MVVM
Model – View – ViewModel

1 MVVM

MVVM heißt **M**odel – **V**iew – **V**iew**M**odel und ist ein Designpattern, das für WPF empfohlen wird. Es ist sehr ähnlich zu **MVC** (Model-View-Controller). Aufgrund der teilweise verschwimmenden Unterschiede wird dieses Pattern z.B. bei Angular nur noch **MVW** = Model-View-Whatever genannt.

1.1 Komponente

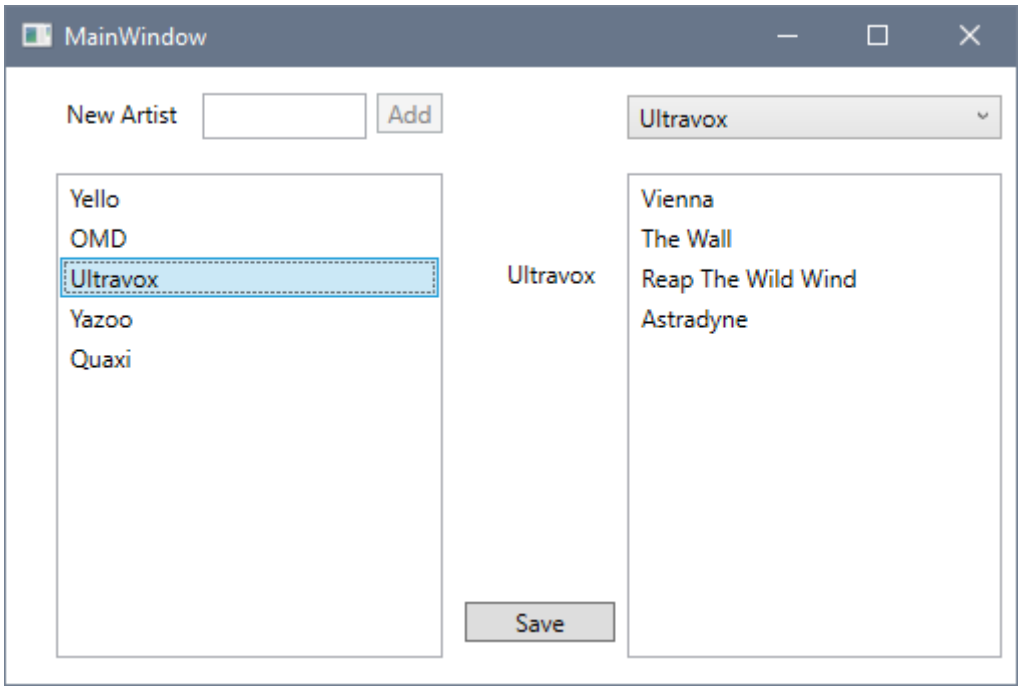
Das Grundlegende Prinzip bei MVVM ist folgendes:

- **Model**: weiß nichts vom View Model: es stellt die sogenannte Business Logic dar und kann z.B. eine Entity Framework Datenbank sein
- **ViewModel**: weiß nichts von der View: Bereitet Daten für die View auf und delegiert Business Logic an das Model. Diese Klasse ist zu programmieren und arbeitet als eine Art Adapter zw. Model und View.
- **View**: interagiert ausschließlich mit dem ViewModel und weiß nichts vom Model. Sie besteht praktisch ausschließlich aus XAML-Code und kann von einem Designer ohne Programmierkenntnisse entworfen werden.

Jedes Window soll dabei genau ein zugehöriges ViewModel haben.

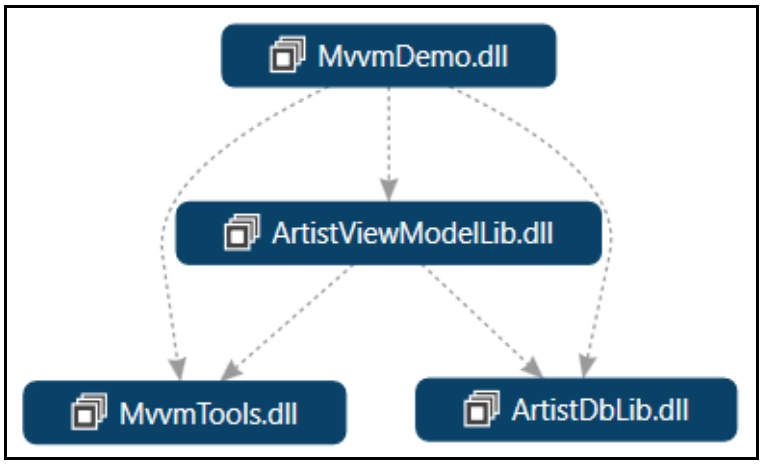
1.2 Vorlage

Als Beispiel soll eine kleine Master-Detail-App für die Anzeige von Musikbands und deren Songs programmiert werden:

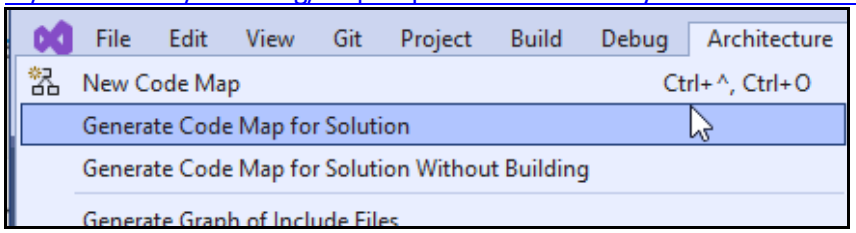


1.3 Abhängigkeiten

Am Ende wird unser Projekt folgende Teilprojekte enthalten:



Dieses Diagramm kann man mit VS Enteprise erzeugen (Details https://docs.microsoft.com/en-us/visualstudio/modeling/map-dependencies-across-your-solutions?view=vs-2022).



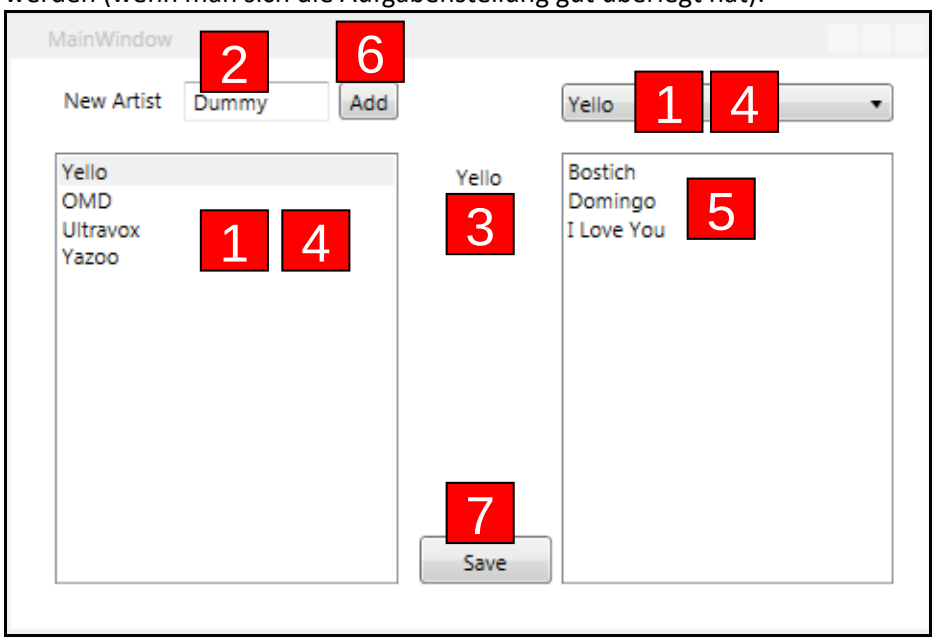
1.4 Vorgangsweise

Mit folgenden Schritten wird das für WPF erreicht:

1. View
   - Entwerfen einer (sehr einfachen) **Skizze** für das UI: Eingabefelder, Listen, Buttons,…
   - Genaues identifizieren aller Anzeigeelemente, die Daten anzeigen bzw. diese verändern sollen
   - Man muss für keines der Controls einen Namen vergeben!
2. Model
   - Wie kann ich meine Daten modellieren?
   - Entsprechendes **Datenmodell** definieren
3. ViewModel-Klasse
   - Ableiten von der Basisklasse `ObservableObject`
   - Für jedes Anzeigefeld (Label, TextBox, ListBox,…) ist eine Property zu definieren
     - incl. `NotifyPropertyChanged` (von Basisklasse)
     - Kann Liste verändert werden? Falls ja ➔ `ObservableCollection`
   - Für jeden Button ist eine ICommand-Property mit einem `RelayCommand` zu definieren
   - Business Logic an das Model delegieren
   - Da ViewModel die View nicht kennt ➔ es darf niemals direkt auf eine TextBox, Button, usw. zugreifen!
4. ViewModel mit Model **initialisieren**
5. View
   - ViewModel auf DataContext des Window setzen
   - Mit **Databinding** Anzeige an ViewModel-Properties binden
   - Mit Databinding Events an ICommand-Properties binden
6. Abschluss
   - Die View so gestalten, dass sie ansprechend aussieht
7. Unit tests
   - es muss nur das ViewModel getestet werden

2 View Erstversion

Als erster Schritt soll also die View entworfen werden. Dabei würde ich bewusst auf zeitraubendes Design mit Layouts verzichten (das sollen dann die Designer machen). Die einzelnen Controls sollen dabei **keine** Namen erhalten!
Diese Komponente stellt den **Startpunkt** der App dar. Sie sieht so aus und muss normalerweise nicht verändert werden (wenn man sich die Aufgabenstellung gut überlegt hat):



Folgende Elemente/Informationen sind enthalten:

#	Bezeichnung	Beschreibung	Anzeige	Eingabe
1	**Artists**	Liste aller Interpreten	x	
2	**NewArtist**	Eingabefeld für neuen Interpret	x	x
3	**SelectedArtist**	Ausgewählter Interpret	x	
4	**SelectedArtist**	Ausgewählter Interpret	x	x
5	**Songs**	Alle Songtitel des ausgewählten Interpreten	x	
6	**AddArtist**	Neuen Interpreten hinzufügen		x
7	**Save**	Speichern		x

Damit ist die View vorerst fertig.

3 Model

Aufgrund der Aufgabenstellung muss ein Datenmodell entworfen werden. In einer sehr einfachen Variante könnte das so aussehen:

```
public class Artist
{
  public string Name { get; set; }
  public List<Song> Songs { get; set; } = new();
  public override string ToString() => Name;
}
```

```
public class Song
{
  public string Name { get; set; }
  public Artist Artist { get; set; }
  public override string ToString() => Name;
}
```

Diese Klassen am besten in einem eigenen Unterverzeichnis/**eigener Library** erstellen.
Die Anbindung an eine Datenbank o.ä. soll für dieses einfache Beispiel nicht berücksichtigt werden. Mit Entity Framework wäre das sehr einfach möglich.
Der Einfachheit halber werden die Daten bei diesem einfachen Beispiel nicht aus einer Datenbank gelesen, sondern direkt programmiert.

```
public class ArtistContext
{
  public List<Artist> Artists { get; set; } = new();
  public List<Song> Songs { get; set; } = new();

  public ArtistContext()
  {
    InitArtistAndHisSongs("Yello", "Bostich", "Domingo", "I Love You");
    InitArtistAndHisSongs("OMD", "Enola Gay", "Electricity", "Genetic Engineering");
    InitArtistAndHisSongs("Ultravox", "Vienna", "The Wall", "Reap The Wild Wind", "Astradyne");
    InitArtistAndHisSongs("Yazoo", "Don't Go", "Nobody's Diary", "Goodbye Seventies", "Only You");
  }

  private void InitArtistAndHisSongs(string artistName, params string[] songNames)
  {
    var artist = new Artist { Name = artistName };
    Artists.Add(artist);
    var songs = songNames.Select(x => new Song { Name = x, Artist = artist });
    Songs.AddRange(songs);
    artist.Songs.AddRange(songs);
  }
}
```

4 ViewModel

MVVM basiert auf genau einem einzigen Event, nämlich **PropertyChanged**. Dieses Event wird im Interface **INotifyPropertyChanged** definiert. Daher müssen ViewModels dieses Interface implementieren.

```
namespace System.ComponentModel
{
    public interface INotifyPropertyChanged
    {
        event PropertyChangedEventHandler? PropertyChanged;
    }
}
```

Die Idee ist, dass das ViewModel immer dieses Event auslöst, wenn sich irgendeine Property ändert. Damit Konsumenten dieses Events wissen, welche Property sich geändert hat, wird der Name der geänderten Property als String in den EventArgs mitgeschickt.

4.1 Vorbereitung: ObservableObject (MvvmTools)

Da geänderte Properties über **INotifyPropertyChanged** die View aktualisieren sollen, bietet sich an, eine einfache Basisklasse zu programmieren, die dieses Interface implementiert.
Diese erwähnte Basisklasse wird oft **ObservableObject** genannt. In der Library **Prism** (stellt Tool-Klassen für MVVM zur Verfügung) wird sie z.B. auch so genannt.

```
public class ObservableObject : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler? PropertyChanged;

  protected void NotifyPropertyChanged(string propertyName)
    => PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

Da diese Klasse in vielen Projekten verwendet werden wird, sollte man sie in eine eigene Klassenbibliothek programmieren – ich habe sie MvvmTools genannt. Ihr findet sie als zip im moodle.

4.2 ViewModel-Klasse

Unser ViewModel wird jetzt von dieser Klasse abgeleitet. Zur besseren Übersicht am besten auch in einer eigenen Library. Achtung: TargetFramework auf **net6-windows** ändern!

```
public class ArtistViewModel: ObservableObject
{
}
```

Da ObservableObject aus MVVM.Tools verwendet wird, nicht vergessen, diese als Referenz hinzuzufügen.

4.2.1 Model

Das ViewModel greift ja auf das Model zu, um an Daten zu kommen. Dieses muss daher als Variable zur Verfügung stehen. Wie das Model initialisiert wird, siehe weiter unten.

```
public class ArtistViewModel : ObservableObject
{
  private ArtistContext? _db;
}
```

4.3 Properties

Für jedes der oben identifizierten Anzeigefelder ist jetzt eine Property zu definieren. Wobei zu überlegen ist:

- Muss bei einer Änderung View aktualisiert werden? Falls ja ➔ **NotifyPropertyChanged** im Setter
- Kann sich der Inhalt einer Collection ändern? Falls ja ➔ **ObservableCollection<T>** statt List<T>

Daraus ergeben sich folgende Properties:

Bezeichnung	Typ	NotifyPropertyChanged	create
Artists	ObservableCollection<Artist>		prop
NewArtist	string	x	propfull
SelectedArtist	string	x	propfull
Songs	List<Song>	x	propfull

Hinweis:

- ObservableCollection liegt im Paket **System.Collections.ObjectModel**
- Für Artists braucht man kein RaisePropertyChangedEvent, weil sich der Zeiger dieser Collection (also das Objekt selbst) nie ändert, wohl aber ihr Inhalt (daher ObservableCollection).

Daraus ergibt sich folgende Implementierung (die Definition der zugehörigen Variablen ist nicht abgebildet):

```
public ObservableCollection<Artist> Artists { get; private set; } = new();
```

```
private string _newArtist = "Dummy";
public string NewArtist
{
  get => _newArtist;
  set
  {
    _newArtist = value;
    NotifyPropertyChanged(nameof(NewArtist));
  }
}
```

```
private List<Song> _songs = new();
public List<Song> Songs
{
  get => _songs;
  set
  {
    _songs = value;
    NotifyPropertyChanged(nameof(Songs));
  }
}
```

Dabei immer die Variablen initialisieren!!!!

4.3.1 Synchronisation anderer Properties

Beim Setter von SelectedArtist ist zusätzlich zu programmieren, dass sich die Songliste (also die Property Songs) ändert:

```
private string _selectedArtist = "";
public string SelectedArtist
{
  get => _selectedArtist;
  set
  {
    _selectedArtist = value;
    Songs = _db?.Songs.Where(x => x.Artist?.Name == _selectedArtist).ToList() ?? new();
    NotifyPropertyChanged(nameof(SelectedArtist));
  }
}
```

Dabei ist wichtig, dass nicht auf die Variable **songs**, sondern auf die Property **Songs** zugewiesen wird! Grund: Im Setter wird das mehrfach erwähnte Event PropertyChanged ausgelöst, was die View zum Aktualisieren auffordert.

5 Initialisierung

Das ViewModel muss noch mit dem Model initialisiert werden. Das wird üblicherweise in der **Codebehind-Datei des Windows** programmiert.

5.1 ViewModel

Im ViewModel braucht natürlich einen entsprechenden Konstruktor:

```
public ArtistViewModel(ArtistContext db)
{
  _db = db;
  Artists = db.Artists.AsObservableCollection();
}
```

5.2 MainWindow

Im MainWindow wird nur noch das ViewModel auf die Property **`DataContext`** der Klasse **`Window`** zugewiesen, also am obersten Knoten der XAML-Struktur. Dadurch haben alle Controls dann Zugriff darauf (siehe unten).

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  var db = new ArtistContext();
  var viewModel = new ArtistViewModel(db);
  this.DataContext = viewModel;
}
```

5.2.1 Variante mit Init

Mit Method chaining kann man sich sämtliche Konstruktoren sparen (der Defaultkonstruktor wird ja automatisch vom Compiler erzeugt).

```
public ArtistViewModel() { }
public ArtistViewModel Init(ArtistContext db)
{
  _db = db;
  Artists = db.Artists.AsObservableCollection();
  return this;
}
```

Im Window-Loaded stellt man dann mit dieser Init-Methode die Verbindung zw. ViewModel und Model her:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  var db = new ArtistContext();
  DataContext = new ArtistViewModel().Init(db);
}
```

5.2.2 Dependency Injection

Auch das ist nicht ideal. Am besten löst man das Problem der Initialisierung mit Dependency Injection. Das wird aber jetzt nicht weiter besprochen.

6 View - DataContext

Nachdem das ViewModel vorbereitet wurde, kann man die View an die jeweiligen Properties und Commands binden. Dazu muss man zuerst der View mitteilen, welches Objekt das ViewModel darstellt (die View kennt das ViewModel, aber nicht umgekehrt).
Dabei hat man zwei Möglichkeiten:

6.1 DesignInstance

Dabei gibt man die Klasse, die als ViewModel dienen soll, als Attribut im <Window>-Tag an.
Syntax: `d:DataContext ="{d:DesignInstance viewmodel:ArtistViewModel}"`

```
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:MvvmDemo"
xmlns:viewmodel="clr-namespace:ArtistViewModelLib;assembly=ArtistViewModelLib"
d:DataContext ="{d:DesignInstance viewmodel:ArtistViewModel}"
mc:Ignorable="d"
Loaded="Window_Loaded"
```

Das Präfix **`viewmodel`** braucht man, wenn das ViewModel in einem eigenen Verzeichnis bzw. Library liegt (was empfohlen wird).

6.1.1 IsDesignTimeCreatable

Durch die Angabe einer DesignInstance für den DataContext wird man im XAML-Editor mit Intellisense unterstützt. Möchte man zusätzlich auch die aktuellen Werte des ViewModels zur Entwicklungszeit sehen, muss man festlegen, dass die Entwicklungsumgebung zu diesem Zeitpunkt ein Objekt des ViewModels erstellen darf (z.B. dass „Dummy“ für NewArtist angezeigt wird). Das geht über das Attribut IsDesignTimeCreatable:
`d:DataContext ="{d:DesignInstance viewmodel:ArtistViewModel, IsDesignTimeCreatable=True}"`
In diesem Fall muss dann aber ein Default-Konstruktor vorhanden sein – man wird aber im Editor mit einer Fehlermeldung darauf hingewiesen.

6.2 Window.DataContext

Eine andere Variante wäre, eine Instanz auf die Property DataContext des <Window>-Tags zu setzen:

```
        xmlns:local="clr-namespace:MvvmDemo"
        xmlns:viewmodel="clr-namespace:ArtistViewModelLib;assembly=ArtistViewModelLib"
        mc:Ignorable="d"
        Loaded="Window_Loaded"
        Title="MainWindow" Height="450" Width="800">
    <Window.DataContext>
        <viewmodel:ArtistViewModel/>
    </Window.DataContext>
    <Grid>
```

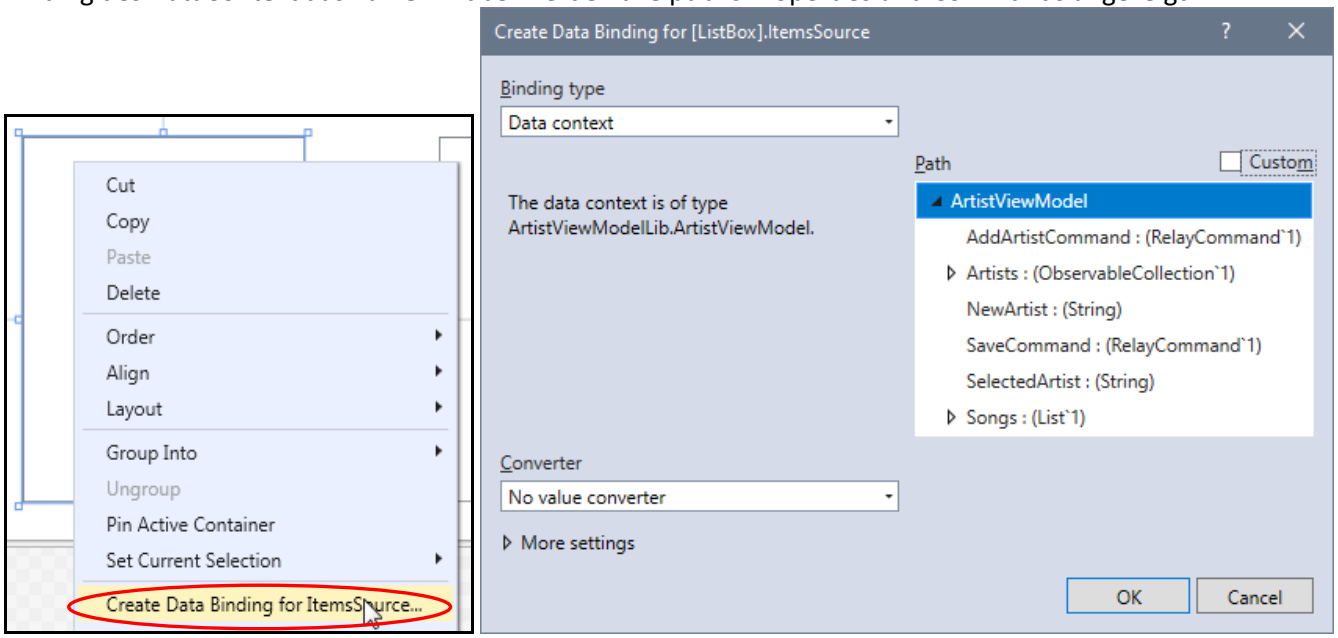
Für das Präfix **`viewmodel`** gilt das selbe wie bei DesignInstance. In diesem Fall muss die ViewModel-Klasse aber einen Default-Konstruktor haben.

7 View - Databinding

Bei der Zuweisung des Databinding hat man mehrere Möglichkeiten.

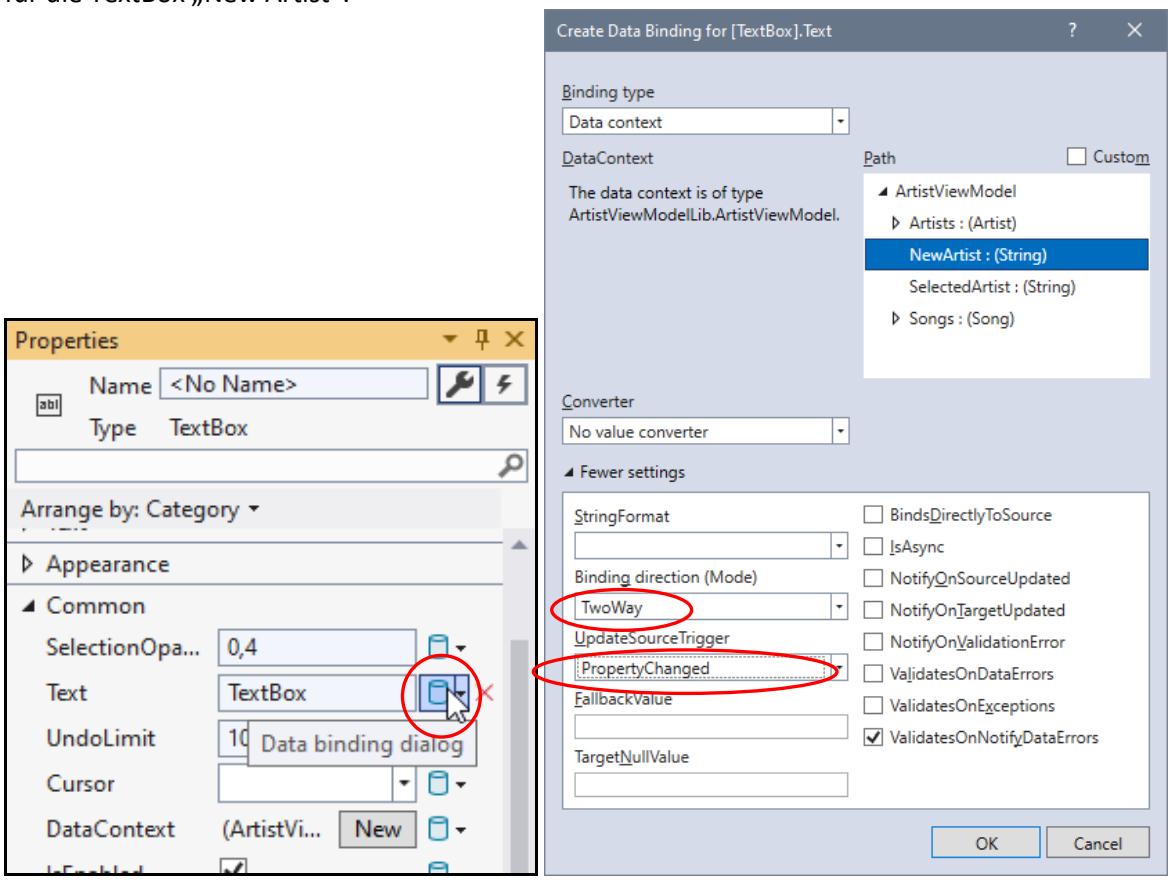
7.1 Kontextmenü

Über das Kontextmenü des Designers kann man für die Content-Property (also ItemsSource für eine ListBox) ein Binding des DataContext auswählen. Dabei werden alle public Properties und Commands angezeigt:



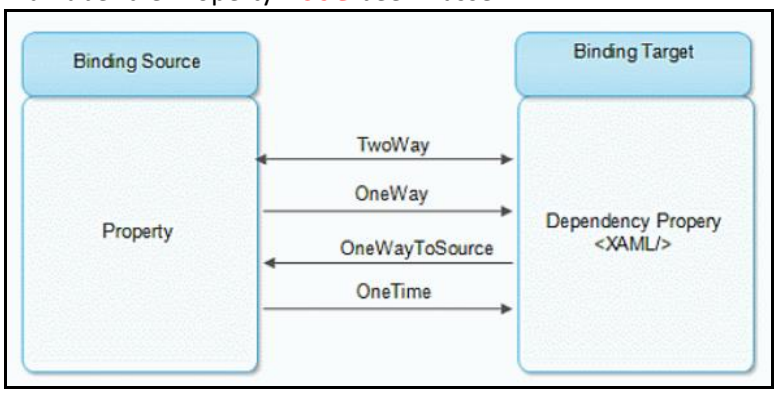
7.2 Properties-Window

Auch im Properties-Window kann das oben gezeigte Databinding-Fenster öffnen und eine Property zuweisen, z.B. für die TextBox „New Artist“:



7.2.1 Mode

Es ist noch interessant, ob das Synchronisieren automatisch auch in der **Rückrichtung** funktioniert, dass man also sowohl die Quelle als auch das Ziel ändern kann und trotzdem die Werte synchron gehalten werden. Das kann man über die Property **`Mode`** beeinflussen:



7.2.2 UpdateSourceTrigger

Eine Frage stellt sich noch, vor allem bei der Eingabe von Werten in eine TextBox: **Wann** wird der neue Wert übernommen? D.h.: Welche Aktion löst das Update aus?
Auch das kann man einstellen, die entsprechende Property heißt **`UpdateSourceTrigger`** und kann folgende Werte annehmen:

- **`PropertyChanged`**: sobald sich der Wert der Property ändert
- **`LostFocus`**: wenn man das Control verlässt
- **`Explicit`**: manuelles Auslösen des Events (wird hier nicht weiter besprochen)

7.3 XAML

Bei beiden Varianten werden Attribute im XAML eingefügt, die man natürlich auch direkt eingeben könnte.

```
<TextBox HorizontalAlignment="Left" VerticalAlignment="Top"
         Height="23" Margin="98,13,0,0" Width="82"
         Text="{Binding NewArtist, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" />
```

Wie schon öfter erwähnt empfiehlt es sich, das Binding-Attribut in eine eigene Zeile zu schreiben.

7.3.1 IsSynchronizedWithCurrentItem

Bei der ListBox wird der Inhalt (=ItemsSource) and die Artists gebunden, das aktuell ausgewählte Element (=SelectedItem) an die Property SelectedArtist.

```
<ListBox HorizontalAlignment="Left" VerticalAlignment="Top"
         Height="242" Width="193" Margin="25,53,0,0"
         ItemsSource="{Binding Artists}"
         IsSynchronizedWithCurrentItem="True"
         SelectedItem="{Binding SelectedArtist, Mode=TwoWay}"/>
```

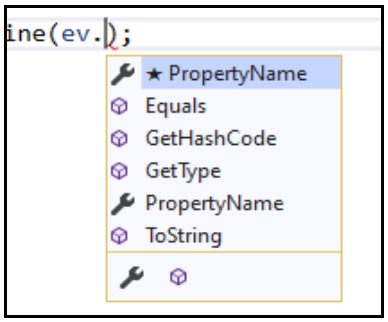
Damit bei Änderung von SelectedArtist auch in der ListBox bzw. ComboBox das richtige Element markiert wird, muss **`IsSynchronizedWithCurrentItem`** auf true gesetzt werden!
Die zweite ListBox bzw. die ComboBox sind analog zu konfigurieren.

7.4 Überprüfen PropertyChanged

Zum besseren Verständnis kann man sich einen Listener auf das eingangs erwähnte einzige relevante Event **`PropertyChanged`** registrieren.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  var db = new ArtistContext();
  var viewModel = new ArtistViewModel().Init(db);
  DataContext = viewModel;
  viewModel.PropertyChanged += (_, e) => System.Console.WriteLine(e.PropertyName);
}
```

Das Event hat auch wirklich nur eine einzige Property, nämlich den Namen der geänderten Property:



8 Commands

Für jedes Benutzer-Event (Button-Klick, Menü-Auswahl, …) ist bei MVVM ein Command zu definieren. Commands müssen das Interface **ICommand** implementieren, das zwei Methoden erzwingt (https://msdn.microsoft.com/de-de/library/system.windows.input.icommand(v=vs.110).aspx):

Bezeichnung	Typ
`CanExecute(Object)`	Defines the method that determines whether the command can execute in its current state.
`Execute(Object)`	Defines the method to be called when the command is invoked.

Wie wir weiter unten sehen werden, ist WPF für die Verwendung von Commands designt.

- Liefert **CanExecute()** false, wird ein Button inaktiv.
- Klickt man auf einen Button, der an ein Command gebunden ist, wird die Methode **Execute()** des Commands aufgerufen.

8.1 RelayCommand (MvvmTools)

Damit man nun nicht für jeden Button eine neue Klasse erzeugen und von ICommand ableiten muss, wird empfohlen, eine entsprechende allgemeine Basisklasse **RelayCommand** zu erzeugen. Wie für ObservableObject gilt auch hier: das gehört in eine Library (am besten die gleiche, wie ObservableObject).
Diese Klasse muss nicht programmiert werden – sie ist in MvvmTools bereits enthalten.

```
public class RelayCommand<T> : ICommand
{
  private readonly Action<T?> _execute;
  private readonly Predicate<T?>? _canExecute;

  public RelayCommand(Action<T?> execute) : this(execute, null)
  { }

  public RelayCommand(Action<T?> execute, Predicate<T?>? canExecute)
  {
    _execute = execute ?? throw new ArgumentNullException(nameof(execute));
    _canExecute = canExecute;
  }

  public void Execute(object? parameter)
  {
    if (parameter == null) _execute(default);
    else _execute((T)parameter);
  }
  public bool CanExecute(object? parameter) =>
                _canExecute == null
            || (parameter == null ? _canExecute(default) : _canExecute((T)parameter));

  public event EventHandler? CanExecuteChanged
  {
    add => CommandManager.RequerySuggested += value;
    remove => CommandManager.RequerySuggested -= value;
  }

}
```

8.2 Verwendung

Mit dieser Vorbereitung können die Command-Properties sehr einfach programmiert werden (**ICommand** liegt in der Assembly **System.Windows.Input**):

```
public ICommand AddArtistCommand => new RelayCommand<string>(
  DoAddArtist,
  x => NewArtist.Trim().Length > 0
);
private void DoAddArtist(string obj)
{
  Artists.Add(new Artist { Name = NewArtist });
  NewArtist = "";
}
```

Wann ist der Aufruf erlaubt?

Oder kürzer:

```
public ICommand AddArtistCommand => new RelayCommand<string>(
  _ => { Artists.Add(new Artist { Name = NewArtist }); NewArtist = ""; },
  x => NewArtist.Trim().Length > 0
);
```

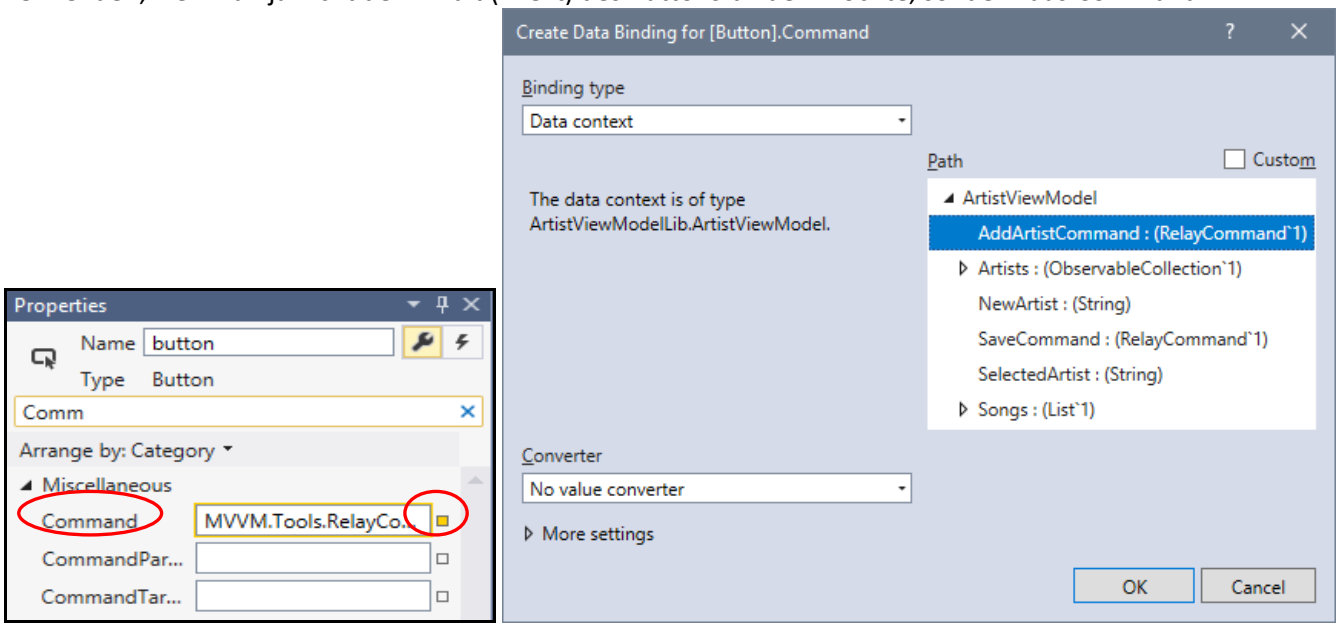
Analog für Save (der zweite Parameter kann fehlen, dann ist der Aufruf immer erlaubt):

```
public ICommand SaveCommand => new RelayCommand<string>(Save);
private void Save(string obj)
{
  //save the data to the database
  Console.WriteLine("Saving...");
}
```

Der AddArtist-Button soll nur dann aktiv sein (=CanExecute retourniert true) wenn NewArtist Zeichen enthält.
Hinweis: Man beachte, dass nicht die TextBox überprüft wird, sondern nur die Property! **Ein ViewModel darf die View nicht kennen**!
Die eigentlichen Execute-Callbacks der RelayCommands führen dann die Business Logic aus. Dieser soll wenn möglich an das Model delegiert werden.

8.3 Binding

Ähnlich funktioniert die Konfiguration der Klick-Events. Dabei darf muss man aber die Property `Command` verwenden, weil man ja nicht den Inhalt (=Text) des Buttons binden möchte, sondern das Command.



Dabei entsteht folgender Code im XAML, den man somit auch direkt eintragen könnte:

```
<Button Content="Add"
        HorizontalAlignment="Left" Margin="185,13,0,0" VerticalAlignment="Top" Width="33"
        Command="{Binding AddArtist, Mode=OneWay}" />
```

9 View ansprechend gestalten

Jetzt könnte man die View noch seinen eigenen Ansprüchen entsprechend verschönern bzw. zumindest mit Layouts vernünftig organisieren.
Man beachte: Sobald die Schnittstelle = Properties u. Commands des ViewModels definiert ist, kann sich parallel zur Programmierung ein Designer um das Aussehen des UI kümmern.