider nicht wie in WPF direkt im Window-Element erfolgen, da diese Klasse in WinUI 3 kurioserweise nicht von *FrameworkElement*, sondern direkt von *Object* abgeleitet ist und somit nicht über eine Eigenschaft namens *Resources* verfügt. Daher bietet es sich an, den gesamten Inhalt des Hauptfensters in einer *Page* zu kapseln, in der sich dann wiederum die gewünschten Ressourcen anlegen lassen.

Wählt der Benutzer einen Menüpunkt der Navigationsstruktur aus, wird das Ereignis *ItemInvoked* des *NavigationView*-Controls ausgelöst. Dieses lässt sich mittels *{x:Bind}*, wie wir bereits gesehen haben, direkt mit einer Methode des *MainViewModel* verknüpfen. Listing 10 zeigt deren Implementierung; *e.InvokedItemContainer* verweist hier auf die Auswahl vom Typ *NavigationViewItem*. Dessen Tag-Eigenschaft war bereits über die Bindung im Template mit der jeweiligen *PageTypeTreeItem*-Instanz verbunden worden, die wiederum für den Aufruf der *IShell.NavigateTo*-Methode eingesetzt wird.

Listing 10: Ein Eventhandler für die Navigation

```
// user clicked on menu item
public void MenuItemInvoked(NavigationView sender,
    NavigationViewItemInvokedEventArgs e)
{
  // click on settings icon?
  if (e.IsSettingsInvoked)
  {
    shell.NavigateTo(typeof(SettingsView),
      "settings");
    return;
  }

  // else get the desired view type
  var treeItem = e.InvokedItemContainer.Tag as
    PageTypeTreeItem;
  if (treeItem?.Data != null)
    shell.NavigateTo(treeItem.Data, treeItem.Title);
}
```

So schließt sich der Kreis von der Definition des *Type*-Objekts der View in der Datenstruktur bis zu deren Anzeige. Anders als im ersten Beispiel, bei dem der Typname im Tag gespeichert wurde und das *Type*-Objekt über Reflection ermittelt wurde, bietet diese Variante weitestgehend die Typsicherheit bereits zur Compile-Zeit.

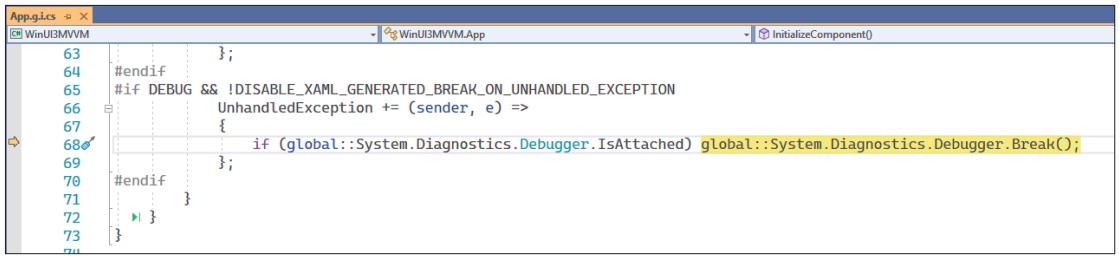
Etwas kurios ist der Umgang mit dem Menüpunkt *Settings*, der über das Zahnradsymbol aufgerufen werden kann. Dieser erfordert eine Sonderbehandlung. Mithilfe der Eigenschaft *NavigationViewItemInvokedEventArgs.IsSettingsInvoked* lässt sich ermitteln, ob eben dieser oder ein anderer Menüpunkt ausgewählt wurde. Wünscht der Benutzer die Anzeige der Einstellungsseite, wird explizit zu dieser View (hier *SettingsView*) navigiert. Ansonsten erfolgt die Navigation wie oben beschrieben.

Das *NavigationView*-Steuerelement kann auch eine *Zurück*-Schaltfläche (Symbol: Pfeil nach links) anzeigen. Diese muss allerdings zunächst einmal über die Eigenschaft *IsBackEnabled* freigeschaltet werden. Klickt der Anwender dann darauf, passiert erst einmal gar nichts. Eine passende Aktion muss man selbst über den Event *BackRequested* in die Wege leiten. Das eingebettete *Frame*-Objekt verfügt hierfür über die Methode *GoBack*. Allerdings muss man dann auch die Auswahl im Menübaum selbst nachziehen. Ob das auch automatisch gehen könnte, war der Dokumentation nicht zu entnehmen.

Unklar ist auch, wieso die Implementierung von *Go Back* beziehungsweise *Show Settings* so unterschiedlich gelöst ist: einmal über einen Event, das andere Mal über eine Fallunterscheidung bei der Navigation. Ob das noch auf Altlasten aus der UWP zurückzuführen ist?

Events durchreichen

Bei der Umschaltung zwischen zwei verschiedenen Ansichten gibt es naturgemäß ein paar Ereignisse, die behandelt werden sollten. Die Ansicht, die verlassen werden soll, sollte darüber benachrichtigt werden und gegebenenfalls ihr Veto einlegen können. Der passende Event hierzu ist *Frame.OnRootFrameNavigating*. Über dessen *NavigatingCancelEventArgs.Cancel*-Eigenschaft kann die laufende Navigation abgebrochen werden. Sinnvoll ist das, wenn beispielsweise ▶



Das begegnet einem bei WinUI 3 leider des Öfteren: nichtssagende Breaks und Fehlermeldungen (Bild 2)

Benutzereingaben noch nicht gespeichert worden sind, aber nicht automatisch verworfen werden können oder sollen.

Auch könnte die Ansicht, zu der navigiert wurde, darüber benachrichtigt werden. Das geschieht mithilfe des Events *Frame.OnRootFrameNavigated*. Beide Ereignisse werden im MainWindow behandelt, dort aber hauptsächlich an das betreffende ViewModel durchgereicht beziehungsweise genutzt, um dessen Eigenschaft *Title* abzurufen und für die Darstellung im Header zu nutzen (Listing 11). *OnNavigatingFrom* ist eine virtuelle Methode von *ViewModelBase* (siehe Listing 3), die bei Bedarf in einem konkreten ViewModel überschrieben werden kann.

Die Views werden im Beispiel in Form von *Page*-Klassen angelegt. Unter Umständen benötigen die ViewModels auch einige Events aus der zugehörigen Page. Im vorliegenden Fall wurden die Ereignisse *PageLoaded* und *PageUnloaded* bereits berücksichtigt. Die Verknüpfung hierzu übernimmt die Extension-Methode *SetupViewModel* (Listing 8). Bei Bedarf lassen sich auf diesem Wege weitere Eventhandler verbinden.

Im Dialog mit dem Benutzer

In Desktop-Anwendungen gibt es viele Situationen, in denen man in einem modalen Dialogfenster Eingaben vom Benutzer abfragen möchte. Während der Inhalt des jeweiligen Dialogfensters natürlich unterschiedlich sein kann, gibt es aber zumeist einige Schaltflächen wie *Ok* oder *Abbrechen*, die immer in gleicher oder ähnlicher Form angezeigt werden sollen. WinUI 3 kennt zu diesem Zweck die Methode *ContentDialog.ShowAsync*. Diese gibt ein *Awaitable* zurück, das im Zusammenhang mit *async/await* verwendet werden kann, um nachfolgenden Code erst nach erfolgter Rückmeldung des Benutzers auszuführen. Das sieht deutlich moderner aus als die altbackenen *ShowDialog*-Methoden aus Windows Forms oder WPF. Es hilft allerdings nur bedingt, da die aufrufende Methode als *async* gekennzeichnet sein muss, in vielen Fällen bei Desktop-Anwendungen aus syntaktischen Gründen aber den Rückgabetyp *void* besitzen muss und somit kein *Task*-Objekt zurückgeben kann.

Listing 12 zeigt eine abstrahierte Umsetzung in Form der Methode *ShowDialog*. Über einen Parameter vom Typ *DialogSettings* werden Einstellungen für den Titel des Dialogfensters sowie für die Darstellung der Schaltflächen mitgegeben. Der generische Typ-Parameter *TDialogView* gibt an, was als Inhalt des Dialogfensters dargestellt werden soll. *TDialogView* muss mittelbar oder unmittelbar von *FrameworkElement* abgeleitet sein. In den meisten Fällen wird man hier eine *Page*-Klasse angeben, die den Dialog inhaltlich definiert und ihrerseits mit einem geeigneten ViewModel verknüpft sein kann.

Listing 11: Vor und nach der Navigation kann Code der ViewModels ausgeführt werden

```
public sealed partial class MainWindow : Window,
    IShell
{
  ...
  // navigation to different page is requested. Can be
  // cancelled via e.Cancel
  private void OnRootFrameNavigating(object sender,
      NavigatingCancelEventArgs e)
  {
    var vm = ((sender as Frame)?.Content as Page)
      ?.DataContext as ViewModelBase;
    if (vm != null)
    {
      vm.OnNavigatingFrom(e);
    }
  }
  // navigation to view completed
  private void OnRootFrameNavigated(object sender,
      NavigationEventArgs e)
  {
    var vm = ((sender as Frame)?.Content as Page)
      ?.DataContext as ViewModelBase;
    if (vm == null)
      nvMain.Header = "no ViewModel";
    else
    {
      nvMain.Header = vm.Title;
      // call a method on the ViewModel if appropriate
    }
  }
}
```

An dieser Stelle sei auf zwei weitere Ungereimtheiten bei WinUI 3 verwiesen: Zum einen ist es erforderlich, die Eigenschaft *XamlRoot* des Dialogs auf *Content.XamlRoot* des Hauptfensters zu setzen. Was das im Hintergrund bewirkt, blieb aber bislang ein Geheimnis der Dokumentation. Vermutlich geht es darum, die Beziehungen zwischen den Fenstern über das Betriebssystem abzubilden (Parent, Child, Window-Handles …). Genaueres ließ sich aber nicht in Erfahrung bringen.

Das andere Problem ergibt sich mit der Größeneinstellung des Dialogfensters. Diese scheint keinesfalls beliebig anpassbar zu sein. Anscheinend gibt es hierbei voreingestellte Obergrenzen für Höhe und Breite. Für Tablets mag das sinnvoll sein, für Desktop-Rechner mit großen Bildschirmen eher nicht. Eine Browser-Darstellung im Dialogfenster sieht dann aus wie in Bild 3 gezeigt. Wie man die Obergrenzen umgehen kann, ließ sich ebenfalls der Dokumentation und den Microsoft-Beispielen bislang nicht entlocken.

Das Dialogfenster stößt an seine Grenzen (Bild 3)

Fazit und Ausblick

Die Frage, ob man bei der Programmierung von WinUI-3-Anwendungen auf bekannte und bewährte Konzepte wie MVVM und Depedency Injection zurückgreifen kann, lässt sich klar mit Ja beantworten. Zwar muss man an manchen Stellen tricksen, aber das ist bei WPF auch nicht immer besser lösbar. Die gezeigten Beispielumsetzungen stellen kein fertiges Framework dar – sie sollen als Inspiration dienen, wie Sie Ihr eigenes Framework für WinUI 3 gestalten könnten. Den vollständigen Code zum Beispiel finden Sie auf GitHub unter [4].

Überraschende Verhaltensweisen und Bugs, die derzeit noch nicht zufriedenstellend dokumentiert sind, sind hoffentlich vorübergehender Natur. Die .NET-Klassen, die hier zum Einsatz kommen, sind oft nur dünne Wrapper um in C++ geschriebene Methoden, die im Untergrund ihre Dienste verrichten. Abstürze auf dieser Ebene sind für .NET-Entwickler allerdings schwer nachzuvollziehen. Hier ist man von anderen Umgebungen Besseres gewöhnt.

Der Grundstein für eine WinUI-3-Anwendung ist gelegt. Jetzt gilt es, sinnvolle Ansichten zu gestalten und über die Navigation verfügbar zu machen. Für die Gestaltung der Oberfläche bietet WinUI 3 „out of the box“ wesentlich mehr als WPF und Co. Einen Einblick in diese Welt der Steuerelemente soll der nächste Artikel dieser Reihe vermitteln. ■

Listing 12: Anzeigen modaler Dialoge

```
/// <summary>
/// Show a view as a content dialog
/// </summary>
/// <typeparam name="TDialogView">type of the
/// view</typeparam>
/// <param name="settings">settings for buttons etc.
/// </param>
/// <returns>user choice</returns>
public async Task<ContentDialogResult>
    ShowDialog<TDialogView>(DialogSettings settings)
    where TDialogView : FrameworkElement, new()
{
  ContentDialog dialog = new ContentDialog()
  {
    Title = settings.Title,
    PrimaryButtonText = settings.PrimaryButtonText,
    CloseButtonText = settings.CloseButtonText,
    SecondaryButtonText =
      settings.SecondaryButtonText,
    DefaultButton = settings.DefaultButton,
  };
  dialog.XamlRoot = this.Content.XamlRoot;
  dialog.Content = new TDialogView()
  {
    MinWidth = 1000,  // does not work!
    MinHeight = 800
  };
  var result = await dialog.ShowAsync();
  // show as modal window
  return result;
}
```

[1] Joachim Fuchs, UI zu gewinnen, dotnetpro 4/2022, Seite 68 ff., www.dotnetpro.de/A2204WinUI3
[2] Joachim Fuchs, Entmystifiziert, dotnetpro 11/2021, Seite 74 ff., www.dotnetpro.de/A2111DI
[3] WinUI 3 XAML Controls Gallery, www.dotnetpro.de/SL2205WinUI3_1
[4] Sourcecode zum Beispielprogramm auf GitHub, www.dotnetpro.de/SL2205WinUI3_2

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 A2205WinUI3