Petri-Netz Editor

in WPF nach dem MVVM Entwurfsmuster

Informatikprojekt

der Hochschule Ravensburg-Weingarten im Studiengang Angewandte Informatik – Profil Spiele und digitale Medien

Eingereicht von

Steffen Koch

Matrikel-Nr: 27754

Inhalt

1 Einführung in WPF

1.1 Allgemein

Die Windows Presentation Foundation (WPF) wurde in der Version 3.0 des .NET Frameworks eingeführt. WPF ist ein modernes Programmiermodell für die Entwicklung von Windows- und Webbrowser-Anwendungen. Als UI-Gestaltungssprache wird XAML verwendet, auf welches in Kapitel 2 eingegangen wird. WPF bietet sehr viele verschiedene UI-Elemente an, wie z.B. klassische Windows-Desktop-Fenster, 2D-Grafiken, 3D-Grafiken, Dokumente, Animationen und Videos.

Außerdem werden viele Visualisierungsmöglichkeiten gegeben, darunter Farbübergänge, Verformungen, Bewegungen, Schatten- und Spiegeleffekte, Bildveränderungen und Überblendeeffekte. Dadurch kann man Elemente beliebig kombinieren, z.B. kann ein Kontrollkästchen Teil eines Textfeldes sein, welches Teil einer Liste ist.

Da die Anzeige der WPF vektorbasiert ist, bietet sie eine gute Darstellung, welche unabhängig von der Größe des Anzeigegeräts ist. Durch die Definitionen von wiederverwendbaren Styles für beliebige UI-Elemente lassen sich einheitliche Gestaltungsmerkmale auf alle Elemente anwenden, was das Trennen von UI und Verhalten fordert. WPF bietet gegenüber dem Vorgänger Windows-Forms ein wesentlich ausgeprägteres Ereignissystem und eine Abstraktion bei der Bindung von UI-Elementen an Befehle, durch das sich Commands und Texte an die UI binden lassen.

1.2 Stärken und Eigenschaften der WPF

Eigenschaft	**Beschreibung**
Flexibles Inhaltsmodell	Die WPF besitzt ein flexibles Inhaltsmodell. In anderen Programmiermodellen, wie z.B. Windows-Forms, konnten beispielsweise Buttons lediglich einen Text oder ein Bild als Inhalt enthalten. Mit dem Inhaltsmodell der WPF kann ein Button – genau wie alle anderen Elemente – einen beliebigen Inhalt haben.
Layout	Die WPF stellt viele Layout-Panels zur Verfügung, um Controls in einer Anwendung dynamisch anzuordnen und zu positionieren. Es lassen sich Layout-Panels auch beliebig ineinander verschachteln, wodurch man sehr komplexe Layouts erstellen kann.
Styles	Ein Style ist eine Sammlung von Eigenschaftswerten. Styles lassen sich einem oder mehreren Elementen der Benutzeroberfläche zuweisen, wodurch deren Eigenschaften dann die im Style definierten Eigenschaften annehmen. Ohne Styles müsste man diese auf allen Objekten setzen. Styles lassen sich auch vererben.
Trigger	Trigger erlauben es, auf deklarativem Weg festzulegen, wie ein Control auf bestimmte Eigenschaftsänderungen oder Events reagieren soll. Trigger werden meistens in einem Style oder einem Template definiert.
Lookless Controls	Custom Controls (TextBoxen, Buttons, SelectBoxen) sind bei der WPF „lookless“, das heißt, sie trennen ihre visuelle Erscheinung von ihrer eigentlichen Logik und ihrem Verhalten. Das Aussehen eines Controls wird mit einem *ControlTemplate* beschrieben. Das Control selber besitzt also kein

	Aussehen, sondern nur Logik und Verhalten. Dadurch kann man durch das Setzen der *Template*-Property das komplette Aussehen eines Controls anpassen.
Daten	Die Elemente einer WPF-Anwendung können mit DataBinding an verschiedene Datenquellen gebunden werden. Dadurch wird Code gespart, der beispielsweise die UI nach einer Änderung aktualisiert. Außerdem können durch DataTemplates das Aussehen von Daten, beispielsweise eines „Wohnort-Objektes“, auf einer Benutzeroberfläche definiert werden.

1.3 Projekt-Aufbau

Um eine WPF-Anwendung zu erstellen, legt man in Visual Studio ein neues Projekt mit dem Typen „WPF-App (.NET Framework) an. Eine WPF Applikation enthält anfangs eine *Application* und ein *Window*, welche jeweils eine XAML-Datei (.xaml) und eine Codebehind-Datei (.xaml.cs) enhalten. Es werden auch vorerst nicht sichtbare Dateien erstellt, welche jedoch für die Entwicklung nicht wichtig sind. In der App.xaml kann man festlegen, welches *Window* beim Start der *Application* geöffnet wird.

2 XAML

2.1 Einführung

Extensible Application Markup Language ist eine Beschreibungssprache zur Gestaltung von grafischen Benutzeroberflächen. XAML ist XML-basiert, was den Einstieg in XAML erleichtert. Mit XAML kann man beispielsweise Benutzeroberflächen, grafische Elemente, Verhaltensweisen, Transformationen, Animationen, Farbverläufe, Mediadateien und vieles mehr definieren. In XAML kann man Eigenschaften von Objekten, wie z.B. den Hintergrund, einfach als Attribute definieren.

Eine XAML-Datei besteht also immer aus einem Parent-Element, welches hier das Window ist.

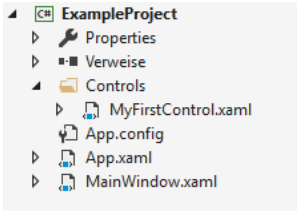
```
<Window x:Class="ExampleProject.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:ExampleProject"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

Im Window sind schon weitere Eigenschaften, wie die Höhe, Breite und einem Titel, definiert. Über die verschiedenen xmlns-Eigenschaften, kann man einen Namespace definieren (In Java: package). Der Standard-Namespace wird mit xmlns definiert, und weitere Namespaces benötigen eigene Namen, wie beispielsweise *x*, *dmc*, *local*. Indem man einen neuen Ordner im Projekt anlegt, legt man

automatisch einen neuen Namespace an. Dieser wird jedoch erst gültig, sobald auch Klassen in diesem Ordner liegen.

Beispielsweise könnte man einen neuen Namespace für Controls anlegen und in diesem ein neues Benutzersteuerelement MyFirstControl anlegen.



Wenn man nun dieses Element dem MainWindow hinzufügen will, muss man zunächst den Namespace definieren. Es wird also ein neuer Namespace mit dem Namen *controls* im *Window* definiert.

```
<Window x:Class="ExampleProject.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:ExampleProject"
        xmlns:controls="clr-namespace:ExampleProject.Controls"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
        <controls:MyFirstControl />
    </Grid>
</Window>
```

MyFirstControl wird also einfach über den Namespace und dem Namen des Controls eingefügt.

2.2 Events

Events funktionieren in XAML sehr einfach. Man muss lediglich eine Funktion im Codebehind anlegen, welche auf das jeweilige Event passt. VisualStudio bietet hier aber Abhilfe, da man sobald man das Event eingegeben hat, sofort die passende Funktion im Codebehind generieren lassen kann. Ein einfaches Klick-Event könnte man wie folgt definieren.

```
<Button Click="Button_OnClick"></Button>
```

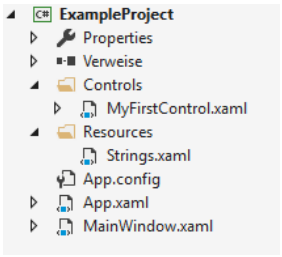
Die passende Funktion dazu würde wie folgt aussehen.

```
private void Button_OnClick(object sender, RoutedEventArgs e)
{

}
```

2.3 Ressourcen

In XAML können Ressourcen Strings, Integers, Farben, Styles und vieles weitere sein. Man kann sie in jeder XAML-Datei definieren, außerdem kann man auch mehrere Ressourcen in eine Datei schreiben, und diese später in eine weitere XAML-Datei einbinden. Beispielsweise kann man einen Ordner „Resources“ erstellen, in diesen fügt man eine Datei von Typ Ressourcenwörterbuch ein.



Ein Ressourcenwörterbuch beginnt mit einem *ResourceDictionary* Objekt, in welchem man beliebig viele Ressourcen definieren kann. Namespaces müssen wieder wie im Window eingebunden werden. Um beispielsweise zwei *Strings* zu definieren, fügt man den system-Namespace hinzu, in welchem die Klasse String liegt. Dann definiert man die beiden Strings. Es **muss** ihnen jedoch noch eine eindeutige ID zugewiesen werden.

```
<ResourceDictionary
           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
           xmlns:local="clr-namespace:ExampleProject.Resources"
           xmlns:system="clr-namespace:System;assembly=mscorlib">

    <system:String x:Key="Firstname">Max</system:String>
    <system:String x:Key="Lastname">Mustermann</system:String>

</ResourceDictionary>
```

Um diese Ressourcen nun zu nutzen, muss man zunächst das Ressourcenwörterbuch einbinden. Dies kann man entweder in jeder Datei machen, was den Zugriff auf die Ressourcen aber auch auf diese Datei beschränkt, oder man bindet es in die App.xaml, was einen globalen Zugriff erlaubt.

```
<Application x:Class="ExampleProject.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:ExampleProject"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
         <ResourceDictionary Source="Resources/Strings.xaml"></ResourceDictionary>
    </Application.Resources>
</Application>
```

Um diese Ressourcen nun zu nutzen, kann man sie mit *StaticResource* oder mit *DynamicResource* ansprechen. Wenn man DynamicResource benutzt, wird eine Änderung der Ressource auch auf die nutzenden Elemente angewandt. Bei StaticResource wird der Wert nur am Anfang geladen.

```
<TextBlock Text="{StaticResource Firstname}" />
```

2.4 Binding

In WPF kann man Eigenschaften von Objekten an Eigenschaften von anderen Objekten binden. Beispielsweise kann man den Titel von einem Window an den Text einer TextBox binden. Somit wird bei jeder Änderung des Textes auch der Titel des Windows verändert. Es lassen sich nur Dependency-Properties binden. Um eine Dependency-Property zu erstellen, benötigt man folgende Syntax.

```
public static readonly DependencyProperty TitleProperty =
            DependencyProperty.Register("Title", typeof(string),
                typeof(MyFirstControl), new FrameworkPropertyMetadata(String.Empty));

public string Title
{
       get { return (string)GetValue(TitleProperty); }
       set { SetValue(TitleProperty, value); }
}
```

In diesem Fall würde man später nur noch auf *Title* zugreifen. *TitleProperty* wird nur für das Dependency-Management verwendet. Bindings werden meistens in XAML definiert. Man gibt einfach den Namen des Elementes an, von dem man die Daten beziehen soll, dann definiert man noch die Eigenschaft, auf welche man „zielen“ will. Nur Eigenschaften, welche gebunden werden, müssen als Dependency-Property definiert sein. In diesem Fall wäre es die Title-Eigenschaft des Windows.

```
<Window x:Class="ExampleProject.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:ExampleProject"
        xmlns:controls="clr-namespace:ExampleProject.Controls"
        mc:Ignorable="d"
        Title="{Binding ElementName=MainTextBox, Path=Text}" Height="450" Width="800">
    <StackPanel>
        <TextBox x:Name="MainTextBox" />
    </StackPanel>
</Window>
```

2.5 Converter

Manchmal hat man das Problem, dass man 2 Eigenschaften aneinanderbinden will, diese aber unterschiedliche Datentypen haben. Für dieses Problem gibt es Converter in der WPF. Ein Converter ist eine Klasse, welche einen Wert in einen anderen Typ umwandeln kann. Um einen Converter zu erstellen, erstellt man einfach eine neue Klasse und implementiert das *IValueConverter* Interface. Dieses hat die beiden Funktionen Convert(object value, ..) und ConvertBack(object value, ..).

Um beispielsweise einen Converter zu implementieren, welcher eine Zahl auf 25% seiner Originalgröße konvertieren soll, würde man folgenden Code verwenden.

```
class NumberConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
                CultureInfo culture)
    {
        double valueDouble = (double) value;
        return valueDouble * 0.25;
    }

    public object ConvertBack(object value, Type targetType, object parameter,
                CultureInfo culture)
    {
        return null;
    }
}
```

Um z.B. eine TextBox in WPF immer 25% der Fenstergröße zu geben, könnte man den Converter wie folgt in XAML nutzen.

```
<Window x:Class="ExampleProject.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:ExampleProject"
        xmlns:controls="clr-namespace:ExampleProject.Controls"
        xmlns:converter="clr-namespace:ExampleProject.Converter"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        mc:Ignorable="d" x:Name="self"
        Title="{Binding ElementName=MainTextBox, Path=Text}" Height="450" Width="800">
    <Window.Resources>
        <converter:NumberConverter x:Key="MyConverter" />
    </Window.Resources>
    <StackPanel>
        <TextBox Width="{Binding ElementName=self, Path=ActualWidth,
Converter={StaticResource MyConverter}}" />
    </StackPanel>
</Window>
```

3 MVVM

Das Model-View-ViewModel-Pattern (MVVM) erlaubt eine sehr gute Trennung von UI-Design und Logik. Es ist eine moderne Variante des Model-View-Controller-Patterns (MVC). Für „größere“ Anwendungen ist MVVM sehr gut geeignet, da es sehr viele Vorteile bringt. Für kleine Anwendungen ist es nicht besonders zu empfehlen, da es einen gewissen Overhead mit sich bringt. Da das Design (Benutzeroberfläche) und die Logik (Event Handler & Co.) komplett getrennt sind, ist eine sehr einfache Zusammenarbeit zwischen Designern und Entwicklern möglich.

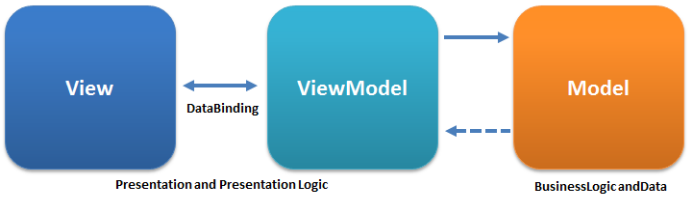


Abbildung 1 *Die Abhängigkeiten beim MVVM-Pattern*

3.1 Model

Das Model hat im MVVM-Pattern die gleiche Rolle wie im MVC-Pattern, nämlich das Kapseln der Daten, welche beispielsweise aus einer Datenbank, XML-Dateien oder anderen Datenquellen stammen können. Ein einfaches Beispiel für ein Model wäre eine Klasse für einen Angestellten.

```
class Employe
{
    public string Firstname { get; set; }
    public string Lastname { get; set; }
    public DateTime DayOfBirth { get; set; }
}
```

3.2 View

Die Aufgabe der View, das Darstellen von Daten, ist im MVVM- und MVC-Pattern auch identisch. In der View befinden sich grafische Elemente wie Buttons, TextBoxen und sonstige Controls. Die View besteht aus XAML-Code (.xaml) und einer Codebehind-Datei (.xaml.cs). Im Codebehind darf nur UI-spezifischer Code enthalten sein und keine Logik. Im XAML-Code wird die Anzeige von Daten über Bindings gelöst.

Eine View für das Anzeigen eines Angestellten könnten man wie folgt definieren.

```
<UserControl x:Class="ExampleProject.View.EmployeView"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:ExampleProject.View"
             xmlns:viewModel="clr-namespace:ExampleProject.ViewModel"
             mc:Ignorable="d" d:DataContext="{d:DesignInstance viewModel:EmployeViewModel}"
             d:DesignHeight="450" d:DesignWidth="800">
    <StackPanel>
        <Label Content="Vorname" />
        <TextBox Text="{Binding Employe.Firstname}" />
        <Label Content="Nachname" />
        <TextBox Text="{Binding Employe.Lastname}" />
        <Label Content="Geburtstag" />
        <TextBlock Text="{Binding Employe.DayOfBirth}" />
    </StackPanel>
</UserControl>
```

3.3 ViewModel

Die View nutzt als Datenquelle im oberen Beispiel eine Klasse vom Typ EmployeViewModel, welche die Logik und die Informationen für die View bereitstellt. Dazu gehören die Daten des Models, Logik und UI-nahe Informationen, beispielsweise ob ein Button aktiviert oder deaktiviert ist. Im ViewModel werden auch Benutzereingaben entgegengenommen, diese werden zwar zuerst von der View entgegengenommen, werden dann aber direkt via Data Binding an das ViewModel weitergeleitet. Ein ViewModel darf nicht von der View wissen, noch darf es grafische Elemente enthalten, sodass diese auch ohne View instanziiert werden und somit getestet werden können. Ein ViewModel für das Anzeigen eines Angestellten könnte wie folgt aussehen.

```
class EmployeViewModel
{
    public Employe Employe { get; set; }
}
```

Das ViewModel hat in diesem Fall eine Property für den aktuell ausgewählten Angestellten. Würde sich nun das ViewModel zur Laufzeit aktualisieren, würde die UI nicht sofort den neuen Employe anzeigen, da das ViewModel noch keine Information an die View sendet, dass sich die Employe Eigenschaft verändert hat. Um dies zu erreichen, muss das ViewModel das Interface *INotifyPropertyChanged* implementieren, welches dann die View informiert. Eine Implementierung würde wie folgt aussehen.

```
class EmployeViewModel : INotifyPropertyChanged
{
    private Employe _employe;
    public Employe Employe
    {
        get { return _employe; }
        set
        {
            _employe = value;
            OnPropertyChanged();
        }
    }
    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged([CallerMemberName] String propertyName = "")
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
     }
}
```

In diesem Fall würde das ViewModel die View aktualisieren. Wie man Commands in einem ViewModel implementiert, wird in Kapitel 4 erläutert.

4 DevExpress

DevExpress MVVM ist ein MVVM-Framework für WPF Anwendungen, welches von der Firma Developer Express Inc. entwickelt wird. Es erleichtert die Entwicklung nach dem MVVM-Pattern extrem, da sehr viele nützliche MVVM-Funktionalitäten genutzt werden können, darunter eine Basisklasse für ViewModels, Commands, mit denen man Funktionen an die View binden kann, Behaviors, welche das Verhalten von visuellen Controls steuern, Services und vieles mehr.

4.1 ViewModels

Das ViewModel ist für die Interaktion zwischen Model und View verantwortlich. DevExpress MVVM bietet hier viele verschiedene Basisklassen an, welche ViewModels nutzen können um Bindable-Properties, Validierung, Commands und vieles mehr zu nutzen.

4.1.1 BindableBase

Die *BindableBase* Klasse implementiert das INotifyPropertyChanged Interface, welches bereits in Kapitel 3.3 angesprochen wurde. BindableBase bietet die beiden Funktionen **GetProperty()** und **SetProperty()** an, welches für eine Bindable-Property wie folgt genutzt werden können.

```
class EmployeViewModel : ViewModelBase
{
    public Employe Employe
    {
        get { return GetProperty(() => Employe); }
        set { SetProperty(() => Employe, value); }
    }
}
```

Anderst wie in Kapitel 3.3, hat man nur eine Property, welche ohne selbst geschriebenen Code die View aktuell hält. Will man beispielsweise auf die Änderung einer Property reagieren, bietet DevExpress MVVM eine einfache gute Lösung. Der Funktion SetProperty kann auch eine CallBack-Funktion übergeben werden, welche aufgerufen wird, wenn sich die Property ändert. Dies kann man beispielsweise so implementieren.

```
class EmployeViewModel : ViewModelBase
{
    public Employe Employe
    {
        get { return GetProperty(() => Employe); }
        set { SetProperty(() => Employe, value, OnEmployeChanged); }
    }
    void OnEmployeChanged()
    {

    }
}
```

4.1.2 ViewModelBase

ViewModelBase erbt von der Klasse BindableBase und hat somit alle Funktionen, welche in Kapitel 4.1.1 erläutert wurden. Außerdem bietet sie große Unterstützung während der Entwicklung, da sie 2 Funktionen, nämlich **OnInitializeInDesignMode()** und **OnInitializeInRuntime()**, anbietet. Manchmal kommt es vor, dass ein ViewModel eine Verbindung zu einer Datenbank aufbaut und Daten bezieht. Wenn man allerdings in Visual Studio im Designer arbeitet, hat man keine Berechtigung um auf eine Datenbank zu verbinden. Für diesen Fall gibt es die Methode OnInitializeInDesignMode(), in welcher man die benötigten Daten mit Dummy-Daten füllen kann. Die richtige Verbindung kann man dann in der Funktion OnInitializeInRuntime() herstellen.

```
class EmployeViewModel : ViewModelBase
{
    public Employe Employe
    {
        get { return GetProperty(() => Employe); }
        set { SetProperty(() => Employe, value); }
    }
    protected override void OnInitializeInDesignMode()
    {
        base.OnInitializeInDesignMode();
        Employe = new Employe() {Firstname = "Max", Lastname = "Mustermann"};
    }
    protected override void OnInitializeInRuntime()
    {
        base.OnInitializeInRuntime();
        Employe = DatabaseController.GetEmploye();
    }
}
```

ViewModelBase bietet auch noch eine gute Implementierung für Commands. Ein Command ist eine Klasse, welche das Interface *ICommand* implementiert. Diese Klasse kann an die Command-Property eines Buttons gebunden werden. Die DevExpress MVVM Implementierung bietet hier aber sehr viel mehr. Ein Command besteht hier aus einer Property vom Typ *DelegateCommand*, welche das Command an sich darstellt. Außerdem gibt es zwei Funktionen, einmal die Funktion welche aufgerufen wird, wenn das Command ausgeführt wird, und einmal eine Funktion, welche angibt, ob das Command denn aktuell ausgeführt werden kann. Dies ist wichtig, denn was wäre, würde man den aktuell ausgewählten Angestellten speichern wollen, obwohl aktuell kein Angestellter ausgewählt wäre? Man müsste dies im Code abfangen. Hier können diese Funktionen Abhilfe schaffen. Man könnte eine einfache Speicherfunktion wie folgt implementieren.

```
class EmployeViewModel : ViewModelBase
{
    public Employe Employe
    {
        get { return GetProperty(() => Employe); }
        set { SetProperty(() => Employe, value); }
    }
    public ICommand SaveCommand { get; private set; }
    public EmployeViewModel()
    {
        SaveCommand = new DelegateCommand(Save, CanSave);
    }
    public void Save()
    {
        DatabaseController.SaveEmploye(Employe);
    }
    public bool CanSave()
    {
        return Employe != null;
    }
}
```

```
<Button Command="{Binding SaveCommand}" Content="Speichern" />
```

Dadurch könnte man das Command nur ausführen, wenn ein Mitarbeiter ausgewählt ist. Außerdem wäre der Button disabled, wenn kein Mitarbeiter ausgewählt wäre. Dadurch wird unnötige Logik im Codebehind gespart.

5 Umsetzung

Das Ziel des Projektes war es, einen Petri-Netz Editor zu programmieren, welcher die standardmäßigen Funktionen eines Petri-Netzes implementiert. Außerdem soll er Petri-Netze nach PNML-Standard lesen und auch als solche speichern können. PNML ist ein Standard für die XML-Speicherung von Petri-Netzen. Für das Projekt sollte außerdem WPF benutzt werden, sowie eine MVVM-Implementierung. Das Ergebnis des Projektes sieht folgendermaßen aus.

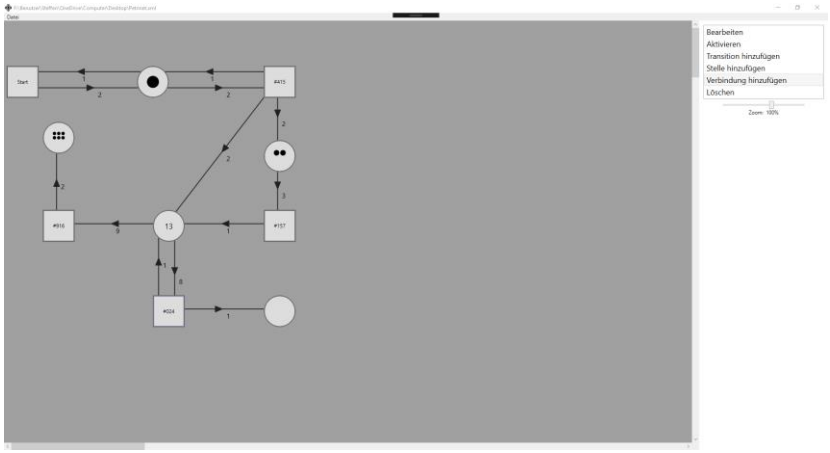


Abbildung 2: *Petri-Netz Editor Window*

5.1 MVVM

5.1.1 Model

Es gibt 5 Klassen, welche die Objekte des Petri-Netzes repräsentieren.

UIPlaceable

Als Basisklasse für Alle Objekte gibt es die Klasse *UIPlaceable*, welche die Position eines Objektes enthält, eine eindeutige Id und eine Beschreibung.

ConnectableBase

Als nächstes gibt es die Klasse *ConnectableBase*, welche von *UIPlaceable* erbt. Sie besitzt eine Liste für die Ausgänge und eine Liste für die Eingänge eines Objektes. Außerdem besitzt sie einen Namen. Die ConnectableBase-Klasse ist die Basisklasse für die Transitionen und die Stellen.

Transition

Eine Transition hat nur eine zusätzliche Property, nämlich *IsExecutable*, welche angibt, ob die Transition aktuell ausführbar ist.

Place

Die Klasse *Place* repräsentiert eine Stelle im Petri-Netz. Sie hat zusätzlich noch eine Property für den aktuellen Wert der Stelle, nämlich *Value*.

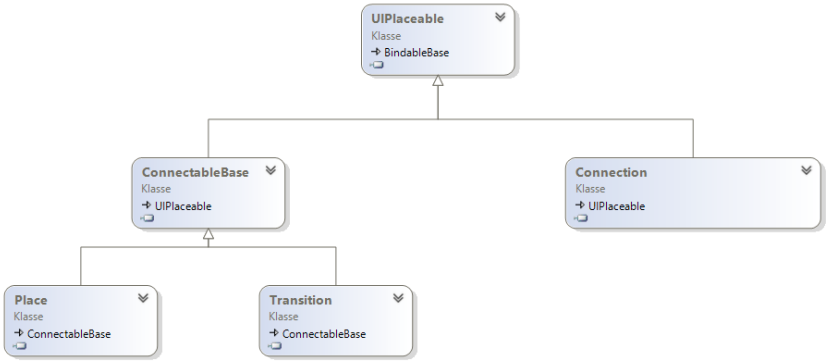


Abbildung 3: *Klassendiagramm Model*

5.1.2 View

In diesem Kapitel wird die Editor-View, welche für die Anzeige des Petri-Netzes verantwortlich ist, erläutert. Die Anzeige der View ist ein Element der Klasse *ItemsControl*, welche als Host für die Objekte dient. Die Objekte, welche in einer Liste gespeichert sind, werden über die Eigenschaft *ItemsSource* an das ItemsControl gebunden. Das sorgt dafür, dass alle Elemente, dargestellt als UI-Elemente, auf dem ItemsControl angezeigt werden. In den Resourcen des ItemsControls werden für die verschiedenen Typen (Transition, Place, Connection) sogenannte DataTemplates definiert. Dies sorgt dafür, dass die Objekte auch als individuelle UI-Objekte angezeigt werden.

```
<DataTemplate  DataType="{x:Type components:Transition}">
    <Border x:Name="MainBorder">
        <Border.Style>
            <Style TargetType="Border">
                <Style.Triggers>
                    <DataTrigger Binding="{Binding Path=DataContext.EditorMode, ElementName=self,
                                 UpdateSourceTrigger=PropertyChanged}" Value="ShowInformation">
                        <Setter Property="dd:DragDrop.IsDragSource" Value="True" />
                    </DataTrigger>
                </Style.Triggers>
            </Style>
        </Border.Style>
        <Grid>
            <controls:ConnectableButton x:Name="MainButton"  EditorMode="{Binding Path=DataContext.EditorMode,
                                        ElementName=self, UpdateSourceTrigger=PropertyChanged}" />
        </Grid>
    </Border>
</DataTemplate>
```

Da der ConnectableButton auf jede Aktion (Editieren, Ausführen, Löschen, usw.) reagieren muss, wurde sein Command je nach aktuellem Status des aktuell ausgewählten Modus, gebunden. Hier ein kleiner Ausschnitt aus den Triggern des ConnectableButtons.

```
<DataTrigger Binding="{Binding Path=DataContext.EditorMode, ElementName=self,
UpdateSourceTrigger=PropertyChanged}" Value="AddConnection">
    <Setter Property="Command" Value="{Binding Path=DataContext.AddConnectionCommand, ElementName=self}" />
    <Setter Property="CommandParameter" Value="{Binding .}" />
</DataTrigger>

<DataTrigger Binding="{Binding Path=DataContext.EditorMode, ElementName=self,
UpdateSourceTrigger=PropertyChanged}" Value="AddConnection">
    <Setter Property="Command" Value="{Binding Path=DataContext.AddConnectionCommand, ElementName=self}" />
    <Setter Property="CommandParameter" Value="{Binding .}" />
</DataTrigger>

<MultiDataTrigger>
    <MultiDataTrigger.Conditions>
        <Condition Binding="{Binding Path=IsMouseOver, RelativeSource={RelativeSource Self}}" Value="true" />
        <Condition Binding="{Binding Path=DataContext.EditorMode, ElementName=self,
UpdateSourceTrigger=PropertyChanged}" Value="Delete" />
    </MultiDataTrigger.Conditions>
    <MultiDataTrigger.Setters>
        <Setter Property="Command" Value="{Binding Path=DataContext.DeleteCommand, ElementName=self}" />
        <Setter Property="CommandParameter" Value="{Binding .}" />
        <Setter Property="IsMarkedAsDelete" Value="True" />
    </MultiDataTrigger.Setters>
</MultiDataTrigger>
```

Um die Objekte auch frei platzieren zu können, wurde dem ItemsControl ein *Canvas* als ItemsPanel gegeben. Ein Canvas ist ein Panel, auf dem Objekte frei, durch Definition der X- und Y-Koordinaten, platziert werden können.

```
<ItemsControl.ItemsPanel>
    <ItemsPanelTemplate>
        <Canvas x:Name="MainCanvas" HorizontalAlignment="Stretch" VerticalAlignment="Stretch" />
    </ItemsPanelTemplate>
</ItemsControl.ItemsPanel>
```

Um den Objekten eine freie Platzierung zu geben, musste noch der ItemContainerStyle, welcher angibt, wo die Elemente platziert werden, gesetzt werden. Dieser wird an die X- und Y-Position der Objekte gebunden.

```
<ItemsControl.ItemContainerStyle>
    <Style TargetType="ContentPresenter">
        <Setter Property="Canvas.Left" Value="{Binding Position.X,
                UpdateSourceTrigger=PropertyChanged}"/>
        <Setter Property="Canvas.Top" Value="{Binding Position.Y,
                UpdateSourceTrigger=PropertyChanged}"/>
    </Style>
</ItemsControl.ItemContainerStyle>
```

5.1.3 ViewModel

Die Logik der Anwendung besteht aus zwei ViewModels, einem ViewModel für das MainWindow, und einem ViewModel für die in Kapitel 5.1.2 genannte Editor-View. In diesem Kapitel wird jedoch nur auf die Klasse *EditorViewModel* eingegangen.

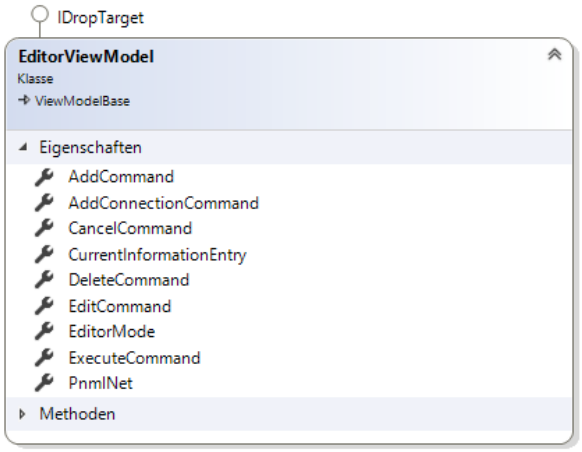


Abbildung 4: *Klassendiagramm EditorViewModel*

Neben den sechs Commands, welche für die Logik des Petri-Netzes verantwortlich sind, gibt es noch eine EditorMode-Property, welche den aktuell gewählten Modus angibt. Als mögliche Modi gibt es Editieren, Aktivieren, Transition hinzufügen, Stelle hinzufügen, Verbindung hinzufügen und Löschen. Außerdem gibt es noch die PnmlNet-Property, welche das Petri-Netz darstellt. Auf ihr befindet sich eine Liste, welche alle Objekte enthält.

Um Fehler bei den dem Hinzufügen einer Verbindung, wie z.B. das Verbinden von zwei Stellen, zu verhindern, wurde die Funktionalität von DevExpress MVVM genutzt. Da der Button jedes Objekt an das *AddConnectionCommand* gebunden ist (wenn Verbindung hinzufügen im EditorMode gewählt wurde), wird auch für jeden Button die *AddConnectionCommandCanExecute* aufgerufen. Diese hat in dem Fall als Parameter das gebundene Objekt (Transition oder Stelle). Da das Hinzufügen einer Verbindung zwei Aktionen (Start wählen und Ziel wählen) benötigt, wird die erste Aktion, das Auswählen eines Startpunktes, im ViewModel in einem Objekt der Klasse *AddConnectionHelper* gespeichert. Dadurch kann man in der CanExecute-Funktion auf die beiden Objekte zugreifen. In dieser Funktion wird dann nur noch geprüft ob die Aktion ausgeführt werden darf.

```
bool AddConnectionCommandCanExecute(ConnectableBase item)
{
    if (AddConnectionHelper.Source == item)
        return false;
    if (AddConnectionHelper.Source != null &&
        AddConnectionHelper.Source.GetType() == item.GetType())
        return false;
    if (AddConnectionHelper.Source != null &&
        AddConnectionHelper.Source.Output.Count(i => i.Destination == item) > 0)
        return false;
    return EditorMode == EditorMode.AddConnection;
}
```

Wie im Klassendiagramm zu sehen, implementiert das ViewModel die Schnittstelle *IDropTarget*, welche für das Drag&Drop Verhalten der Anwendung verantwortlich ist.
Die Schnittstelle kommt aus einer Open-Source Bibliothek, welche den Namen **gong-wpf-dragdrop** (https://github.com/punker76/gong-wpf-dragdrop) trägt. Die Schnittstelle besteht aus zwei Funktionen, nämlich Drop und DragOver, welche beide als Parameter ein Objekt vom Typ IDropInfo haben. Dieses Objekt bietet, im Kontext mit den gebundenen Objekten, eine Property für das Objekt, welches aktuell verschoben wird.

5.2 XML

Das Ziel dieses Projektes war es auch, die Daten im XML-Format zu speichern. Außerdem sollen die Dateien dem PNML-Standard entsprechen. Für die Umsetzung in das XML-Format wurden die Klassen des Namespaces **System.Xml.Serialization** verwendet.

5.2.1 PNML

Der Aufbau eines PNML-Netzes ist in XML wie folgt definiert.

```
<?xml version="1.0"?>
<pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml">
  <net id="net" type="http://www.pnml.org/version-2009/grammar/ptnet">
    <name><text>processes_0_1_min_lang.dot</text></name>
    <page id="page-0">
      <name><text>a page has no name</text></name>
      <place id="p112">
        <name><text>112</text></name>
        <initialMarking><text>0</text></initialMarking>
      </place>
      <transition id="t_6_d_1">
        <name><text>d_1</text></name>
     </transition>
      <transition id="t_0_">
        <arc id="out_t_3_a_1_p112" source="t_3_a_1" target="p112">
          <inscription><text>1</text></inscription>
        </arc>
    </page>
  </net>
```

Dadurch mussten drei Extraklassen angelegt werden. Eine für den Namen, eine für den Wert einer Stelle und einer für den Wert einer Transition. Diese wurden beispielsweise wie folgt implementiert.

```
public class PlaceInitialMarking : BindableBase
{
    [XmlElement("text")]
    public int Text
    {
        get { return GetProperty(() => Text); }
        set { SetProperty(() => Text, value); }
    }
    public PlaceInitialMarking(int value) { Text = value; }
    public PlaceInitialMarking() { }
}
```

5.2.2 Serialisierung

Die XML-Serialisierung in C# ist relativ einfach. Man muss seine Klassen nur mit Annotationen versehen. Für Attribute kann man einfach die XmlAttribute-Annotation verwenden. Für Elemente nutzt man einfach die XmlElement-Annotation.

```
[XmlType("UIPlaceable")]
[XmlInclude(typeof(Transition)), XmlInclude(typeof(Place)), XmlInclude(typeof(Connection)),
XmlInclude(typeof(ConnectableBase))]
public class UIPlaceable : BindableBase
{
    [XmlAttribute("id")]
    public string Id
    {
        get { return GetProperty(() => Id); }
        set { SetProperty(() => Id, value); }
    }

    [XmlElement("Description")]
    public string Description
    {
        get { return GetProperty(() => Description); }
        set { SetProperty(() => Description, value); }
    }
```

Für das Root-Objekt einer XML-Datei, in diesem Fall für das pnml Objekt, kann man einfach die XmlRoot-Annotation verwenden.

```
[XmlRoot("pnml", Namespace = "http://www.pnml.org/version-2009/grammar/pnml", IsNullable = false)]
public class PnmlNet : BindableBase
{

        // ...
}
```

Um die Objekte auch als XML-Dateien zu speichern und diese auch zu lesen, wird ein Objekt der Klasse *XmlSerializer* verwendet. Das Speichern eines Objektes besteht nur aus vier Zeilen Code, welche aus dem Initialisieren des Writers, dem Erstellen einer Datei, dem Serialisieren des Objektes und dem Schliessen der Datei, bestehen.

```
public void SaveToXML(PnmlNet pnmlNet, string fileName)
{
    XmlSerializer writer = new XmlSerializer(typeof(PnmlNet));
    FileStream file = File.Create(fileName);
    writer.Serialize(file, pnmlNet);
    file.Close();
}
```

Das Lesen einer XML-Datei besteht aus dem Öffnen der Datei, dem Initialisieren des Serialisierers, dem Deserialisieren des Objektes und dem Schliessen der Datei.

```
public PnmlNet ReadFromXML(string fileName)
{
    var fileStream = new FileStream(fileName, FileMode.Open, FileAccess.Read, FileShare.Read);
    XmlSerializer deserializer = new XmlSerializer(typeof(PnmlNet));
    var net =  (PnmlNet)deserializer.Deserialize(fileStream);
    fileStream.Close();
    return net;
}
```

6 Quellen

- Thomas Claudius Huber: **Windows Presentation Foundation 4.5**. Galileo Computing
- MVVM Architektur: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel#/media/File:MVVMPattern.png
- MVVM: http://www.it-designers-gruppe.de/fileadmin/Inhalte/Studentenportal/Das_Architekturmuster_MVVM__Text___1_.pdf