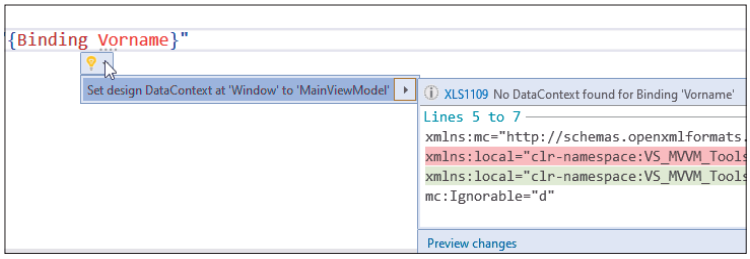
MVVM-CODEGENERATOR IN VISUAL STUDIO 2019

Fast übersehen

Visual Studio bietet nun Funktionen, um vom XAML-Code aus Code in den ViewModels zu generieren. Das kann eine Menge Tipparbeit sparen.

Bei der Fülle an Neuerungen, die kontinuierlich in Microsofts Entwicklungsumgebung einfließen, werden kleine Details leicht übersehen. Seit Version 16.9 unterstützt Visual Studio die automatische Code-Generierung aus dem XAML-Editor heraus, die auf eine typische Model-View-ViewModel-Architektur (MVVM) abzielt. Damit lassen sich auf Basis von Binding-Ausdrücken Properties und Commands automatisch in einem ViewModel generieren. Die Unterstützung findet man für WPF, UWP und Xamarin.Forms-Anwendungen.



Visual Studio kann den benötigten Eintrag für den Designtime-DataContext automatisch erstellen (Bild 1)

Microsoft hat diese Information gut versteckt in den Anmerkungen zu Version 16.9 unter dem Titel „MVVM Tooling“ in zwei Sätzen zusammengefasst [1]. Erwartet man die angebotene Unterstützung im XAML-Editor nicht, wird man sie vermutlich auch kaum wahrnehmen. Hätte mich nicht der aufmerksame Chefredakteur eines bekannten deutschen .NET-Magazins auf den in [2] genannten Artikel hingewiesen, wäre mir die Neuerung gar nicht aufgefallen und diesen Artikel gäbe es nicht. Werfen wir also mal einen Blick auf diese versteckten Gimmicks, und zwar aus der Perspektive eines WPF-Programmierers.

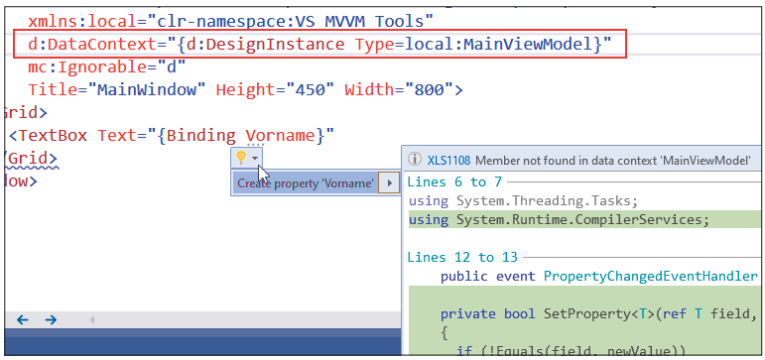
Vorbereitende Maßnahmen

Der neue Automatismus benötigt eine syntaktische Verbindung zwischen XAML-Code und einem geeigneten ViewModel. Je nach Technologie kann diese unterschiedlich ausfallen. Bei WPF reicht es aus, ein ViewModel direkt der *DataContext*-Eigenschaft zuzuweisen, etwa der eines Windows im XAML-Code. In vielen Fällen wird das allerdings nicht möglich sein, da Visual Studio dann das ViewModel auch zur Design-Zeit für die Darstellung im Designer instanzieren muss. Wenn das aber wegen externer Abhängigkeiten (Datenbankverbindung, Internetzugang und andere) nicht geht, kann man auch auf den aus Blend bekannten *DesignTime-DataContext* zurückgreifen. Dieser wird nur vom Blend- oder Visual-Studio-Designer benutzt und zur Laufzeit ignoriert.

Als minimale Voraussetzung muss die ViewModel-Klasse das Interface *INotifyPropertyChanged* implementieren. Ist eine solche Klasse bereits vorhanden, lässt sich die Verknüpfung im XAML-Code mit einem Klick erzeugen (Bild 1). Ein weiterer Mausklick generiert dann die Eigenschaft (Bild 2). Im Vorschaufenster des Tools werden die sich ergebenden Änderungen visualisiert. Der generierte Code enthält die priva-

Listing 1: Generierter Code

```
// Der generierte Code enthält die Methode
// SetProperty sowie die gewünschte
// Eigenschaft (hier Vorname)
class MainViewModel : INotifyPropertyChanged {
  public event PropertyChangedEventHandler
      PropertyChanged;
  private bool SetProperty<T>(ref T field, T newValue,
      [CallerMemberName] string propertyName = null)
  {
    if (!Equals(field, newValue)) {
      field = newValue;
      PropertyChanged?.Invoke(
          this, new PropertyChangedEventArgs(
          propertyName));
      return true;
    }
    return false;
  }
  private string vorname;
  public string Vorname {
    get => vorname;
    set => SetProperty(ref vorname, value);
  }
}
```



Ein Klick, und die Property wird im ViewModel erstellt (Bild 2)

te Methode *SetProperty*, die fortan zum Setzen automatisch erzeugter Eigenschaften verwendet wird (Listing 1). Die Property (hier *Vorname*) wird in C#-6-Syntax deklariert, definiert ein privates Feld und greift in Getter und Setter darauf zu.

Mit dieser Vorgehensweise lassen sich nun beliebige Eigenschaften automatisch generieren, sobald man im XAML-Code die entsprechenden Binding-Ausdrücke formuliert. Dabei werden die Typen der Properties entsprechend den Datentypen der Bindungsziele, also der Dependency Properties auf der linken Seite des Bindings, übernommen. Für die *Text*-Eigenschaft einer TextBox erhält die Property den Typ *String*, für die *Content*-Eigenschaft eines Buttons analog dazu den Typ *Object*. Diese Typen muss man bei Bedarf später von Hand im ViewModel anpassen.

Spätestens beim zweiten ViewModel ist es störend, dass erneut die Methode *SetProperty* generiert wird. Das lässt sich aber leicht ändern. Verlagert man die Implementierung in eine Basisklasse, wie in Listing 2 zu sehen, und ändert man die Sichtbarkeit entsprechend, dann erkennt der Automat dies und unterlässt eine erneute Implementierung.

Alles, was mittelbar oder unmittelbar *INotifyPropertyChanged* implementiert, wird von Visual Studio als potenzieller Kandidat für das Setzen des *DataContext* angesehen. Das führt schnell zur Verwirrung, da im Tool-Fenster neben dem Menüeintrag zum Anlegen der Property auch noch angeboten wird, den *DataContext* neu zu setzen (Bild 3). Und dort werden alle diese Klassen aufgelistet. Hier ist mal wieder ein Tunnelblick gefragt, um nicht versehentlich den Bindungskontext zu wechseln.

Auf Kommando

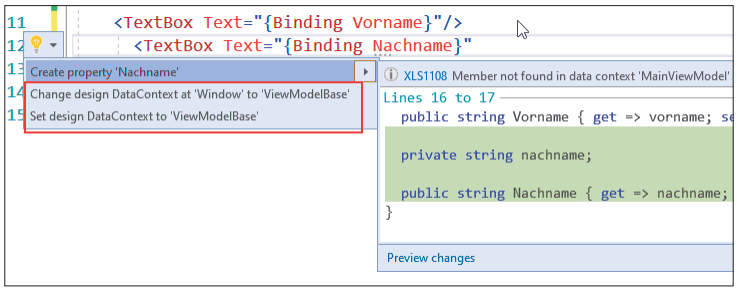
Commands werden in diesem Zusammenhang ebenfalls unterstützt. Wieder sucht der Automat nach Klassen, die eine bestimmte Schnittstelle implementieren. In diesem Fall ist es *ICommand*. Hat man eine solche Klasse (vergleiche *ActionCommand* in Listing 3) ins Projekt eingebunden, kann sie automatisch verwendet werden. Sobald man die *Command*-Eigenschaft eines Buttons oder eines Menu Items bindet, wird das Erzeugen des passenden ViewModel-Codes angebo- ▶

Listing 2: SetProperty in einer Basisklasse

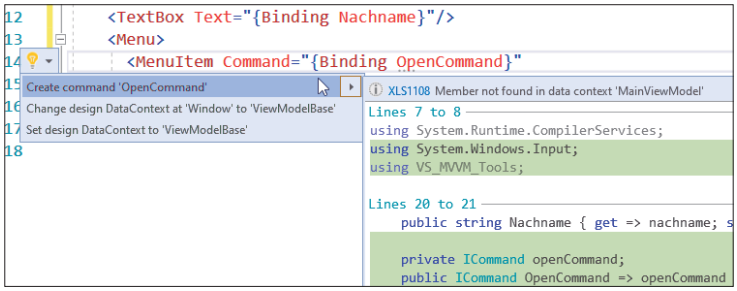
```
// Die Implementierung von SetProperty in einer Basis-
// klasse wird vom Generator erkannt und verwendet.
class MainViewModel : ViewModelBase
{
  private string vorname;
  public string Vorname { get => vorname;
    set => SetProperty(ref vorname, value); }
  private string nachname;
  public string Nachname { get => nachname;
    set => SetProperty(ref nachname, value); }
}

class ViewModelBase : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler
      PropertyChanged;
  protected bool SetProperty<T>(ref T field,
      T newValue,
      [CallerMemberName] string propertyName = null)
  {
    if (!Equals(field, newValue)) {
      field = newValue;
      PropertyChanged?.Invoke(this,
        new PropertyChangedEventArgs(propertyName));
      return true;
    }
    return false;
  }
}
```

Listing 3: Einfache Implementierung von ICommand

```
class ActionCommand : ICommand {
  private readonly Action action;
  public ActionCommand(Action action) {
    this.action = action;
  }
  private bool isEnabled;
  public bool IsEnabled {
    get { return isEnabled; }
    set { isEnabled = value; }
  }
  public event EventHandler CanExecuteChanged;
  public bool CanExecute(object parameter) {
    return isEnabled;
  }
  public void Execute(object parameter) {
    action?.Invoke();
  }
}
```



Kann verwirrend werden: Alles, was INotifyPropertyChanged implementiert, wird als Alternative für den DataContext vorgeschlagen (Bild 3)



Auch Commands lassen sich in WPF generieren (Bild 4)

ten (Bild 4). Den generierten Code sehen Sie in Listing 4. Er ist im Rahmen einer .NET-5-Anwendung entstanden, die auf C# 8 aufsetzt. In .NET-Framework-4.x-Programmen wird die klassische Form für die Code-Generierung verwendet, die Sie in Listing 5 sehen.

Sollte es mehrere Klassen im Projekt geben, die *ICommand* implementieren, dann ist unter Umständen wieder Handarbeit gefragt. Visual Studio bietet beim Anlegen der Command-Eigenschaften keine Auswahl an, sondern entscheidet sich selbst für irgendeine der *ICommand*-Implementierungen. Wenn das regelmäßig die falsche ist, dann ist der nachträgliche Aufwand möglicherweise größer als der vermeintliche Nutzen.

DataTemplates und Styles

Ob die Automatismen auch, wie in [2] beschrieben, innerhalb von *DataTemplate*- und *Style*-Definitionen funktionieren, scheint von der Technologie abzuhängen. In WPF sieht es bislang nicht so aus, als könnte Visual Studio die benötigten Properties in den jeweiligen Datenklassen generieren, auch wenn die Datentypen zur Entwurfszeit bekannt sind und IntelliSense sie berücksichtigt. Anders sieht es offenbar im Fall des in [2] beschriebenen Beispiels in Xamarin.Forms aus. Unter der Überschrift „Code Generation for Models“ ist dort zu sehen, dass die Tools im XAML-Editor von Xamarin durchaus aktiv werden und die Code-Generierung unterstützen.

Listing 4: Von VS generierte Command-Property

```
class MainViewModel : ViewModelBase
{
  ...
  private ICommand openCommand;
  public ICommand OpenCommand =>
    openCommand ??= new ActionCommand(Open);
  private void Open() { ... }
}
```

Listing 5: Klassische Command-Property

```
private ICommand cancelCommand;
public ICommand CancelCommand
{
  get {
    if (cancelCommand == null) {
      cancelCommand =
        new MyActionCommand(Cancel);
    }
    return cancelCommand;
  }
}
private void Cancel() { ... }
```

Fazit

Es hängt stark vom Workflow des Entwicklers ab, wie groß der Nutzen der neuen XAML-Unterstützung in Visual Studio ist. Wer oft mit XAML-Code beginnt und erst dann das zugehörige ViewModel programmiert, kann hier profitieren. Zwar muss der generierte Code vermutlich in den meisten Fällen angepasst werden, kann aber durchaus eine hilfreiche Grundlage sein. Für diejenigen, die erst die ViewModels bauen und danach die Oberfläche designen, ist der Nutzen deutlich geringer. Gefällt einem der generierte Code nicht, dann schaut man ebenfalls in die Röhre. Ändern lassen sich die Vorlagen bislang nicht, aber vielleicht habe ich das ja auch übersehen und ein aufmerksamer Leser weist mich darauf hin. ■

[1] What's New in Visual Studio 2019 version 16.9, www.dotnetpro.de/SL2110Xaml1

[2] Code Generation from XAML in Visual Studio is Mind-blowing Awesome, www.dotnetpro.de/SL2110Xaml2

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 A2110Xaml