

Bernhard Baltes-Götz und Paul Frischknecht

Einführung in Avalonia UI 12

2026 (Rev. 261109)

Copyright © 2026, Bernhard Baltes-Götz & Paul Frischknecht



Dieses Manuskript steht unter der **Creative Commons License BY-NC-SA 4.0 International**.

Lizenzmerkmale:

- Namensnennung
- Nicht-kommerziell
- Weitergabe unter gleichen Bedingungen

Link zur Lizenz: https://creativecommons.org/licenses/by-nc-sa/4.0/deed.de

Die wichtigsten Lizenzbedingungen:

- **Lizenzgewährung**
  Das Manuskript darf ganz oder in Teilen, unverändert oder modifiziert verwendet, vervielfältigt und weitergegeben werden, aber nur für nicht-kommerzielle Zwecke.
- **Namensnennung**
  Bei einer Weitergabe des Manuskripts (auch in veränderter Form) müssen die folgenden Angaben beibehalten werden:
  - Die Namen der Autoren
  - Die Copyright-Angabe
- **Weitergabe unter gleichen Bedingungen**
  Eine veränderte Version des Manuskripts muss unter einer Creative-Commons - Lizenz der vorliegenden oder einer späteren Version mit den gleichen Lizenzelementen oder unter einer BY-NC-SA - kompatiblen Lizenz stehen.

Jede andere Nutzung erfordert eine Genehmigung durch die Autoren (Kontakt: baltes@bebagoe.de).

Vorwort

Obwohl die ***W**indows **P**resentation **F**oundation* (***WPF***) als Basis für kommerzielle, industrielle und wissenschaftliche .NET - Desktopanwendungen außerordentlich erfolgreich ist, bestehen relevante Einschränkungen:

- WPF ist nur unter Windows verwendbar, was die Förderung von Open Source – Betriebssystemen behindert. Microsoft bietet seit .NET 6 mit ***MAUI*** (***M**ulti-platform **A**pp **UI***) eine Multiplattform-Lösung an, unterstützt dabei aber weder Linux noch die Ausführung im Browser.
- Die Abhängigkeit von einer einzigen, außerhalb von Europa ansässigen Software-Firma ist im Hinblick auf die europäische IT-Souveränität nicht optimal.

Mit Avalonia UI ist in Europa eine WPF-Weiterentwicklung mit einer erstklassigen Unterstützung für Linux entstanden, die auch alle anderen signifikanten Plattformen berücksichtigt (macOS, Windows, Android, iOS, Browser). Für die zahlreichen Entwickler[1] mit WPF-Erfahrung bietet Avalonia UI neue Optionen ohne große Migrationsprobleme.

Avalonia UI basiert auf der von Microsoft entwickelten, mittlerweile von der .NET Foundation[2] getragenen und mit einem liberalen Open-Source – Lizenzmodell (MIT) ausgestatteten .NET – Technologie. Microsoft wird diese Technologie sicher engagiert weiterentwickeln. Dass die Community im Bedarfsfall Lücken erfolgreich schließen kann, hat die Entwicklung der plattformunabhängigen Open Source - Alternative **Mono**[3] zum damals noch auf Windows eingeschränkten .NET Framework bewiesen.

Das Manuskript behandelt **Avalonia UI 12** auf der Basis von **C# 14** und **.NET 10**. Über die vorausgesetzten Basistechniken informieren die ersten 15 Kapitel im kostenlosen E-Book von Baltes-Götz & Frischknecht (2026).

Als Entwicklungsumgebung nutzen wir das Visual Studio 2026 in der Community Edition. Die im Manuskript verwendete Software ist frei verfügbar, wobei die kostenlose Nutzung des Avalonia-Plugins für das Visual Studio auf nicht-kommerzielle Projekt eingeschränkt ist.

Die aktuelle Version dieses Manuskripts ist zusammen mit den behandelten Beispielen und Lösungsvorschlägen zu Übungsaufgaben hier zu finden:

https://bebagoe.de/avalonia/

Hinweise auf Fehler und Mängel im Text werden unter der Mail-Adresse

baltes@bebagoe.de

dankbar entgegengenommen.

Trier (D) und Aadorf (CH), im November 2026 Bernhard Baltes-Götz & Paul Frischknecht

---

[1] Zur Vermeidung von sprachlichen Umständlichkeiten verwenden wir in diesem Manuskript meist die männliche Form. Trotz großer Sympathie für das Ziel einer geschlechtsneutralen Sprache scheint uns gegenwärtig die männliche Form das kleinere Übel zu sein.

[2] https://dotnetfoundation.org/

[3] http://www.mono-project.com/, https://en.wikipedia.org/wiki/Mono_(software)

Inhaltsverzeichnis

VORWORT III

1 GUI-PROGRAMMIERUNG MIT AVALONIA UI 1

1.1 Besonderheiten von GUI-Anwendungen 1
1.1.1 Vergleich zwischen GUI- und Konsolenanwendungen 1
1.1.2 Steuerelemente im Vergleich zu anderen Objekten 3

1.2 WPF als richtungsweisende GUI-Technologie der .NET - Plattform 3

1.3 Avalonia UI als modernisiertes Multiplattform WPF 4

1.4 Alternative GUI-Technologien für .NET - Software 7

2 ERSTELLUNGSWERKZEUGE UNTER WINDOWS 11

2.1 .NET SDK 11

2.2 Microsoft Visual Studio Community 2026 für Windows 14

2.3 Visual Studio - Erweiterung für Avalonia UI 15

2.4 Erstellungswerkzeuge erkunden 17
2.4.1 Projekt anlegen 18
2.4.2 Erwerb einer Community-Lizenz 21
2.4.3 XAML-Vorschau und Dateien im Erstellungsordner 22
2.4.4 Bedienoberfläche entwerfen 23
2.4.4.1 Arbeitshilfen und angestrebtes Ergebnis 24
2.4.4.2 Eigenschaften des Anwendungsfensters 25
2.4.4.3 Label mit der Beschriftung "DM:" 26
2.4.4.4 Wechsel zum horizontalen Layout des GUI-Editors 27
2.4.4.5 TextBox zur Aufnahme des DM-Betrags 29
2.4.4.6 Label mit der Beschriftung "Euro:" 30
2.4.4.7 Label-Steuerelement mit Rahmen zur Anzeige des Euro-Betrags 30
2.4.4.8 Button-Steuerelement zur Anforderung der Konvertierung 31
2.4.5 Behandlungsmethode zum Click-Ereignis des Befehlsschalters erstellen 33
2.4.6 Fokuserwerb durch die TextBox beim Start 34
2.4.7 Programm erstellen und starten 36
2.4.8 Projektdatei 38
2.4.8.1 Ausgabetyp 39
2.4.8.2 Zielframework 40
2.4.8.3 Assemblyname 41
2.4.8.4 Weitere Einstellungen 41
2.4.9 Avalonia aktualisieren 42

2.5 Dokumentation und Quellcode 43
2.5.1 Beschreibungen und Anleitungen 43
2.5.2 API-Dokumentation 44
2.5.3 API-Quellcode 46

2.6 Blicke hinter die Kulissen mit ILSpy 47

2.7 Übungsaufgaben zum Kapitel 2 50

3 VERÖFFENTLICHUNG EINER ANWENDUNG FÜR LINUX 51

4 TECHNISCHE GRUNDLAGEN 53

4.1 Software-Stapel einer Avalonia-Anwendung 53

4.2 Stammbaum der Avalonia-Steuerelemente 53

4.3 Single-Thread – Architektur und Dispatcher 55

4.4 Die Klasse AppBuilder 57

4.5 IClassicDesktopStyleApplicationLifetime 58
4.5.1 Kreation durch die Methode StartWithClassicDesktopLifetime() 59
4.5.2 Ereignisse 60
4.5.3 Eigenschaften 60
4.5.4 Shutdown() 60

4.6 Die Klasse Window 61
4.6.1 Besonderheiten des Hauptfensters 61
4.6.2 Eigenschaften 62
4.6.3 Ereignisse 63

4.7 Die Klasse Application 63
4.7.1 Initialisierungsmethoden 63
4.7.2 Eigenschaften 64

4.8 NuGet-Pakete installieren 64

4.9 MessageBox-Ersatz 65

5 XAML 71

5.1 Elementare Regeln zum Aufbau einer XML-Datei 71

5.2 XAML-Beschreibung 72
5.2.1 Wurzelelemente und XML-Namensräume 72
5.2.2 Instanzelemente 75
5.2.3 Eigenschaftsausprägungen zuweisen 76
5.2.3.1 Attributsyntax 76
5.2.3.2 Eigenschaftselementsyntax 77
5.2.3.3 Inhaltseigenschaft 78
5.2.3.4 Textinhaltsyntax 79
5.2.3.5 Kollektionssyntax 79
5.2.3.6 Markup-Erweiterungen 80
5.2.4 XAML-Registrierung von Ereignisbehandlungsmethoden 81

5.3 XAML-Verarbeitung 81

6 CODE-BEHIND - DATEIEN 83

6.1 Definition der Anwendungsklasse 83

6.2 Definition der Hauptfensterklasse 84
6.2.1 Beiträge des Entwicklers 85
6.2.2 Ergänzungen durch das Erstellungssystem 86
6.2.2.1 Generierter C# - Quellcode 86
6.2.2.2 Vom XamlX-Compiler erstellter IL-Code 88

7 ROUTINGEREIGNISSE 91

7.1 Routingstrategien 92
7.1.1 Registrierte Ereignisrouten ermitteln 93
7.1.2 Delegatensignaturen der Behandlungsmethoden 94

7.2 Einsatzfälle für Routingereignisse 95

7.3 Beobachtungsstudie 96

7.4 Klassenbezogene Behandlungsmethoden 101

7.5 Routingereignisse definieren 104

7.6 Angefügte Ereignisse 105

8 ABHÄNGIGKEITSEIGENSCHAFTEN 109

8.1 AvaloniaProperty - Varianten 109

8.2 Registrierung und CLR-Verpackung 111

8.3 Eigenschaftsübertragung auf enthaltene Elemente 114

8.4 Angefügte Eigenschaften 115

9 DATENBINDUNG 119

9.1 Personenverwaltungsbeispiel 119

9.2 Übersetzte Datenbindung 121

9.3 Bindung an eine Objekteigenschaft 121

9.4 Bindung an eine Kollektion mit komplexem Elementtyp 122

9.5 Bindung an ein anderes Steuerelement 124

9.6 Änderungsbenachrichtigungen 127
9.6.1 INotifyPropertyChanged 128
9.6.2 INotifyCollectionChanged 130

9.7 Datenvalidierung 131

10 LAYOUTCONTAINER 135

10.1 Panel 136

10.2 Grid 137
10.2.1 Zeilen und Spalten definieren 137
10.2.2 Ortsangaben 139
10.2.3 Mehrzellige Elemente 139
10.2.4 GridSplitter 140

10.3 DockPanel 141

10.4 StackPanel 143

10.5 WrapPanel 145

10.6 RelativePanel 146

10.7 UniformGrid 148

10.8 Canvas 149

10.9 Geschachtelte Layoutcontainer 151

11 STEUERELEMENTE 153

11.1 Abstammungsverhältnisse 153

11.2 Verwendung 154
11.2.1 Instanziieren 154
11.2.2 Elementare Eigenschaften 156
11.2.3 Ereignisbehandlung 158
11.2.3.1 Behandlungsmethoden definieren und zuweisen 158
11.2.3.2 On-Methoden überschreiben 160

11.3 Standardkomponenten 164
11.3.1 Schaltflächen 164
11.3.1.1 Beschriftete Schaltflächen 164
11.3.1.2 Standard- und Escape-Schalter 165
11.3.1.3 Alt-Zugriffstaste 167
11.3.1.4 Bilder auf Schaltflächen 167
11.3.2 Kontrollkästchen und Optionsfelder 169
11.3.3 Texteingabefelder 172
11.3.4 Tastaturfokus 175
11.3.5 Listen- und Kombinationsfelder 177
11.3.5.1 ListBox mit Items befüllen oder an eine externe Kollektion binden 178
11.3.5.2 ComboBox mit Items befüllen oder an eine externe Kollektion binden 180
11.3.5.3 Gewähltes Item ermitteln 182

11.4 Übungsaufgaben zum Kapitel 11 183

12 MVVM 187

12.1 Verteilung von Zuständigkeiten auf lose gekoppelte Komponenten 188
12.1.1 Modell 188
12.1.2 Ansichtsmodell (ViewModel) 189
12.1.3 Ansicht (View) 191
12.1.4 Variable Aufgabenverteilung 192

12.2 Beispiel 193

12.3 Kommandobindung 198
12.3.1 ICommand 199
12.3.2 Separate ICommand-Implementationen 200
12.3.2.1 Kommandoklasse `AddPersonCmd` 200
12.3.2.2 Kommandoklasse `DeletePersonCmd` 203
12.3.2.3 Ansichtsmodellklasse 204
12.3.2.4 XAML-Code der Ansicht 205
12.3.3 Eigene Kommandoklasse RelayCommand 207
12.3.4 Kommandobindung an eine Methode 209

12.4 MVVM Community Toolkit 210
12.4.1 ObservableObject 211
12.4.2 ObservableValidator 215
12.4.3 Toolkit-Kommandoklasse RelayCommand 217
12.4.4 Verbesserte Bedienung und Validierung 219
12.4.4.1 Neustart 220
12.4.4.2 Validierung im ViewModel 222
12.4.4.3 Toolkit-Attribut RelayCommand 223
12.4.4.4 Bearbeitungsmodi 225
12.4.4.5 Umsetzung der Bearbeitungsmodi durch die Ansicht 227
12.4.4.6 Alterserfassung per String 230
12.4.5 ObservableRecipient 233

13 WEITERE STEUERELEMENTE 235

13.1 Menüoptionen 235
13.1.1 Hauptmenü 235
13.1.2 Kontextmenü 238
13.1.3 Flyout-Menü 239

13.2 ToolTip 239

ANHANG 241

A. Erwerb einer Community-Lizenz für Avalonia UI 241

B. Visueller Baum und DevTools 250

C. Parcel-Werkzeug zum Erstellen von Installationspaketen 254

D. Lösungsvorschläge zu den Übungsaufgaben 259
Kapitel 2 (Erstellungswerkzeuge unter Windows) 259
Kapitel 11 (Steuerelemente) 259

LITERATUR 261

STICHWORTREGISTER 263

1 GUI-Programmierung mit Avalonia UI

Moderne Anwendungsprogramme benötigen eine grafische Bedienoberfläche (ein ***GUI***, ***G****raphical* ***U****ser* ***I****nterface*) mit standardisierten Bedienelementen zur Benutzerinteraktion und Datenpräsentation (Texteingabefelder, Befehlsschalter, Kontrollkästchen und Optionsfelder, Auswahllisten, Tabellen zur Datenpräsentation, Menüs usw.). Von standardisierten Bedienelementen profitieren Entwickler und Anwender:

- Entwickler können dank direkt verwendbarer und dabei flexibel konfigurierbarer Komponenten die Bedienoberfläche einer Anwendung zügig aufbauen.
- Weil die Steuerelemente intuitiv und in verschiedenen Programmen weitgehend konsistent zu bedienen sind, erleichtern sie dem Anwender den Umgang mit Software.

Die .NET – Plattform hat von Beginn an die Entwicklung von GUI-Software ermöglicht, allerdings lange Zeit nur für Windows. Seit .NET 6 bietet Microsoft mit ***MAUI*** (***M****ulti-platform* ***A****pp* ***UI***) ein Multiplattform – GUI-Framework mit Unterstützung der Betriebssysteme Android, iOS, macOS und Windows an. Unter Linux und in einem Web-Browser sind MAUI-Anwendungen *nicht* ausführbar. Mit Avalonia UI steht eine Open Source – Lösung zur Entwicklung von .NET – Software mit grafischer Bedienoberfläche für *alle* signifikanten Plattformen (inklusive Linux und Browser) zur Verfügung.

Das vorliegende Manuskript beschreibt die Version 12 von Avalonia UI und setzt dabei Erfahrungen mit der Programmiersprache C# voraus, die man nötigenfalls z. B. über das kostenlos verfügbare E-Book von Baltes-Götz & Frischknecht (2026) erwerben bzw. auffrischen kann.

1.1 Besonderheiten von GUI-Anwendungen

Zunächst widmen wir uns der Aufgabe, einem Leser mit Erfahrungen bei der objektorientierten Programmierung von Konsolenanwendungen die Besonderheiten von GUI-Anwendungen zu erläutern. Texte zur Einführung in die objektorientierte Programmierung verwenden meist Konsolenanwendungen zur Erläuterung der vielfach anspruchsvollen Konzepte (z. B. Vererbung, Polymorphie, Generizität, Schnittstellen, Kollektionen, Delegaten, Multithreading).

1.1.1 Vergleich zwischen GUI- und Konsolenanwendungen

Im Vergleich zu Konsolenprogrammen geht es bei GUI-Anwendungen nicht nur intuitiver, sondern vor allem auch ereignisreicher[1] und mit mehr Mitspracherechten für die Anwender zu. Ein Konsolenprogramm entscheidet selbst darüber, welche Anweisung als nächstes ausgeführt wird, und wann der Benutzer eine Eingabe machen darf. Um seine Aufgaben zu erledigen, verwendet ein Konsolenprogramm diverse Dienste des Laufzeitsystems, z. B. bei der Aus- oder Eingabe von Zeichen.

Für den Ablauf einer Anwendung mit grafischer Benutzeroberfläche ist ein **ereignisorientiertes und benutzergesteuertes Paradigma** wesentlich, wobei das Laufzeitsystem als Vermittler oder (seltener) als Quelle von Ereignissen in erheblichem Maße den Ablauf mitbestimmt, indem es Methoden der GUI-Anwendung aufruft. Auslöser von Ereignissen ist in der Regel der Benutzer, der mit der Hilfe von Eingabegeräten wie Maus, Tastatur, Touchscreen etc. praktisch permanent in der Lage ist, unterschiedliche Wünsche zu artikulieren. Ein GUI-Programm präsentiert mehr oder weniger viele Bedienelemente, die dem Anwender das Auslösen von Ereignissen ermöglichen. Das Programm wartet die meiste Zeit darauf, auf ein vom **Benutzer** ausgelöstes **Ereignis** mit einer vorbereiteten Behandlungsmethode zu reagieren. Allerdings kann ein GUI-Programm situationsabhängig unterschiedliche Bedienelemente anbieten.

---

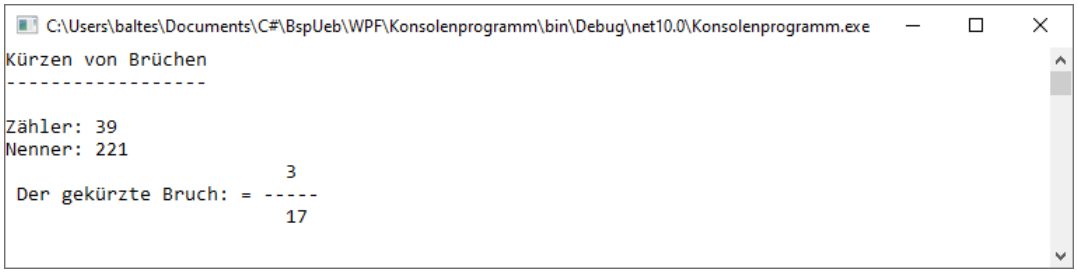
[1] Allerdings kann auch eine .NET - Konsolenanwendung auf Ereignisse reagieren (z. B. auf das Erstellen oder Löschen von Dateien).

Im Vergleich zu einem Konsolenprogramm ist bei einem GUI-Programm die dominante Richtung im Kontrollfluss zwischen Anwendung und Laufzeitsystem invertiert. Die Ereignisbehandlungsmethoden einer GUI-Anwendung sind Beispiele für sogenannte *Call Back - Routinen*. Man spricht auch vom *Hollywood-Prinzip*, weil man in dieser Gegend oft nach der Devise kommuniziert: *Don't call us. We call you.*

Während sich ein Konsolenprogramm gegenüber dem Anwender autoritär und gegenüber dem Laufzeitsystem fordernd verhält, präsentiert ein GUI-Programm dem Anwender Service-Angebote und befolgt kooperativ die Anweisungen des Laufzeitsystems:

- Eine Konsolenanwendung diktiert den Ablauf und erlaubt dem Benutzer gelegentlich eine Eingabe. Um seinen Job erledigen zu können, verlangt das Programm Dienstleistungen vom Laufzeitsystem, z. B.: „Bitte den nächsten Tastendruck übermitteln." Das Laufzeitsystem erledigt solche Anforderungen und gibt die Kontrolle dann wieder an die Konsolenanwendung zurück. Eine Konsolenanwendung benimmt sich so, als wäre sie das einzige Anwendungsprogramm und hätte das Laufzeitsystem exklusiv als Dienstleister zur Verfügung.
- Eine GUI-Anwendung besteht hingegen aus einer Sammlung von Ereignisbehandlungsmethoden, wobei die zugehörigen Ereignisse meist aus Initiativen des Benutzer resultieren. Das Laufzeitsystem registriert die Ereignisse und ruft daraufhin Methoden des GUI-Programms auf.

Zur Illustration betrachten wir eine Konsolen- und eine GUI-Anwendung zum Kürzen von Brüchen. Das Konsolenprogramm



diktiert den gesamten Ablauf:

- Es fragt nach dem Zähler.
- Es fragt nach dem Nenner.
- Es schreibt den gekürzten Bruch auf die Konsole.

Wenn der Benutzer z. B. nach der Eingabe des Nenners den Zähler noch einmal ändern möchte, dann muss er das Programm beenden und neu starten.

Im Unterschied zu diesem **programmgesteuerten Ablauf** diktiert bei der folgenden GUI-Variante

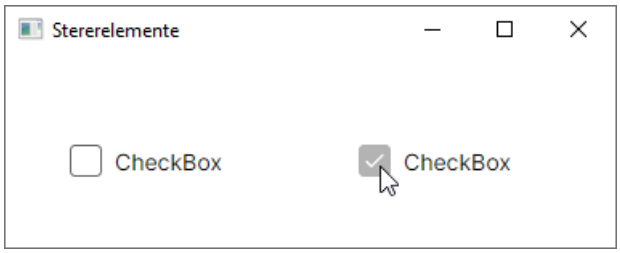


der Benutzer das Geschehen. Er kann die vier Bedienelemente (zwei Texteingabefelder und zwei Schaltflächen) in beliebiger Reihenfolge verwenden, wobei das Programm mit seinen Ereignisbehandlungsmethoden reagiert (**benutzergesteuerter Ablauf**).

1.1.2 Steuerelemente im Vergleich zu anderen Objekten

Als Besonderheiten der Steuerelementklassen (z. B. im Vergleich zu Klassen wie **String** oder **List<T>**) sind zu nennen:

- Ihre Objekte können **auf dem Bildschirm auftreten** und dabei selbständig **mit dem Benutzer interagieren**. Wenn wir z. B. ein Kontrollkästchen (ein Objekt der Klasse **CheckBox**) in ein Fenster einbauen, dann erscheint bzw. verschwindet bei einem Mausklick die Markierung,



  ohne dass wir uns um diese Anpassung der Optik kümmern müssen. Hinter dieser Verhaltenskompetenz steckt eine aufwändige, von den Entwicklern des GUI-Frameworks (z. B. Avalonia UI) geleistete Grafikprogrammierung.
- Die Steuerelemente kommunizieren über **Ereignisse** mit anderen Klassen. Um z. B. eine Benachrichtigung über die gesetzte Markierung eines Kontrollkästchens zu erhalten, registriert man bei Verwendung von Avalonia UI eine den Delegatentyp **RoutedEventHandler** erfüllende Behandlungsmethode bei seinem **Checked**-Ereignis. Bei den Ereignissen von Avalonia UI - Steuerelementen handelt es sich meist um **Routingereignisse** (siehe Kapitel 7).

1.2 WPF als richtungsweisende GUI-Technologie der .NET - Plattform

Zu der GUI-Technik *Windows Forms* (kurz: *WinForms*) aus dem .NET Framework 1.0 erschien 2006 die technisch überlegene ***W**indows **P**resentation **F**oundation* (***WPF***). Sie hat Innovationen gebracht, die bis heute in vielen .NET – GUI-Frameworks anzutreffen sind, z. B.:

- Die GUI-Deklaration in einem ***XML***-Dialekt (*e**X**tensible **M**arkup **L**anguage*) namens ***XAML*** (*e**X**tensible **A**pplication **M**arkup **L**anguage*) ist rationell und erleichtert die Kooperation von Software-Entwicklern und Grafikdesignern. Vor allem fördert sie die in großen Projekten essenzielle Trennung der Anwendungslogik vom Layout der Bedienoberfläche.
- Durch die **Bindung** von Steuerelementen an Datenobjekte oder andere Steuerelemente erhält man eine stets aktuelle Anzeige. Als Datenbindungsziele spielen die zur Ergänzung der .NET – Eigenschaften eingeführten **Abhängigkeitseigenschaften** eine wichtige Rolle (siehe Kapitel 8).
- Verfahren zur **Validierung von Benutzereingaben** verbessern die Datenqualität.
- Die zur Ergänzung der .NET – Ereignisse eingeführten **Routingereignisse** lassen sich auf verschiedenen Ebenen einer Steuerelementhierarchie behandeln.

- Das WPF-Framework dient nicht nur zum rationellen Erstellen von ergonomischen Bedienoberflächen, sondern unterstützt auch die ***MVVM***-Anwendungsstruktur (***M**odel-**V**iew-**V**iew**M**odel*) mit einer Trennung zwischen ...
  - der Bedienoberfläche (Ansicht),
  - den Klassen im sogenannten *Ansichtsmodell* (engl. *View Model* oder zusammengeschrieben als *ViewModel*), die sich um die Präsentationslogik (angezeigte Daten, verfügbare Bedienelemente) kümmern und Kommandos ausführen,
  - sowie den Klassen im *Modell*, die sich um die Anwendungslogik kümmern.

  Die Trennung der Funktionen (engl.: *separation of concerns*) ist für professionelle (komplexe) Anwendungen hochrelevant (siehe Kapitel 12). In einer kleinen Anwendung lohnt sich der damit verbundene Aufwand allerdings nicht.

1.3 Avalonia UI als modernisiertes Multiplattform WPF

Das maßgeblich von der Firma **AvaloniaUI OÜ** aus Tallinn (Estland)[1] entwickelte **Avalonia UI** erlaubt als Framework bzw. Bibliothek ...

- mit Open-Source – Lizenzmodell (MIT, selbe Lizenz wie .NET/C#)
- die Entwicklung von .NET – GUI – Software mit moderner MVVM-Anwendungsarchitektur
- für diverse Plattformen[2]
  - iOS (ab Version 13)
  - Android (ab Version 5.0)
  - Windows (ab Version 8.1)
  - macOS (ab Version 10.14)
  - Linux (z. B. Debian ab Version 9, Ubuntu ab Version 16.04, Fedora ab Version 30)
  - WebAssembly (Browser)

  unter Verwendung derselben Codebasis.

Wenn keine Verwechslungsgefahr besteht, ist im Manuskript der Kürze halber gelegentlich von *Avalonia* die Rede (ohne *UI*).

Dass Avalonia UI von dem (auf Windows beschränkten) WPF-Framework der Firma Microsoft inspiriert ist, verrät schon die Bezeichnung, die Microsoft während der WPF-Entwicklung als Codenamen verwendete. Avalonia UI hat wesentliche WPF-Säulen übernommen:

- die Open Source – Software .NET[3] als Basis (siehe Baltes-Götz & Frischknecht 2026)
- die GUI-Deklaration per ***XAML*** (*e**X**tensible **A**pplication **M**arkup **L**anguage*)
- Routingereignisse
- Abhängigkeitseigenschaften
- Datenbindung und Validierung
- die MVVM-Anwendungsarchitektur

---

[1] https://avaloniaui.net/
Eine ***OÜ*** (***osaü**hing*) ist die estnische Entsprechung der deutschen GmbH (Gesellschaft mit beschränkter Haftung).

[2] https://de.wikipedia.org/wiki/Avalonia_UI

[3] https://dotnet.microsoft.com/en-us/platform/open-source

Jedoch geht Avalonia UI u. a. zugunsten der Plattformunabhängigkeit auch eigene Wege, z. B.:

- Rendering der Bedienoberfläche durch die 2D-Grafikbibliothek *Skia*
  Weil Avalonia nicht die nativen Bedienelemente der Plattformen nutzt, sondern mit Skia eigene, überall identische Elemente zeichnet, reduziert sich der Testaufwand für eine Multiplattform-Anwendung erheblich. Wenn ein GUI-Framework (wie z. B. MAUI, siehe Abschnitt 1.4) die nativen Bedienelemente der Plattformen nutzt, dann muss man das Verhalten eines Programms auf jeder Plattform testen. Bei der von Avalonia UI (und auch von Googles Multiplattform-Lösung *Flutter*) genutzten einheitlichen Skia-Technik muss man allerdings den Nachteil der Abweichung vom jeweiligen Plattform-Standard in Kauf nehmen. Im Vergleich zu Flutter hat Avalonia UI den Vorteil, dass man C# verwenden kann und nicht die spezielle Programmiersprache *Dart* erlernen muss.
- Stile mit Internet-Techniken
  Avalonia verwendet eine an ***CSS*** (***C****ascading* ***S****tyle* ***S****heets*) angelehnte Gestaltung von Steuerelementen durch Klassen, Selektoren und Pseudoklassen. Zu dieser sehr gelungenen Lösung tauchen leider aus Zeitgründen im Manuskript nur einige Beispiele auf.
- Weiterentwicklung von Bedienelementen

Wer ein modernisiertes und von Windows emanzipiertes WPF sucht, das zudem einen Beitrag zur europäischen IT-Souveränität im Bereich der Anwendungs-Software leistet, findet in Avalonia UI eine attraktive Option.

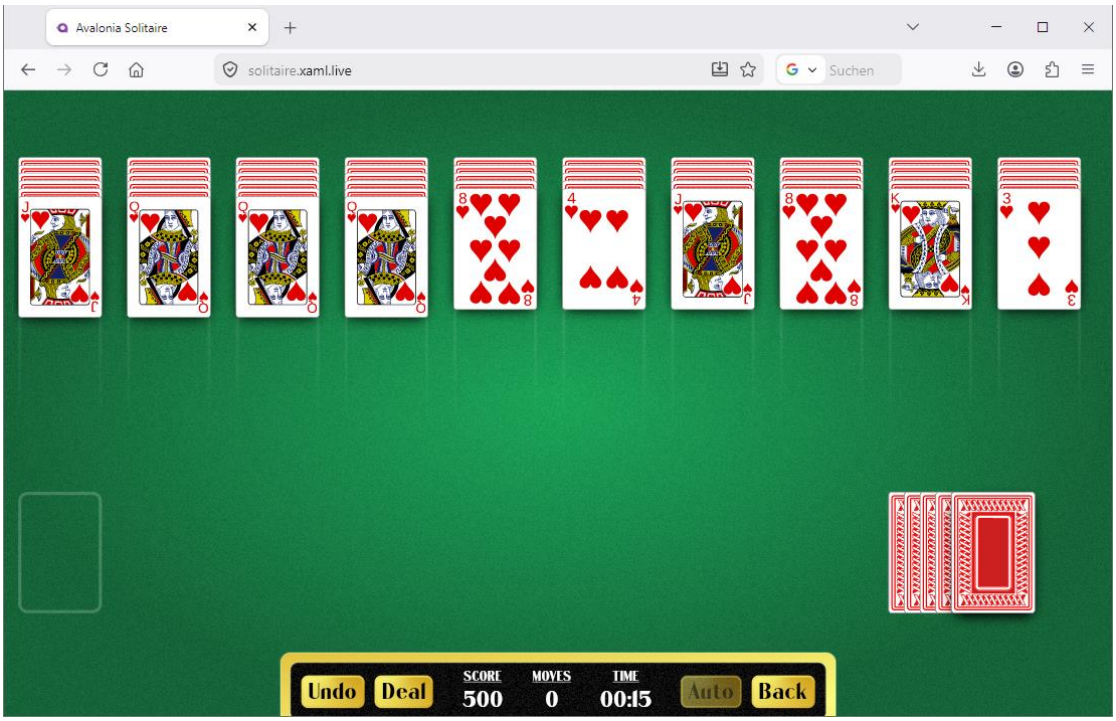
Avalonia UI ist kostenlos unter der MIT-Lizenz auch für kommerzielle Projekte nutzbar, wobei sich die als Motor und Koordinator der Entwicklung tätige Firma Avalonia OÜ durch Zusatzleistungen finanziert:

- Kostenpflichtige Plus- und Pro-Versionen bieten verbesserte Entwicklungswerkzeuge, Premium-Steuerelemente und Datenvisualisierungen an.
- Außerdem ist mit Avalonia XPF eine kostenpflichtige Software zum unaufwändigen Portieren von WPF-Programmen auf Linux und macOS im Angebot.

Zum Leistungsumfang der kostenlosen Edition von Avalonia UI gehören u.a. ....

- eine moderne GUI-Bibliothek,
- auf nicht-kommerzielle Projekte beschränkte Plugins für das Visual Studio und andere Entwicklungsumgebungen (Visual Studio Code, JetBrains Rider) mit hilfreichen Werkzeugen,
- Unterstützung der MVVM-Anwendungsarchitektur (per MVVM Community Toolkit oder ReactiveUI),
- Eingabevalidierung über die Schnittstelle **INotifyDataErrorInfo**,
- eine XAML-Vorschau,
- identische Steuerelemente auf allen Plattformen aufgrund der überall genutzten Skia-Grafik von Google,
- AOT-Übersetzung.

Eine mit Avalonia UI erstellte WebAssembly-Anwendung (WASM) muss vor der Ausführung im Browser zusammen mit dem WebAssembly-Bytecode der Mono-Runtime und der Skia-Bibliothek heruntergeladen werden. Ein Anwendungsbeispiel mit dem Kartenspiel Solitär



benötigt beim Laden von der Seite

https://solitaire.xaml.live/

ca. 5 Sekunden bis zum Start.[1] Das ist akzeptabel beim Starten einer Anwendung, die man längere Zeit nutzen möchte. Avalonia UI verwendet keine HTML-Bedienelemente, sondern zeichnet (engl.: *rendert*) die gesamte Bedienoberfläche auf ein **canvas**-Element, z. B.:

```
<canvas id="canvasvtbunvbs" class="avalonia-canvas" style="width: 100%;
height: 100%; position: absolute;" width="1102" height="144"></canvas>
```

Das sorgt für die Konsistenz mit Desktop-Betriebssystemen und verhindert eine Abhängigkeit vom konkreten Browser. Damit entfallen Dienstleistungen der Browser (z. B. automatisches Ausfüllen von Textfeldern, Rechtschreibunterstützung, Übersetzung), die bei einer Anwendung (siehe obiges Beispiel mit dem Spiel Solitär) allerdings wenig relevant sind. Insgesamt ist die WebAssembly-Option von Avalonia UI ungeeignet für typische Webseiten, bei denen kurze Ladezeiten und die Suchmaschinenoptimierung eine Rolle spielen. Neben der technisch bedingten Startverzögerung haben wir eine relativ hohe CPU-Belastung von älteren Klientenrechnern festgestellt (ca. 30%). Daher ist bei einer mit Avalonia UI erstellen WASM-Anwendung die ***AOT***-Übersetzung (***A**head **O**f **T**ime*, siehe Abschnitt 2.5.2 in Baltes-Götz & Frischknecht 2026) zu empfehlen. Obwohl noch Raum für Verbesserungen besteht, ist Avalonia 12 eine sinnvolle Option, um eine Anwendung mit geringem Aufwand in den Browser bringen (z. B. zur Demonstration).

Der wichtigste Einsatzort von Avalonia UI ist sicher nicht der Browser, sondern der Desktop unter Linux, macOS oder Windows, wo Avalonia-Anwendungen eine gute Leistung zeigen. Ein Blick in den Avalonia-Ausstellungsraum für besonders gelungenen Anwendungen

https://avaloniaui.net/showcase

zeigt (am 3.7.2026) überwiegend Desktopanwendungen:

---

[1] Der Quellcode ist hier zu finden:
https://github.com/AvaloniaUI/Solitaire/tree/master/Solitaire

Plattform	Zahl der Anwendungen
Linux	53
macOS	57
Windows	69
Android	3
iOS	1
WebAssembly	3

Abgesehen von 10 Ausnahmen unterstützen die Anwendungen mindestens zwei Plattformen:

Anzahl der Plattform	Zahl der Anwendungen
1	10 (meist Windows)
2	14 (meist macOS und Windows)
3	43 (meist Linux, macOS und Windows)
4	4
5	2

Weil hinter der 2013 entstandenen Open Source – Lösung Avalonia UI kein Weltkonzern steht, sondern eine relativ kleine Firma, ist keine langjährige Weiterentwicklung garantiert. Allerdings hat auch der IT-Gigant Microsoft bei GUI-Lösungen für .NET in den letzten Jahren einige Investitionsruinen zu verantworten (z. B. UWP, Silverlight).

Insgesamt ist Avalonia UI eine Multiplattform-Option mit guten Zukunftsaussichten und spezieller Attraktivität für Entwickler mit WPF-Erfahrung. Es sind Standard-Steuerelemente in großer Zahl vorhanden, doch ist das Angebot an kommerziellen Steuerelementen für spezielle Zwecke noch begrenzt (z. B im Vergleich zu WPF). Allerdings sind diese kostenpflichtigen Angebote nicht für jedes Projekt relevant.

1.4 Alternative GUI-Technologien für .NET - Software

Microsoft feilt permanent an GUI-Techniken für die .NET - Plattform, wobei eine Evolution mit gelegentlichen Fehlschlägen stattfindet. Seit .NET 6 hat Microsoft mit ***MAUI*** (***M**ulti-platform **A**pp **UI***) eine auf mehreren Plattformen nutzbare grafische Bedienoberfläche im Angebot, sodass die seit .NET Core ausgebaute Plattformunabhängigkeit nicht mehr auf Konsolen- und Serveranwendungen beschränkt ist. Allerdings bleiben bei MAUI das Betriebssystem Linux und die Web-Browser außen vor. Eine Artikelserie von Holger Schwichtenberg aus dem Jahr 2025 beschreibt eine beeindruckende Vielfalt von GUI-Optionen für .NET-Entwickler.[1] Das Wissen um diese Alternativen ist wichtig bei der Auswahl einer Option, für die Lektüre des vorliegenden Manuskripts über Avalonia UI aber irrelevant.

Es folgt eine Kurzfassung des Abschnitts 12.1 aus Baltes-Götz & Frischknecht (2026) mit einem Vergleich der GUI-Optionen aus der Perspektive von Entwicklern mit WPF-Erfahrung:

- Avalonia UI
  Avalonia bietet eine auf diversen Plattformen einsetzbare und zudem modernisierte WPF-Alternative, die wir im vorliegenden Manuskript ausführlich darstellen.

[1] https://www.heise.de/hintergrund/GUI-Frameworks-fuer-NET-Teil-1-Windows-Forms-10364458.html

- WPF
  Das WPF-Framework ist aktuell (im Sommer 2026) für die Entwicklung von *Windows*-Anwendungen immer noch eine gute Wahl. Für seine Verwendung sprechen die immer noch moderne Technologie (siehe Abschnitt 1.2), die guten Entwicklungswerkzeuge im Visual Studio (vor allem der grafische UI-Designer) und die große Anzahl von Firmen, die professionelle WPF-Steuerelemente für komplexe Aufgaben als Ergänzung zu der von Microsoft gelieferten Grundausstattung anbieten. Nachteilig ist vor allem die Beschränkung auf Windows.
- WinForms
  Die 2002 mit dem auf Windows beschränkten .NET *Framework* 1.0 eingeführte GUI-Technik *Windows Forms* hat sich dank einer treuen Anhängerschaft bis heute behauptet, obwohl schon 2006 das technisch überlegene WPF erschien. Mittlerweile sind beide GUI-Techniken auch für das moderne .NET verfügbar, aber weiterhin nur unter Windows.
- UWP und WinUI 2
  Mit Windows 8 hat Microsoft unter Bezeichnungen wie *Metro App*, *Modern App* oder *Store App* ein neues Design und Bedienkonzept für Windows-Anwendungen eingeführt. Es entstand ein neues Anwendungsmodell (statt Win32-API) und ein neues Laufzeitsystem namens *Windows-Runtime* (*WinRT*), das Anwendungen in einem Sandkasten mit beschränktem Zugriff auf die Systemumgebung betreibt.[1] Zusammen mit Windows 10 hat Microsoft das auf dem Laufzeitsystem WinRT basierende Anwendungsmodell weiterentwickelt zur ***U**niversal **W**indows **P**latform* (***UWP***) . Microsoft hat das UWP-Anwendungsmodell und seine Grafikbibliothek WinUI 2 zwar noch nicht als *veraltet* (engl.: *deprecated*) herabgestuft, empfiehlt jedoch für neue Windows-Desktopanwendungen eindeutig das Windows App SDK mit der Grafikbibliothek WinUI 3.[2]
- Windows App SDK und WinUI 3
  Zur Überwindung der Trennung von Win32- und UWP-Anwendungen hat Microsoft im Jahr 2021 das *Windows App SDK* (früherer Name: *Project Reunion*) mit der Grafikbibliothek WinUI 3 als Kernbestandteil präsentiert.[3] WinUI 3 ist wie WPF auf Windows beschränkt.
- MAUI
  Seit .NET 6 sind dank einer Technik namens ***MAUI*** (***M**ulti-platform **A**pp **UI***) C# - Anwendungen mit grafischer Bedienoberfläche möglich, die mit identischem Quellcode unter Android, iOS, macOS und Windows laufen.[4] Linux und Web-Browser sind auch in der .NET - Version 10 noch außen vor. Wichtige Gemeinsamkeiten von MAUI und Avalonia UI sind die GUI-Deklaration per XAML und die Unterstützung der MVVM-Anwendungsarchitektur. Allerdings verwendet MAUI eine von Avalonia UI (und WPF) deutlich abweichende XAML-Variante.

---

1 https://en.wikipedia.org/wiki/Features_new_to_Windows_8

2 https://learn.microsoft.com/en-us/windows/apps/get-started/windows-developer-faq
Wie diese Webseite erläutert, wird die zusammen mit der Metro- bzw. WinUI-Bedienung entwickelte Windows Runtime (WinRT) jetzt vom Windows App SDK genutzt.

3 https://en.wikipedia.org/wiki/Windows_App_SDK

4 https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui

- Uno
  Uno ist wie Avalonia UI eine als Open Source (Apache-Lizenz) verfügbare Multiplattform-GUI-Technik für .NET – Software. Aktuell (im Sommer 2026) ist Avalonia UI für Entwickler mit WPF-Erfahrung gegenüber Uno zu bevorzugen, denn:
  - Die Avalonia UI - Entwicklung hat sich vor allem an WPF (mit dem früheren Codenamen *Avalon*) orientiert, während Uno mit UWP bzw. WinUI als Vorbild entstand, sodass beim XAML-Dialekt von Avalonia UI eine höheren WPF-Kompatibilität besteht.
  - Avalonia UI bietet eine bessere Unterstützung des MVVM-Entwurfsmusters.
- Webanwendung mit verteilter Architektur und SPA
  Zur Verwendung eines Browsers als Bedienoberfläche bietet Microsoft mehrere, zu *ASP.NET Core* gehörende Lösungen, z. B.:[1]
  - Eine Webanwendung mit verteilter Anwendungsarchitektur bestehend aus ...
    - der in C# programmierten und auf einem Server bzw. auf mehreren Servern ausgeführten Geschäftslogik samt Datenverwaltung
    - und einer klientenseitigen Bedienoberfläche in einem Web-Browser.

    Das Erstellen der vom Browser unter Verwendung von offenen Standards (HTML, CSS) ausgeführten Bedienoberfläche kann auf dem Server, auf dem Klienten oder kooperativ erfolgen. Baltes-Götz & Frischknecht (2026) beschreiben im Kapitel 22 eine mit *ASP.NET Core Razor Pages* realisierte Lösung, wobei der HTML/CSS - Code im Wesentlichen auf dem Server entsteht. Im Browser sorgen die JavaScript-Bibliothek *jQuery* sowie die CSS-Bibliothek *Bootstrap* für Interaktivität und dynamische Formatierung ohne Server-Zugriffe (z. B. bei der Eingabevalidierung).
  - Man kann in C# klientenseitig vom Browser ausgeführte ***S**ingle **P**age Web-**A**pps* (***SPA***s) mit *Blazor WebAssembly* - Technik erstellen.

  Bei den beiden dargestellten *Lösungen* ist im Unterschied zu einer WebAssembly-Anwendung mit Avalonia UI *keine* XAML-Deklaration im Spiel. Die Bedienoberfläche ist mit HTML und CSS zu erstellen.

[1] Holger Schwichtenberg beschreibt in seiner 2025 veröffentlichten Artikelserie über Bedienoberflächen für .NET – Software sechs Lösungen:
https://www.heise.de/hintergrund/GUI-Frameworks-fuer-NET-Teil-4-Webframeworks-10438855.html

2 Erstellungswerkzeuge unter Windows

In diesem Kapitel beschreiben wir kostenlose Werkzeuge zum Entwickeln von Avalonia – Anwendungen in der Programmiersprache C#.

2.1 .NET SDK

Die von Microsoft im frei verfügbaren **.NET SDK** (***S**oftware **D**evelopment **K**it*) angebotenen Werkzeuge zum Erstellen von .NET – Software sind auch für Avalonia UI relevant:[1]

- das als ***CLI*** (***C**ommand **L**ine **I**nterface*) bezeichnete Programm **dotnet.exe** mit zahlreichen Optionen zum Erstellen und Veröffentlichen von .NET - Programmen
- das Erstellungssystem **MSBuild**
  Es erfährt aus der Projektdatei z. B. den Ausgabetyp (GUI-Programm oder Bibliothek), die erforderliche .NET – Version und die beteiligten NuGet-Pakete.
- der C# - Compiler **Roslyn**
- eine Laufzeitumgebung (***CLR***, ***C**ommon **L**anguage **R**untime*) zur Ausführung von .NET - Programmen

In der Praxis der Software-Entwicklung verwendet man eine auf diesen Werkeugen aufsetzende Entwicklungsumgebung (***IDE***, ***I**ntegrated **D**evelopment **E**nvironment*) und kommt folglich mit den Werkzeugen nur selten direkt in Kontakt. MSBuild ist über die Projektdatei im Arbeitsalltag präsent, und das CLI (dotnet.exe) kommt z. B. beim Veröffentlichen einer Anwendung zum Einsatz (siehe Kapitel 3).

Bei der Installation der für Windows verfügbaren und im Manuskript verwendeten Entwicklungsumgebung Visual Studio Community (siehe Abschnitt 2.2) wird auch das aktuelle .NET SDK eingerichtet bzw. aktualisiert. Daher verzichten wir darauf, das .NET SDK samt Laufzeitumgebung von der Webseite

https://dotnet.microsoft.com/

zu beziehen und separat zu installieren.

Unabhängig vom Installationsverfahren ist das .NET – SDK unter Windows hier zu finden:

C:\program files\dotnet\sdk

Microsoft veröffentlicht in jedem November eine neue .NET - Hauptversion, die alternierend mit einer einfachen oder langen Unterstützungsdauer ausgestattet ist:[2]

- **STS** (***S**tandard **T**erm **S**upport*)
  18 Monate Versorgung mit Sicherheits-Updates
- **LTS** (***L**ong **T**erm **S**upport*)
  36 Monate Versorgung mit Sicherheits-Updates

Wie die folgende Tabelle für die .NET - Versionen 8 bis 10 zeigt, hält Microsoft den Veröffentlichungsplan ein:

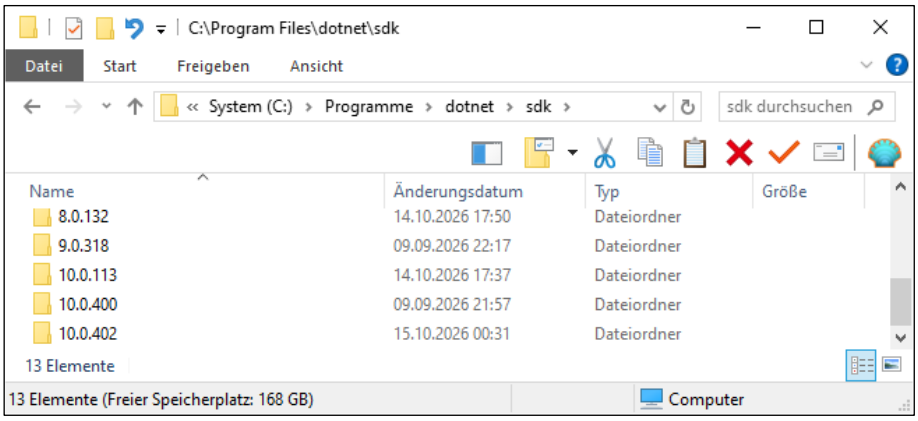
---

[1] Die Bezeichnung *.NET – SDK* bezieht manchmal die Laufzeitumgebungen ein (z. B. beim Download-Angebot), beschränkt sich in anderen Fällen aber auf die Werkzeuge zum Erstellen von .NET – Software.

[2] https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-core

Version	Erschienen	Typ der Unterstützung	Ende der Unterstützung
.NET 10	11.11.2025	LTS	14.11.2028
.NET 9	12.11.2024	STS	10.11.2026
.NET 8	14.11.2023	LTS	10.11.2026

Sind auf einem Windows-Rechner die im Oktober 2026 unterstützten Versionen installiert, dann zeigt sich im Installationsordner das folgende Bild:



Das zum Erstellen und Ausführen von .NET- und damit auch von Avalonia-Anwendungen geeignete Kommandozeilenprogramm **dotnet.exe**, das sogenannte **CLI** (***C**ommand **L**ine **I**nterface*), landet hier:

C:\program files\dotnet\sdk\dotnet.exe

Ein während der Installation automatisch vorgenommener Eintrag in den Suchpfad für ausführbare Programme ermöglicht den bequemen Aufruf von **dotnet.exe** in jedem Konsolenfenster.

Über das Kommando **dotnet --list-sdks** erfährt man die auf einem Rechner installierten .NET – SDKs, z. B.:

```
C:\Users\baltes>dotnet --list-sdks
8.0.132 [C:\Program Files\dotnet\sdk]
9.0.318 [C:\Program Files\dotnet\sdk]
10.0.113 [C:\Program Files\dotnet\sdk]
10.0.402 [C:\Program Files\dotnet\sdk]
```

Die Bestandteile einer SDK-Versionsangabe (z. B. 10.0.401) haben die folgenden Bedeutungen:

- Die beiden ersten, durch einen Punkt getrennten Ziffern geben die unterstützte .NET – Haupt - bzw. – Nebenversion an (im Beispiel: 10.0).
- Dann folgt nach einem weiteren Punkt die sogenannte *Featuregruppe*, die mit 1 startet und beim vierteljährlichen Erscheinen einer neuen Visual Studio - Nebenversion ansteigt (im Beispiel: 4).
- Am Ende steht (ohne trennenden Punkt) die beim monatlichen Wartungs-Update heraufgesetzte Patch-Version (im Beispiel 01).[1]

Mit dem Kommando **dotnet --list-runtimes** ist zu erfahren, welche .NET – Laufzeitumgebungen auf einem Rechner vorhanden sind, z. B.:

[1] https://learn.microsoft.com/en-us/dotnet/core/porting/versioning-sdk-msbuild-vs

```
C:\Users\baltes>dotnet --list-runtimes
Microsoft.NETCore.App 8.0.31 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 8.0.32 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 9.0.20 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 10.0.12 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 10.0.13 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.WindowsDesktop.App 8.0.31 [C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 8.0.32 [C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 9.0.20 [C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 10.0.12 [C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 10.0.13 [C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop.App]
```

Die Laufzeitumgebungen sind für verschiedene Anwendungstypen zuständig:

- **.NETCore.App 10.0.13**
  Hier befinden sich die Plattform-unabhängigen ***BCL***-Typen (***B**ase **C**lass **L**ibrary*) zur Ausführung von .NET - Anwendungen.
- **WindowsDesktop.App 10.0.13**
  Weil wir bei der Visual Studio – Installation als **Workload** (Einsatzfeld) auch die **.NET - Desktopentwicklung** wählen (siehe Abschnitt 2.2), befindet sich auch die .NET – Laufzeitumgebung für Windows-Anwendungen mit grafischer Bedienoberfläche in **WinForms- oder WPF-Technik** auf dem Rechner. Avalonia verwendet diese Laufzeitumgebung aufgrund seiner Plattform-unabhängigen Technik aber *nicht*.

Die Versionierung der Laufzeitumgebungen verwendet das Schema

Hauptversion.Nebenversion.Patch

Jede Laufzeitumgebung befindet sich in einem eigenen Ordner, z. B.:

C:\Program Files\dotnet\shared\Microsoft.NETCore.App\10.0.13

Mit dem Kommando **dotnet sdk check** lässt die Aktualität der installierten SDKs und Laufzeitumgebungen prüfen, z. B.:

```
C:\Windows\System32>dotnet sdk check
.NET SDKs:
Version       Status
-----------------------------------------------------------
8.0.132       Die Unterstützung von .NET 8.0 läuft in Kürze aus.
9.0.318       Die Unterstützung von .NET 9.0 läuft in Kürze aus.
10.0.113      Auf dem neuesten Stand.
10.0.402      Auf dem neuesten Stand.

.NET-Runtimes:
Name                              Version      Status
---------------------------------------------------------------------------------
Microsoft.NETCore.App             8.0.32    Die Unterstützung von .NET 8.0 läuft in Kürze aus.
Microsoft.WindowsDesktop.App      8.0.32    Die Unterstützung von .NET 8.0 läuft in Kürze aus.
Microsoft.NETCore.App             9.0.20    Die Unterstützung von .NET 9.0 läuft in Kürze aus.
Microsoft.WindowsDesktop.App      9.0.20    Die Unterstützung von .NET 9.0 läuft in Kürze aus.
Microsoft.NETCore.App             10.0.13   Auf dem neuesten Stand.
Microsoft.WindowsDesktop.App      10.0.13   Auf dem neuesten Stand.
```

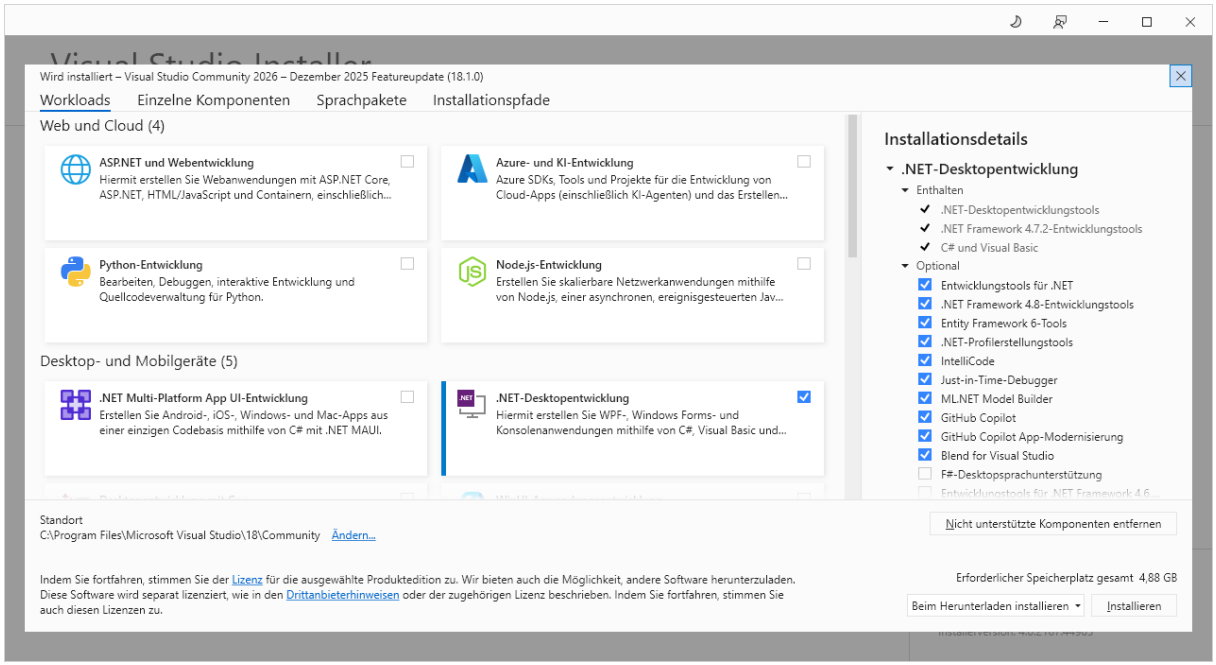
2.2 Microsoft Visual Studio Community 2026 für Windows

Im Manuskript kommt zur Entwicklung von Avalonia-Anwendungen die Entwicklungsumgebung Visual Studio Community 2026 zum Einsatz. Auf der Webseite

https://visualstudio.microsoft.com/downloads/

ist der Web-Installer **VisualStudioSetup.exe** verfügbar, der die benötigten Dateien während der Installation aus dem Internet bezieht.

Während der Installation durch den **Visual Studio Installer** erscheint der folgende Dialog zur Wahl der **Workloads** (Einsatzfelder):



Um gelegentlich einen vergleichenden Blick auf die WPF-Entwicklung werfen zu können, installieren wir die **.NET - Desktopentwicklung**, die für Avalonia *nicht* erforderlich ist.[1]

Wer Avalonia UI - Anwendungen nicht nur für die Desktop-Betriebssysteme Linux, macOS und Windows, sondern auch für die Mobilsysteme Android und iOS entwickeln möchte, sollte die **.NET Multi-Platform App UI Entwicklung** markieren. Zwar entstehen mit Avalonia UI keine MAUI-Anwendungen, doch lassen sich auf diese Weise die für Android und iOS erforderlichen SDKs, Laufzeitkomponenten und Emulatoren bequem installieren (Del Sole 2025). Für die Entwicklung von Anwendungen für iOS benötigt man allerdings zusätzlich einen macOS-Rechner mit der Entwicklungsumgebung Xcode. Wir verzichten auf die MAUI-Installation, weil das vorliegende E-Book aus Zeitgründen keine mobilen Anwendungen behandelt.

Zum Entwickeln von WebAssembly-Anwendungen, sind die optionalen Angebote im **Visual Studio Installer** *nicht* relevant. Dazu benötigt man die per CLI (**dotnet.exe**) zu installierenden **wasm-tools**.[2]

Bei Bedarf kann man den **Visual Studio Installer** später über einen Link im Windows-Startmenü erneut starten, um den Installationsumfang zu erweitern.

---

1 https://docs.avaloniaui.net/docs/platform-specific-guides/windows

2 https://docs.avaloniaui.net/docs/platform-specific-guides/webassembly

2.3 Visual Studio - Erweiterung für Avalonia UI

Wir ergänzen nun ...

- Avalonia-Projektvorlagen für .NET – Anwendungen[1]
- die Visual Studio – Erweiterung für Avalonia UI.

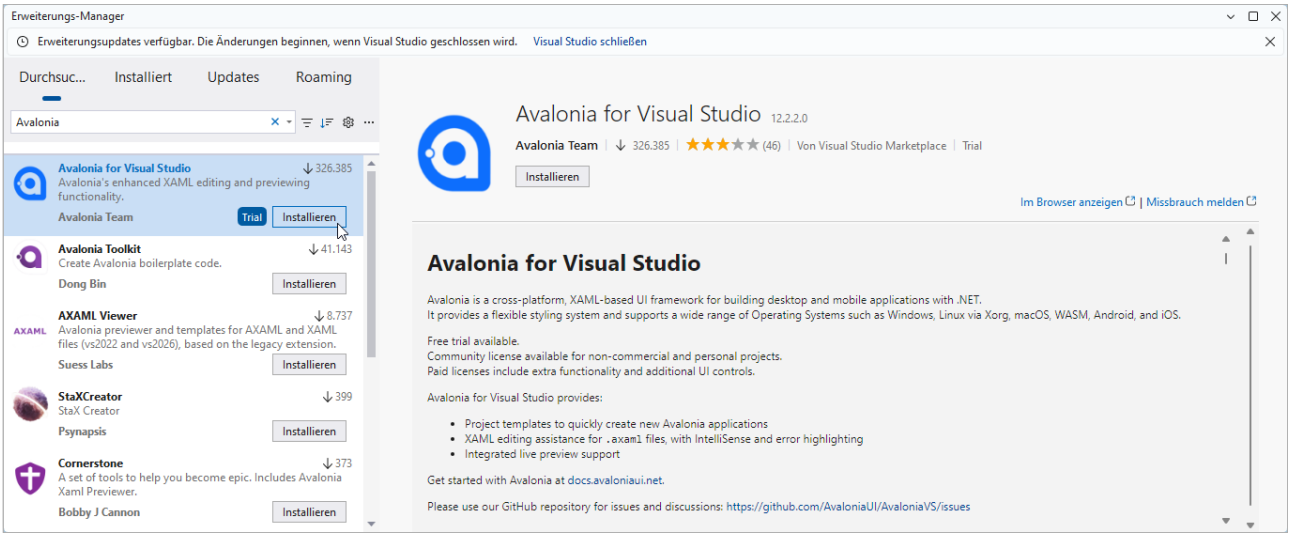
Weil die Avalonia-Erweiterung für das von uns benutzte Visual Studio die Avalonia-Projektvorlagen mitbringt, ist keine separate Installation der Projektvorlagen über das folgende **dotnet**-Kommando erforderlich:

```
>dotnet new install Avalonia.Templates
```

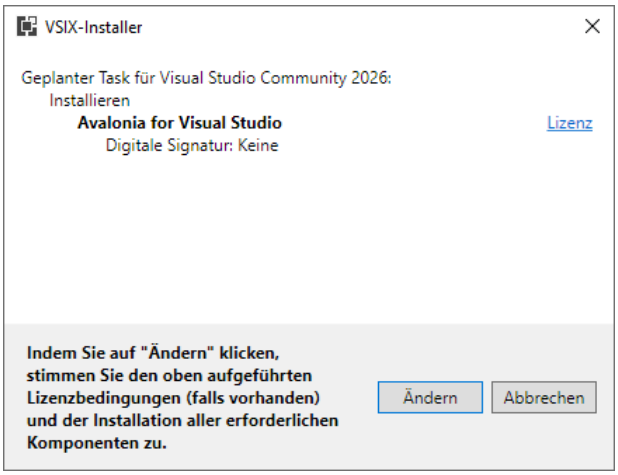
Wir starten das Visual Studio und installieren die Avalonia UI - Erweiterung über den Menübefehl[2]

Erweiterungen > Erweiterungen verwalten

Wir tragen **Avalonia** ins Suchfeld ein und **installieren** die Version 12.2.2.0:



Man muss das Visual Studio beenden, um die Installation der Erweiterung durch den **VSIX Installer** zu ermöglichen:

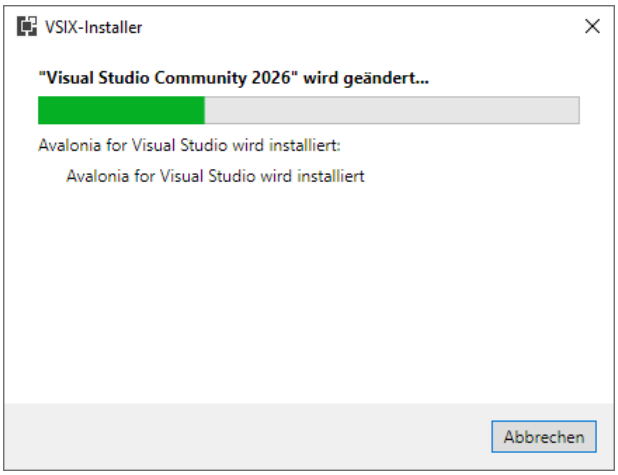


Nach einem Klick auf **Ändern** beginnt die Installation:

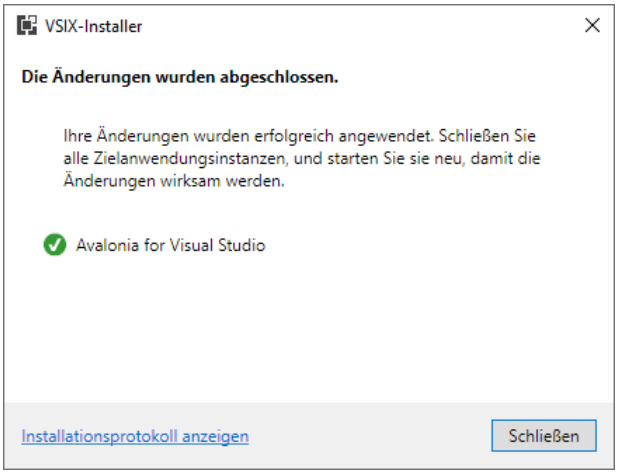
---

1 https://docs.avaloniaui.net/docs/get-started/install-avalonia

2 https://docs.avaloniaui.net/docs/get-started/set-up-your-ide



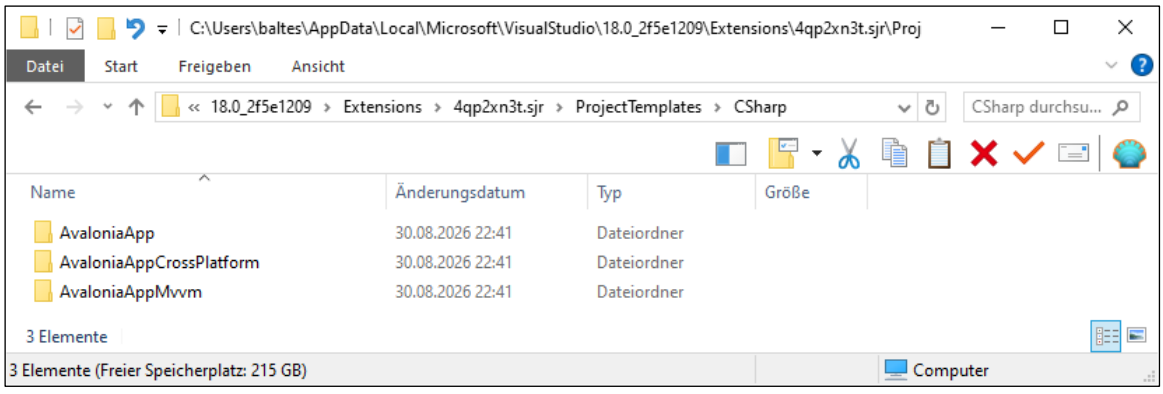
Anschließend startet man das Visual Studio neu:



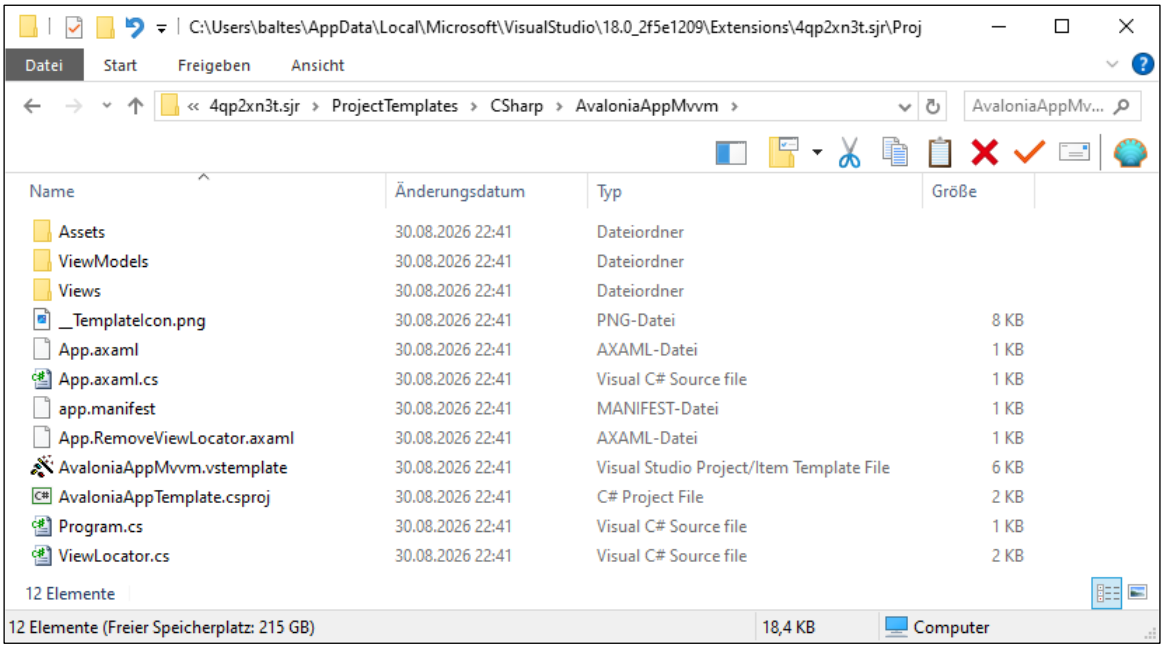
Bei einer Installation über die Erweiterung zum Visual Studio 2026 landen die Avalonia-Projektvorlagen hier:

C:\Users\<*Benutzername*>\AppData\Local\Microsoft\VisualStudio\18.0_2f5e1209\Extensions\<*zufälliger Ordnername*>\ProjectTemplates\CSharp

Eine Inspektion der Ordner



zeigt die Bestandteile einer Projektvorlage, z. B. bei der Vorlage **Avalonia .NET MVVM App**:



2.4 Erstellungswerkzeuge erkunden

In diesem Abschnitt lernen Sie elementare Arbeitsschritte und Werkzeuge kennen. Wir lassen zunächst ein Hallo-Programm weitgehend automatisch erstellen, produzieren aber auch schon eine nützliche Anwendung, die DM-Beträge in Euro-Beträge wandelt:[1]



[1] Benötigt wird ein solches Programm z. B. noch bei der Auszahlung von Erbschaftsbeträgen aufgrund von Testamenten, die in der DM-Ära entstanden sind.

2.4.1 Projekt anlegen

Wir klicken im Visual Studio – Startdialog



auf

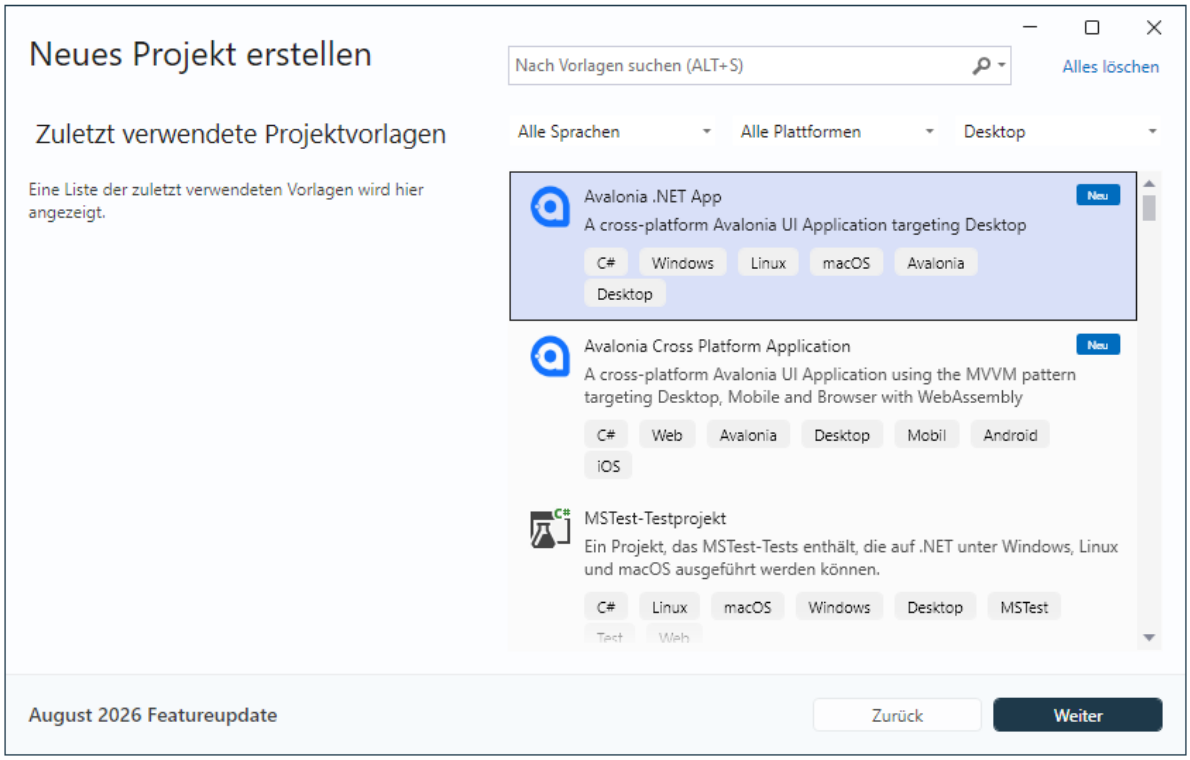
Neues Projekt erstellen

oder nutzen den Menübefehl

Datei > Neu > Project/Solution

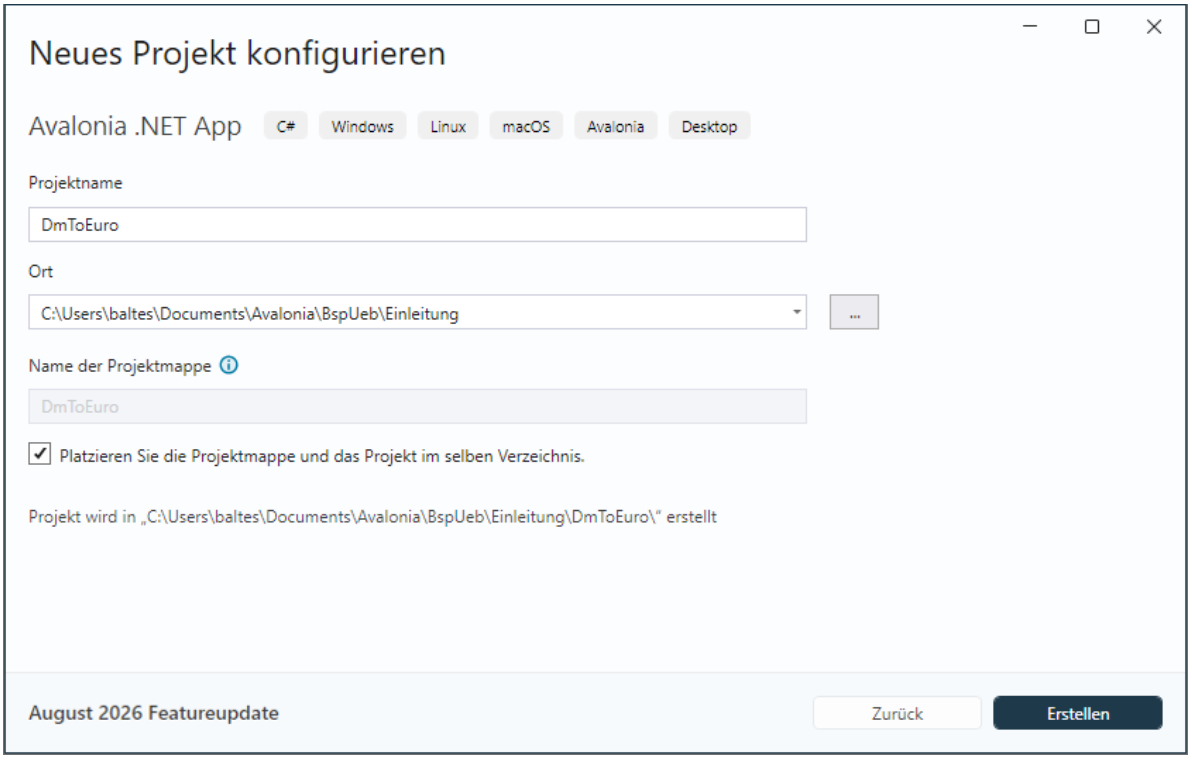
in der bereits aktiven Entwicklungsumgebung.

Im Dialog **Neues Projekt stellen** wählen wir von den Optionen zum Anwendungstyp **Desktop** (siehe Drop-Down –Liste oben rechts)



die Projektvorlage für eine **Avalonia .NET App** in C#. Damit entsteht eine unter Linux, macOS und Windows einsetzbare GUI-Anwendung. Auf die MVVM-Architektur (siehe Kapitel 12) verzichten wir vorläufig noch, damit die Lernkurve nicht zu steil wird.

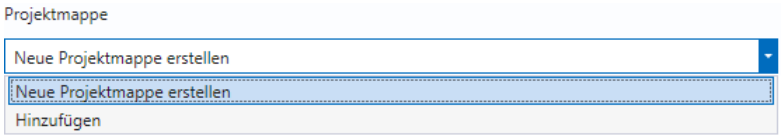
Wir machen **weiter** und legen im nächsten Dialog



den **Projektnamen** `DmToEuro` sowie einen **Ort** (ein übergeordnetes Verzeichnis) für den Projektordner fest, z. B.

C:\Users\<*Benutzername*>\Documents\Avalonia\BspUeb

Jedes Visual Studio - Projekt gehört zu einer **Projektmappe**, die eine Familie von zusammengehörigen Projekten verwaltet (z. B. ein Anwendungsprogramm und eine Klassenbibliothek). Ist beim Erstellen eines neuen Projekts keine Mappe geöffnet, dann legt das Visual Studio eine Mappe an. Anderenfalls kann man das neue Projekt in einer neuen Mappe erstellen oder in die geöffnete Mappe hinzufügen:



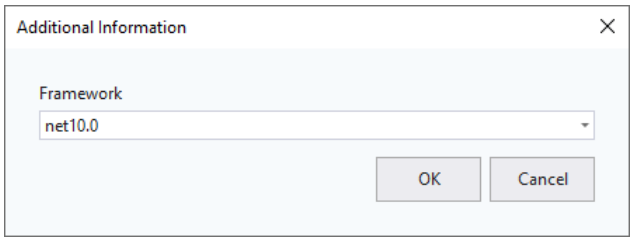
Die englischsprachige Version der Entwicklungsumgebung bezeichnet eine Projektmappe als *Solution*, und gelegentlich taucht die deutsche Übersetzung *Lösung* auf.

Die Projektmappen der Manuskriptbeispiele enthalt meist nur ein einziges Projekt. In dieser Situation ist es überflüssig, im Ordner einer Projektmappe einen Unterordner für das einzige enthaltene Projekt anzulegen. Daher markieren wir das Kontrollkästchen **Platzieren Sie die Projektmappe und das Projekt im selben Verzeichnis**. Im gemeinsamen Verzeichnis für das Projekt und die Projektmappe erscheinen:

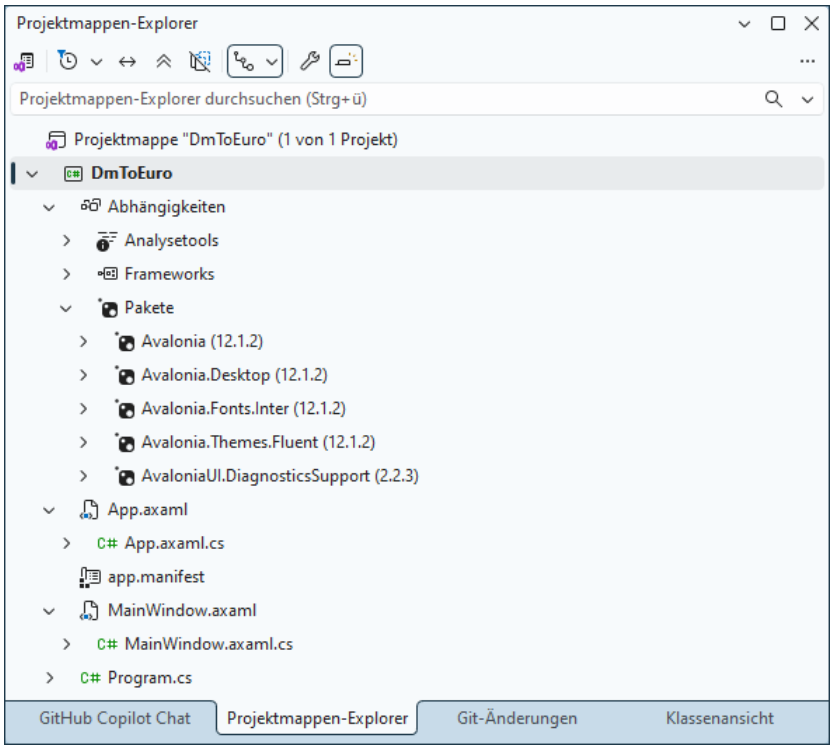
- die Projektdatei (im Beispiel **DmToEuro.csproj**)
- die Projektmappendatei (im Beispiel **DmToEuro.slnx**) mit einer von der englischen Bezeichnung *solution* abgeleiteten Namenserweiterung

Beide Dateien haben das im Abschnitt 5.1 beschriebene XML-Format.

Nach einem Mausklick auf **Erstellen** akzeptieren wir im nächsten Dialog als **Framework** das aktuelle und mit LTS-Support ausgestattete .NET 10.0:



Einige Sekunden nach einem Mausklick auf **OK** entsteht das Projekt, und das Visual Studio präsentiert am rechten Fensterrand im **Projektmappen-Explorer** eine Baumansicht zur Projektmappenverwaltung. Hier erscheint ein Eintrag für jedes Projekt in der potenziell aus mehreren Projekten bestehenden Projektmappe. Die Mappen zu unseren Beispielen enthalten in der Regel nur *ein* Projekt, und die Namen von Mappe und Projekt sind identisch, z. B.:



Der **Projektmappen-Explorer** zeigt fast denselben Inhalt wie bei einem WPF-Projekt (ohne MVVM-Architektur). Wir sehen:[1]

- die vorausgesetzten Bibliotheken (**Abhängigkeiten**)
  Im Vergleich zu einem WPF-Projekt fehlt die Abhängigkeit vom Framework **Microsoft.WindowsDesktop.App**. Stattdessen bestehen Abhängigkeiten von den NuGet-Paketen, auf denen Avalonia UI basiert.
- die XAML-Dateien zur Bedienoberfläche und zur Anwendung
  Mit der *EXtensible Application Markup Language* werden wir uns im Verlauf des Manuskripts noch mehrfach befassen. Im Vergleich zu einem WPF-Projekt fällt die abweichende Namenserweiterung auf (**axaml** statt **xaml**).

[1] Ein Ordner mit dem fertigen Visual Studio – Projekt ist hier zu finden:
…\BspUeb\Einleitung\DmToEuro

- die C# - Quellcodedateien
  Abweichend von WPF ist die Quellcodedatei **Program.cs** vorhanden. Sie enthält die Startklasse **Program** mit der Startmethode **Main()**, die eine Anwendungskonfiguration vornimmt:

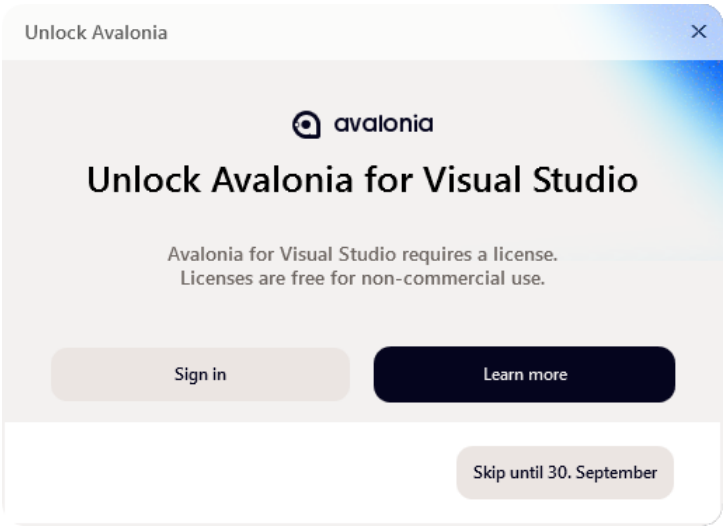
```
class Program {
    [STAThread]
    public static void Main(string[] args) => BuildAvaloniaApp()
        .StartWithClassicDesktopLifetime(args);

    public static AppBuilder BuildAvaloniaApp()
        => AppBuilder.Configure<App>()
            .UsePlatformDetect()
#if DEBUG
            .WithDeveloperTools()
#endif
            .WithInterFont()
            .LogToTrace();
}
```

  Die Methode **UsePlatformDetect()** sorgt für die zur vorgefundenen Plattform (Linux, macOS, Windows) passende Konkretisierung von abstrakt definierten GUI-Funktionen (z. B. Fensterbehandlung, Zwischenablage, Dateiauswahldialog).
  Der **WithInterFont()** - Aufruf bewirkt die Nutzung der von Rasmus Andersson entworfenen Open Source – Schriftartfamilie **Inter** auf allen unterstützten Plattformen.[1] Im Unterschied zur Verwendung von Plattform-spezifischen Schriftfamilien erhält man ein einheitliches Erscheinungsbild.

2.4.2 Erwerb einer Community-Lizenz

Beim Wechsel zum Registerblatt mit der Datei **MainWindow.axaml** (XAML-Deklaration der Bedienoberfläche, siehe Abschnitt 2.4.4) erscheint ein Hinweis auf das erforderlichen Entsperren der Visual Studio – Erweiterung von Avalonia UI:

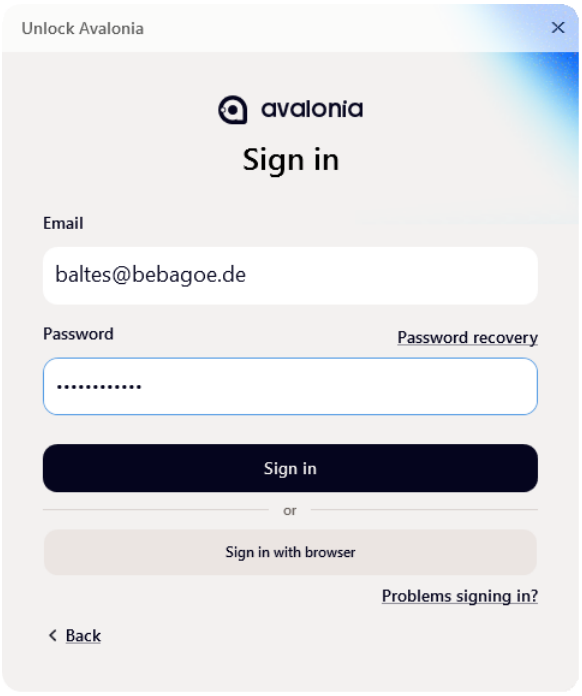


Dazu genügt eine **Community-Lizenz**, und die erforderlichen Schritte zum Erwerb sind im Anhang A beschrieben.

Mit der (ein Jahr lang gültigen) Community-Lizenz lässt sich die Visual Studio – Erweiterung nach einem Klick auf **Sign in** entsperren, was nur beim ersten Anmelden erforderlich ist:

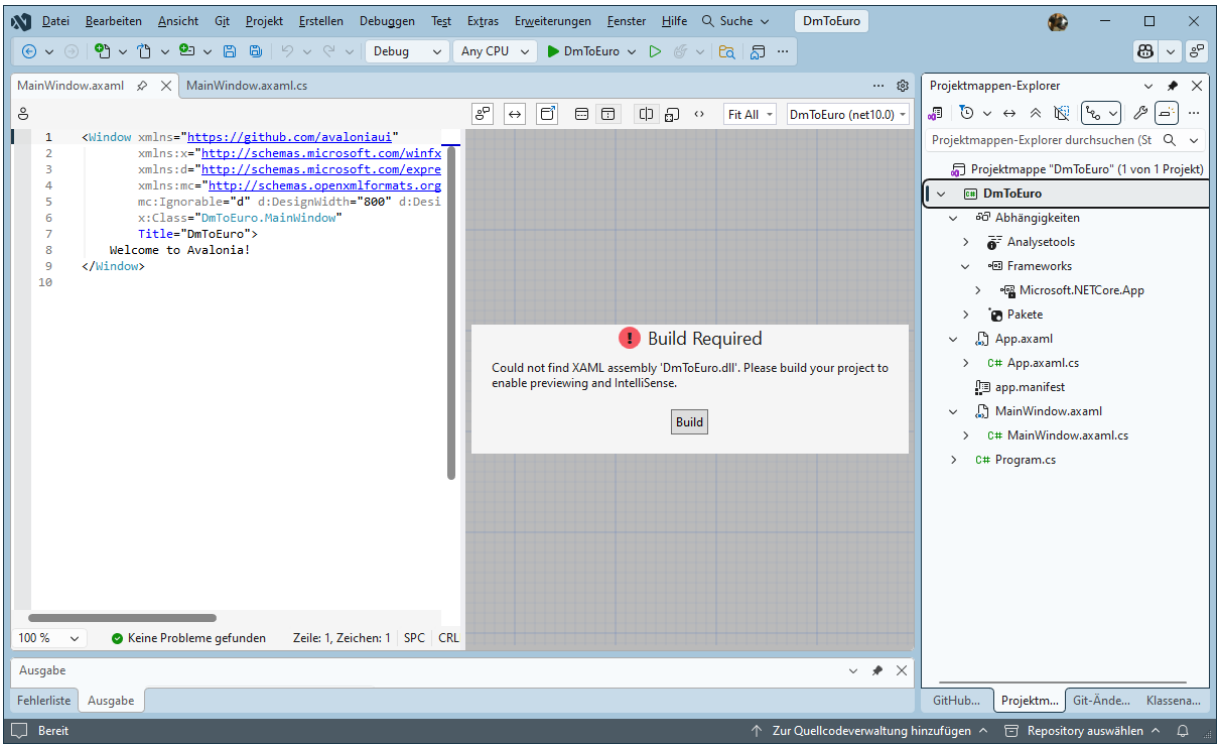
---

[1] https://rsms.me/inter/

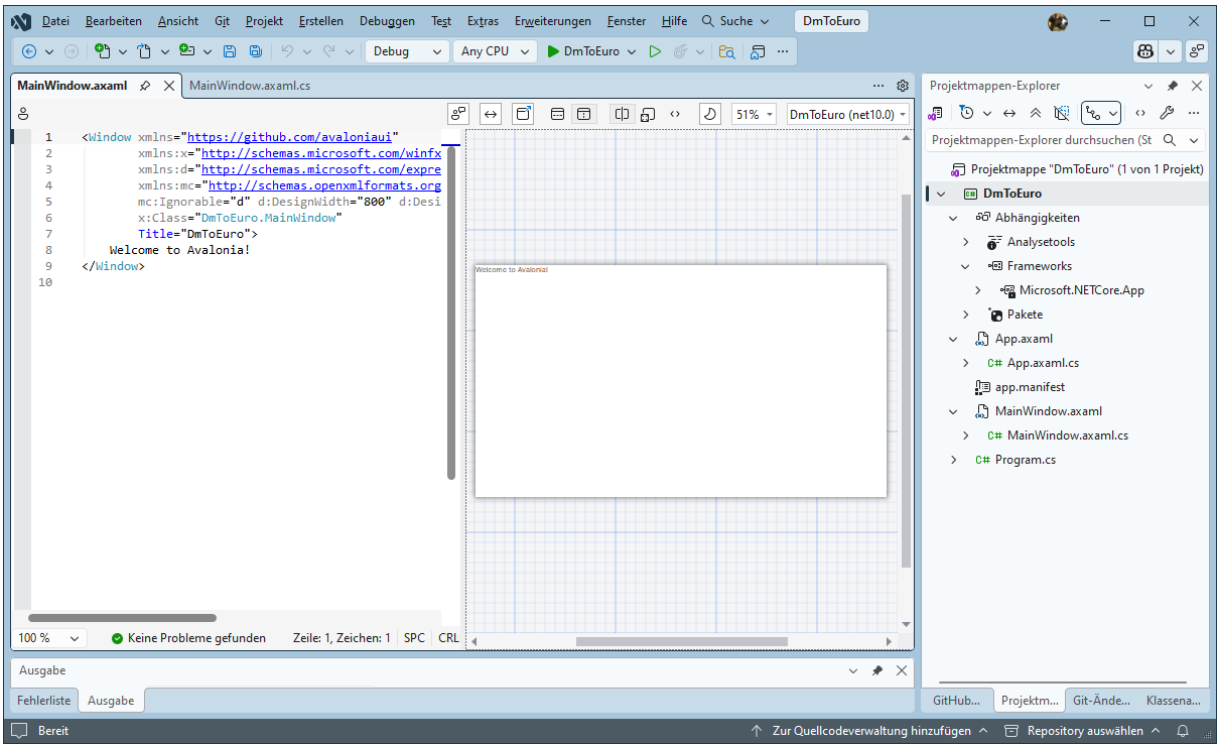


2.4.3 XAML-Vorschau und Dateien im Erstellungsordner

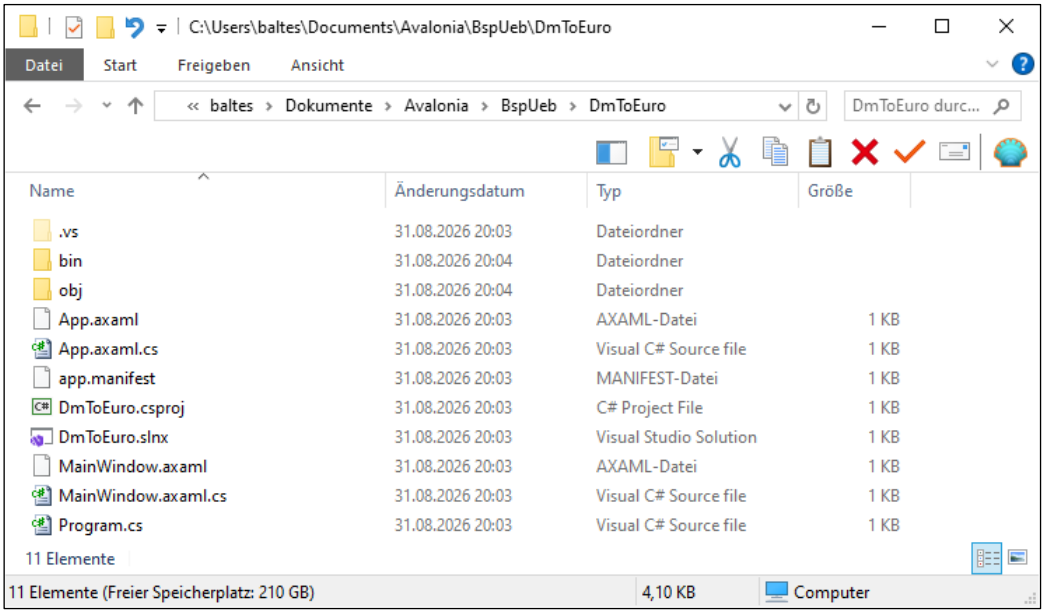
Damit die XAML-Vorschau klappt, muss man das Programm (mit dem Assembly **DmToEuro.dll**) erstellen lassen, z. B. per Klick auf den Schalter **Build**:



Nun ist die Benutzeroberfläche des (bereits startfähigen) Programms zu sehen, die sich im Beispiel noch auf einen kurzen Text beschränkt: Welcome to Avalonia!



Weil wir uns *gegen* einen Projekt*unter*ordner im Projektmappenordner entschieden haben, resultiert ein gemeinsamer Ordner für das Projekt und die Projektmappe:



2.4.4 Bedienoberfläche entwerfen

Wir machen uns nun daran, das Anwendungsfenster unseres Währungskonverters mit den benötigten Bedienelementen auszustatten, die man auch als *Steuerelemente* (engl.: *controls*) bezeichnet. Für WPF enthält das Visual Studio einen GUI-Designer, den man fast wie ein Grafikprogramm verwendet und Steuerelemente per Drag & Drop auf die Fensteroberfläche befördert, wobei der Designer die Deklaration der Bedienoberfläche in der *EXtensible Application Markup Language* (XAML) erledigt. Einen analogen Komfort bietet Avalonia UI nur im Rahmen der

kostenpflichtigen Plus-Lizenz.[1] Wir sind also gezwungen, die XAML-Beschreibungssprache zu lernen, wobei Avalonia das Fensterdesign durch eine Vorschau, Code-Vervollständigung per IntelliSense und das Markieren (Unterschlängeln) von Fehlern unterstützt. Dabei resultiert erfahrungsgemäß ein besserer XAML-Code als durch Drag & Drop plus Positionieren und Dimensionieren per Maus.

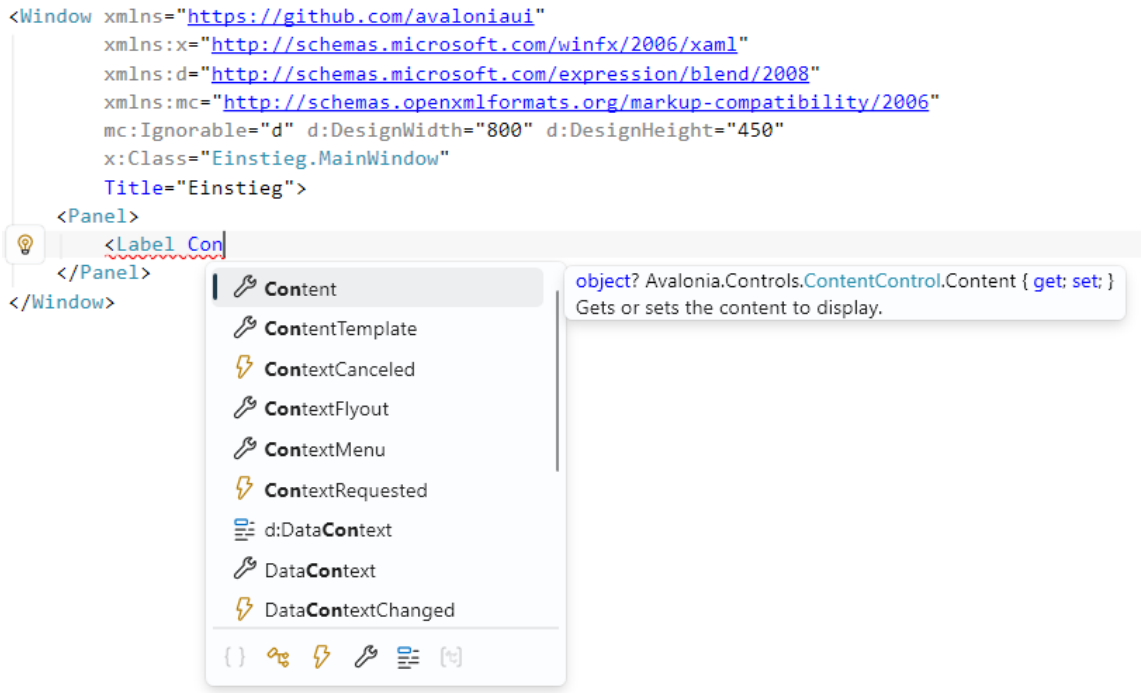
2.4.4.1 Arbeitshilfen und angestrebtes Ergebnis

Trotz des fehlenden Drag&Drop – Designers stehen wir beim Fensterdesign nicht ohne Hilfe da:

- Die IntelliSense-Funktion im Visual Studio hilft beim Erstellen der XAML-Syntax. Wenn man z. B. das Startelement eines **Panel**-Layoutcontainers (siehe Abschnitt 10.1) einfügt, dann erscheint automatisch das ergänzende Endelement:

  ```
  <Panel></Panel>
  ```

  IntelliSense schlägt zulässige Syntaxerweiterungen vor, die man per Doppelklick oder Tabulatortaste übernehmen kann, z. B.:

  Die Liste passt sich dynamisch an die bisherige Eingabe an.

- Fehlerhaften XAML-Code markiert die Entwicklungsumgebung schon bei der Eingabe durch rotes Unterschlängeln:

  ```
  <Panel>
      <Label Kontent="DM:"
  </Panel>
  ```

- In der XAML-Vorschau zeigt Avalonia UI das Resultat von korrekter Syntax oder eine Fehlermeldung.
- Schließlich übernimmt der Copilot mit vorgeschlagenen Vervollständigungen einige Arbeit.

Mit diesen Hilfen ist eine Bedienoberfläche per XAML-Syntax erheblich einfacher zu deklarieren als durch C# - Anweisungen.

---

[1] https://docs.avaloniaui.net/tools

Zur Realisation des Währungskonverters



werden wir anschließend in der Datei **MainWindow.axaml** den initial vorhandenen Text

```
Welcome to Avalonia!
```

durch eine Instanz des Layoutmanagers **Panel** ersetzen, der die Steuerelemente aufnimmt:

- ein **Label**-Steuerelement mit der Beschriftung "DM:"
- ein **TextBox**-Steuerelement zum Eintragen des DM-Betrags
- ein **Label**-Steuerelement mit der Beschriftung "Euro:"
- ein **Label**-Steuerelement mit Rahmen zur Anzeige des Euro-Betrags
- ein **Button**-Steuerelement zur Veranlassung der Konvertierung

2.4.4.2 Eigenschaften des Anwendungsfensters

Im **Window**-Startelement des XAML-Codes...

- ersetzen wir das Attribut **d:DesignWidth** durch das Attribut **Width** mit dem Wert "500", Das legt die initiale (vom Benutzer änderbare) Breite des Fensters fest.
- ersetzen wir das Attribut **d:DesignHeight** durch das Attribut **Height** mit dem Wert "180", Das legt die initiale (vom Benutzer änderbare) Höhe des Fensters fest.
- ändern wir den Wert des Attributs **Title** auf "DM → Euro".
  Den rechtsgerichteten Pfeil erhält man unter Windows so:
  - Ziffernblock per **Num**-Taste einschalten (**Num**-LED leuchtet)
  - **Alt**-Taste drücken und festhalten
  - auf dem Ziffernblock der Tastatur nacheinander die Tasten **2** und **6** drücken
  - **Alt**-Taste loslassen
  - Ziffernblock per **Num**-Taste ausschalten

Im XAML-Fenster ergibt sich der folgende Zwischenstand:

```
<Window xmlns="https://github.com/avaloniaui"
        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"
        mc:Ignorable="d" Width="500" Height="180"
        x:Class="DmToEuro.MainWindow"
        Title="DM → Euro">
    Welcome to Avalonia!
</Window>
```

Weil wir in der Regel die XML-Namensräume

```
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

nicht benötigen, lässt sich das **Window**-Startelement entschlacken:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="DmToEuro.MainWindow"
        Title="DM → Euro" Width="500" Height="180">
    Welcome to Avalonia!
</Window>
```

2.4.4.3 Label mit der Beschriftung "DM:"

Das Startelement des **Label**-Objekts mit der Beschriftung "DM:" soll ...

- ein **Content**-Attribut mit dem Wert "DM:" erhalten,
- ein **HorizontalAlignment**-Attribut mit dem Wert "**Left**" erhalten,
  Damit ist das **Label** am linken Container-Rand fixiert.
- ein **VerticalAlignment**-Attribut mit dem Wert "**Top**" erhalten,
  Damit ist das **Label** am oberen Container-Rand fixiert.
- ein **Margin**-Attribut mit dem Wert "40,40,0,0" erhalten,
  Somit hat das **Label** zum linken bzw. oberen Container-Rand den Abstand 40.
- ein **Width**-Attribut mit dem Wert "60" erhalten,
  Das legt die Breite des Steuerelements fest.
- ein **Height**-Attribut mit dem Wert "30" erhalten.
  Das legt die Höhe des Steuerelements fest.

Wir ersetzen den initial vorhandenen Text

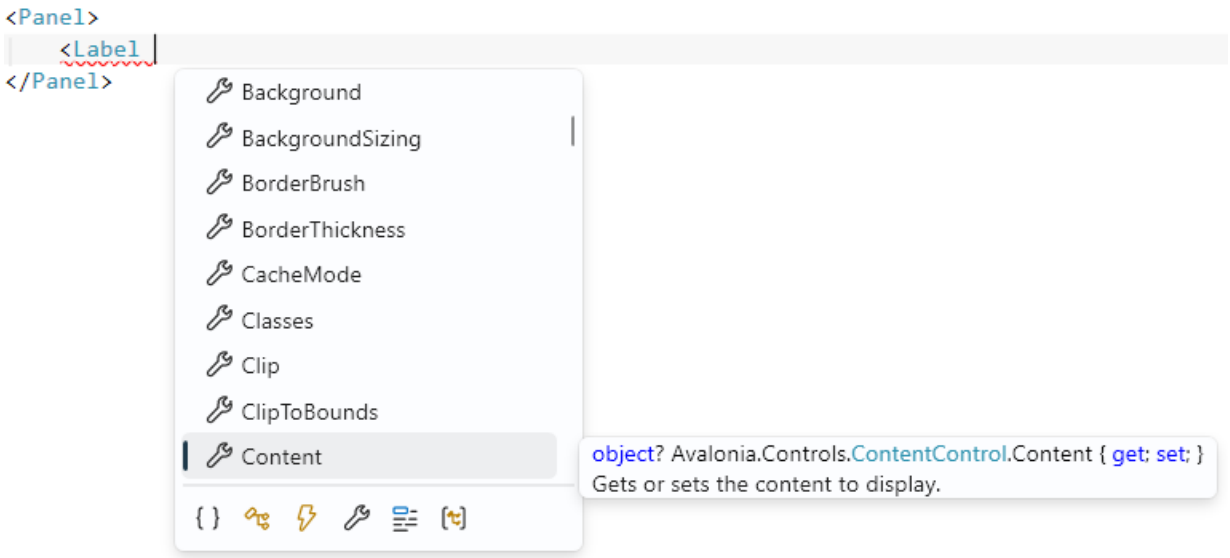
```
Welcome to Avalonia!
```

durch das Start- und das Endelement eines **Panel**-Layoutcontainers:

```
<Panel>

</Panel>
```

Zu dem hier eingefügten **Label**-Startelement präsentiert IntelliSense die zulässigen Erweiterungen in einer Liste, aus der wir das **Content**-Attribut per Tabulatortaste wählen:



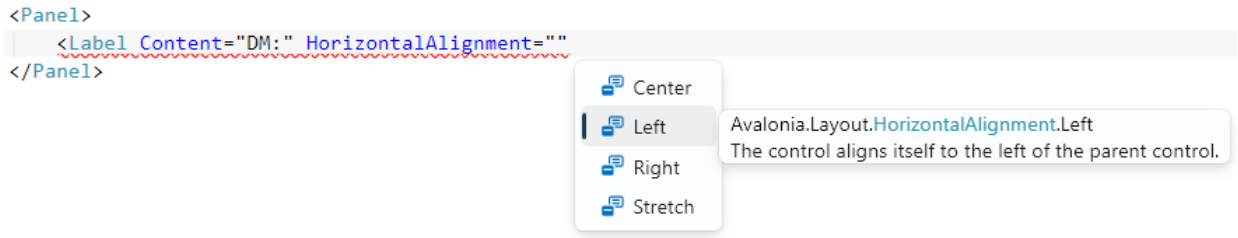
Wir setzen ein Leerzeichen hinter

```
<Label Content="DM:"
```

und ergänzen Attribut **HorizontalAlignment**

```
<Label Content="DM:" HorizontalAlignment="Left"
```

mit der Hilfe von IntelliSense

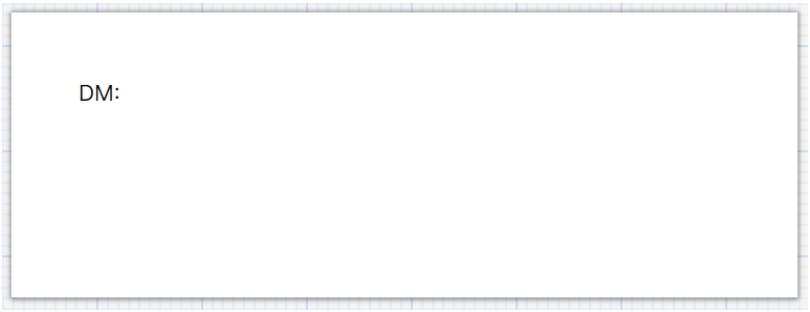


Analog ergänzen wir die Attribute **VerticalAlignment**, **Margin**, **Width** und **Height**:

```
<Label Content="DM:" HorizontalAlignment="Left" VerticalAlignment="Top"
       Margin="40,40,0,0" Width="60" Height="30" />
```

Weil das **Label**-Element über die Attribute hinaus keinen Inhalt hat, benötigt es kein Endelement. In diesem Fall muss man ersatzweise im Startelement vor die terminierende eckige Klammer einen Schrägstrich setzen.

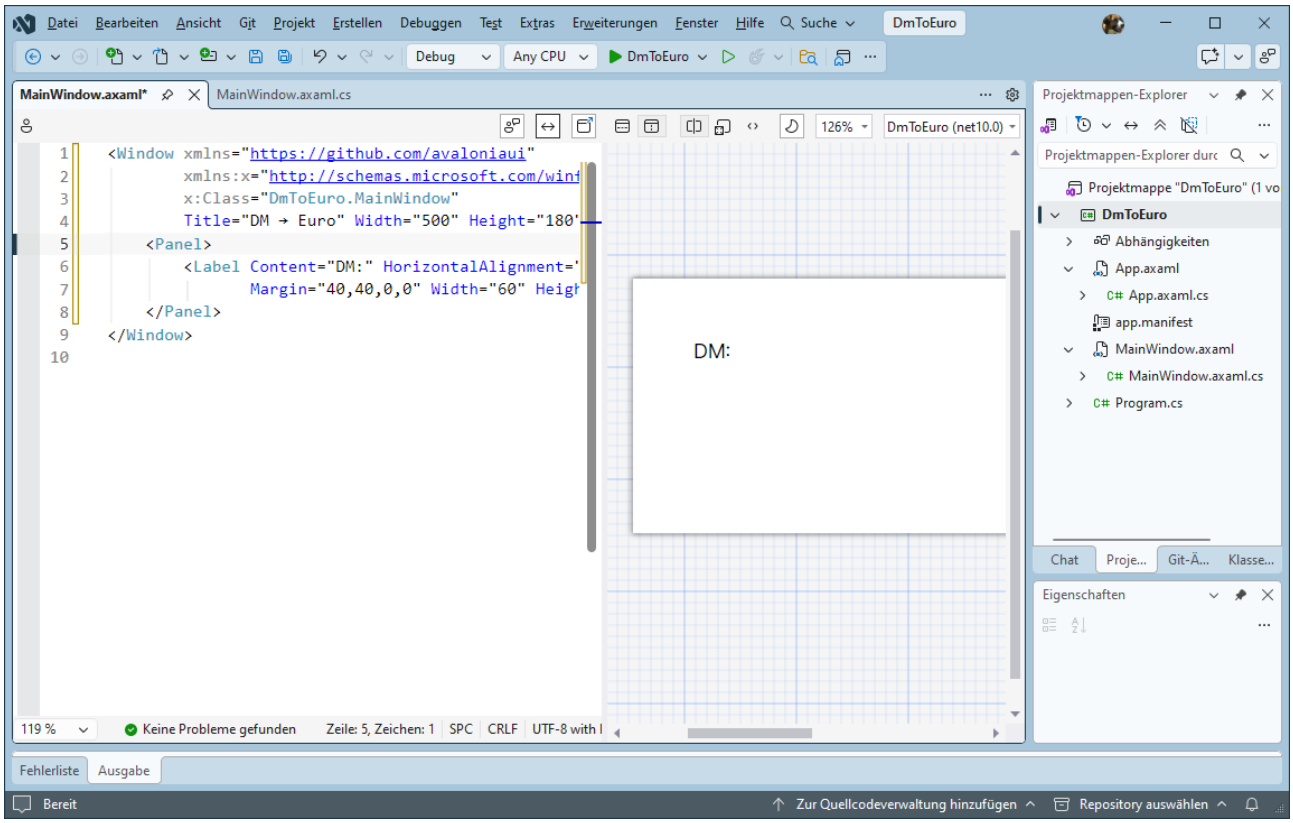
In der XAML-Vorschau von Avalonia UI ist nun das fertige **Label**-Steuerelement zu sehen:



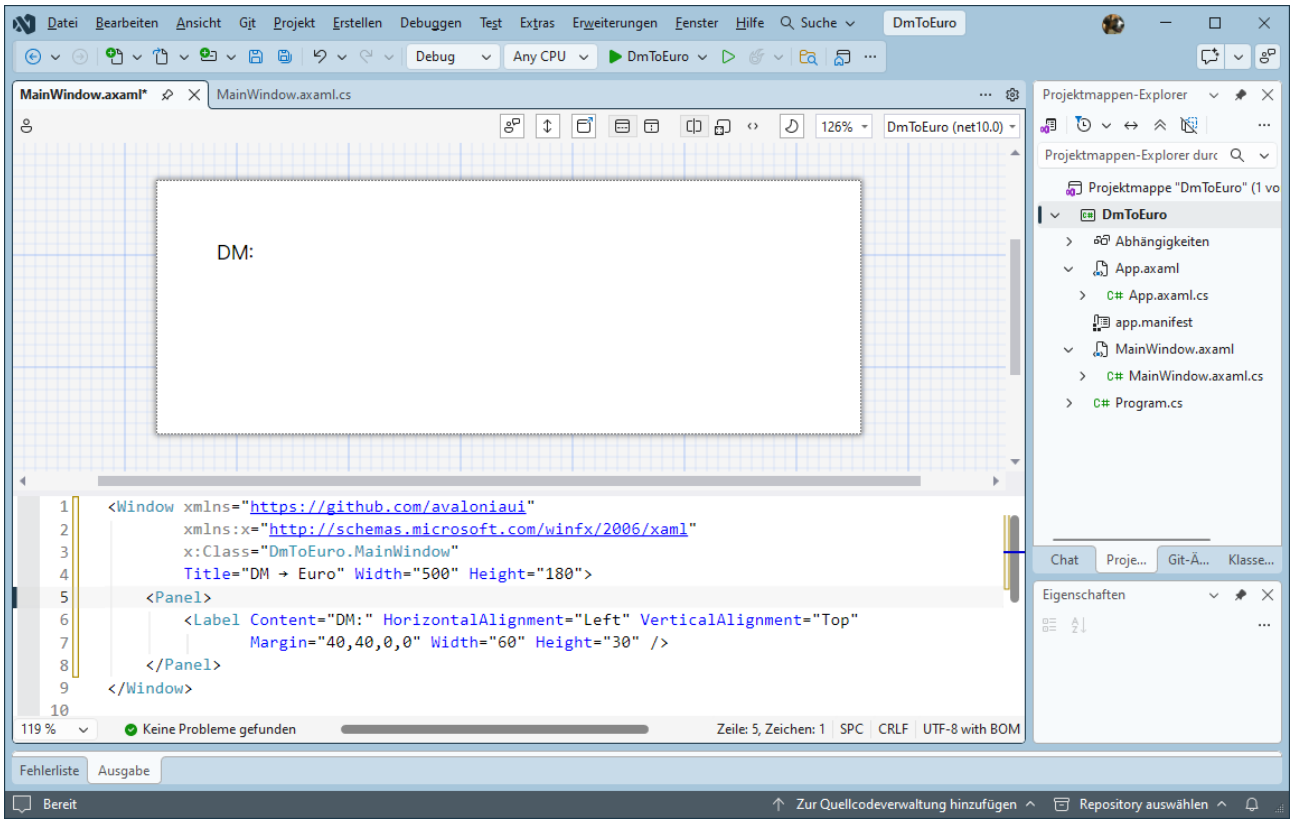
Mit den XAML-Syntaxregeln werden wir uns im Kapitel 5 gründlich beschäftigen.

2.4.4.4 Wechsel zum horizontalen Layout des GUI-Editors

Beim voreingestellten vertikalen Layout des GUI-Editors ist das Editieren des XAML-Codes durch den beschränkten horizontalen Platz erschwert:



Nach dem Wechsel zum horizontalen Layout über den Schalter  und einem Platztausch von XAML-Code und Vorschau über den Schalter  fällt das Editieren des XAML-Codes leichter:

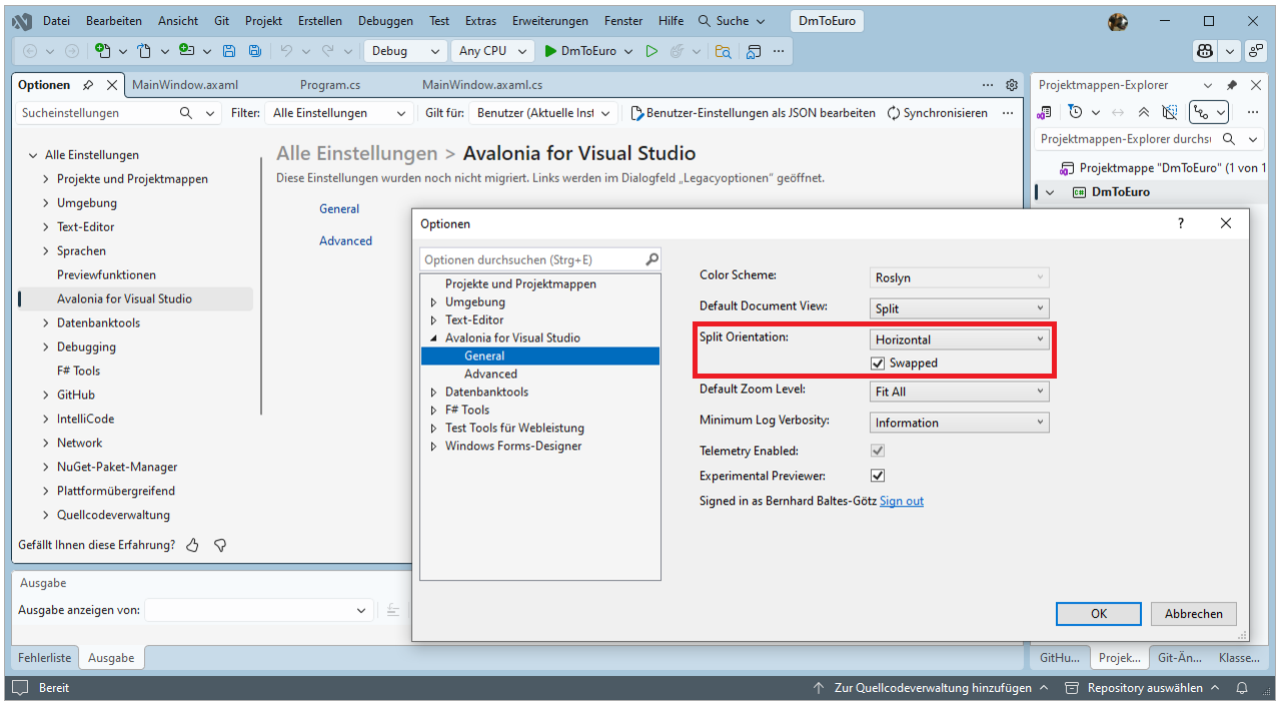


Um diese Einstellung permanent zu machen, nimmt man nach

Extras > Optionen >Avalonia for Visual Studio > General

im **Optionen**-Dialog die folgenden Einstellungen vor:

- Wahl von **Horizontal** in der Drop-Down – Liste **Split Orientation**
- Markierung von **Swapped**



2.4.4.5 TextBox zur Aufnahme des DM-Betrags

Auch für die **TextBox** zur Aufnahme des zu konvertierenden DM-Betrags genügt ein Startelement. Es erhält ...

- ein **Name**-Attribut mit dem Wert "`eingabe`", um das Steuerelement in C# - Anweisungen ansprechen zu können,
- ein **HorizontalAlignment**-Attribut mit dem Wert "**Left**",
  Damit ist die **TextBox** am linken Container-Rand fixiert.
- ein **VerticalAlignment**-Attribut mit dem Wert "**Top**",
  Damit ist die **TextBox** am oberen Container-Rand fixiert.
- ein **Margin**-Attribut mit dem Wert "130,36,0,0",
  Somit hat die **TextBox** zum linken bzw. oberen Container-Rand den Abstand 130 bzw. 36.
- ein **Width**-Attribut mit dem Wert "90",
  Das legt die Breite fest.
- ein **Height**-Attribut mit dem Wert "30",
  Das legt die Höhe fest.
- ein **Padding**-Attribut mit dem Wert "5",
  Das legt den Innenrand (Abstand des Texts vom Rahmen) fest.
- ein **TextAlignment**-Attribut mit dem Wert "**Right**".
  Das sorgt für die bei Zahlen übliche rechtsbündige Anzeige.
- ein **VerticalContentAlignment**-Attribut mit dem Wert "**Center**" erhalten.

Im XAML-Code kommt das folgende Element hinzu:

```
<TextBox Name="eingabe" HorizontalAlignment="Left" VerticalAlignment="Top"
         Margin="130,36,0,0" Width="90" Height="30" Padding="5" TextAlignment="Right"
         VerticalContentAlignment="Center"/>
```

2.4.4.6 Label mit der Beschriftung "Euro:"

Das Startelement des Labels mit der Beschriftung "Euro:" soll ...

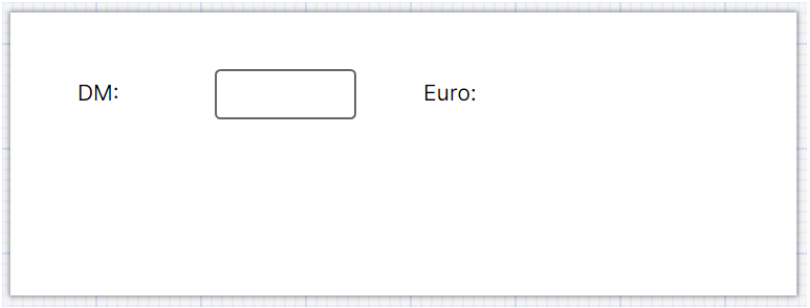
- ein **Content**-Attribut mit dem Wert "Euro:" erhalten,
- ein **HorizontalAlignment**-Attribut mit dem Wert "**Left**" erhalten,
  Damit ist das **Label** am linken Container-Rand fixiert.
- ein **VerticalAlignment**-Attribut mit dem Wert "**Top**" erhalten,
  Damit ist das **Label** am oberen Container-Rand fixiert.
- ein **Margin**-Attribut mit dem Wert "260,40,0,0" erhalten,
  Somit hat das **Label** zum linken bzw. oberen Container-Rand den Abstand 260 bzw. 40.
- ein **Width**-Attribut mit dem Wert "60" erhalten,
  Das legt die Breite fest.
- ein **Height**-Attribut mit dem Wert "30" erhalten.
  Das legt die Höhe fest.

Statt die Deklaration komplett neu zu erstellen,

```
<Label Content="Euro:" HorizontalAlignment="Left" VerticalAlignment="Top"
   Margin="260,40,0,0" Width="60" Height="30" />
```

kann man die Deklaration des Labels mit der Beschriftung "DM:" kopieren und dann anpassen.

Wir haben mittlerweile den folgenden Stand erreicht:



2.4.4.7 Label-Steuerelement mit Rahmen zur Anzeige des Euro-Betrags

Das **Label**-Steuerelement mit Rahmen zur Anzeige des Euro-Betrags soll ...

- ein **Name**-Attribut mit dem Wert "`ausgabe`" erhalten, um das Steuerelement in C# - Anweisungen ansprechen zu können,
- ein **HorizontalAlignment**-Attribut mit dem Wert "**Left**" erhalten,
- ein **VerticalAlignment**-Attribut mit dem Wert "**Top**" erhalten,
- ein **Margin**-Attribut mit dem Wert "350,36,0,0" erhalten,
  Somit hat das **Label** zum linken bzw. oberen Container-Rand den Abstand 350 bzw. 36.
- ein **Width**-Attribut mit dem Wert "90" erhalten,
- ein **Height**-Attribut mit dem Wert "32" erhalten,
- ein **BorderBrush**-Attribut mit dem Wert "**Black**" und ein **BorderThickness**-Attribut mit dem Wert "1" erhalten,
  Durch die Umrandung ist das **Label** auch dann sichtbar, wenn es keinen Euro-Betrag anzeigt.
- ein **HorizontalContentAlignment**-Attribut mit dem Wert "**Right**" erhalten,
  Das sorgt für die bei Zahlen übliche rechtsbündige Anzeige.
- ein **VerticalContentAlignment**-Attribut mit dem Wert "**Center**" erhalten.

Ein **Label**-Objekt hat per Voreinstellung den Innenrand 5,, der sich u. a. auf die Textanzeige auswirkt. Daher haben wir zum **TextBox**-Objekt denselben Innenrand vereinbart.

Im XAML-Code kommt das folgende Element hinzu:

```
<Label Name="ausgabe" HorizontalAlignment="Left" VerticalAlignment="Top"
       Margin="350,36,0,0" Width="90" Height="32"
       BorderBrush="Black" BorderThickness="1"
       HorizontalContentAlignment="Right" VerticalContentAlignment="Center"/>
```

2.4.4.8 Button-Steuerelement zur Anforderung der Konvertierung

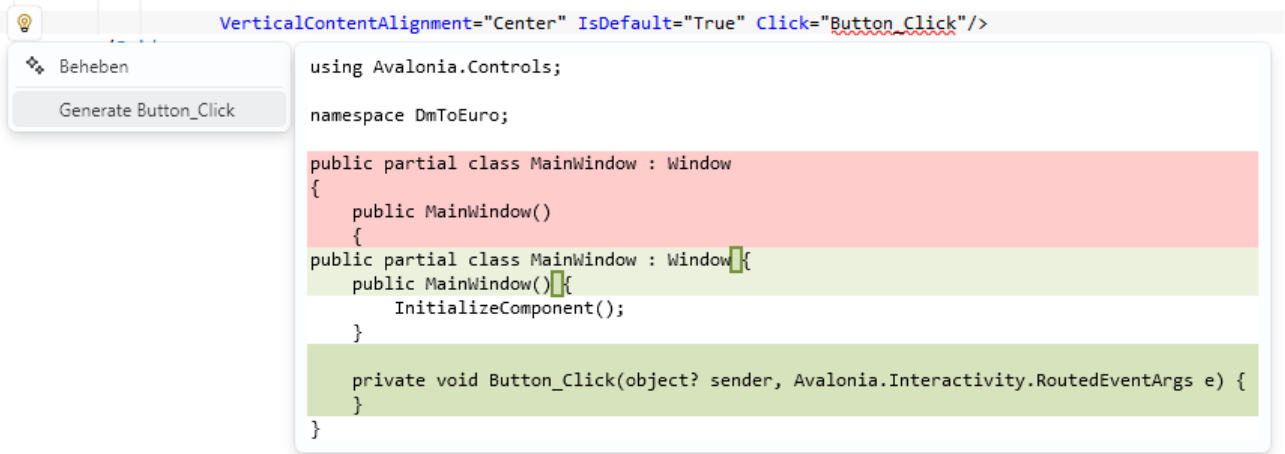
Das **Button**-Steuerelement zur Anforderung der Konvertierung soll ...

- ein **Content**-Attribut mit dem Wert "`Konvertieren`" erhalten.
- ein **HorizontalAlignment**-Attribut mit dem Wert "**Left**" erhalten.
- ein **VerticalAlignment**-Attribut mit dem Wert "**Top**" erhalten.
- ein **Margin**-Attribut mit dem Wert "130,110,0,0" erhalten. Somit hat der **Button** zum linken bzw. oberen Container-Rand den Abstand 130 bzw. 110.
- ein **Width**-Attribut mit dem Wert "90" erhalten.
- ein **Height**-Attribut mit dem Wert "30" erhalten.
- ein **FontSize**-Attribut mit dem Wert "12" erhalten. Der Schalter soll dieselbe Breite haben wie die **TextBox**. Um die Beschriftung *Konvertieren* vollständig anzeigen zu können, wählen wir den Schriftgrad 12.
- ein **VerticalContentAlignment**-Attribut mit dem Wert **Center** erhalten. Das sorgt für die vertikale Zentrierung der **Button**-Beschriftung.
- ein **IsDefault**-Attribut mit dem Wert "**True**" erhalten. Wenn die Eigenschaft **IsDefault** den Wert **True** erhält, dann lässt sich der Schalter im Programm auch per **Enter**-Taste betätigen.
- ein **Click**-Attribut mit dem Wert "`Button_Click`" erhalten. Die in der Datei **MainWindow.axaml.cs** definierte Klasse `MainWindow`, die das Anwendungsfenster repräsentiert, wird gleich eine Instanzmethode namens `Button_Click()` erhalten, die nach der Betätigung des Schalters startet (siehe Abschnitt 2.4.5).

Avalonia kritisiert den Namen der noch nicht definierten Ereignisbehandlungsmethode:

```
<Button Content="Konvertieren" HorizontalAlignment="Left" VerticalAlignment="Top"
        Margin="130,110,0,0" Width="90" Height="30" FontSize="12"
        VerticalContentAlignment="Center" IsDefault="True" Click="Button_Click"/>
```

Befindet sich die Einfügemarke im rot unterschlängelten Bereich, dann erscheint eine Glühbirne am linken Rand:



Nach einem Mausklick auf die Glühbirne erscheint der Vorschlag **Generate Button_Click**. Befindet sich der Mauszeiger über diesem Vorschlag, dann erscheint eine Vorschau auf die Ausführung, die ein Klick auf den Vorschlag auslöst. Dann legt das Avalonia-Plugin in der Datei **MainWindow.axaml.cs** mit einer partiellen Definition der von **Window** (Namensraum **Avalonia.Controls**) abstammenden Fensterklasse `MainWindow` die Behandlungsmethode an:[1]

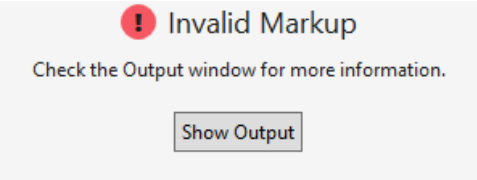
```
using Avalonia.Controls;

namespace DmToEuro;

public partial class MainWindow : Window {
  public MainWindow() {
    InitializeComponent();
  }

  private void Button_Click(object? sender, Avalonia.Interactivity.RoutedEventArgs e) {
  }
}
```

Wenn die XAML-Vorschau trotzdem noch die Fehlermeldung



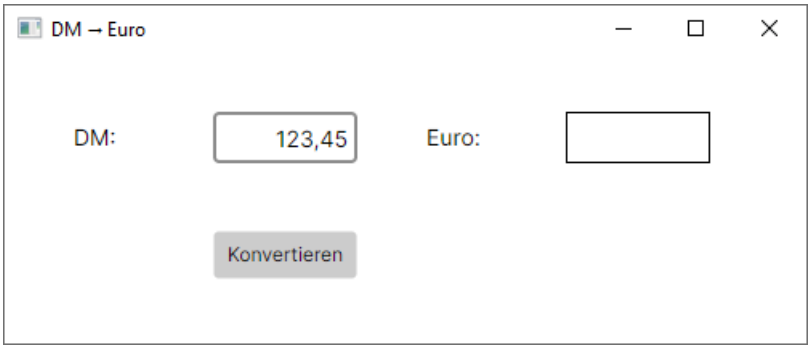
anzeigt, dann hilft die Neuerstellung des Programms über den Menübefehl:

Erstellen > Projektmappe neu erstellen

Die XAML-Vorschau zeigt mittlerweile die geplante Bedienoberfläche:



Nach einem Programmstart (z. B. mit der Tastenkombination **Strg+F5**) ist auch die Titelzeile zu sehen:



[1] Ein Fragezeichen am Ende eines Typnamens (z. B. **object**) signalisiert für die Variable die explizite **null**-Zulässigkeit (siehe z. B. Abschnitt 15.1 in Baltes-Götz & Frischknecht 2026).

Im nächsten Abschnitt vervollständigen wir die Behandlungsmethode zum **Click**-Ereignis des Schalters.

2.4.5 Behandlungsmethode zum Click-Ereignis des Befehlsschalters erstellen

Nun ist die Bedienoberfläche des entstehenden Programms in einem akzeptablen Zustand, und wir können uns um die Funktionalität kümmern. Es ist eine Behandlungsmethode für das **Click**-Ereignis des **Button**-Objekts zu erstellen zu vervollständigen. Sie soll …

- beim **TextBox**-Objekt die aktuell eingetragene Zeichenfolge erfragen,
- diese Zeichenfolge nach Möglichkeit in eine Zahl vom Typ **decimal** wandeln,
- das Ergebnis durch den DM-Euro - Umrechnungsfaktor 1,95583 dividieren,
- den Euro-Betrag in eine Zeichenfolge wandeln
- und das umrahmte **Label**-Objekt auffordern, diese Zeichenfolge anzuzeigen.

Deklarieren Sie im Rumpf der im Abschnitt 2.4.4.8 entstandenen Methode `Button_Click()` eine lokale Variable mit dem Typ **decimal** und dem Namen `faktor`:

```
private void Button_Click(object? sender, Avalonia.Interactivity.RoutedEventArgs e) {
    const decimal faktor = 1.95583m;
```

Wenn Sie die Anweisung mit einem Zuweisungsoperator fortsetzen, hängt die Reaktion der Entwicklungsumgebung von der Zusammensetzung und vom Erfahrungsstand des Assistententeams aus IntelliSense, IntelliCode und Copilot ab (siehe Abschnitt 3.3 in Baltes-Götz & Frischknecht 2026). Im Beispiel hat der KI-Assistent Copilot „geistesgegenwärtig“ den Umrechnungsfaktor von DM in Euro als **decimal**-Literal (mit dem Suffix **m**) vorgeschlagen. Es spricht nichts dagegen, den Vorschlag per Tabulatortaste zu übernehmen.

In der folgenden Komplettierung der Methode `Button_Click()`

```
private void Button_Click(object? sender, Avalonia.Interactivity.RoutedEventArgs e) {
    const decimal faktor = 1.95583m;
    if (!decimal.TryParse(eingabe.Text, out decimal betrag))
        return;
    ausgabe.Content = (betrag / faktor).ToString("0.00");
}
```

dient die Methode **TryParse()** der Struktur **Decimal** dazu, den Inhalt der **TextBox** in den Typ **decimal** zu wandeln und das Ergebnis in der lokalen, inline deklarierten Variablen `betrag` abzulegen. Misslingt die Konvertierung, dann endet die Methode mit **return**. Auf eine Fehlermeldung für den Benutzer verzichten wir hier der Kürze halber, weil die Avalonia-Entsprechung der WPF-Klasse **MessageBox** eine NuGet-Installation erfordert (siehe Abschnitt 4.9).

Wir dividieren den Betrag durch den Umrechnungsfaktor und übergeben das Resultat an das **Label**-Objekt `ausgabe` als neuen Wert der Eigenschaft **Content**. Dabei sorgt die Methode **ToString()** aufgrund der als Parameter erhaltenen Formatierungszeichenfolge für eine Euro-Ausgabe mit zwei Dezimalstellen.

Wenn der Quellcodeeditor die Instanzvariablen `eingabe` und `ausgabe` trotz **Name**-Attribute im XAML-Code durch Unterschlängeln als unbekannt kritisiert,

```
private void Button_Click(object? sender, Avalonia.Interactivity.RoutedEventArgs e) {
    const decimal faktor = 1.95583m;
    if (!decimal.TryParse(eingabe.Text, out decimal betrag))
        return;
    ausgabe.Content = (betrag / faktor).ToString("0.00");
}
```

dann hilft die Neuerstellung des Programms über den Menübefehl:

Erstellen > Projektmappe neu erstellen

Wir lassen das nun fertiggestellte Programm über die Tastenkombination **Strg+F5** erstellen und ausführen:



2.4.6 Fokuserwerb durch die TextBox beim Start

Es wäre nett, wenn nach dem Start des Programms das Textfeld für den DM-Betrag den Tastaturfokus hätte, sodass der Benutzer unmittelbar mit der Eingabe beginnen könnte, ohne zuvor den Tastaturfokus (z. B. per Maus) setzen zu müssen. Eine Lösungsmöglichkeit besteht darin, das zu privilegierende Steuerelement über die Methode **Focus()** der Klasse **UIElement** aufzufordern, den Tastaturfokus zu übernehmen, z. B.:

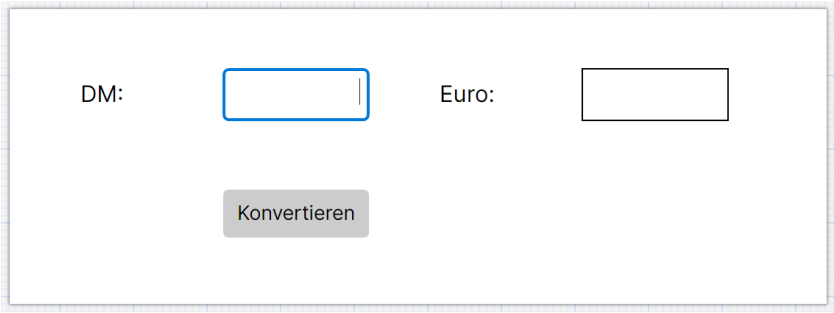
```
eingabe.Focus();
```

Wir veranlassen die Ausführung dieser Anweisung beim Laden des Anwendungsfensters, indem wir ...

- in der Klasse `MainWindow` eine Ereignisbehandlungsmethode zum **Loaded**-Ereignis der Fensterklasse erstellen und die Anweisung dort einfügen:
  ```
  private void Window_Loaded(object sender, RoutedEventArgs e) {
      eingabe.Focus();
  }
  ```
- im XAML-Code zum **Window**-Element die Ereignisbehandlungsmethode registrieren:
  ```
  <Window xmlns="https://github.com/avaloniaui"
          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
          x:Class="DmToEuro.MainWindow"
          Title="DM → Euro" Width="500" Height="180" Loaded="Window_Loaded">
  ```

Im Unterschied zur **Click**-Behandlung beim Schalter (siehe Abschnitt 2.4.4.8) haben wir diesmal die Hilfe der Entwicklungsumgebung per Schnellaktion zu einem absichtlich verursachten Fehler *nicht* benötigt.

In der XAML-Vorschau ist die Fokusvergabe an einer blauen Umrandung zu erkennen:



Durch einen Mausklick auf den Schalter verliert die **TextBox** den Fokus. Um dieses (reguläre) Verhalten der Bedienoberfläche zu kompensieren, kann man am Ende der **Click**-Ereignisbehandlung den Fokus an die **TextBox** übertragen

```
private void Button_Click(object? sender, RoutedEventArgs e) {
    const decimal faktor = 1.95583m;
    if (!decimal.TryParse(eingabe.Text, out decimal betrag))
        return;
    ausgabe.Content = (betrag / faktor).ToString("0.00");
    eingabe.Focus();
}
```

oder im XAML-Code dem Schalter die Fokusbefähigung entziehen:

```
<Button Content="Konvertieren" HorizontalAlignment="Left" VerticalAlignment="Top"
        Margin="130,110,0,0" Width="90" Height="30"
        FontSize="12" VerticalContentAlignment="Center" IsDefault="True"
        Focusable="False" Click="Button_Click" />
```

Wir bevorzugen die zweite Lösung.

Der XAML-Code ist nun komplett,

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="DmToEuro.MainWindow"
        Title="DM → Euro" Width="500" Height="180" Loaded="Window_Loaded">
    <Panel>
        <Label Content="DM:" HorizontalAlignment="Left" VerticalAlignment="Top"
               Margin="40,40,0,0" Width="60" Height="30" />
        <TextBox Name="eingabe" HorizontalAlignment="Left" VerticalAlignment="Top"
                 Margin="130,36,0,0" Width="90" Height="30" Padding="5"
                 TextAlignment="Right" />
        <Label Content="Euro:" HorizontalAlignment="Left" VerticalAlignment="Top"
               Margin="260,40,0,0" Width="60" Height="30" />
        <Label Name="ausgabe" HorizontalAlignment="Left" VerticalAlignment="Top"
               Margin="350,36,0,0" Width="90" Height="32" BorderBrush="Black"
               BorderThickness="1" HorizontalContentAlignment="Right" />
        <Button Content="Konvertieren" HorizontalAlignment="Left"
                VerticalAlignment="Top" Margin="130,110,0,0" Width="90" Height="30"
                FontSize="12" VerticalContentAlignment="Center" IsDefault="True"
                Focusable="False" Click="Button_Click" />
    </Panel>
</Window>
```

und die Anwendung ist praxistauglich.

Es überrascht, dass die fokussierte **TextBox** nicht blau umrandet ist (wie in der Vorschau), sondern grau:



Unter Linux verhält sich das Programm erwartungsgemäß (siehe Kapitel 3). Laut Google-KI übernimmt Avalonia UI das in Windows als Akzentfarbe eingestellte Grau.

2.4.7 Programm erstellen und starten

In der Regel benötigen wir kein Debugging (siehe Abschnitt 3.3.5 in Baltes-Götz & Frischknecht 2026) und starten Programme daher mit der Tastenkombination **Strg+F5**, mit dem Symbolschalter ▷ oder mit dem Menübefehl

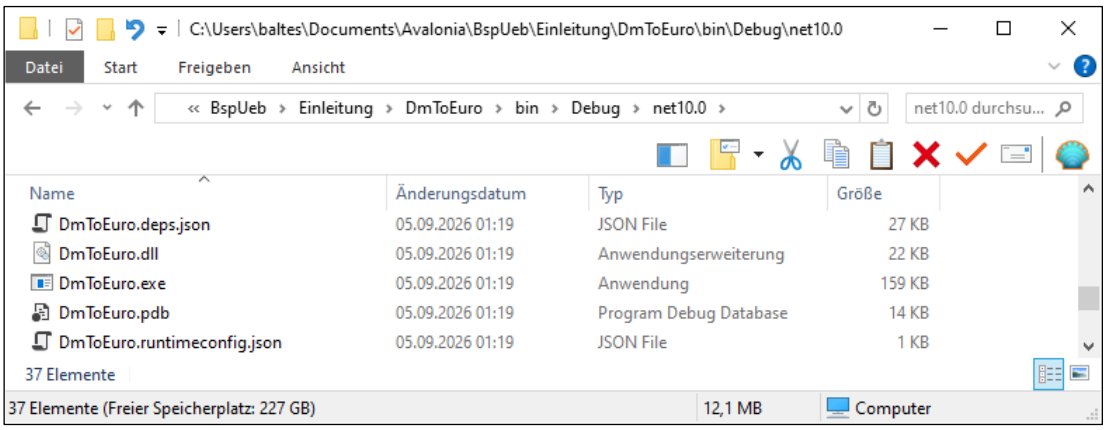
Debuggen > Starten ohne Debugging

Dann verzichtet das Visual Studio während der Programmausführung auf die Anzeige diagnostischer Informationen.

Das Erstellungssystem MSBuild verwendet aber die **Debug**-Konfiguration und den Ausgabe- bzw. Erstellungsordner

...\bin\Debug\net10.0

Im Beispiel enthält der Erstellungsordner u. a. die folgenden Dateien:



Auch im Hinblick auf eine Veröffentlichung von Programmen müssen wir uns mit diesen Dateien genauer beschäftigen, als es z. B. bei den nur temporär (während der Erstellung) relevanten Dateien im **obj**-Projektunterordner der Fall ist. Die folgenden fünf Dateien sind auch im Erstellungsordner einer WPF-Anwendung mit demselben Funktionsumfang zu finden:

- **DmToEuro.dll**
  Das erstellte Assembly mit dem ausführbaren Programm befindet sich in der Datei **DmToEuro.dll**.
- **DmToEuro.exe**
  Diese Datei ermöglicht unter Windows den bequemen Start durch das Abschicken des Dateinamens in einem Konsolenfenster

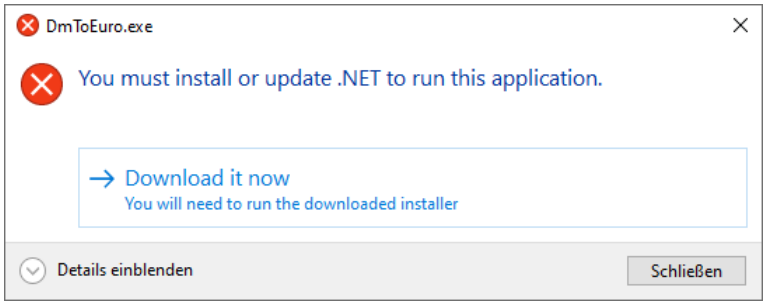
  ```
  >dmtoeuro
  ```

  oder per Doppelklick auf den Dateinamen im Windows-Explorer. Für den Assembly-Start per dotnet-CLI in einem Konsolenfenster

  ```
  >dotnet dmtoeuro.dll
  ```

  benötigt man die **exe**-Datei nicht.

- **DmToEuro.runtimeconfig.json**
  Diese für die Ausführung des Assemblies unverzichtbare Konfigurationsdatei informiert über die erforderliche Version der .NET Core – Laufzeitumgebung mit den Plattform-unabhängigen Basistypen. Wenn auf einem Kundenrechner die benötigte Version der Laufzeitumgebung fehlt, dann erscheint beim Startversuch die folgende Fehlermeldung:



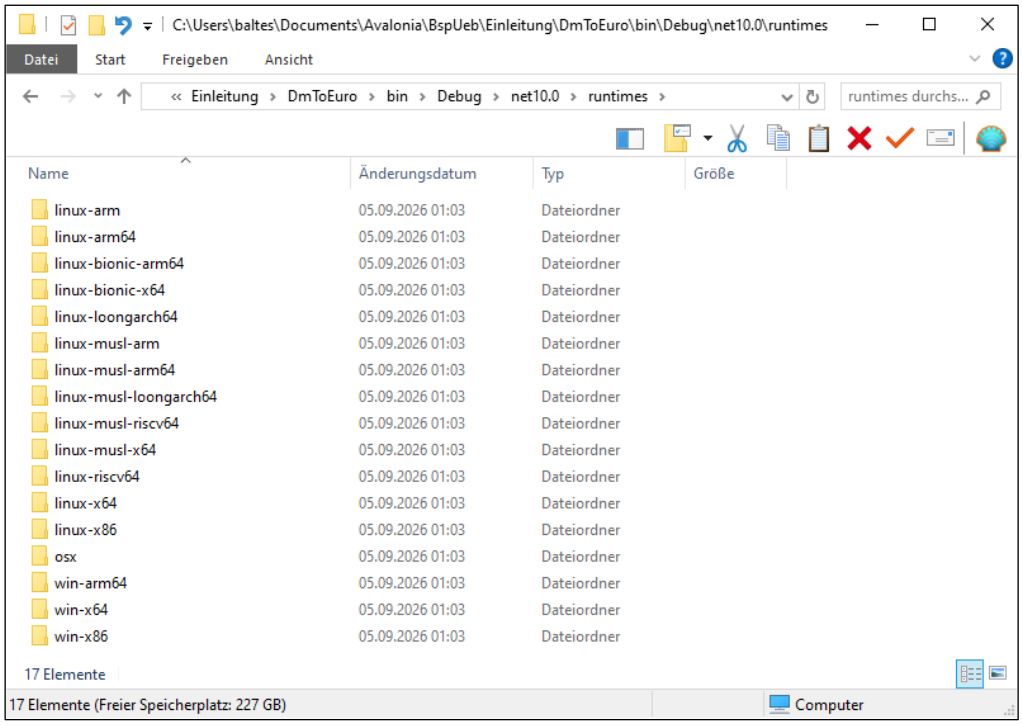
- **DmToEuro.deps.json**
  Diese Datei informiert über Assemblies außerhalb der Standardbibliothek, von denen das Assembly **DmToEuro** abhängig ist. Solche externen Assemblies stammen meist aus NuGet-Paketen. Bei einer Avalonia-Anwendung sind mehrere NuGet-Pakete beteiligt (siehe Abschnitt 2.4.8).
- **DmToEuro.pdb**
  Die **pdb**-Datei (***P**rogram **D**atab**a**se*) enthält Informationen, die das *Debugging* (die Fehlersuche) und das *Profiling* (die Leistungsoptimierung) unterstützen. Sie ist für die Ausführung des Assemblies nicht erforderlich, sorgt aber z. B. bei einem Ausnahmefehler für die Ausgabe einer informativen Aufrufersequenz mit Lokalisation der Fehlerstelle im eigenen Quellcode (Datei und Zeile):

```
Unhandled exception. System.ArgumentNullException: Value cannot be null. (Parameter 'owner')
   at Avalonia.Controls.Window.ShowCore[TResult](Window owner, Boolean modal)
   at Avalonia.Controls.Window.ShowDialog[TResult](Window owner)
   at Avalonia.Controls.Window.ShowDialog(Window owner)
   at MsBox.Avalonia.MsBox`3.ShowWindowDialogAsync(Window owner)
   at MsgBoxDemo.MainWindow.ButtonDialog_Click(Object sender, RoutedEventArgs e) in
     C:\Users\ ... \MainWindow.axaml.cs:line 27
```

  Ohne **pdb**-Datei fehlt dem Unfallbericht die Lokalisation des Fehlers im eigenen Quellcode:

```
Unhandled exception. System.ArgumentNullException: Value cannot be null. (Parameter 'owner')
   at Avalonia.Controls.Window.ShowCore[TResult](Window owner, Boolean modal)
   at Avalonia.Controls.Window.ShowDialog[TResult](Window owner)
   at Avalonia.Controls.Window.ShowDialog(Window owner)
   at MsBox.Avalonia.MsBox`3.ShowWindowDialogAsync(Window owner)
   at MsgBoxDemo.MainWindow.ButtonDialog_Click(Object sender, RoutedEventArgs e)
```

Während sich im Erstellungsordner der analogen WPF-Anwendung nur die fünf beschriebenen Dateien befinden (Gesamtumfang ca. 180 KB), enthält der Avalonia-Erstellungsordner insgesamt 563 MB. Grund ist die Unterstützung von mehreren Plattformen, und der Unterordner **runtimes** nimmt mit 551 MB fast den gesamten Raum ein:



Für den Einsatz unter Windows (64 Bit) benötigt man nur die Laufzeitumgebung im Ordner **win-x64**, und der Platzbedarf der Anwendung schrumpft auf 130 MB. Die WPF-Anwendung belegt hauptsächlich deswegen weniger Platz, weil sie die Desktop-Laufzeitumgebung des lokalen Rechners im Ordner

C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop.App\10.0.13

mit ca. 95 MB verwendet, während die Avalonia-Anwendung ihre eigene Desktop-Laufzeitumgebung nutzt. Die Plattform-unabhängige .NET Core - Laufzeitumgebung des lokalen Rechners im Ordner

C:\Program Files\dotnet\shared\Microsoft.NETCore.App\10.0.13

wird von beiden Anwendungen verwendet.

2.4.8 Projektdatei

Auf den Erstellungsprozess (das Verhalten von MSBuild und Compiler) wirken sich die Einstellungen in der Projektdatei mit der Namenserweiterung **csproj** aus, die sich im Projektordner befindet. Hier ist die Datei **DmToEuro.csproj** aus dem Währungskonverterbeispiel zu sehen:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <RootNamespace>DmToEuro</RootNamespace>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ApplicationManifest>app.manifest</ApplicationManifest>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Avalonia" Version="12.1.1" />
    <PackageReference Include="Avalonia.Desktop" Version="12.1.1" />
    <PackageReference Include="Avalonia.Themes.Fluent" Version="12.1.1" />
    <PackageReference Include="Avalonia.Fonts.Inter" Version="12.1.1" />
    <PackageReference Include="AvaloniaUI.DiagnosticsSupport" Version="2.2.3">
      <IncludeAssets Condition="'$(Configuration)' != 'Debug'">None</IncludeAssets>
      <PrivateAssets Condition="'$(Configuration)' != 'Debug'">All</PrivateAssets>
    </PackageReference>
  </ItemGroup>
</Project>
```

Im Vergleich zu einem WPF-Projekt mit ähnlichem Funktionsumfang sind mehrere NuGet-Pakete vorhanden, auf denen Avalonia basiert.

Bei der Arbeit mit dem Visual Studio kann man die Projektdatei per Doppelklick auf den Projektnamen im **Projektmappen-Explorer** öffnen und editieren.

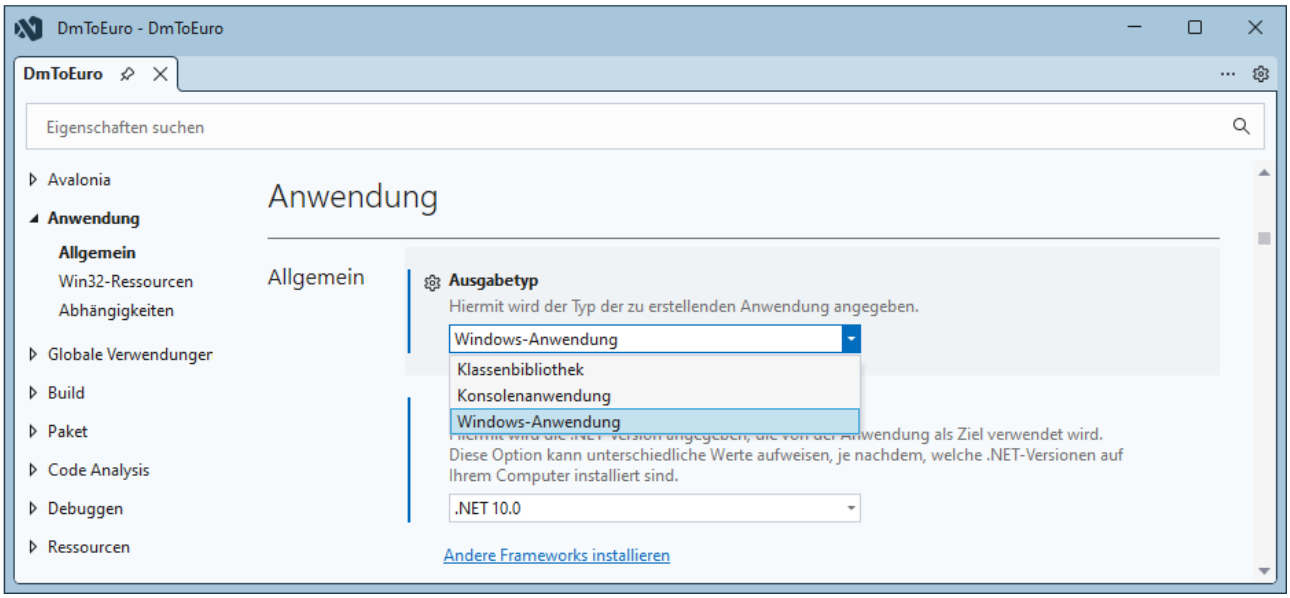
Die wichtigsten Optionen lassen sich etwas bequemer über den Projekteigenschaftendialog der Entwicklungsumgebung ändern, den man im Beispiel mit dem Menübefehl

Projekt > DmToEuro Eigenschaften

erreicht. In diesem Dialog vorgenommene Änderungen wirken sich sofort auf die Projektdatei aus und beeinflussen somit das MSBuild-Erstellungssystem sowie den Compiler.

2.4.8.1 Ausgabetyp

Im Projekteigenschaftendialog kann man den Ausgabetyp einstellen, ohne die korrekten Namen der Alternativen (**Exe**, **WinExe**, **Library**) kennen zu müssen:



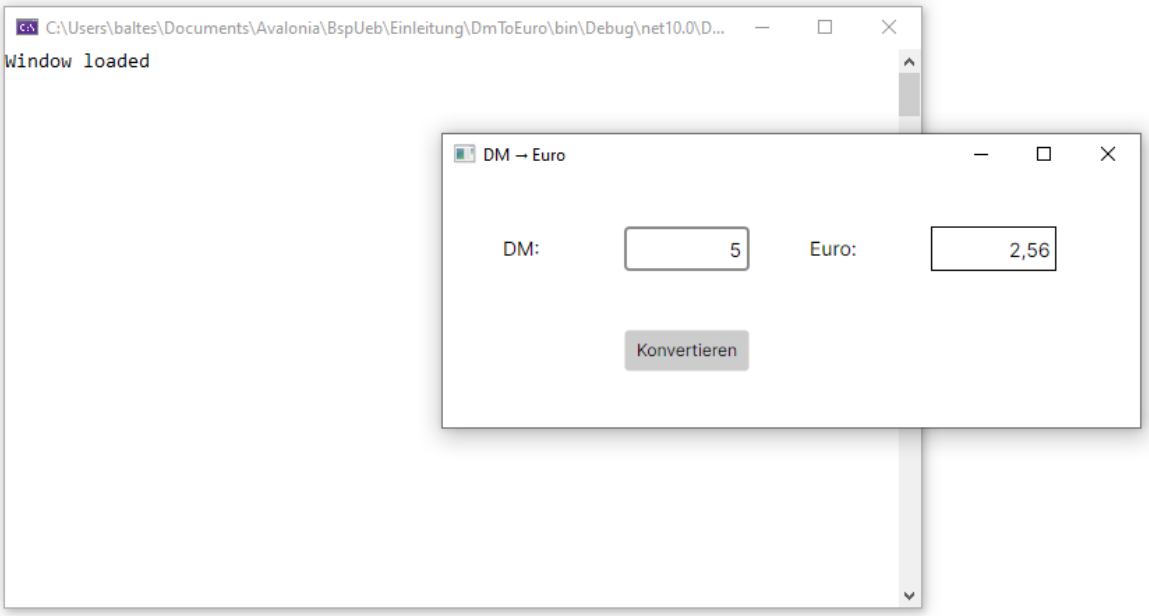
In der Projektdatei **DmToEuro.csproj** zum gerade als Beispiel dienenden Programm `DmToEuro` resultiert aus der Wahl der Alternative **Windows-Anwendung** das folgende **PropertyGroup**-Element:

```
<OutputType>WinExe</OutputType>
```

Wenn man (z. B. durch die Wahl der Alternative **Konsolenanwendung** im Projekteigenschaftendialog) den Ausgabetyp **Exe** einträgt,

```
<OutputType>Exe</OutputType>
```

dann erscheint beim Starten des Programms neben dem Anwendungsfenster mit den GUI-Bedienelementen zusätzlich ein Konsolenfenster:

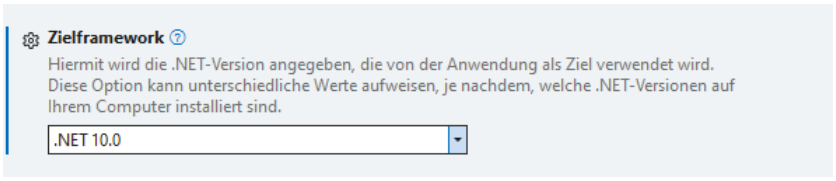


Bei einem GUI-Programm ist ein Konsolenfenster nicht grundsätzlich überflüssig, weil hier z. B. bei einem Laufzeitfehler eine informative Beschreibung inklusive der betroffenen Aufrufersequenz erscheint. Außerdem kann ein Programm neben seinen GUI-Aktivitäten auch Konsolenausgaben vornehmen (z. B. über **WriteLine()** - Aufrufe). Ohne Konsolenfenster bleiben Laufzeitfehlermeldungen mit ihren diagnostischen Informationen (z. B. zur Fehlerstelle in der Quellcodedatei) sowie **WriteLine()** – Ausgaben im Verborgenen.[1]

2.4.8.2 Zielframework

Im **Anwendungs**-Bereich des Projekteigenschaftendialogs geht es in der Drop-Down - Liste **Zielframework** um die .NET – Version, die …

- zum Erstellen verwendet
- und damit auch minimal auf einem ausführenden Rechner vorausgesetzt wird, falls man die Laufzeitumgebung nicht zusammen mit dem Programm ausliefert (siehe Kapitel 3 zum Veröffentlichen einer Anwendung).



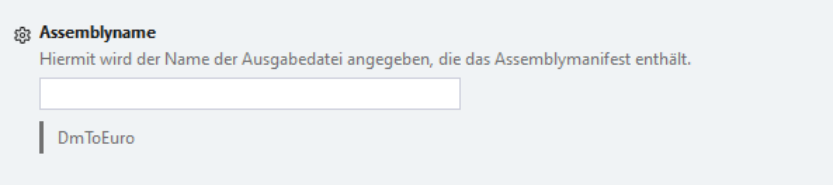
In der Projektdatei wirken sich Angaben zur .NET – Version auf das **PropertyGroup**-Element **TargetFramework** aus, z. B.:

```
<TargetFramework>net10.0</TargetFramework>
```

[1] Ein Avalonia-Programm hat aber durchaus die Möglichkeit, Laufzeitfehler in einem GUI-Fenster anzuzeigen (siehe Abschnitt 4.9).

2.4.8.3 Assemblyname

Das Textfeld **Assemblyname** im **Anwendung**s-Bereich des Projekteigenschaftendialogs zeigt, dass per Voreinstellung das erstellte Assembly seinen Namen vom Projekt übernimmt:



Ein alternativer Name führt in der Projektdatei zum **PropertyGroup**-Element **AssemblyName**, z. B.:

```
<AssemblyName>Konverter</AssemblyName>
```

2.4.8.4 Weitere Einstellungen

Der Projekteigenschaftendialog enthält neben den eben beschriebenen Erstellungsoptionen noch weitere Einstellungen, z. B.:

- **Standardnamespace**
  Namensräume verhindern Namenskollisionen, weil z. B. identisch benannte Klassen aus zwei verschiedenen Namensräumen unterschiedliche *vollqualifizierte* Namen besitzen. Von der Entwicklungsumgebung angelegte Typdefinitionen verwenden als Namensraum den Eintrag im Feld **Standardnamespace**, der initial mit dem Projektnamen identisch ist:

  Die dunkelblaue blaue Markierung am linken Rand zeigt an, dass ein Projektdateieintrag zum Namensraum vorhanden ist:

  ```
  <RootNamespace>DmToEuro</RootNamespace>
  ```

  Auf bereits vorhandene Quellcodedateien wirkt sich eine Änderung des Standardnamensraums *nicht* aus.

- **NULL-Werte zulassen**
  Seit C# 8 bzw. .NET 5 kann man durch das Projektdatei-Element **Nullable** bekunden, dass Referenztypen den Wert **null** per Voreinstellung *nicht* annehmen sollen, und dass **null**-zulässige Referenztypen (engl. *nullable reference types*) durch ein **?**-Suffix am Namensende zu kennzeichnen sind:

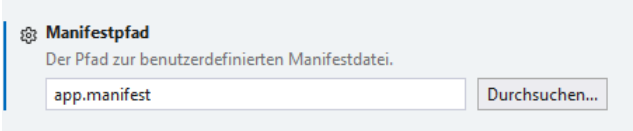
  ```
  <Nullable>enable</Nullable>
  ```

  Dieses Projektdateielement kann man über die folgende Einstellung im **Anwendung**s-Bereich des Projekteigenschaftendialogs veranlassen:

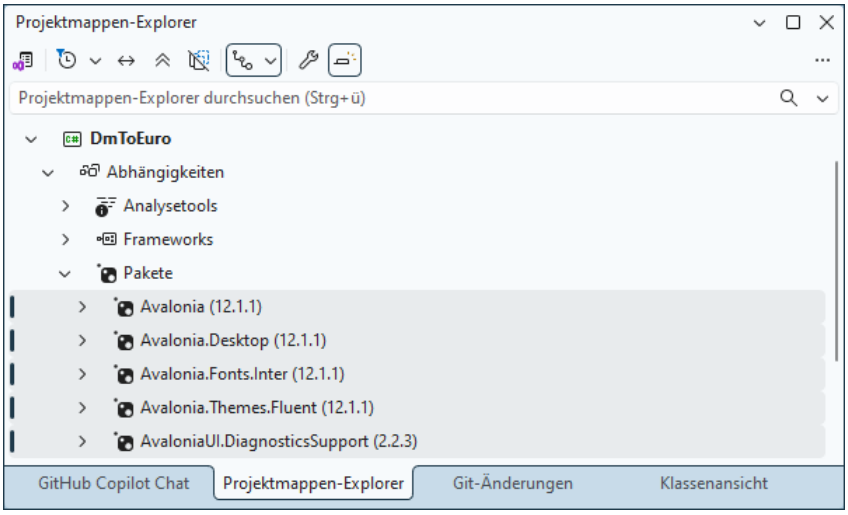
  

  Damit ist der sogenannte *Nullable-Kontext* projektweit eingestellt.

- **Manifest**
  Die Windows-spezifische Manifestdatei legt u. a. die unterstützten Windows-Versionen fest. Das Projektdateielement
  ```
  <ApplicationManifest>app.manifest</ApplicationManifest>
  ```
  lässt sich über die folgende Einstellung im **Anwendung**s-Bereich des Projekteigenschaftendialogs anfordern:



Die am Projekt beteiligten NuGet-Pakete



sind in der Projektdatei registriert, sodass die Entwicklungsumgebung beim Öffnen des Projekts die Existenz der Pakete prüfen kann:

```
<ItemGroup>
  <PackageReference Include="Avalonia" Version="12.1.1" />
  <PackageReference Include="Avalonia.Desktop" Version="12.1.1" />
  <PackageReference Include="Avalonia.Themes.Fluent" Version="12.1.1" />
  <PackageReference Include="Avalonia.Fonts.Inter" Version="12.1.1" />
  <PackageReference Include="AvaloniaUI.DiagnosticsSupport" Version="2.2.3">
    <IncludeAssets Condition="'$(Configuration)' != 'Debug'">None</IncludeAssets>
    <PrivateAssets Condition="'$(Configuration)' != 'Debug'">All</PrivateAssets>
  </PackageReference>
</ItemGroup>
```

Die in einem Projekt verwendete Avalonia-Version ist am einfachsten über die Projektdatei festzustellen.

2.4.9 Avalonia aktualisieren

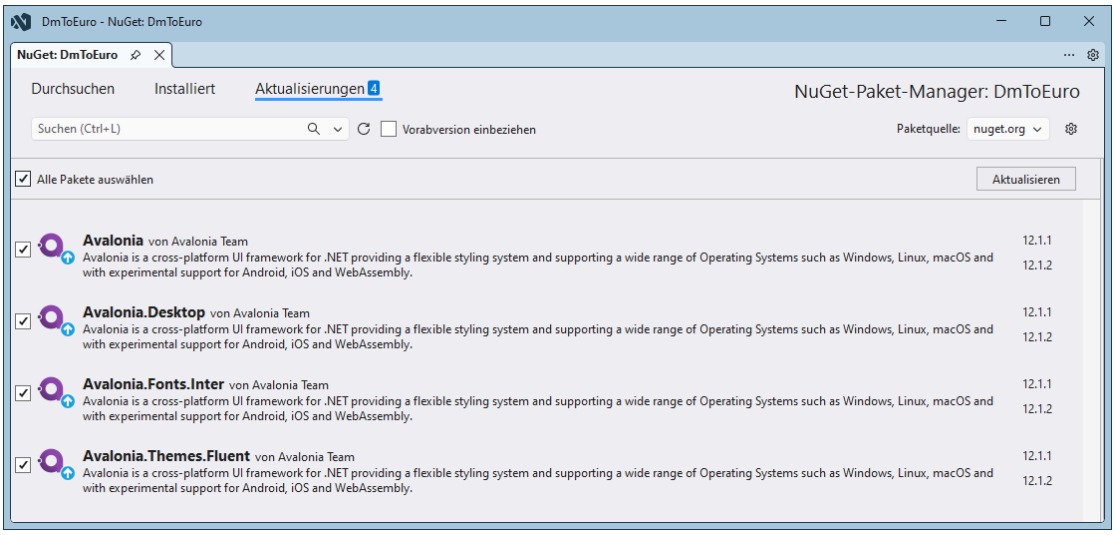
Das Visual Studio aktualisiert die NuGet-Pakete mit Avalonia *nicht* automatisch, weil inkompatible Änderungen (engl.: *breaking changes*) das Projekt gefährden könnten. Das manuelle Aktualisieren ist mit dem NuGet-Paketmanager besonders bequem zu realisieren. Man startet ihn über das Item

NuGet-Pakete verwalten

aus dem Kontextmenü zum Projektknoten

Abhängigkeiten > Pakete

im **Projektmappen-Explorer**. Im Fenster des Paketmanagers wählt man die Registerkarte **Aktualisierungen**:



Anschließend sind in der Projektdatei die neuen Versionen zu sehen:

```
<ItemGroup>
  <PackageReference Include="Avalonia" Version="12.1.2" />
  <PackageReference Include="Avalonia.Desktop" Version="12.1.2" />
  <PackageReference Include="Avalonia.Themes.Fluent" Version="12.1.2" />
  <PackageReference Include="Avalonia.Fonts.Inter" Version="12.1.2" />
  <PackageReference Include="AvaloniaUI.DiagnosticsSupport" Version="2.2.3">
    <IncludeAssets Condition="'$(Configuration)' != 'Debug'">None</IncludeAssets>
    <PrivateAssets Condition="'$(Configuration)' != 'Debug'">All</PrivateAssets>
  </PackageReference>
</ItemGroup>
```

2.5 Dokumentation und Quellcode

2.5.1 Beschreibungen und Anleitungen

Auf den Webseiten zu Avalonia UI ist eine umfangreiche Dokumentation zu finden, z. B.:

- Einstieg
  https://docs.avaloniaui.net/docs/welcome
- Tutorials
  https://docs.avaloniaui.net/docs/samples-tutorials

Weitere Informationsquellen im Internet, z. B.:

- Biswanger, G. & König, R. (2025). *Moderne Cross-Plattform Apps mit .NET und Avalonia UI*
  https://www.youtube.com/watch?v=YNms_s0h6vI
- König, R. (2025). *Avalonia: Das bessere WPF für moderne Cross-Platform-Apps*
  https://basta.net/blog/avalonia-cross-platform-dotnet/
- König, R. (2026). *Avalonia 12 wird zur echten Cross-Platform-Alternative*
  https://basta.net/blog/avalonia-12-navigation-webview-responsive-design/
- Labinator.com (2025). *C# Guide*, Kapitel 10 über Avalonia UI
  https://csharp.labinator.com/docs/avalonia-ui
- Menzel, M. (2025). *Anwendungen mit Avalonia UI erstellen*
  https://t2informatik.de/blog/anwendungen-mit-avalonia-ui-erstellen/

Bücher über Avalonia UI:

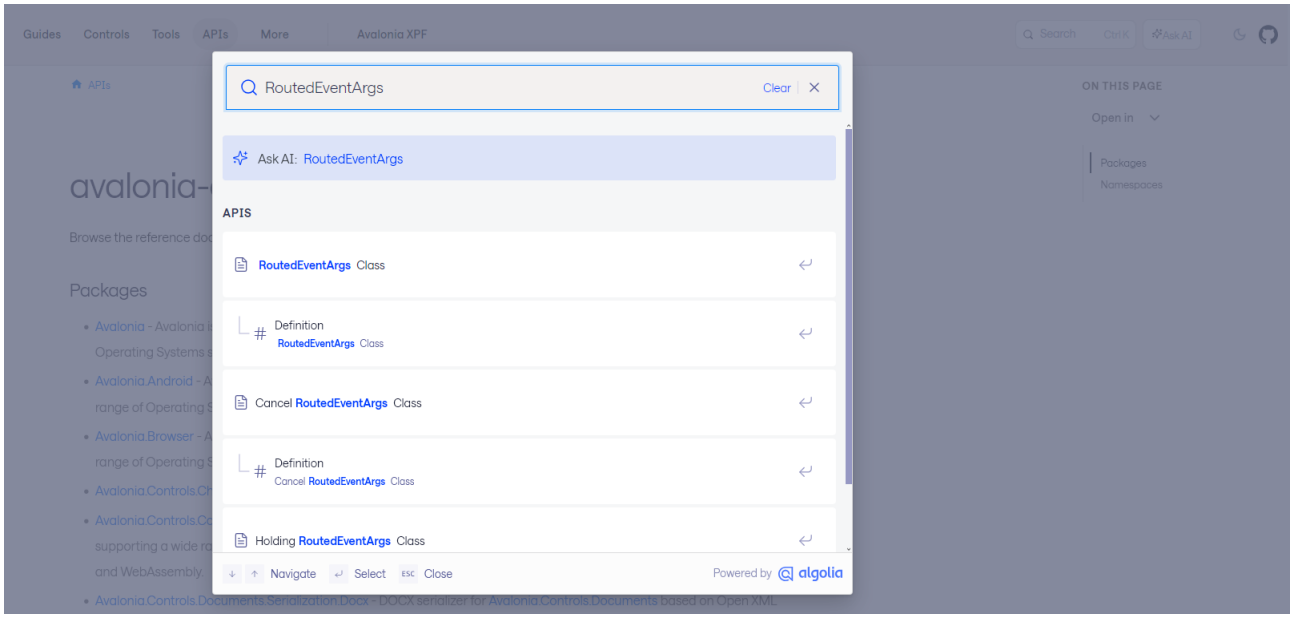
- Bandiera, R. (2025). *Avalonia UI. Cross-platform desktop applications in C#*
- Del Sole, A. (2025). *Avalonia UI Succinctly*
- Johnson, R. (2025). *Avalonia Development Essentials*

2.5.2 API-Dokumentation

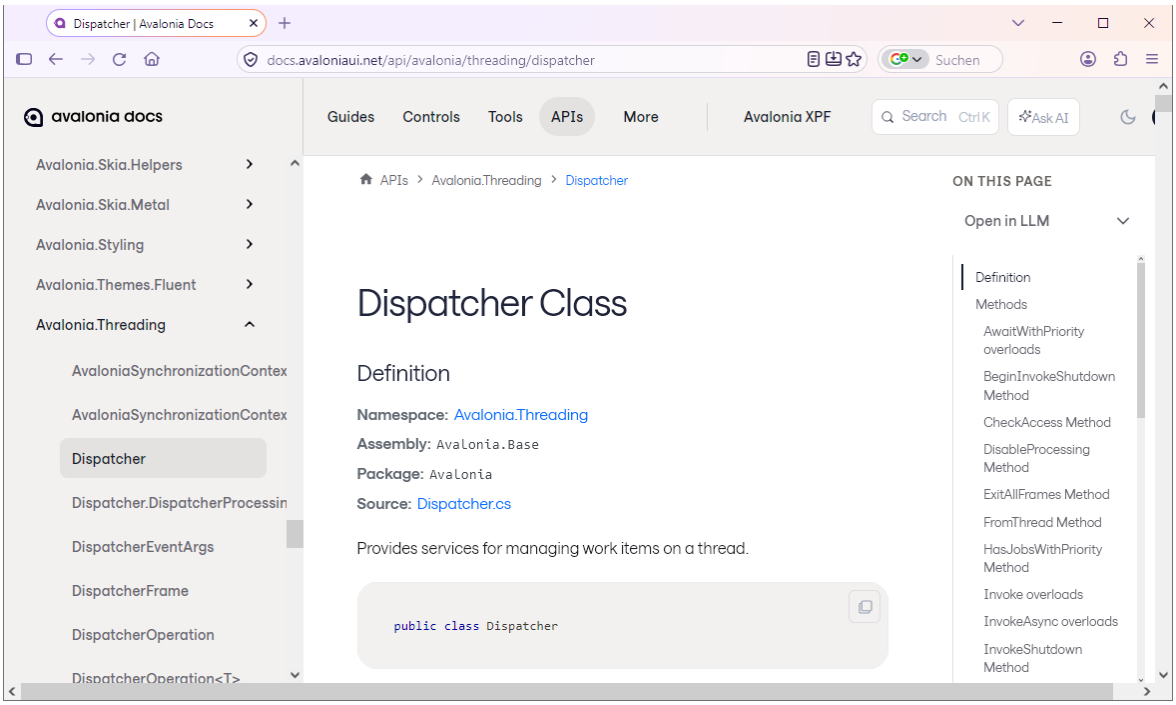
Die API-Dokumentation ist hier zu finden

https://docs.avaloniaui.net/api/

und verfügt über eine Suchfunktion, z. B.:



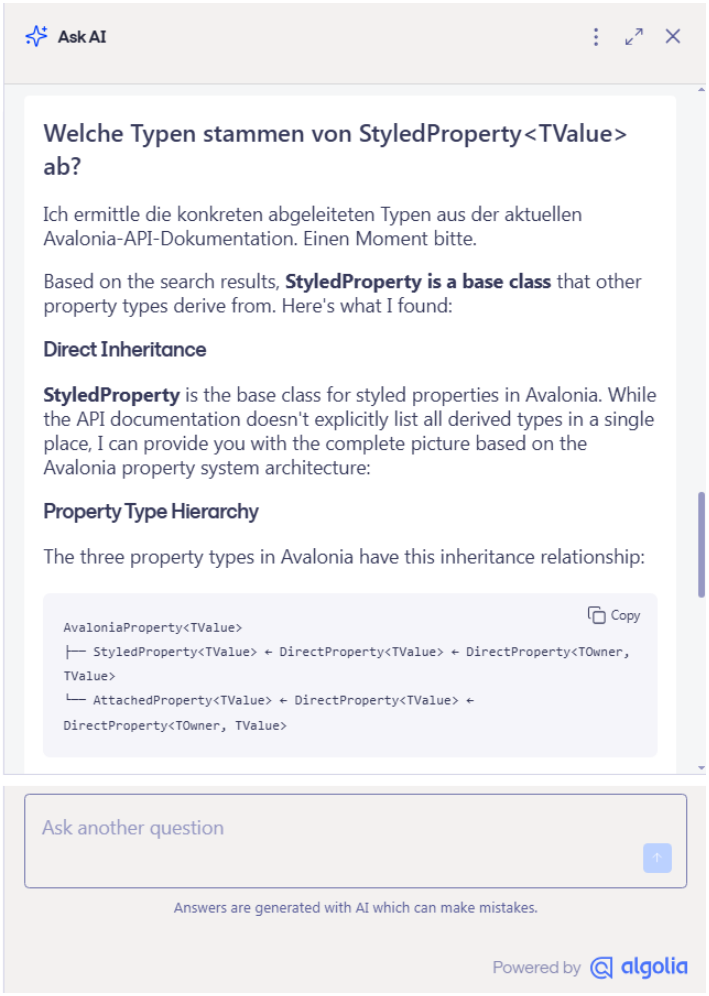
Wir geben im Manuskript zu Typen in der Regel den Namensraum an, sodass man auch bei erfolgloser Suche die benötigte API-Information über die Navigationsleiste am linken Fensterrand zügig ansteuern kann, z. B.:



Auf der API-Webseite steht auch ein KI-Assistent bereit, um die Suche in der Dokumentation zu vereinfachen. Er hat die deutsche Frage

> Welche Typen stammen von StyledProperty<TValue> ab?

verstanden und in Englisch beantwortet. Die gelieferte Klassenhierarchie ist allerdings unklar bzw. unpräzise:



Oft sind z. B. die von ChatGPT im Rahmen einer kostenlosen Nutzung gelieferten Informationen nützlicher. Zu der obigen Frage erhält man eine korrekte und gut lesbare Darstellung der Klassenhierarchie:



Leider ist es *nicht* möglich, im Visual Studio die Dokumentation zu Avalonia-Typen über das Positionieren der Einfügemarke und die Funktionstaste **F1** anzufordern.

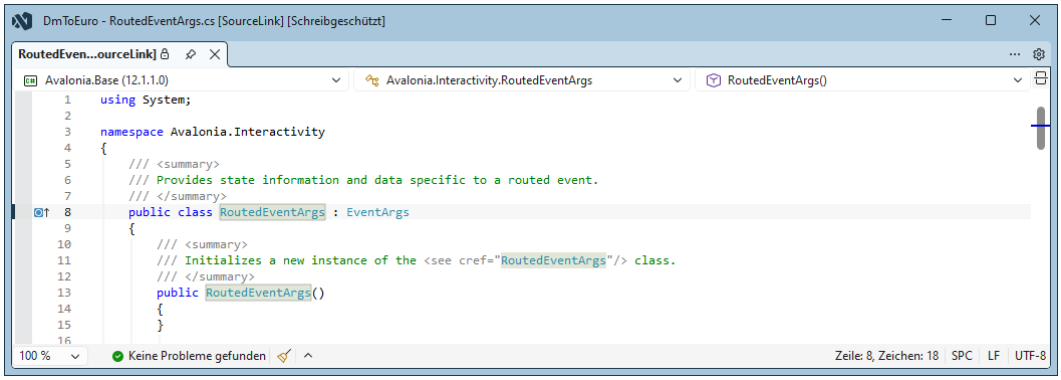
2.5.3 API-Quellcode

Um den Avalonia – API-Quellcode einzusehen, bewegt man den Mauszeigen über den interessierenden Bestandteil, sodass eine QuickInfo erscheint, z. B.:

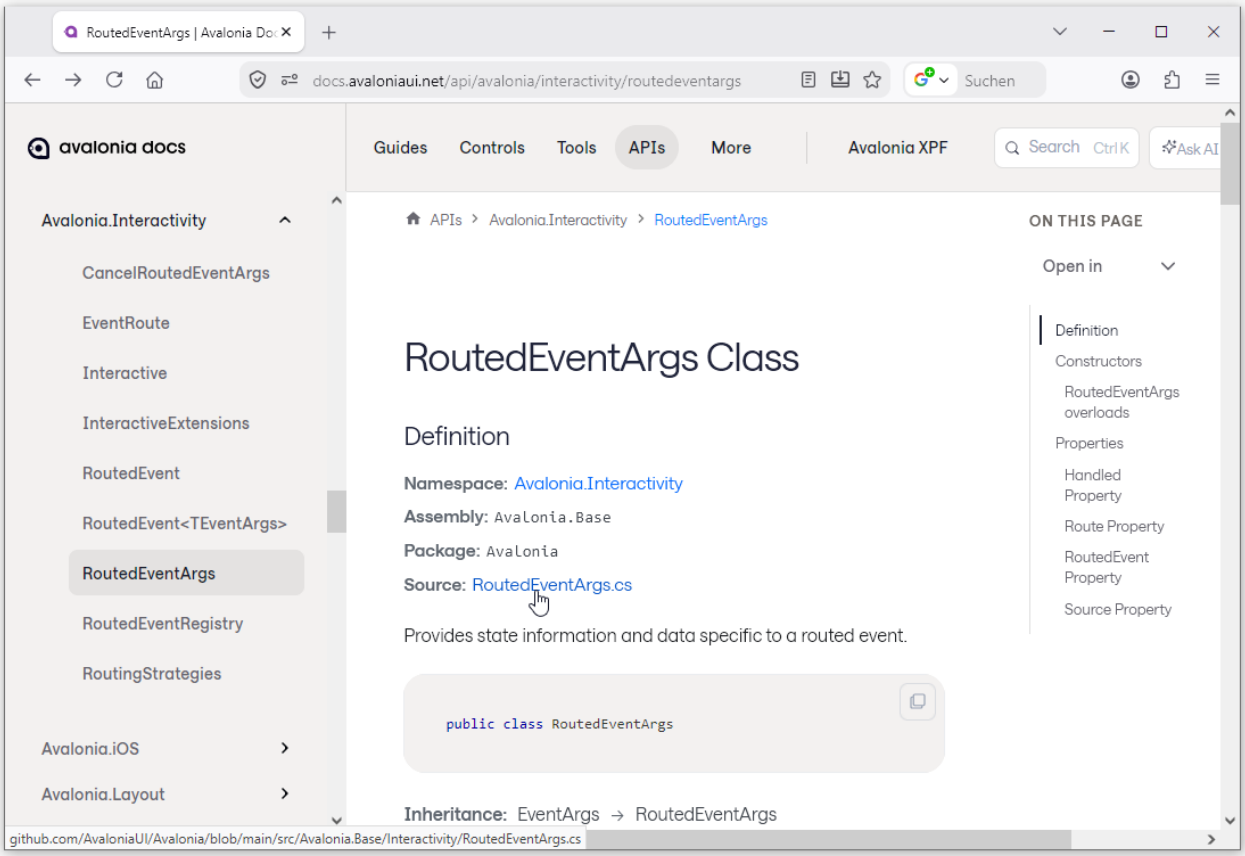
```
private void Button_Click(object? sender, RoutedEventArgs e) {
    const decimal faktor = 1.95583m;
    if (!decimal.TryParse(eingabe.Text, out deci
        return;
    ausgabe.Content = (betrag / faktor).ToString("0.00");
}
```

class Avalonia.Interactivity.RoutedEventArgs
Provides state information and d… routed event.
RoutedEventArgs

Nach einem Mausklick auf den interessierenden Bestandteil erscheint der Quellcode in einem Editor-Registerblatt, z. B.:



Die API-Dokumentation zu Avalonia-Typen enthält in der Regel einen Link zum Quellcode auf GitHub, z. B.:



2.6 Blicke hinter die Kulissen mit ILSpy

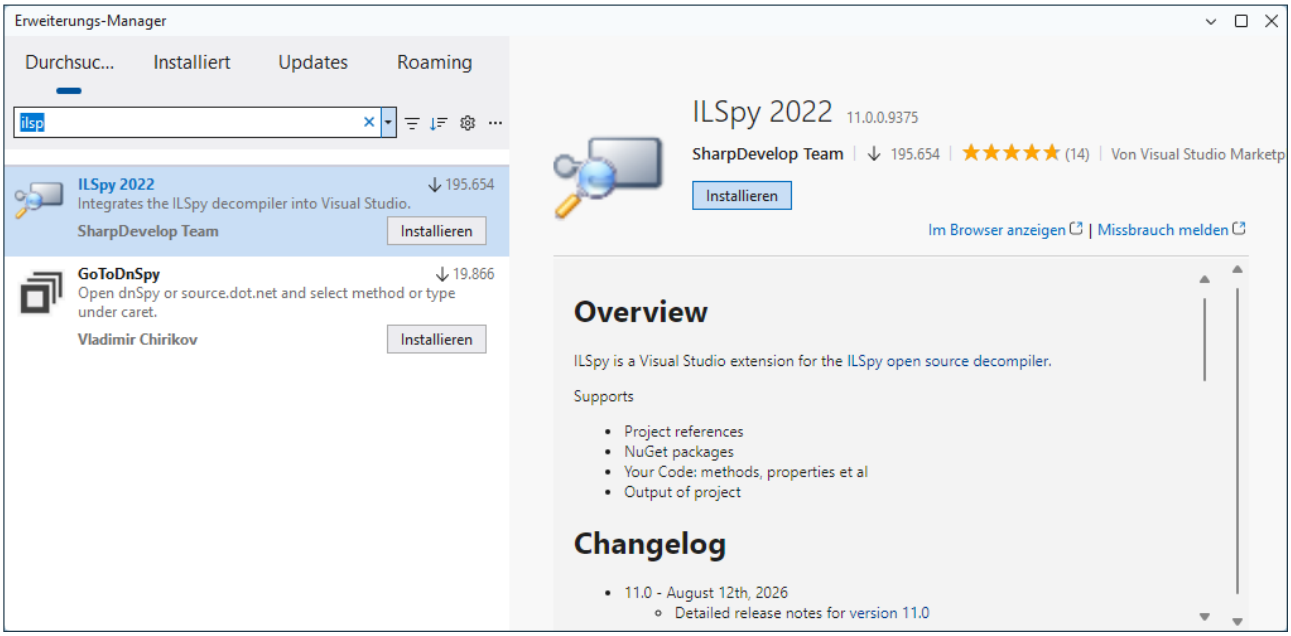
In diesem Abschnitt beschreiben wir den Bezug und die Installation des Programms ILSpy, das etliche Kompetenzen zur gründlichen Analyse von .NET - Assemblies besitzt, z. B.:

- Anzeige des IL-Codes der enthaltenen Klassen
- Dekompilierung (Rückübersetzung) des IL-Codes in C# - Code
- Anzeige der Metadaten

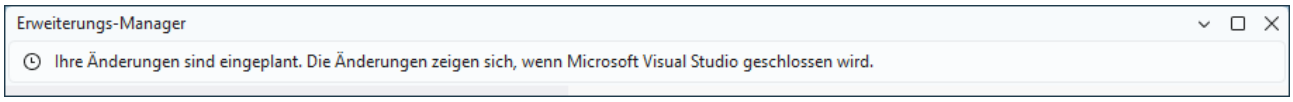
ILSpy ist eine vom SharpDevelop-Team[1] entwickelte Open Source – Software, die man als Visual Studio - Erweiterung (Plugin) oder als eigenständiges Programm nutzen kann.[2] Wir verwenden die zuerst genannte Option und starten im Visual Studio die Installation der Erweiterung über den Menübefehl

Erweiterungen > Erweiterungen verwalten

Wir suchen nach der Erweiterung **ILSpy 2022** und **installieren** die Version 11.0.0.9375:



Man muss das Visual Studio beenden,

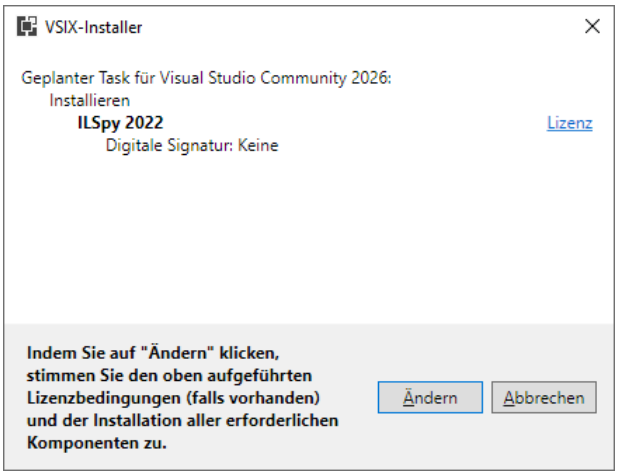


um die Installation der Erweiterung durch den **VSIX Installer** zu ermöglichen:

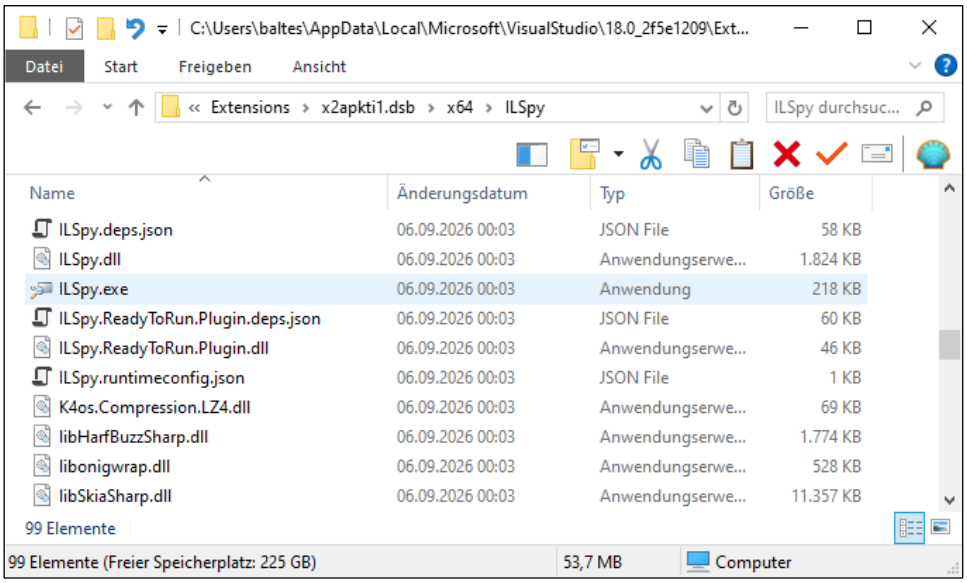
---

[1] Wer schon länger in der .NET – Szene unterwegs ist, kann sich vielleicht an die Entwicklungsumgebung **SharpDevelop** erinnern (https://sourceforge.net/projects/sharpdevelop/files/).

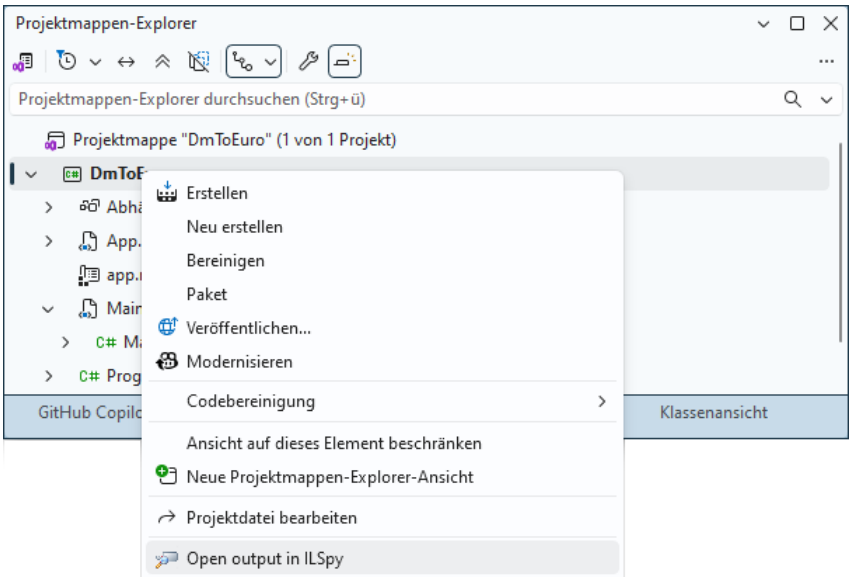
[2] https://sourceforge.net/projects/ilspy.mirror/



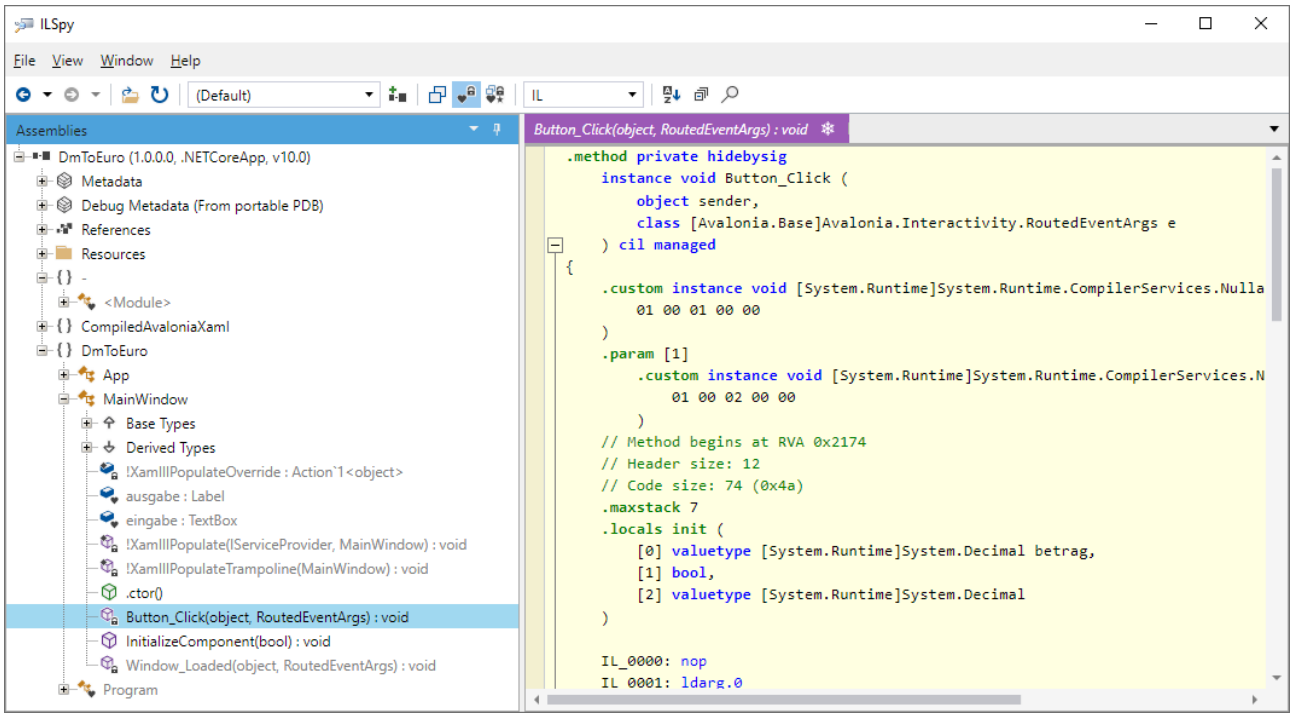
Die Installation landet im Windows-Benutzerprofil, z. B.:



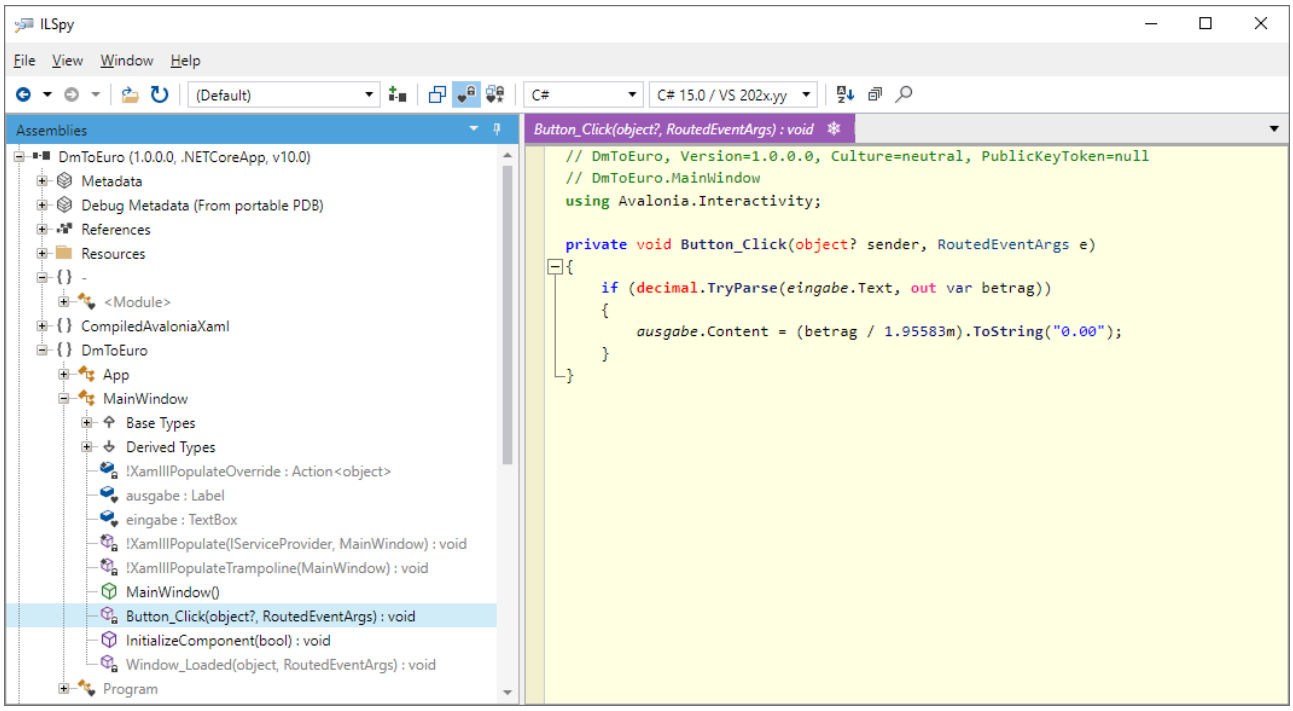
Nach dem nächsten Start der Entwicklungsumgebung kann man über das Kontextmenü zu einem Projekt



das zugehörige Assembly in ILSpy öffnen. Hier ist der IL-Code zur Methode `Button_Click()` der Klasse `MainWindow` aus dem Währungskonverter-Projekt zu sehen (vgl. Abschnitt 2.4):



Ist von einem Assembly nur der IL-Code vorhanden (die ausführbare **dll**-Datei), dann kann ILSpy den Quellcode ziemlich gut rekonstruieren, z. B.:



Wer seinen C# - Quellcode geheim halten (und sein geistiges Eigentum schützen) möchte, der muss einen sogenannten *Obfuskator* verwenden.[1] Solche Programme verändern den IL-Code, um die Rückübersetzung zu erschweren.

[1] https://de.wikipedia.org/wiki/Obfuskation_(Software)#Java-Bytecode_und_MSIL
https://learn.microsoft.com/en-us/visualstudio/ide/dotfuscator/

2.7 Übungsaufgaben zum Kapitel 2

1) Installieren Sie nach Möglichkeit auf Ihrem PC das Visual Studio Community 2026 (siehe Abschnitt 2.2) und die Avalonia-Erweiterung (siehe Abschnitt 2.3).

2) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Avalonia UI ist eine frei verwendbare Open Source - Software, doch darf man die Avalonia-Erweiterung für das Visual Studio *nur für nicht-kommerzielle* Projekte einsetzen.
2. In neuen Avalonia-Projekten ist per Voreinstellung der sogenannte *Nullable-Kontext* eingestellt, so dass **null**-zulässige Referenztypen durch ein **?**-Suffix am Namensende zu kennzeichnen sind.
3. Im Quellcodeeditor kann man die Dokumentation zu einem Avalonia - API-Bestandteil anfordern, indem man die Einfügemarke auf den Bestandteil setzt und die Funktionstaste **F1** drückt.
4. ILSpy kann den IL-Code eines Programms in C# zurückübersetzen.

3 Veröffentlichung einer Anwendung für Linux

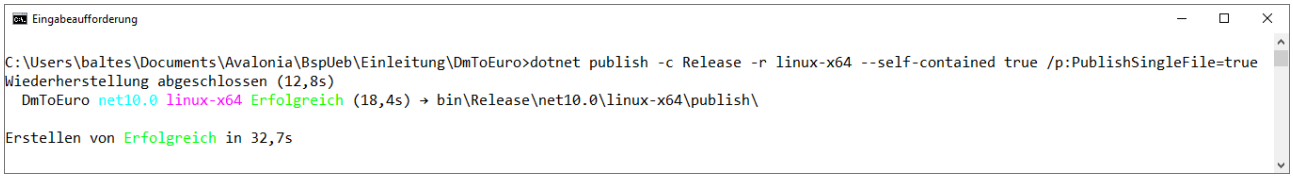
Während Microsofts Multiplattform – GUI-Lösung MAUI keine Unterstützung für das Betriebssystem Linux bietet, lässt sich eine Avalonia UI – Anwendung mit geringem Aufwand auch unter Linux nutzen. Wir veröffentlichen den im Abschnitt 2.4 erstellten Währungskonverter zur Nutzung unter Linux und demonstrieren die Verwendung unter Ubuntu 24.[1]

Bei der Veröffentlichung einer .NET – Anwendung sind etliche Optionen bzw. Einstellungen relevant (siehe Kapitel 23 in Baltes-Götz & Frischknecht 2026).

In einem auf den Projektordner positionieren Konsolenfenster führen wir das Kommando

```
>dotnet publish -c Release -r linux-x64 --self-contained true /p:PublishSingleFile=true
```

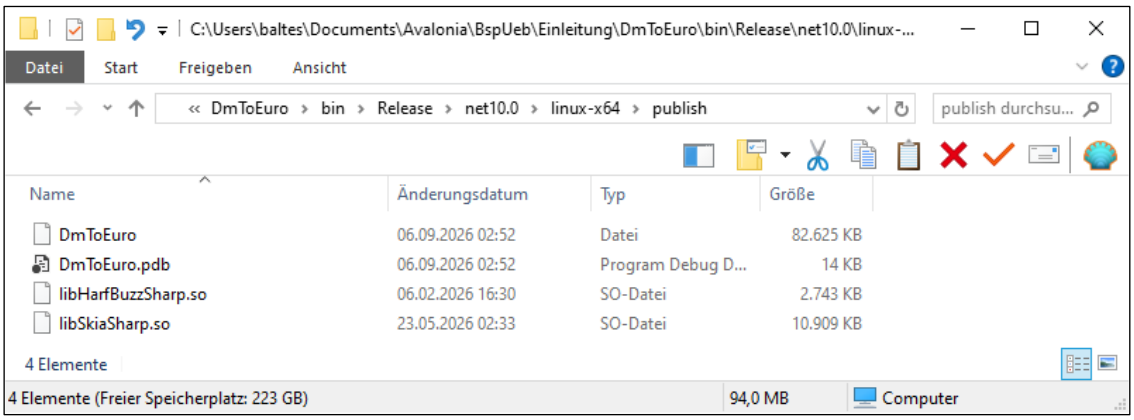
aus, um eine unter Linux-x64 lauffähige Einzeldateianwendung zu erstellen.



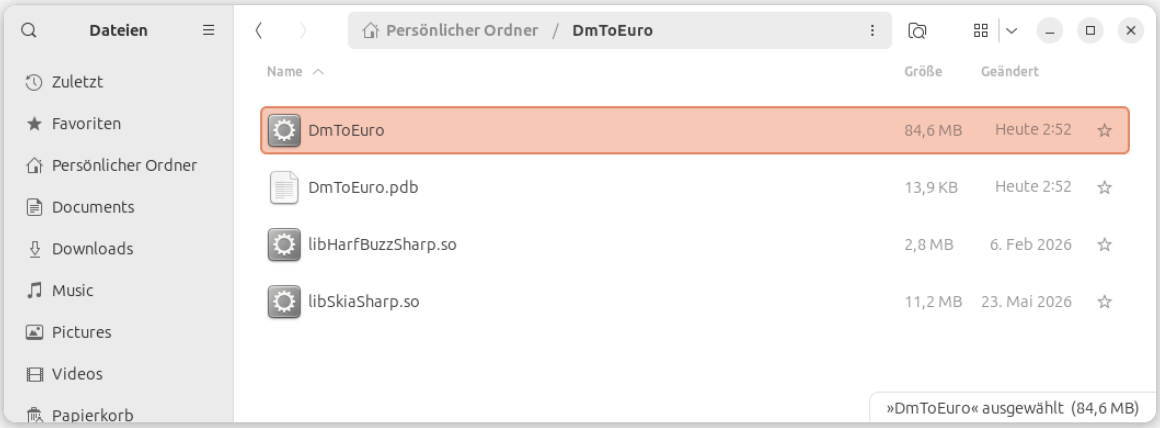
Der Ordner

...\bin\Release\net10.0\linux-x64\publish

enthält das Programm zusammen mit der Laufzeitumgebung in der Datei **DmToEuro**:



Wir kopieren den kompletten Ordner auf den Linux-Rechner:



---

[1] Wertvolle Hinweise zum Vorgehen hat die folgende Webseite geliefert:
https://forum.combit.net/t/running-the-avalonia-samples-on-linux-or-macos/13104

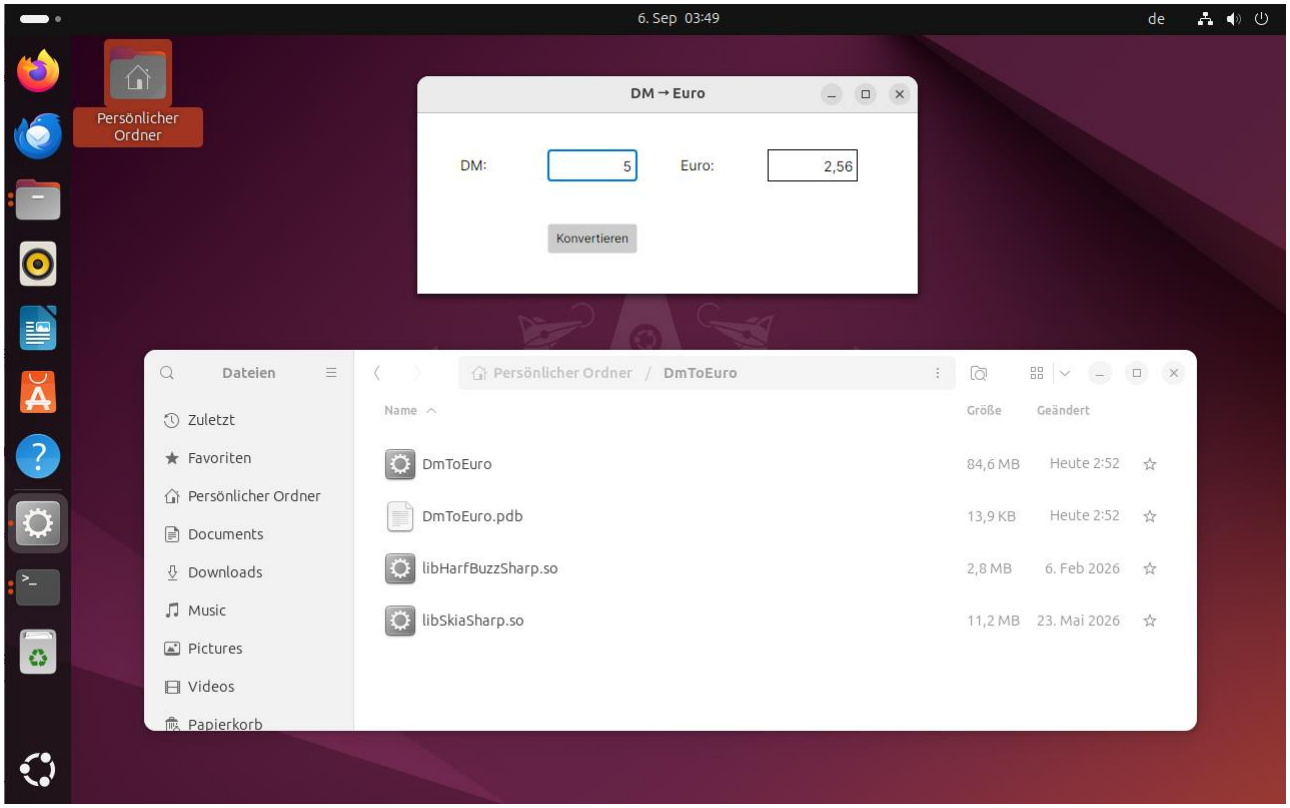
In einem Linux-Terminal deklarieren wir das Programm **DmToEuro** mit dem folgenden **chmod**-Kommando als ausführbar:

```
>chmod +x DmToEuro
```

Zum Starten des Programms setzt man im Ubuntu-Dateimanager einen Doppelklick auf die ausführbare Datei oder startet das Programm im Terminal:

```
>./DmToEuro
```

Wie von Avalonia versprochen, resultiert unter Ubuntu-Linux dasselbe Layout wie unter Windows:



Für ein professionelles Programm ist die Erstellung eines Pakets für eine grafische Linux-Bedienoberfläche (z. B. Gnome oder KDE) wegen der besseren Systemintegration (z. B. Desktop-Verknüpfung, (De)installation per Paketmanager) zu empfehlen. Für Debian und Ubuntu sowie verwandte Distributionen eignen sich **deb**-Pakete.

Man kann die Pakete manuell erstellen, muss dazu aber einigen Aufwand betreiben.[1] Das von AvaloniaUI OÜ angebotene Programm **Parcel** vereinfacht das Erstellen von Installationspaketen (siehe Anhang C). Es ist für Einzelpersonen im Rahmen der Community-Lizenz kostenlos verfügbar,[2] erstellt mit dieser Lizenz aber nur Installationspakete für das Betriebssystem, auf dem Parcel läuft. Da wir das nur für Windows verfügbare Visual Studio als Entwicklungsumgebung verwenden und die Community-Lizenz von Avalonia UI nutzen, können wir im Anhang C nur die Kreation eines Installationsprogramms für Windows vorführen. Mit der für 15 € pro Monat erhältlichen Plus-Lizenz von Avalonia UI lassen sich mit Parcel auf einem Entwicklungsrechner Installationspakete für Linux, macOS und Windows erstellen.

---

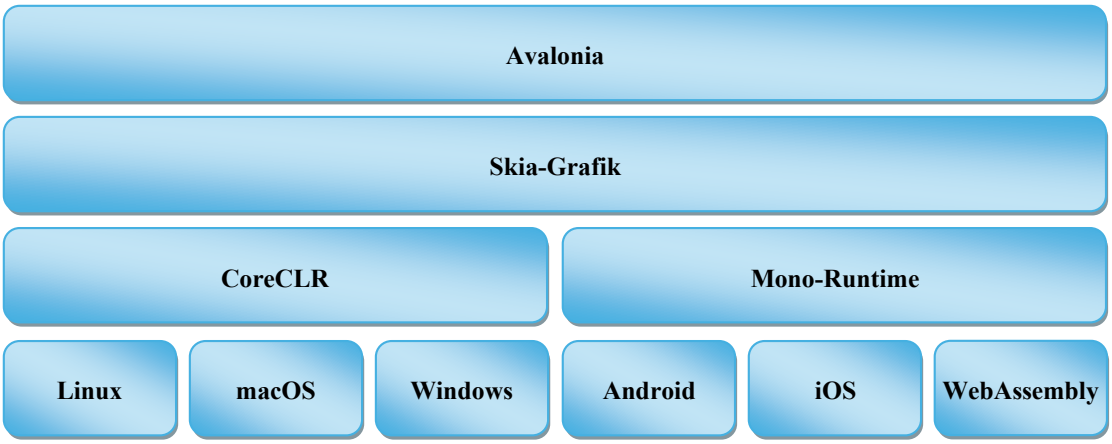
1 https://docs.avaloniaui.net/docs/deployment/linux

2 https://avaloniaui.net/pricing?audience=individual

4 Technische Grundlagen

4.1 Software-Stapel einer Avalonia-Anwendung

In der folgenden Abbildung sind für die unterstützten Plattformen die Software-Stapel einer Avalonia-Anwendung zu sehen:



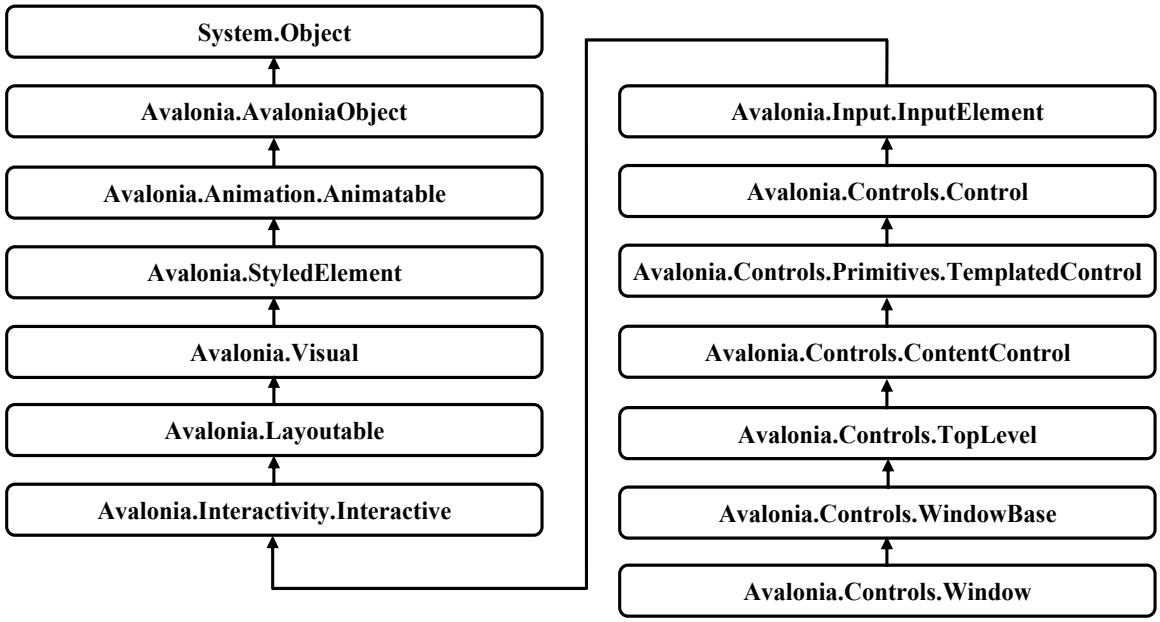
Für die auf allen Plattformen zum Zeichnen der Bedienoberfläche genutzte Skia-Grafik von Google, die auch im Browser *Chrome* und in der Multiplattform-Lösung *Flutter* zum Einsatz kommt, ist zur Vermeidung von Rucklern die Ablösung durch die *Impeller*-Grafik geplant.

Über die beiden, in aktuellen .NET – Anwendungen tätigen Laufzeitumgebungen **CoreCLR** und **Mono-Runtime** informiert z. B. der Abschnitt 2.2 in Baltes-Götz & Frischknecht (2026).

Wir beschränken uns in diesem Manuskript auf die drei Desktop-Plattformen Linux, macOS und Windows.

4.2 Stammbaum der Avalonia-Steuerelemente

Im Beispiel des Abschnitts 2.4 haben wir die Klasse **Window** (Namensraum **Avalonia.Controls)** als Basisklasse des Anwendungsfensters kennengelernt. In der folgenden Abbildung ist der Stammbaum der Klasse zu sehen:



Viele Klassen aus dieser Hierarchie werden uns im Verlauf des Manuskripts als Basisklassen bzw. Funktionalitätsspender für GUI-Komponenten begegnen. Die Basisklasse einer Komponente vermittelt einige Information über ihre Kompetenzen, z. B. ...

- stammt die zur Anzeige von Bildern dienende Klasse **Image** von **Control** ab,
- während **Label** die Basisklasse **ContentControl** hat, also nicht nur Text anzeigen kann, sondern ein beliebiges Objekt.

Obwohl die folgenden Anmerkungen oberflächlich bleiben, lassen sich doch einige für Avalonia zentrale Begriffe verorten (z. B. Abhängigkeitseigenschaft, Routingereignis). Manche Klassen (z. B. **Animatable**) tauchen im Manuskript nicht mehr auf, andere werden später gründlich behandelt:[1]

- **AvaloniaObject** (Namensraum **Avalonia**)[2]
  Diese Klasse entspricht der WPF-Klasse **DependencyObject**, erfüllt aber zusätzlich die Schnittstelle **INotifyPropertyChanged** (vgl. Abschnitt 9.6.1). Ihre Objekte können *Abhängigkeitseigenschaften* (*AvaloniaProperties*) anbieten (siehe Kapitel 7.6). Diese Eigenschaften spielen in Avalonia eine zentrale Rolle und haben viele Vorteile im Vergleich zu den gewöhnlichen CLR-Eigenschaften (*Common Language Runtime*). Auf die Ausprägung einer Abhängigkeitseigenschaft können sich verschiedene Einflussquellen auswirken (z. B. Datenbindung, ein übergeordnetes Steuerelement, ein Stil). Die meisten Eigenschaften von Avalonia-Steuerelementen sind als Abhängigkeitseigenschaften realisiert.
- **Animatable** (Namensraum **Avalonia.Animation**)[3]
  Das ist die Basisklasse für animationsfähige Objekte.
- **StyledElement** (Namensraum **Avalonia**)[4]
  Diese Klasse unterscheidet sich von ihrer Basisklasse **Animatable** u. a. durch ...
  - die für Datenbindungen wichtige Eigenschaft **DataContext**,
  - die Implementation der Schnittstelle **ILogical**, sodass Objekte zu Knoten im logischen Baum der Steuerelemente werden lönnen,
  - die Implementation der Schnittstelle **IStyleHost**, sodass die Anwendung von Stilen möglich ist.
- **Visual** (Namensraum **Avalonia**)[5]
  Objekte dieser Klasse besitzen elementare Grafikkompetenzen (z. B. Ausgabe auf dem Bildschirm).
- **Layoutable** (Namensraum **Avalonia.Layout**)[6]
  Diese Klasse steuert Layout-bezogene Kompetenzen bei.
- **Interactive** (Namensraum **Avalonia.Interactivity**)[7]
  Das ist die Basisklasse für Objekte, die Routingereignisse auslösen können (siehe Kapitel 7).
- **InputElement** (Namensraum **Avalonia.Input**)[8]
  Das ist die Basisklasse für Objekte, die Eingabeereignisse (generiert per Tastatur, Maus, Touch-Screen etc.) verarbeiten können.

---

1 Über weitere Details informiert die folgende Webseite:
https://docs.avaloniaui.net/docs/fundamentals/architecture

2 https://docs.avaloniaui.net/api/avalonia/avaloniaobject

3 https://docs.avaloniaui.net/api/avalonia/animation/animatable

4 https://docs.avaloniaui.net/api/avalonia/styledelement

5 https://docs.avaloniaui.net/api/avalonia/visual

6 https://docs.avaloniaui.net/api/avalonia/layout/layoutable

7 https://docs.avaloniaui.net/api/avalonia/interactivity/interactive

8 https://docs.avaloniaui.net/api/avalonia/input/inputelement

- **Control** (Namensraum **Avalonia.Controls**)[1]
  Das ist die Basisklasse für Avalonia-Steuerelemente.
- **TemplatedControl** (Namensraum **Avalonia.Controls.Primitives**)[2]
  Dies ist die Basisklasse für Steuerelemente, deren Optik von einem **IControlTemplate**-Objekt abhängt.
- **ContentControl** (Namensraum **Avalonia.Controls**)[3]
  Von **ContentControl** abgeleitete Steuerelementklassen enthalten die Eigenschaft **Content** vom Typ **Object** und können ein beliebiges Objekt aufnehmen. Ein Objekt der Klasse **Window** enthält z. B. als **Content** meist einen Layoutcontainer (z. B. aus der Klasse **Grid**). Zur Darstellung eines komplexen Datenobjekts weist man der Eigenschaft **ContentTemplate** ein Objekt vom Typ **IDataTemplate** zu. Es enthält Steuerelemente, die an Eigenschaften des Datenobjekts gebunden sind und diese darstellen.
- **TopLevel** (Namensraum **Avalonia.Controls**)[4]
  Ein TopLevel-Objekt erscheint auf der Bedienoberfläche, ist aber kein positionierbares Fenster.
- **WindowBase** (Namensraum **Avalonia.Controls**)[5]
  Dies ist ein positionierbares Top-Level - Fenster.

4.3 Single-Thread – Architektur und Dispatcher

Wie die meisten modernen GUI-Frameworks ist auch in Avalonia UI aus Konsistenzgründen das Single-Thread - Prinzip realisiert, wobei alle mit GUI-Elementen interagierenden Methoden im selben Thread (also sequentiell) ausgeführt werden.[6] Auf ein GUI-Element darf nur derjenige Thread zugreifen, der es erzeugt hat. Ein Verstoß gegen diese Regel führt zu einer **InvalidOperationException**.

In der Regel ist in einer Avalonia-Anwendung nur *ein* UI-Thread tätig. Das gilt auch dann, wenn in der Bedienoberfläche mehrere Fenster geöffnet sind.

Zum UI-Thread gehört ein Objekt der Klasse **Dispatcher** (dt.: *Verteiler*) im Namensraum **Avalonia.Threading**, das die Zugriffsexklusivität sicherstellt und weitere Aufgaben erfüllt.[7] Man kann ein GUI-Programm als eine Ansammlung von Methoden zur Behandlung von Ereignissen auffassen. Durch die vom Betriebssystem registrierten Benutzeraktivitäten (z. B. Mausklicks, Tastendrücke) und sonstige Ursachen entstehen Ereignisse, die zu einem permanenten Strom von Nachrichten führen. Der Dispatcher verarbeitet die eintreffenden Nachrichten in einer **while**-Schleife. Er ...

- leitet Nachrichten an registrierte Behandlungsmethoden weiter,
- empfängt Delegatenobjekte zur Ausführung im UI-Thread und verwaltet sie in prioritätsabhängigen Warteschlangen.

Der Dispatcher des UI-Threads ist über die statische Eigenschaft **UIThread** der Klasse **Dispatcher** (Namensraum **Avalonia.Threading**) ansprechbar:

---

1 https://docs.avaloniaui.net/api/avalonia/controls/control
2 https://docs.avaloniaui.net/api/avalonia/controls/primitives/templatedcontrol
3 https://docs.avaloniaui.net/api/avalonia/controls/contentcontrol
4 https://docs.avaloniaui.net/api/avalonia/controls/toplevel
5 https://docs.avaloniaui.net/api/avalonia/controls/windowbase
6 https://docs.avaloniaui.net/docs/app-development/threading
7 https://docs.avaloniaui.net/api/avalonia/threading/dispatcher

public static Dispatcher UIThread { get; }

Sofern nur ein Dispatcher vorhanden ist, liefert die **Dispatcher**-Instanzeigenschaft der Klasse **AvaloniaObject** dasselbe Ergebnis:

public Dispatcher Dispatcher { get; set; }

Über einen an dieses Objekt gerichteten **CheckAccess()** - Aufruf

public bool CheckAccess()

bringt eine Methode in Erfahrung, ob sie im UI-Thread läuft und auf die Bedienoberfläche zugreifen darf.

Die im UI-Thread ausgeführten Methoden haben das Recht zum Zugriff auf die Bedienoberfläche. Allerdings muss sich bei diesen Methoden der Zeitaufwand in Grenzen halten (maximal 100 Millisekunden), damit die Bedienoberfläche flüssig reagiert. Solange eine Methode läuft (z. B. gestartet als Reaktion auf ein Ereignis), kann der UI-Thread nicht auf andere Ereignisse reagieren. Auch ist keine Aktualisierung der Anzeige möglich, zu der z. B. das Laufzeitsystem auffordert, weil ein bisher verdeckter Fensterbereich sichtbar geworden ist. Zeitaufwändige Arbeiten (z. B. Netzwerk-, Datei oder Datenbankzugriffe, aufwändige Berechnungen) gehören daher in einen Arbeits-Thread. In der Regel müssen irgendwann Ergebnisse der Hintergrundtätigkeit an der Oberfläche erscheinen, wobei wegen der eingangs genannten Regel aus dem Arbeits-Thread keine direkten Zugriffe auf Steuerelemente möglich sind. Mit Hilfe der **Dispatcher**-Methode **Post()**

public void Post(Action *action*, **Avalonia.Threading.DispatcherPriority** *priority* = **default)**

kann man eine Methode in die Warteschlange des UI-Threads stellen, ohne auf die Ausführung zu warten, z. B.:

```
Avalonia.Threading.Dispatcher.UIThread.Post(() => ausgabe.Content = "0,00");
```

Der Parameter vom Typ **DispatcherPriority** ist optional und hat die Voreinstellung **default**.

Weil eine Avalonia-Anwendung in der Regel nur einen UI-Thread und damit nur einen **Dispatcher** besitzt, lässt sich dessen **Post()** – Methode auch über die **Dispatcher**-Eigenschaft eines Steuerelements ansprechen, z. B.:

```
ausgabe.Dispatcher.Post(() => ausgabe.Content = "0,00");
```

Duch die Verwendung von **InvokeAsync()** statt **Post()** erhält man als Rückgabe ein Objekt der Klasse **DispatcherOperation** im Namensraum **Avalonia.Threading**,

public Avalonia.Threading.DispatcherOperation InvokeAsync(Action *action***)**

dessen Eigenschaft **Status** über den Stand der Bearbeitung informiert:

public Avalonia.Threading.DispatcherOperationStatus Status { get; set; }

Wenn eine aufwändige Aufgabe permanent im UI-Thread auszuführen ist (z. B. die Rechtschreibekontrolle in einem Textprogramm), dann empfiehlt sich die folgende Technik:[1]

- Die ausführende Methode erledigt eine kurze Arbeitsepisode und beendet sich dann.
- Ist die Arbeit noch nicht erledigt, dann setzt die Methode mit einem **InvokeAsync()** – Aufruf an den **Dispatcher** des UI-Threads ein Delegatenobjekt mit einem erneuten Aufruf der Methode in die Warteschlange unerledigter Arbeiten. Um die Benutzeroberfläche wenig zu stören, eignet sich eine niedrige Priorität für das Delegatenobjekt (z. B. **DispatcherPriority.SystemIdle**).

[1] https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/threading-model

4.4 Die Klasse AppBuilder

In der **Main()** – Methode der Startklasse **Program** einer Avalonia UI – Desktop-Anwendung erstellt die statische Methode **Configure<App>()** der Klasse **AppBuilder** ein Objekt der Klasse **AppBuilder**, das ...

- die Anwendung durch einen Aufruf der Methode **UsePlatformDetect()** mit den Diensten des Betriebssystems verbindet[1]
- und die Anwendung mit dem Lebenszyklusmodell **IClassicDesktopStyleApplicationLifetime** startet (siehe Abschnitt 4.5).

Als Multiplattformlösung muss Avalonia UI die Lebenszyklusmodelle für Desktop-, Mobil- und Browser-Anwendungen unterstützen. Das Manuskript beschränkt sich auf Desktop-Anwendungen für die Betriebssysteme Linux, macOS und Windows.

In der Avalonia-Startklasse **Program** kann man für eine Desktopanwendung die folgende, vereinfachte **Main()** – Methode verwenden:

```
class Program {
    [STAThread]
    public static void Main(string[] args) {
        AppBuilder.Configure<App>()
            .UsePlatformDetect()
            .StartWithClassicDesktopLifetime(args);
    }
}
```

Das Attribut **System.STAThread** ist (ohne Bezug zum Thema das aktuellen Abschnitts) unter Windows für die Methode **Main()** erforderlich, um eine **COMException** bei der Nutzung von ***COM***-Objekten (***C****omponent* ***O****bject* ***M****odel*) zu verhindern.[2] Eine Avalonia-Anwendung verwendet unter Windows COM-Objekte z. B. beim Schreibzugriff auf die Zwischenablage.

Ein Aufruf der statischen und generischen **AppBuilder**-Methode **Configure<TApp>()** mit einem auf die Basisklasse **Application** im Namensraum **Avalonia** (siehe Abschnitt 4.7) eingeschränkten Typparameter liefert eine **AppBuilder**-Instanz mit der **TApp**-Konkretisierung als Wert der Eigenschaft **ApplicationType**:

```
public static AppBuilder Configure<TApp>()
    where TApp : Application, new() {
    return new AppBuilder() {
        ApplicationType = typeof(TApp),
        _appFactory = () => new TApp()
    };
}
```

Diese Instanz führt die Methode **UsePlatformDetect()** aus

---

1 https://docs.avaloniaui.net/api/avalonia/appbuilder

2 https://docs.microsoft.com/en-us/windows/desktop/com/single-threaded-apartments

```
public static AppBuilder UsePlatformDetect(this AppBuilder builder) {
  LoadHarfBuzz(builder);

  if (OperatingSystem.IsWindows()) {
    LoadWin32(builder);
    LoadSkia(builder);
  } else if (OperatingSystem.IsMacOS()) {
    LoadAvaloniaNative(builder);
    LoadSkia(builder);
  } else if (OperatingSystem.IsLinux()) {
    LoadX11(builder);
    LoadSkia(builder);
  } else {
    Logger.TryGet(LogEventLevel.Warning, LogArea.Platform)?.Log(builder,
     "Avalonia.Desktop package was referenced on non-desktop platform or it isn't supported");
  }

  return builder;
}
```

und verbindet dabei die Anwendung mit den Techniken des Betriebssystems (mit dem sogenannten *Plattform-Backend*) zur Nutzung von Diensten wie:[1]

- Verwaltung der Fenster
- Zwischenablage
- Standarddialoge zur Dateiauswahl etc.

Zu Beginn lädt **UsePlatformDetect()** die Bibliothek **HarfBuzz** zur Darstellung von Schriften, die Folgen von Unicode-Zeichen für jedes Schriftsystem und jede Sprache in korrekt formatierte und positionierte Glyphen wandelt.[2] Allen Plattformen gemeinsam ist das Laden der universellen Skia - Rendering-Technik.

Nach dem Laden des Plattform-Backends gibt **UsePlatformDetect()** die per Parameter erhaltene **AppBuilder**-Instanz zurück und ermöglicht somit eine Fortsetzung der Methodenverkettung. In obiger **Main()** – Methode folgt der für Desktopanwendungen obligatorische Aufruf der Methode **StartWithClassicDesktopLifetime()**. Diese Methode kehrt erst beim Anwendungsende zurück.

4.5 IClassicDesktopStyleApplicationLifetime

Die von Avalonia unterstützten Betriebssysteme unterscheiden sich bei den Ereignissen und Methoden mit Einfluss auf das Starten und das Beenden von Anwendungen. Dem trägt Avalonia Rechnung durch Anwendungs- bzw. Lebenszyklusmodelle, die durch Schnittstellen definiert sind:

- **IControlledApplicationLifetime**
- **IClassicDesktopStyleApplicationLifetime**
- **ISingleViewApplicationLifetime**
- **IActivityApplicationLifetime**

Die Wahl eines Lebenszyklusmodells ist ein wesentlicher Schritt bei der Anwendungskonfiguration durch den **AppBuilder**, und das Ergebnis bestimmt den Datentyp der Instanz mit dem Lebenszyklusmodell der Anwendung. Auf diese Instanz zeigt die Eigenschaft **ApplicationLifetime** der Klasse **Application**, die die Anwendung im Avalonia-Framework repräsentiert und zentrale Dienste bereitstellt (siehe Abschnitt 4.7).

---

[1] Es handelt sich um eine in der Klasse **AppBuilderDesktopExtensions** (Namensraum **Avalonia**) definierte Erweiterungsmethode, die wie eine Instanzmethode der Klasse **AppBuilder** verwendbar ist (zu Erweiterungsmethoden siehe z. B. Abschnitt 7.15.3 in Baltes-Götz & Frischknecht 2026).

[2] https://harfbuzz.github.io/

Das Manuskript behandelt ausschließlich das Lebenszyklusmodell der Desktopanwendungen für Linux, macOS und Windows, das die Schnittstelle **IClassicDesktopStyleApplicationLifetime** definiert.

4.5.1 Kreation durch die Methode StartWithClassicDesktopLifetime()

Der im Abschnitt 4.4 beschriebene **AppBuilder** führt nach der Methode **UsePlatformDetect()** die Methode **StartWithClassicDesktopLifetime()** aus:[1]

public static int StartWithClassicDesktopLifetime(this AppBuilder *builder*, **string[]** *args*)

Sie sorgt in einer Avalonia - Desktop-Anwendung ...

- für die Anzeige des Hauptfensters
- und die Inbetriebnahme der Ereignisschleife durch den Dispatcher (siehe Abschnitt 4.3).

Insbesondere ist als Folge dieses Methodenaufrufs die von der **Application**-Eigenschaft **ApplicationLifetime** referenzierte Instanz mit dem Lebenszyklusmodell der Anwendung vom Typ **IClassicDesktopStyleApplicationLifetime** mit der Basisschnittstelle **IControlledApplicationLifetime**. Das ist der Standard für die im Manuskript behandelten Avalonia-Anwendungen.

Eine Inspektion der Quellcodedatei **ClassicDesktopStyleApplicationLifetime.cs** zeigt, dass die von **StartWithClassicDesktopLifetime()** indirekt aufgerufene Methode **StartCore()** für die Anzeige des Hauptfensters (**ShowMainWindow()** - Aufruf) und für den Start der Ereignisschleife sorgt (**MainLoop()** - Aufruf):[2]

```
internal int StartCore(string[] args) {
    BeforeInit();
    AfterInit();
    Startup?.Invoke(this, new ControlledApplicationLifetimeStartupEventArgs(args));
    _cts = new CancellationTokenSource();

    ShowMainWindow();
    Dispatcher.UIThread.MainLoop(_cts.Token);
    Environment.ExitCode = _exitCode;
    return _exitCode;
}
```

Im Abschnitt 4.7.1 ist zu erfahren, dass bei einer Desktop-Anwendung das Objekt zum Anwendungsfenster in der Methode **OnFrameworkInitializationCompleted()** entsteht. Vielleicht wundert sich jemand darüber, dass die von **StartWithClassicDesktopLifetime()** aufgerufene Methode **StartCore()** das „erst“ in **OnFrameworkInitializationCompleted()** erzeugte Fenster zur Anzeige bringt. Die Methode **OnFrameworkInitializationCompleted()** wird jedoch *nicht nach*, sondern *während* der Tätigkeit von **StartWithClassicDesktopLifetime()** aufgerufen. Das folgt schon daraus, dass **StartWithClassicDesktopLifetime()** bis zum Ende der Anwendung aktiv bleibt. Im Normalfall existiert das Fensterobjekt also beim **StartCore()** – Aufruf.

In den folgenden Abschnitten stellen wir wichtige Mitglieder der Schnittstelle **IClassicDesktopStyleApplicationLifetime** vor.

---

[1] Es handelt sich um eine in der Klasse **ClassicDesktopStyleApplicationLifetimeExtensions** (Namensraum **Avalonia**) definierte Erweiterungsmethode, die wie eine Instanzmethode der Klasse **AppBuilder** verwendbar ist (zu Erweiterungsmethoden siehe z. B. Abschnitt 7.15.3 in Baltes-Götz & Frischknecht 2026).

[2] https://github.com/AvaloniaUI/Avalonia/blob/main/src/Avalonia.Controls/ApplicationLifetimes/ClassicDesktopStyleApplicationLifetime.cs

4.5.2 Ereignisse

Eine Instanz vom Typ **IClassicDesktopStyleApplicationLifetime** Instanz löst Ereignisse aus, deren Behandlung für ein Programm relevant sein kann. Die folgenden Ereignisse sind in der Basisschnittstelle **IControlledApplicationLifetime** definiert:

- **Startup**
  Das Ereignis tritt in der Startphase auf. Gegenüber seiner Behandlung ist in der Regel die im Abschnitt 4.7.1 beschriebene Methode **OnFrameworkInitializationCompleted()** zu bevorzugen.
- **Exit**
  Das Programm steht unmittelbar vor dem nicht mehr zu verhindernden Ende, kann aber z. B. noch Daten sichern oder Ressourcen wie (Netzwerk- oder Datenbankverbindungen) freigeben.

Das in der Schnittstelle **IClassicDesktopStyleApplicationLifetime** definierte Ereignis **ShutdownRequested** tritt ein, wenn der Benutzer im Begriff ist, seine Desktop-Sitzung zu beenden (durch Abmelden vom Betriebssystem oder Herunterfahren des Rechners).

4.5.3 Eigenschaften

Wie sich das Schließen von Fenstern auf die Terminierung der Anwendung auswirkt, hängt von der **IControlledApplicationLifetime**-Eigenschaft **ShutdownMode** ab. Sie ist vom Typ der gleichnamigen Enumeration im Namensraum **Avalonia.Controls.ApplicationLifetimes** mit den folgenden Werten:

- **OnLastWindowClose** (= Voreinstellung von **ShutdownMode**)
  Avalonia beendet die Anwendung beim Schließen des letzten Fensters.
- **OnMainWindowClose**
  Avalonia beendet die Anwendung beim Schließen des Hauptfensters.
- **OnExplicitShutdown**
  Diese Option deaktiviert die automatische Beendigung durch das Schließen des letzten Fensters oder des Hauptfensters, sodass ein expliziter Aufruf der **IControlledApplicationLifetime**-Methode **Shutdown()** per Programm für die Terminierung erforderlich ist (siehe Abschnitt 4.5.4).

Über die folgenden Eigenschaften der Schnittstelle **IClassicDesktopStyleApplicationLifetime** lassen sich die Fenster einer Anwendung ansprechen:

- Die Eigenschaft **MainWindow** zeigt auf das Hauptfenster.
- Die Eigenschaft **Windows** zeigt auf eine Liste mit allen Top-Level - Fenstern.

Über die Eigenschaft **Args** bietet eine **IClassicDesktopStyleApplicationLifetime**-Instanz den Zugriff auf die Befehlszeilenargumente:

```
public String[] Args { get; set; }
```

4.5.4 Shutdown()

Die **IControlledApplicationLifetime**-Methode **Shutdown()** beendet die Anwendung und übergibt einen *Exit-Code* (alias: *Return-Code*) mit einer Information über den (Miss)erfolg der Programmtätigkeit an das Betriebssystem (z. B. 0: alles gut gegangen, 1: Beendigung mit Fehler):

```
public void Shutdown(int exitCode)
```

Einen Aufruf der **Application**-Methode **Shutdown()** können sowohl die CLR (*Common Language Runtime*) als auch das Programm veranlassen. Das dabei gefeuerte Ereignis **Exit** gibt dem Programm Gelegenheit für Abschlussmaßnahmen.

Das Laufzeitsystem ruft **Shutdown()** z. B. in den folgenden Fällen auf:

- Der Benutzer schließt das letzte Fenster der Anwendung, und die Eigenschaft **ShutdownMode** der **IControlledApplicationLifetime**-Instanz hat den voreingestellten Wert **OnLastWindowClose**.
- Der Benutzer schließt das Hauptfenster der Anwendung, und die Eigenschaft **ShutdownMode** hat den Wert **OnMainWindowClose**.
- Im UI-Thread tritt eine unbehandelte Ausnahme auf, und es ist …
  - entweder keine Behandlungsmethode zum **Dispatcher**-Ereignis **UnhandledException**-vorhanden,
  - oder die vorhandene Methode erklärt die zugrundeliegende Ausnahme *nicht* als behandelt.

  Im Abschnitt 4.9 ist eine Behandlungsmethode zum **UnhandledException**-Ereignis zu sehen, die das Ereignis als behandelt erklärt und anschließend auf Wunsch des Benutzers die Anwendung per **Shutdown()** beendet.
- Die Betriebssystem-Sitzung endet, weil sich der Benutzer abmeldet, oder das Betriebssystem herunterfährt, und das daraufhin ausgelöste Ereignis **ShutdownRequested** (siehe Abschnitt 4.5.2) bleibt entweder unbehandelt oder wird ohne Widerspruch gegen das Sitzungsende behandelt.[1]

Eine Avalonia-Anwendung über die statische **Environment**-Methode **Exit()** zu beenden, ist in der Regel *nicht* sinnvoll, weil diese Methode den kompletten Prozess mit allen vorhandenen Threads abrupt abwürgt.

4.6 Die Klasse Window

Alle Fenster eines Avalonia-Programms stammen von der Klasse **Window** ab, deren Stammbaum wir schon aus Abschnitt 4.2 kennen.[2] Zur Vereinfachung der Darstellung bezeichnen wir ab jetzt ...

- das in der Startphase eines Avalonia-Programms erzeugte Objekt der Hauptfensterklasse als *Fensterobjekt*,
- das zugehörige Fenster als *Hauptfenster*
- und das die Anwendung im Avalonia-Framework repräsentierende Objekt der Klasse **Application** als *Anwendungsobjekt*.

4.6.1 Besonderheiten des Hauptfensters

Als Besonderheiten des Hauptfensters bzw. des zugehörigen Fensterobjekts sind zu nennen:

- In einer Desktop-Anwendung sorgt die vom **AppBuilder** ausgeführte Methode **StartWithClassicDesktopLifetime()** für das Erscheinen des Hauptfensters, das man also nicht selbst über die **Window**-Methode **Show()** veranlassen muss (siehe Abschnitt 4.5.1). Für das *Erstellen* des Fensterobjekts ist aber der Entwickler verantwortlich (siehe Abschnitt 4.7.1).

---

[1] https://docs.avaloniaui.net/api/avalonia/controls/applicationlifetimes/iclassicdesktopstyleapplicationlifetime#shutdownrequested-event

[2] https://docs.avaloniaui.net/api/avalonia/controls/window

- Ist für ein einfaches Programm die Separation von GUI-Design und Anwendungslogik nicht erforderlich, dann realisiert man wesentliche Teile der Anwendungslogik durch Ereignisbehandlungsmethoden in der Hauptfensterklasse (in der Code-Behind – Datei, siehe Kapitel 6).
- Beim Schließen des Hauptfensters endet das Programm, wenn die **Application**-Eigenschaft **ApplicationLifetime** vom Typ **IClassicDesktopStyleApplicationLifetime** ist und deren Eigenschaft **ShutdownMode** vom Typ der gleichnamigen Enumeration im Namensraum **Avalonia.Controls.ApplicationLifetimes** ...
  - den Wert **OnMainWindowClose** besitzt,
  - oder den voreingestellten Wert **OnLastWindowClose** besitzt, und nur das Hauptfenster vorhanden ist.

Unsere Beispielprogramme kommen meist mit *einem* Fenster aus. Selbstverständlich kann eine Avalonia-Anwendung auch *mehrere* Fenster verwenden, wobei in der Regel trotzdem nur *ein* UI-Thread zum Einsatz kommt.

4.6.2 Eigenschaften

Es folgt eine kurze Auswahl der **Window**-Eigenschaften:

- **Title**
  Diese Eigenschaft legt die Titelzeilenbeschriftung des Fensters fest.
- Die von **StyledElement** geerbte Eigenschaft **Resources** zeigt auf ein Objekt vom Typ **IResourceDictionary** (Namensraum **Avalonia.Controls**) mit Ressourcen (z. B. Farbdefinitionen) zur gemeinsamen Nutzung durch alle Steuerelemente des Fensters. Man definiert diese Instanz in der XAML-Datei zum Fenster, z. B.:

```
<Window.Resources>
    <SolidColorBrush x:Key="BerlinerBlau" Color="#003153"/>
</Window.Resources>
```

  In mehreren Fenstern einer Anwendung zu nutzende Ressourcen bringt man besser in der XAML-Datei zum Anwendungsobjekt unter (siehe Abschnitt 4.7.2).
- Die von **StyledElement** geerbte Eigenschaft **Styles** zeigt auf ein Objekt vom gleichnamigen Kollektionstyp im Namensraum **Avalonia.Styling**, das Stile zur gemeinsamen Nutzung durch alle Steuerelemente des Fensters enthält. Man definiert diese Instanz in der XAML-Datei zum Fenster, z. B.:

```
<Window.Styles>
    <Style Selector="Button.berlin">
        <Setter Property="Background" Value="{StaticResource BerlinerBlau}"/>
        <Setter Property="Foreground" Value="Yellow"/>
        <Setter Property="FontSize" Value="25"/>
        <Setter Property="CornerRadius" Value="15"/>
    </Style>
</Window.Styles>
```

  Während eine Ressource (ein **IResourceDictionary**-Element) einen Wert enthält, den man über seinen Schlüssel (**x:Key**) einer Steuerelementeigenschaft zuweisen kann, ist ein Stil (ein **Styles**-Element) eine Sammlung von Eigenschaft-Wert – Paaren, die (wie in CSS) über den **Selector** jedem Steuerelement mit passendem **Classes**-Attribut zugeordnet wird.
  In mehreren Fenstern einer Anwendung zu nutzende Stile bringt man besser in der XAML-Datei zum Anwendungsobjekt unter (siehe Abschnitt 4.7.2).

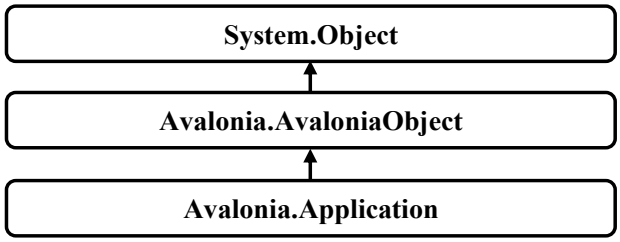
4.6.3 Ereignisse

In der Klasse **Window** sind aufgrund des umfangreichen Stammbaums zahlreiche, meist geerbte Ereignisse verfügbar. Diese Ereignisse sind in **Window** oder in der unmittelbaren Basisklasse **WindowBase** definiert, z. B.:

- **Activated**, **Deactivated**
  Das Fenster ist in den Vordergrund geholt bzw. von dort verdrängt worden.
- **Closing**
  Das Fenster hat einen **Close()** - Aufruf erhalten. In der Ereignisbehandlung lässt sich das Schließen verhindern.

4.7 Die Klasse Application

Eine Avalonia-Anwendung benötigt ein Objekt aus der Klasse **Application** oder einer Spezialisierung, das wir im Manuskript als *Anwendungsobjekt* bezeichnen.[1] Im Vergleich zum Stammbaum der Klasse **Window**, der uns im Abschnitt 4.2 zur Vorstellung wichtiger Steuerelement-Basisklassen gedient hat, ist die Ahnenreihe der Klasse **Application** kurz:



Das die Anwendung im Avalonia-Framework repräsentierende Anwendungsobjekt enthält wichtige Methoden sowie Eigenschaften.

4.7.1 Initialisierungsmethoden

Die **Application**-Methoden **Initialize()** und **OnFrameworkInitializationCompleted()** kommen regelmäßig zum Initialisieren einer Avalonia-Anwendung zum Einsatz:

```
public partial class App : Application {
    public override void Initialize() { ... }
    public override void OnFrameworkInitializationCompleted() { ... }
}
```

Die Methode **Initialize()** bietet Gelegenheit, den vom XamlX-Compiler (siehe Abschnitt 5.3) erstellten ***IL***-Code (***I**ntermediate **L**anguage Code*) zu laden, z. B.:

```
public override void Initialize() {
    AvaloniaXamlLoader.Load(this);
}
```

Mit dem Aufruf der Methode **OnFrameworkInitializationCompleted()** signalisiert Avalonia den Abschluss seiner Initialisierung, sodass man sich um die Erstellung des Fensterobjekts und andere Vorbereitungen kümmern kann. In einer Desktop-Anwendung entsteht per Voreinstellung die folgenden Methode:

```
public override void OnFrameworkInitializationCompleted() {
    if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
        desktop.MainWindow = new MainWindow();
    base.OnFrameworkInitializationCompleted();
}
```

---

[1] https://docs.avaloniaui.net/api/avalonia/application

Im Abschnitt 4.9 erweitern wir in einer Variante des im Abschnitt 2.4 entwickelten Währungskonverters die Methode **OnFrameworkInitializationCompleted()** um die Registrierung einer Behandlungsmethode beim **UnhandledException**-Ereignis des **Dispatcher-**Objekts zum UI-Thread, um die Benutzer im Fall einer unbehandelten Ausnahme mit einer Dialogbox informieren zu können.

4.7.2 Eigenschaften

Die Klasse **Application** enthält Eigenschaften zur Verwaltung von Ressourcen, Stilen und Themen sowie für andere Zwecke, z. B.

- Die statische **Application**-Eigenschaft **Current** zeigt auf das Anwendungsobjekt.
- Die Eigenschaft **ApplicationLifetime** zeigt auf eine Instanz mit dem Lebenszyklusmodell der Anwendung, die im Fall einer Desktop-Anwendung vom Typ **IClassicDesktopStyleApplicationLifetime** ist (siehe Abschnitt 4.5).
- Die Eigenschaft **Resources** zeigt auf ein Objekt vom Typ **IResourceDictionary** (Namensraum **Avalonia.Controls**) mit Ressourcen (z. B. Farbdefinitionen) zur gemeinsamen Nutzung durch alle Fenster und Steuerelemente der Anwendung. Man definiert eine anwendungsbezogene **IResourceDictionary**-Instanz meist in der XAML-Datei zum Anwendungsobjekt, z. B.:

```
<Application.Resources>
    <SolidColorBrush x:Key="BerlinerBlau" Color="#003153"/>
</Application.Resources>
```

- Die Eigenschaft **Styles** zeigt auf ein Objekt vom gleichnamigen Kollektionstyp im Namensraum **Avalonia.Styling**, das Stile zur gemeinsamen Nutzung durch alle Fenster und Steuerelemente der Anwendung enthält. Man definiert eine anwendungsbezogene **Styles**-Kollektion meist in der XAML-Datei zum Anwendungsobjekt, z. B.:

```
<Application.Styles>
    <Style Selector="Button.berlin">
        <Setter Property="Background" Value="{StaticResource BerlinerBlau}"/>
        <Setter Property="Foreground" Value="Yellow"/>
        <Setter Property="FontSize" Value="25"/>
        <Setter Property="CornerRadius" Value="15"/>
    </Style>
    <FluentTheme />
</Application.Styles>
```

  Während eine Ressource (ein **IResourceDictionary**-Element) einen Wert enthält, den man über seinen Schlüssel (**x:Key**) einer Steuerelementeigenschaft zuweisen kann, ist ein Stil (ein **Styles**-Element) eine Sammlung von Eigenschaft-Wert – Paaren, die (wie in CSS) über den **Selector** jedem Steuerelement mit passendem **Classes**-Attribut zugeordnet wird.
- Die Eigenschaft **RequestedThemeVariant** zeigt auf die in einer Anwendung eingestellte Themenvariante. Die in Avalonia eingebauten Themen (Pakete mit visuellen Einstellungen) **SimpleTheme** und **FluentTheme** unterstützen die Varianten **Dark** und **Light**.

4.8 NuGet-Pakete installieren

Wir erläutern die gelegentlich erforderliche Installation eines NuGet-Pakets am Beispiel des Pakets **Avalonia.Themes.Simple**. Für den Wechsel vom voreingestellten **FluentTheme** auf das **SimpleTheme**

FluentTheme



SimpleTheme



genügt es nicht, im Element **Application.Styles** die Voreinstellung

```
<Application.Styles>
    <FluentTheme />
</Application.Styles>
```

auf

```
<Application.Styles>
    <SimpleTheme />
</Application.Styles>
```

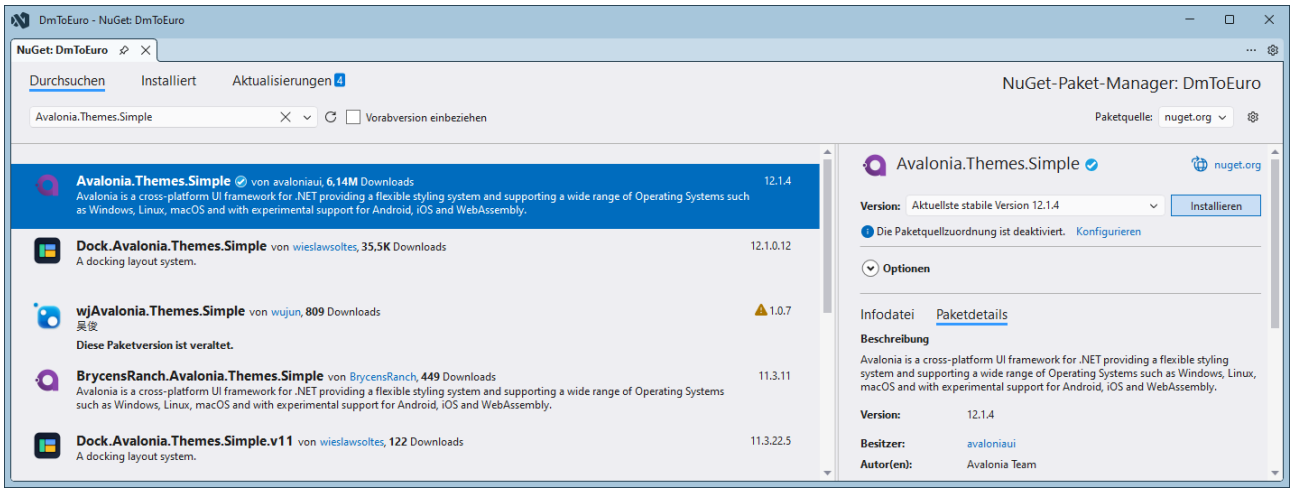
zu ändern. Zusätzlich muss man im Projekt das NuGet-Paket **Avalonia.Themes.Simple** installieren. Dazu startet man den NuGet-Paketmanager über das Item

NuGet-Pakete verwalten

aus dem Kontextmenü zum Projektknoten

Abhängigkeiten > Pakete

im **Projektmappen-Explorer**. Im Fenster des Paketmanagers wählt man die Registerkarte **Durchsuchen**, trägt den Paketnamen **Avalonia.Themes.Simple** in das Suchfeld ein und klickt auf den Schalter **Installieren**:



Diese NuGet-Paketinstallation ist in jedem Projekt zu wiederholen, kostet aber nur wenige Sekunden.

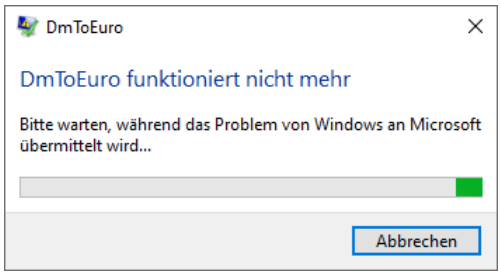
4.9 MessageBox-Ersatz

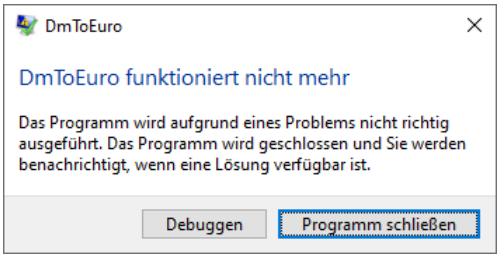
Wie im Abschnitt 4.7.1 zu erfahren war, ruft Avalonia UI nach der Framework-Initialisierung die Methode **OnFrameworkInitializationCompleted()** auf. Die **Application**-Ableitung eines typischen Desktop-Projekts enthält eine Überschreibung dieser Methode, die das Fensterobjekt erstellt und nach Bedarf weitere Initialisierungen vornimmt. Wir registrieren in einer Variante des im Abschnitt 2.4 erstellten Währungskonverters eine Behandlungsmethode beim **UnhandledException**-

Ereignis des **Dispatcher**-Objekts zum UI-Thread, um die Benutzer im Fall einer unbehandelten Ausnahme mit einer Dialogbox informieren zu können:[1]

```
public override void OnFrameworkInitializationCompleted() {
    if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
        desktop.MainWindow = new MainWindow();
    Dispatcher.UIThread.UnhandledException += OnUnhandledException;
    base.OnFrameworkInitializationCompleted();
}
```

Wenn im UI-Thread, der sich um die Bedienoberfläche und die Interaktion mit dem Benutzer kümmert, eine unbehandelte Ausnahme auftritt, dann besteht die Standardreaktion unter Windows darin, das Programm nach der Anzeige einer Infodialogbox zu beenden:





Die folgende Behandlungsmethode für das **UnhandledException**-Ereignis

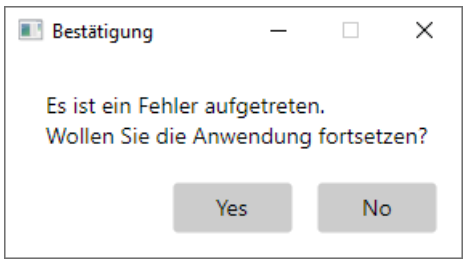
```
private async void OnUnhandledException(object sender,
                                        DispatcherUnhandledExceptionEventArgs e) {
    if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop) {
        e.Handled = true;
        var box = MessageBoxManager.GetMessageBoxStandard("Bestätigung",
                  "Es ist ein Fehler aufgetreten.\nWollen Sie die Anwendung fortsetzen?",
                  ButtonEnum.YesNo);
        var result = await box.ShowWindowAsync();
        if (result == ButtonResult.No)
            desktop.Shutdown();
    }
}
```

erklärt die zugrundeliegende Ausnahme als behandelt

```
e.Handled = true;
```

und lässt den Benutzer über die Terminierung der Anwendung entscheiden.

Dieser Dialog

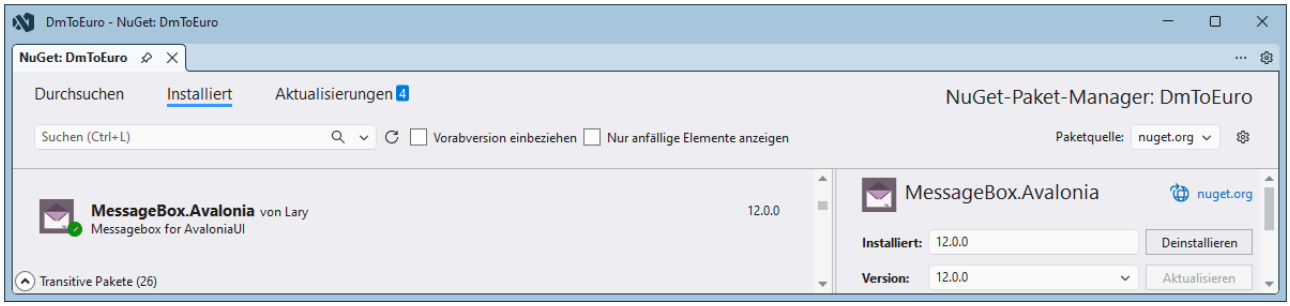


ist mit dem Lieferumfang von Avalonia 12 nicht zu realisieren, und die wesentliche Aufgabe des aktuellen Abschnitts besteht darin, eine Alternativlösung für die in Avalonia 12 fehlende Entsprechung zur WPF-Klasse **MessageBox** vorzustellen.

---

[1] Die Methode stammt aus einem Programm, das sich im folgenden Visual Studio – Projekt befindet:
...\BspUeb\Grundlagen\DmToEuro.UnhandledException

Man kann eine eigene **Window**-Ableitung definieren (siehe Übungsaufgabe im Abschnitt 11.4) oder eine fertige und als Nuget-Paket verfügbare Lösung verwenden.[1] Wir verwenden in der Regel die zweite Lösung und installieren das von der (in Amsterdam ansässigen) IT-Firma JetBrains beigesteuerte Paket **MessageBox.Avalonia** über den **NuGet-Paket-Manager**:



Diese NuGet-Paketinstallation ist in jedem Projekt zu wiederholen, kostet aber nur wenige Sekunden.

Die Methode **GetMessageBoxStandard()** der statischen Klasse **MessageBoxManager**

public static IMsBox<ButtonResult> GetMessageBoxStandard(
MessageBoxStandardParams *@params***)**

liefert ein Objekt vom Typ **IMsBox<ButtonResult>**, das u. a. die folgenden Methoden beherrscht:[2]

- **public Task<ButtonResult> ShowWindowAsync()**
  Diese Methode zeigt ein Fenster als nicht-modalen Dialog, wobei das auslösende Fenster uneingeschränkt bedienbar bleibt.
- **Task<ButtonResult> ShowWindowDialogAsync(Window** *owner***)**
  Es erscheint ein Fenster als modaler Dialog, der bis zum Schließen das auslösende Fenster blockiert, sodass man es z. B. nicht bewegen kann.

Beide Methoden werden asynchron ausgeführt, belasten den UI-Thread also nicht.

Die Dokumentation des Anbieters empfiehlt dringend, die Methoden grundsätzlich per **await**-Operator aufzurufen:

> Always use await when calling MessageBox methods. Never use .Result or .Wait() on the Task, as this will cause a deadlock and freeze your UI.

Ein Methodenaufruf per **await**-Operator ist nur in einer Methode mit dem Modifikator **async** möglich, z. B.:

```
private async void ButtonWindow_Click(object? sender, RoutedEventArgs e) {
    var box = MessageBoxManager.GetMessageBoxStandard("Nachfrage",
                                "Gefällt Ihnen das Programm?",
                                ButtonEnum.YesNo, MsBox.Avalonia.Enums.Icon.Question);
    var result = await box.ShowWindowAsync();
    antwort = (int)result;
}
```

Um den Unterschied zwischen einem modalen und einem nicht-modalen Dialogfenster zu erläutern, definieren wir durch den folgenden XAML-Code[3]

---

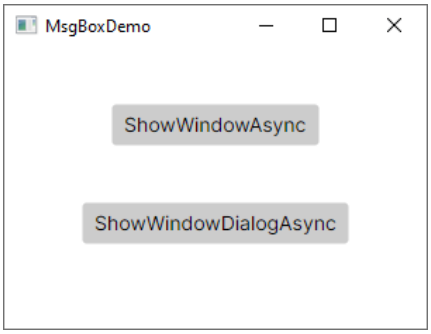
1 https://docs.avaloniaui.net/troubleshooting/controls/messagebox

2 https://github.com/AvaloniaCommunity/MessageBox.Avalonia

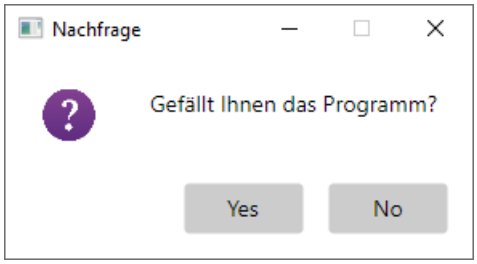
3 Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Grundlagen\MsgBoxDemo

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Width="300" Height="200"
        x:Class="MsgBoxDemo.MainWindow"
        Title="MsgBoxDemo">
    <StackPanel>
        <Button Content="ShowWindowAsync" HorizontalAlignment="Center" Margin="40"
                Click="ButtonWindow_Click" />
        <Button Content="ShowWindowDialogAsync" HorizontalAlignment="Center"
                Click="ButtonDialog_Click" />
    </StackPanel>
</Window>
```

ein Hauptfenster mit zwei **Button**-Objekten:



Das **Click**-Ereignis des oberen Schalters wird durch die eben vorgestellte Methode `ButtonWindow_Click()` behandelt. Es erscheint ein zusätzliches Fenster, wobei das Hauptfenster uneingeschränkt verwendbar bleibt:



Zum unteren Schalter des Hauptfensters gehört die folgenden **Click**-Behandlungsmethode:

```
private async void ButtonDialog_Click(object? sender, RoutedEventArgs e) {
    var box = MessageBoxManager.GetMessageBoxStandard("Nachfrage",
                               "Gefällt Ihnen das Programm?",
                               ButtonEnum.YesNo, MsBox.Avalonia.Enums.Icon.Question);
    if (GetTopLevel(this) is not Window window)
        return;
    var result = await box.ShowWindowDialogAsync(window);
    antwort = (int)result;
}
```

Sie startet dasselbe Nachfragefenster als modalen Dialog, und man kann das Hauptfenster bis zum Schließen des Dialogs *nicht* ansprechen (z. B. nicht verschieben).

Dank **await**-Operator erhält man den vom Benutzer gewählten Schalter über ein Ergebnis vom Enumerationstyp **ButtonResult** mit den Werten

Ok, **Yes**, **No**, **Abort**, **Cancel**, **None**

Durch optionale Parameter der Methode **GetMessageBoxStandard()** kann man ...

- durch einen Wert der Enumeration **ButtonEnum** die Ausstattung des Dialogfensters mit Schaltflächen festlegen:
  Ok, **YesNo**, **OkCancel**, **OkAbort**, **YesNoCancel**, **YesNoAbort**
- durch einen Wert der Enumeration **Icon** ein im Dialogfensters anzuzeigendes Symbol wählen:
  None, **Battery**, **Database**, **Error**, **Folder**, **Forbidden**, **Info**, **Plus**, **Question**, **Setting**, **SpeakerLess**, **SpeakerMore**, **Stop**, **Stopwatch**, **Success**, **Warning**, **Wifi**

Beide Enumerationen befinden sich im Namensraum **MsBox.Avalonia.Enums**. Trotz einer **using**-Direktive für diesen Namensraum muss man den Namen der Enumeration **Icon** in einer Methode einer von **Window** abstammenden Klasse vollqualifiziert schreiben, weil die Klasse **Window** eine Eigenschaft namens **Icon** besitzt (siehe Methode `ButtonDialog_Click()`).

5 XAML

Microsoft hat die XML-basierte *eXtensible Application Markup Language* (***XAML***) zur Deklaration der Bedienoberfläche einer WPF-Anwendung entwickelt und verwendet sie mittlerweile auch für andere GUI-Frameworks (vgl. Abschnitt 1.4). Avalonia UI hat die XAML-Definition von Microsoft weitgehend unverändert übernommen hat, was für Entwickler mit WPF-Erfahrungen von Vorteil ist.

Wie in WPF ist auch in Avalonia UI eine imperative GUI-Kreation durch C# - Anweisungen möglich, doch hat die Deklaration per XAML deutliche Vorteile:

- Vereinfachung der GUI-Gestaltung durch die übersichtliche Darstellung des Baums mit den Bedienelementen eines Fensters
- Trennung von Anwendungslogik und GUI-Design
  Das ermöglicht professionelle Anwendungsarchitekturen wie ***MVVM*** (***M**odel-**V**iew-**V**iew**M**odel*, siehe Kapitel 12) und erleichtert die Zusammenarbeit von Software-Entwicklern und Grafikdesignern.

Weil Avalonia UI für die Gestaltung der Bedienoberfläche statt eines WYSIWYG-Designers (wie WPF) nur eine Vorschau bietet (vgl. Abschnitt 2.4), sind Kennnisse der XAML-Syntax unerlässlich. Im Vergleich zur Übernahme von Bedienelementen per Drag & Drop sowie zur Positionierung und Dimensionierung per Maus resultiert aus der manuellen Erstellung aber ein besserer XAML-Code ohne überflüssige bzw. störende Effekte ungeplanter Mausaktionen.

5.1 Elementare Regeln zum Aufbau einer XML-Datei

XAML ist ein Dialekt der zur Deklaration von hierarchisch strukturieren Daten geeigneten *eXtensible Markup Language* (***XML***), und das XML-Format kommt auch in .NET – Konfigurationsdateien zum Einsatz (z. B. in der Projektdatei). Daher lohnt es sich, im aktuellen Abschnitt elementare Regeln zum Aufbau einer XML-Datei zu erläutern.

Eine XML-Datei enthält Text und ist auch für die Lektüre durch Menschen geeignet. In der ersten Zeile steht oft die (nicht obligatorische) **XML-Deklaration** mit einer Versions- und Codierungsangabe, z. B.:

```
<?xml version="1.0" encoding="utf-8" ?>
```

Bei XAML-Dateien fehlt die XML-Deklaration allerdings.

Jede XML-Datei besteht aus hierarchisch verschachtelten Elementen und enthält nur *ein Wurzelelement*. In einer XAML-Datei zur Deklaration eines Avalonia-Fensters hat das Wurzelelement den Namen **Window** (passend zur Klasse **Window** im Namensraum **Avalonia.Controls**):

```
<Window . . . >
  <Panel>
    <Button . . . >
      . . .
    </Button>
      . . .
  </Panel>
</Window>
```

Wie das Beispiel **Window** zeigt, besteht ein XML-Element *mit* Inhalt (z. B. mit untergeordneten Elementen) aus:

- Startkennung (oft als *Start-Tag* bezeichnet)
- Inhalt
- Endkennung (oft als *End-Tag* bezeichnet)

Start- und Endkennung sind durch Paare spitzer Klammern begrenzt und enthalten den Namen des Elements. In der Endkennung folgt auf die einleitende spitze Klammer ein Schrägstrich.

In der Startkennung eines Elements sind **Attribute** erlaubt, die aus Name-Wert - Paaren bestehen und durch mindestens ein Leerzeichen zu trennen sind. Das folgende **Button** - Element zur Deklaration einer Schaltfläche besitzt z. B. die Attribute **HorizontalAlignment** und **VerticalAlignment** zur horizontalen bzw. vertikalen Ausrichtung des Steuerelements:

```
<Button Name="button" HorizontalAlignment="Center" VerticalAlignment="Center">
    . . .
</Button>
```

Alle Attributwerte sind durch (doppelte oder einfache) Anführungszeichen zu begrenzen.

Ein Element *ohne* untergeordnete Elemente oder sonstige Inhalte kann sich auf die Startkennung beschränken, die dann mit einem Schrägstrich zu enden hat, z. B. beim folgenden **Image** – Element:

```
<Image Source="avares://Beobachtungsstudie/Assets/avalonia.png" />
```

Wie beim C# - Quellcode ist auch beim XML-Code die **Groß-/Kleinschreibung relevant** (mit seltenen Ausnahmen, auf die wir ggf. hinweisen).

Das folgende Beispiel zeigt, wie ein Kommentar in einer XML-Datei auszusehen hat:

```
<!-- Kommentar -->
```

Um einen markierten Zeilenbereich einer X(A)ML-Datei als Kommentar zu vereinbaren oder diese Vereinbarung aufzuheben, verwendet man im Visual Studio dieselben Tastenkombinationen wie bei C# - Code:

- Vereinbaren: **Strg+K**, **Strg+C**
- Vereinbarung aufheben: **Strg+K**, **Strg+U**

5.2 XAML-Beschreibung

Das Erstellen der Bedienoberfläche kann in einer Avalonia-Anwendung auch per C# - Quellcode erfolgen, doch ist die XAML-Deklaration in der Regel zu bevorzugen. Man bringt den XAML-Code zu einem Fenster in einer Textdatei mit der UTF-8 - Codierung und der Namenserweiterung **axaml** unter. Neben den Fensterklassen profitiert in einer Avalonia-Anwendung auch die Anwendungsklasse von der Deklaration per XAML.

5.2.1 Wurzelelemente und XML-Namensräume

Als Wurzelelement einer XAML-Datei in einem Avalonia-Programm kommen in Frage:

- **Application**
  Die zur Deklaration der (von **Application** abstammenden Anwendungsklasse) dienende XAML-Datei besitzt ein Wurzelelement namens **Application**, wobei die folgenden Attribute regelmäßig vorhanden sind:
  - Die Attribute **xmlns**, **xmlns:x** vereinbaren XML-Namensräume, wobei der Standardnamensraums kein Präfix benötigt (siehe unten).
  - Das Attribut **x:Class** nennt die Anwendungsklasse (Standardname: `App`) samt .NET-Namensraum. Am Anfang des Attributnamens steht das Namensraumpräfix **x**.
  - Das Attribut **RequestedThemeVariant** wählt eine Variante des in einer Anwendung eingestellten Themes (Pakets mit visuellen Einstellungen). Avalonia enthält für das **FluentTheme** und das und **SimpleTheme** (siehe Abschnitt 4.8) jeweils die Varianten **Dark** und **Light**.

Außerdem kann man …

- Behandlungsmethoden zu Ereignissen der Anwendungsklasse durch weitere Attribute
- sowie Anwendungsstile (z. B. das zu verwendende Theme) im geschachtelten Element **Application.Stiles**

deklarieren.

Welchen XAML-Code das Visual Studio für eine neue Avalonia-Anwendung aufgrund der Projektvorlage **Avalonia .NET App** erzeugt, betrachten wir am Beispiel des im Abschnitt 2.4 erstellten Währungskonverters:

```
<Application xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="DmToEuro.App"
             RequestedThemeVariant="Default">

    <Application.Styles>
        <FluentTheme />
    </Application.Styles>
</Application>
```

Der Name einer XAML-Datei entsteht aus dem Namen der deklarierten Klasse durch Anhängen der Erweiterung **axaml**, sodass bei der Anwendungsklasse per Voreinstellung **App.axaml** resultiert.

- **Window**
  Die zur Deklaration einer (von **Window** abstammenden Fensterklasse) dienende XAML-Datei besitzt ein Wurzelelement namens **Window**. Weil die Klasse zum Hauptfenster einer Anwendung per Voreinstellung den Namen `MainWindow` erhält, landet der zugehörige XAML-Code in der Datei **MainWindow.axaml**. Aufgrund der Projektvorlage **Avalonia .NET App** entsteht für das Hauptfenster der folgende XAML-Code:

```
<Window xmlns="https://github.com/avaloniaui"
        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"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="DmToEuro.MainWindow"
        Title="DmToEuro">
    Welcome to Avalonia!
</Window>
```

Bei Verzicht auf die meist überflüssigen Namensräume **xmlns:d** und **xmlns:mc** sowie auf die Attribute **DesignWidth** und **DesignHeight**, die sich nur auf die XAML-Vorschau auswirken, kann man die Startkennung zum **Window**-Element vereinfachen, z. B.:[1]

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class=" DmToEuro.MainWindow"
        Title="DmToEuro" Width="800" Height="450">
    Welcome to Avalonia!
</Window>
```

Das Attribut **x:Class** nennt die Hauptfensterklasse (Standardname: `MainWindow`) samt .NET-Namensraum.

[1] https://docs.avaloniaui.net/docs/app-development/xaml-preview-and-design-settings

Die Startkennung zu einem XAML - Wurzelelement enthält mehrere **xmlns**-Attribute zur Deklaration von **XML-Namensräumen** unter Verwendung der folgenden Syntax:

xmlns[**:***prefix*]="*URI*"

Ein XML-Namensraum legt erlaubte Elemente und Attribute fest; ein Präfix dient zur Vermeidung von Namenskollisionen.

Relevante .NET – Namensräume (insbesondere die zu Avalonia UI oder zum Projekt gehörenden) werden auf XML-Namensräume abgebildet. Es wäre allerdings sehr umständlich, für die auf verschiedene .NET - Namensräume verteilten Avalonia UI - Typen jeweils das korrekte Präfix angeben zu müssen. Daher gehören *alle* .NET - Namensräume mit Avalonia UI - Typen zu einem einzigen XAML-Namensraum. Dies ist möglich, weil keine Namenskollisionen auftreten. Um die Verwendung des sehr wichtigen XAML-Namensraums mit den WPF-Typen zu erleichtern, hat er *kein* Präfix und fungiert folglich als **XAML-Standardnamensraum**:

```
xmlns="https://github.com/avaloniaui"
```

Die Bezeichnung für einen XML-Namensraum verwendet das ***URI***-Format (***U**niform **R**esource **I**dentifier*), um Eindeutigkeit sicherzustellen. Jedoch existiert im Internet nicht unbedingt ein Ort mit diesem Namen.

Für den projektspezifischen .NET – Namensraum, der u. a. die Anwendungs- und die Fensterklasse enthält, entsteht bei Verwendung der Anwendungsvorlage **Avalonia .NET MVVM App** ein XML-Namensraum mit dem Präfix **local**, z. B. beim Projekt `AvaloniaMvvmApp`:

```
xmlns:local="using:AvaloniaMvvmApp"
```

Alternativ zur empfohlenen Syntax mit dem Schlüsselwort **using** ist auch eine Variante mit dem in WPF üblichen Schlüsselwort **clr-namespace** erlaubt, z. B.:

```
xmlns:local="clr-namespace:AvaloniaMvvmApp"
```

Neben den Namensräumen für .NET - Typen gibt es einen weiteren wichtigen XML-Namensraum. Er enthält die XAML-Sprachdefinition und verwendet das Präfix **x**:[1]

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Die Vereinbarung von XML-Namensräumen erfolgt meist im Start-Tag des Wurzelelements, ist aber grundsätzlich in jedem Start-Tag erlaubt.

Es folgt eine Liste mit oft benötigten **x**-Attributen:

- **x:Class**
  Im Wurzelelement nennt das **x:Class** - Attribut die anwendungsspezifische Ableitung der Basisklasse **Application** bzw. **Window** samt .NET - Namensraum, z. B.:
  ```
  x:Class="DmToEuro.MainWindow"
  ```

[1] Weitere Informationen über die XML-Namensräume mit den Avalonia UI - Klassen bzw. mit der XAML-Sprachdefinition finden sich hier:
https://docs.avaloniaui.net/docs/xaml/namespaces

- **x:Key**
  Zur Verwaltung von Ressourcen (z. B. Farbdefinitionen) kann man auf verschiedenen Ebenen einer Avalonia-Anwendung (z. B. im **Window**- oder **Application**-Element) eine Kollektion aus der Klasse **ResourceDictionary** definieren:

```
<Window x:Class= ...

    <Window.Resources>
        <SolidColorBrush x:Key="BerlinerBlau" Color="#003153" />
        <SolidColorBrush x:Key="Warnung" Color="DarkRed" />
    </Window.Resources>
     . . .
</Window>
```

  Wie man ein Ressourcen-Objekt (z. B. vom Typ **SolidColorBrush**) einer Eigenschaft zuweist, ist im Abschnitt 5.2.3.6 über die Markup-Erweiterungen zu erfahren.

- **x:Name**
  Über dieses Attribut vereinbart man einen Namen für die Instanz, die aus einem XAML-Instanzelement entsteht (siehe Abschnitt 5.3), z. B.:

```
<TextBox x:Name="eingabe" ... />
```

  Es resultiert eine Instanzvariable mit dem gewählten Namen, die in der Code-Behind - Datei (siehe Kapitel 6) einen Instanzzugriff ermöglicht. Man muss nicht unbedingt für jede aus einem XAML-Element resultierende Instanz einen Namen vereinbaren. Es ist z. B. kein Name erforderlich, um bei den Ereignissen eines Steuerelements Behandlungsmethoden per XAML-Code zu registrieren.

  Wenn ein Typ die Eigenschaft **Name** besitzt, dann kann man statt des XAML-Attributs **x:Name** auch das Attribut **Name** verwenden, z. B.:

```
<TextBox Name="eingabe" ... />
```

5.2.2 Instanzelemente

Mit Ausnahme der Eigenschaftselemente (siehe Abschnitt 5.2.3.2) repräsentiert jedes XAML-Element eine Instanz (z. B. ein Objekt) mit einem beliebigen, in einem zugänglichen Assembly realisierten .NET - Typ. In der Startkennung eines Instanzelements ist nach der öffnenden spitzen Klammer der Typname anzugeben.

Sind keine untergeordneten Elemente oder sonstige Inhalte vorhanden, dann ist die Startkennung mit einem Schrägstrich und einer schließenden spitzen Klammer zu komplettieren, z. B.

```
<Label Content="DM:" HorizontalAlignment="Left" VerticalAlignment="Top"
       Margin="40,40,0,0" Width="60" Height="30" />
```

Sind untergeordnete Elemente (z. B. Steuerelemente in einem Layoutcontainer) oder ein sonstiger Inhalt (z. B. ein Text) vorhanden, dann beendet man die Startkennung mit einer schließenden spitzen Klammer, lässt die untergeordneten Elemente bzw. den sonstigen Inhalt folgen und ergänzt eine Endkennung, die zwischen einem Paar spitzer Klammern einen Schrägstrich und den Element- bzw. Typnamen enthält, z. B.:

```
<Panel>
    <Label Content="DM:" HorizontalAlignment="Left" VerticalAlignment="Top"
           Margin="40,40,0,0" Width="60" Height="30" />
    <TextBox x:Name="eingabe" HorizontalAlignment="Left" VerticalAlignment="Top"
             Margin="130,36,0,0" Width="90" Height="30" Padding="5"
             TextAlignment="Right" VerticalContentAlignment="Center"/>
     . . .
</Panel>
```

Ein Instanzelement stellt den Auftrag an die XAML-Verarbeitung dar, eine Instanz vom angegebenen Typ zu erstellen. Die Instanzkreation findet zur Laufzeit über einen Aufruf des parameterlosen Konstruktors des jeweiligen Typs statt (siehe Abschnitt 5.3).

Im Unterschied zu einem Instanzelement deklariert ein Eigenschaftselement keine Instanz, sondern eine Eigenschaft einer Instanz. In der Regel hat diese Eigenschaft keinen elementaren Typ, denn in diesem Fall wäre ein Attribut bequemer als ein Eigenschaftselement. Im folgenden Beispiel erhält die **Button**-Eigenschaft **Background** als Wert eine Referenz auf ein Objekt vom Typ **LinearGradientBrush**, und dieses Objekt wird wiederum per Instanzelement deklariert:

```
<Button Content="Stop" Margin="10" FontSize="48" Foreground="WhiteSmoke">
    <Button.Background>
        <LinearGradientBrush>
            <GradientStop Offset="0.0" Color="Red" />
            <GradientStop Offset="1.0" Color="Blue" />
        </LinearGradientBrush>
    </Button.Background>
</Button>
```

Der Objektgraph eines Fensters lässt sich mit XAML-Instanzelementen erheblich übersichtlicher entwerfen als mit C# - Syntax.

5.2.3 Eigenschaftsausprägungen zuweisen

Um eine Instanzeigenschaft mit einem Wert zu versorgen, genügt oft die Zuweisung einer Zeichenfolge (per Attributsyntax, siehe Abschnitt 5.2.3.1), während in anderen Fällen eine Member-Instanz zu erstellen ist (per Eigenschaftselementsyntax, siehe Abschnitt 5.2.3.2).

5.2.3.1 Attributsyntax

Vielen Instanzeigenschaften kann man in der Instanzelement-Startkennung über die XML-Attributsyntax einen Wert zuweisen, wobei auf den Namen der Eigenschaft das Zuweisungszeichen und eine durch (einfache oder doppelte) Anführungszeichen begrenzte Zeichenfolge mit dem Wert folgen, z. B.

```
<TextBlock Text="Stop" Margin="5" VerticalAlignment="Center" />
```

Zwischen zwei Attributen hat mindestens ein Leerzeichen zu stehen, während ein Komma an dieser Stelle einen Syntaxfehler verursacht.

Bei der Wandlung einer Zeichenfolge in den jeweiligen Eigenschaftstyp sind **Typkonverter** im Spiel. Im letzten Beispiel muss aus der Zeichenfolge zur **TextBlock**-Eigenschaft **Margin**, die für einen Abstand zur Umgebung sorgt, eine Instanz vom Strukturtyp **System.Windows.Thickness** entstehen. Den zuständigen Konverter kennt Avalonia wie in vielen vergleichbaren Fällen, sodass wir uns nicht darum kümmern müssen.[1] Um zu einem Eigenschaftstyp einen benutzerdefinierten Konverter festzulegen, dekoriert man den Eigenschaftstyp mit einem passenden **TypeConverter**-Attribut, z. B.:[2]

```
[TypeConverter(typeof(TemperatureConverter))]
public struct Temperature { . . . }
```

---

1 https://docs.avaloniaui.net/docs/xaml/type-converters

2 Zu C# - Attributen siehe z. B. Kapitel 14 in Baltes-Götz & Frischknecht 2026.

5.2.3.2 Eigenschaftselementsyntax

Die Attributsyntax eignet sich *nicht* dazu, um einer Eigenschaft eine komplexe Member-Instanz zuzuweisen. In diesem Fall ordnet man dem Instanzelement ein *Eigenschaftselement* unter, dessen Name nach dem Schema

Typname.Eigenschaftsname

zu bilden ist. Als Inhalt des Eigenschaftselements ist ein Instanzelement mit dem Typ der Eigenschaft zu erstellen.

Der folgenden XAML-Code weist der **Content**-Eigenschaft eines **Button**-Steuerelements ein **StackPanel**-Layoutobjekt (siehe Abschnitt 10.4) zur Verwaltung der **Button**-Oberfläche zu:

```
<Button Name="button" Background="WhiteSmoke">
    <Button.Content>
        <StackPanel Orientation="Horizontal">
            . . .
        </StackPanel>
    </Button.Content>
</Button>
```

Beschränkt man sich bei der **Content**-Eigenschaft eines **Button**-Instanzelements auf eine Beschriftung, dann kommt man mit der Attributsyntax aus, z. B.:

```
<Button Name="button" Content="Stop" Background="WhiteSmoke"/>
```

Die Attributsyntax ist eine bequeme, per Typkonverter (vgl. Abschnitt 5.2.3.1) ermöglichte Kurzform der Eigenschaftselementsyntax. Im letzten Beispiel erhält die **Button**-Instanzeigenschaft **Background** vom Schnittstellentyp **IBrush** (Namensraum **Avalonia.Media**) per Attributsyntax als Wert ein Objekt vom Typ **SolidColorBrush**, und man spart sich die folgende, aufwändigere Eigenschaftselementsyntax:

```
<Button Name="button" Content="Stop">
    <Button.Background>
        <SolidColorBrush Color="WhiteSmoke"/>
    </Button.Background>
</Button>
```

Mit Ausnahme des Wurzelelements treten alle Instanzelemente einer XAML-Datei bei der Zuweisung einer Eigenschaftsausprägung auf, wobei die Zuweisung im Fall der sogenannten *Inhaltseigenschaft* eines Typs implizit geschieht (siehe Abschnitt 5.2.3.3). Im folgenden Beispiel erhält die Inhaltseigenschaft **Content** der Klasse **Window** ein Objekt der Layoutcontainerklasse **Panel** als Wert, und der Inhaltseigenschaft **Children** der Klasse **Panel** wird eine Liste von Steuerelementen zugewiesen:

```
<Window ...>
    <Panel>
        <Label Name="label" Content="Suchen bis (2 bis 2146435070)" . . . />
        <Button Name="btnStart" Content="Suche starten"  . . . />
        <TextBox Name="textBox" . . . />
        <ListBox Name="listBox"  . . ./>
    </Panel>
</Window>
```

Die explizite, in der Praxis nicht anzutreffende Schreibweise macht deutlich, dass Eigenschaftszuweisungen im Spiel sind:

```
<Window ...>
    <Window.Content>
        <Panel>
            <Panel.Children>
                <Label Name="label" Content="Suchen bis (2 bis 2146435070)" . . . />
                <Button Name="btnStart" Content="Suche starten"  . . . />
                <TextBox Name="textBox" . . . />
                <ListBox Name="listBox"  . . ./>
           </Panel.Children>
        </Panel>
    </Window.Content>
</Window>
```

Die Eigenschaft **Children** besitzt den Kollektionstyp **Controls** (Namensraum **Avalonia.Controls**), der von **AvaloniaList<Control>** abstammt. Dank der Kollektionssyntax (siehe Abschnitt 5.2.3.5) kann man die Instanzelemente in der Kollektion auflisten.

5.2.3.3 Inhaltseigenschaft

Jede Avalonia-Klasse kann von ihren Instanzeigenschaften genau eine als sogenannte *Inhaltseigenschaft* (engl.: *content property*) deklarieren. Dazu ist in der Typdefinition der Inhaltseigenschaft das Attribut **Content** (Namensraum **Avalonia.Metadata**) voranzustellen. Die Klasse **TextBlock** (Namensraum **Avalonia.Controls**) hat z. B. die Inhaltseigenschaft **Inlines**:

```
[Content]
public InlineCollection? Inlines {
    get => _inlines;
    set => SetAndRaise(InlinesProperty, ref _inlines, value);
}
```

In der Klasse **ContentControl** ist **Content** als Inhaltseigenschaft deklariert:

```
[Content]
public object? Content {
    get => GetValue(ContentProperty);
    set => SetValue(ContentProperty, value);
}
```

Ist ein Kindelement als Wert der Inhaltseigenschaft anzugeben, dann kann man auf das Kennungspaar gemäß XAML-Eigenschaftselementsyntax (siehe Abschnitt 5.2.3.2) verzichten. Weil die Klasse **Button** indirekt von der Klasse **ContentControl** abstammt, lässt sich das im Abschnitt 5.2.3.2 präsentierte Beispiel mit einer Deklaration für die **Button**-Eigenschaft **Content**

```
<Button Name="button" Background="WhiteSmoke">
    <Button.Content>
        <StackPanel Orientation="Horizontal">
            . . .
        </StackPanel>
    </Button.Content>
</Button>
```

unter Verzicht auf das Kennungspaar

```
<Button.Content> . . . </Button.Content>
```

einfacher schreiben:

```
<Button Name="button" Background="WhiteSmoke">
    <StackPanel Orientation="Horizontal">
     . . .
    </StackPanel>
</Button>
```

5.2.3.4 Textinhaltsyntax

Bei einigen Eigenschaftselementen kann man Text *ohne* Begrenzung durch Anführungszeichen als Wert angeben, z. B.:

```
<Button>
    <Button.Content>
        Knopf
    </Button.Content>
</Button>
```

Für den Datentyp der betroffenen Eigenschaft muss eine von den folgenden Bedingungen erfüllt sein:

- Einer Eigenschaft von diesem Typ kann man eine Zeichenfolge zuweisen, was z. B. beim Typ **Object** (dem Typ der **ContentControl**-Eigenschaft **Content**) der Fall ist.
- Für den Typ ist eine Klasse mit entsprechenden Kompetenzen als Typkonverter definiert (vgl. Abschnitt 5.2.3.1), z. B.:

  ```
  <Button.Background>
      LightBlue
  </Button.Background>
  ```

- Der Typ ist im XAML-Sprachumfang bekannt, was bei vielen elementaren Datentypen (z. B. **Int32**, **Double**) der Fall ist, z. B.:

  ```
  <Button.Height>
      40
  </Button.Height>
  ```

Man wendet die Textinhaltsyntax meist bei der *Inhaltseigenschaft* an (siehe Abschnitt 5.2.3.3), sodass sich das erste Beispiel des aktuellen Abschnitts vereinfachen lässt:

```
<Button>Knopf</Button>
```

Diese Lösung ist sogar kürzer als die äquivalente Attributsyntax:

```
<Button Content="Knopf"/>
```

5.2.3.5 Kollektionssyntax

Ist eine Eigenschaft vom Typ einer Kollektion, dann kann man...

- auf ein Instanzelement zur Kollektion verzichten
- und die per **Add()** in die Kollektion aufzunehmenden Instanzelemente direkt auflisten.

Im folgenden Beispiel erhält ein **Button**-Steuerelement zur Hintergrundgestaltung als **Background**-Wert per Eigenschaftselementsyntax ein **LinearGradientBrush**-Objekt. Dieses Objekt definiert einen Farbverlauf, und seine **GradientStops**-Eigenschaft ist vom Typ **GradientStopCollection**. Man benötigt kein Instanzelement zur Kollektion, sondern kann die Kollektionselemente vom Typ **GradientStop** direkt auflisten:

```
<Button Content="Stop" Margin="10" FontSize="48" Foreground="WhiteSmoke">
    <Button.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStop Offset="0.0" Color="Red" />
                <GradientStop Offset="1.0" Color="Blue" />
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Button.Background>
</Button>
```

Weil **GradientStops** die Inhaltseigenschaft von **LinearGradientBrush** ist, lässt sich das Beispiel noch einfacher formulieren:

```
<Button Content="Stop" Margin="10" FontSize="48" Foreground="WhiteSmoke">
    <Button.Background>
        <LinearGradientBrush>
            <GradientStop Offset="0.0" Color="Red" />
            <GradientStop Offset="1.0" Color="Blue" />
        </LinearGradientBrush>
    </Button.Background>
</Button>
```

5.2.3.6 Markup-Erweiterungen

Mit den bisher vorgestellten Techniken zur Festlegung von Eigenschaftsausprägungen ist es *nicht* möglich, ...

- eine Referenz zu einem bereits existenten Objekt zuzuweisen,
- eine Verbindung zu einer Quelle von variablen Werten herzustellen (Datenbindung).

Dies gelingt mit einer sogenannten *Markup-Erweiterung*, die man meist per Attributsyntax unter Verwendung einer geschweift eingeklammerten Zeichenfolge realisiert. Im folgenden Beispiel erhält die Eigenschaft **Background** mit Hilfe der Klasse **StaticResourceExtension** eine Referenz auf ein anderenorts definiertes Objekt:

```
Background="{StaticResource BgColor}"
```

Bei Verwendung der Attributsyntax gibt man den Namen der Markup-Erweiterungsklasse unmittelbar nach der öffnenden geschweiften Klammer an und darf dabei den Namensteil *Extension* (falls vorhanden) weglassen (siehe Beispiel).

Besonders wichtig sind die folgenden Markup-Erweiterungsklassen:

- **StaticResourceExtension**
  Im Abschnitt 5.2.1 war zu erfahren, dass man zur Verwaltung von Ressourcen (z. B. Farbdefinitionen) auf verschiedenen Ebenen einer Avalonia-Anwendung (z. B. im **Application**-Element) ein Kollektionsobjekt aus der Klasse **ResourceDictionary** definieren kann. Im folgenden Beispiel wird das von der **Application**-Eigenschaft **Resources** referenzierte **ResourceDictionary**-Objekt mit anwendungsglobal verwendbaren Ressourcen per Eigenschaftselement- und Kollektionssyntax befüllt. Für ein Ressourcen-Objekt ist der zur Ansprache erforderliche Schlüssel anzugeben, z. B. für ein Objekt der zur Oberflächengestaltung geeigneten Klasse **SolidColorBrush**:

  ```
  <Application xmlns=https://github.com/avaloniaui . . . >

      <Application.Resources>
          <SolidColorBrush x:Key="BgColor" Color="WhiteSmoke"/>
      </Application.Resources>

      <Application.Styles>
          <FluentTheme />
      </Application.Styles>
  </Application>
  ```

  Über die Markup-Erweiterungsklasse **StaticResourceExtension** lässt sich ein Objekt der Ressourcen-Kollektion anwendungsglobal verwenden, wobei nach der Markup-Erweiterungsklasse der Schlüssel anzugeben ist, z. B.:

  ```
  <Window xmlns=https://github.com/avaloniaui . . .
      Background="{StaticResource BgColor}">
    . . .
  </Window>
  ```

- **Binding**
  In der folgenden **ComboBox**-Deklaration

```
<ComboBox Name="ComboBox" ItemsSource="{Binding Path=Personen}"
          DisplayMemberBinding="{Binding Path=FirstName}" />
```

  erhält ein Objekt der Klasse **Binding** per Markup-Erweiterung den Auftrag, die **ItemsSource**-Eigenschaft des Steuerelements mit einer beobachtbaren Personenkollektion

```
public ObservableCollection<Person> Personen { get; } = [];
```

  zu verbinden. Die **DisplayMemberBinding**-Eigenschaft sorgt dafür, dass von der aktuell gewählten Person der Vorname zu sehen ist, z. B.:



  Eine Begrenzung der **Path**-Werte durch Anführungszeichen ist weder erforderlich noch erlaubt, weil es ansonsten zu einer schwer interpretierbaren Verschachtelung von Anführungszeichen käme. Zudem darf man den **Path**-Eigenschaftsnamen weglassen:

```
<ComboBox Name="ComboBox" ItemsSource="{Binding Personen}"
          DisplayMemberBinding="{Binding FirstName}" />
```

  Mit der Datenbindung werden wir uns im Kapitel 9 ausführlich beschäftigen. Dort sind u.a. die Voraussetzungen für die im Beispiel verwendete **Binding**-Syntax beschrieben:
  - die in der Klasse **Avalonia.StyledElement** definierte Eigenschaft **DataContext**
  - das XAML-Attribut **x:DataType**

  Das XAML-Attribut **x:DataType** ist für die seit Avalonia 12 voreingestellte *kompilierte* Bindung erforderlich, die im Vergleich zu der in WPF per Voreinstellung (ohne AOT-Übersetzung) verwendeten Bindung *per Reflexion* für eine Fehlererkennung zur Übersetzungszeit und für ein verbessertes Laufzeitverhalten sorgt.[1]

5.2.4 XAML-Registrierung von Ereignisbehandlungsmethoden

Per Attributsyntax kann man nicht nur einer Eigenschaft, sondern auch einem Ereignis einen Wert zuweisen. Dabei ist der Name einer Behandlungsmethode (ohne Parameterklammern) anzugeben, z. B.:

```
<Button Content="Konvertieren"  . . . IsDefault="True" Click="Button_Click"/>
```

Implementiert wird eine solche Behandlungsmethode in der Code-Behind - Datei zum XAML-Code (siehe Kapitel 6).

Diese Praxis ist in kleinen Programmen angemessen, verletzt aber das Prinzip der Trennung von Anwendungslogik und GUI-Gestaltung. Im Zusammenhang mit der MVVM-Anwendungsarchitektur stellen wir die alternative Lösung per Kommandobindung vor (siehe Abschnitt 12.3).

5.3 XAML-Verarbeitung

Bei der XAML-Verarbeitung unterscheidet sich Avalonia UI erheblich von WPF:[2]

- Beim Erstellen eines Programms übersetzt der in Avalonia enthaltene **XamlX-Compiler** den XAML-Code in ***IL***-Code (***I**mmediate **L**anguage*)
- Die in WPF im Erstellungsprozess aus der XAML-Dateien erstellten und als Ressourcen in das erzeugte Assembly einbezogenen ***BAML***-Dateien (***B**inary **A**pplication **M**arkup **L**anguage*) gibt es in Avalonia UI nicht.

---

[1] https://docs.avaloniaui.net/docs/avalonia12-breaking-changes

[2] https://docs.avaloniaui.net/docs/xaml/compilation

- Zur Laufzeit entstehen (ohne die Notwendigkeit zur Interpretation von XAML-Code) die visuellen Bäume der Fenster durch die vom Anwendungsobjekt aufgerufene statische Methode **Load()** der Klasse **AvaloniaXamlLoader**:

```
public partial class App : Application {
  public override void Initialize() {
    AvaloniaXamlLoader.Load(this);
  }

  public override void OnFrameworkInitializationCompleted() { . . . }
}
```

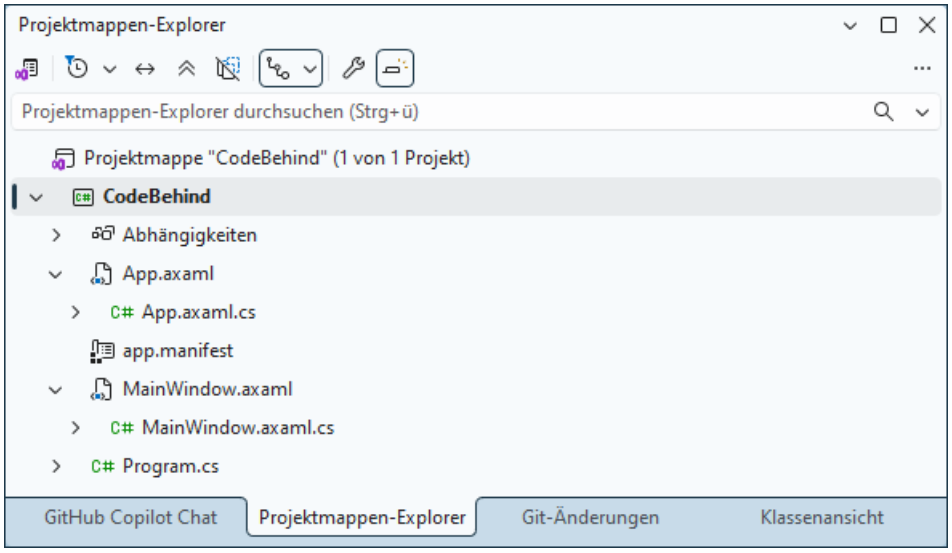
Mit der Rolle der Methode **OnFrameworkInitializationCompleted()** bei der Anwendungsinitialisierung haben wir uns im Abschnitt 4.7.1 beschäftigt.

6 Code-Behind - Dateien

Ein vom Visual Studio aufgrund der Vorlage **Avalonia .NET App** erstelltes Projekt enthält die folgenden XAML-Dateien:

- **App.axaml** zur Deklaration von Anwendungsklassen-Eigenschaften mit Bezug zur Bedienoberfläche (z. B. **RequestedThemeVariant**, **Application.Styles**)
- **MainWindow.axaml** zur Deklaration der Hauptfensterklassen-Bedienoberfläche

Wie der **Projektmappen-Explorer** nach dem Aufklappen der Zweige zu den beiden XAML-Dateien zeigt, existiert jeweils eine begleitende C# - Quellcodedatei mit einer partiellen Definition der Anwendungs- bzw. Hauptfensterklasse:



Für diese sogenannten *Code-Behind - Dateien* sind die Entwickler zuständig, während das Erstellungssystem MSBuild unter Beteiligung von Avalonia-spezifischen Erstellungsoperationen die partiellen Klassendefinitionen (z. B. aufgrund der XAML-Verarbeitung) vervollständigt. Der XamlX-Compiler in Avalonia erstellt allerdings im Unterschied zu seinem WPF-Gegenstück nicht zu jeder XAML-Datei eine C# - Quellcodedatei, sondern übersetzt den XAML-Code teilweise direkt in ***IL***-Code (***I**ntermediate **L**anguage Code*).

6.1 Definition der Anwendungsklasse

In der Datei **App.axaml.cs** befindet sich eine partielle Definition der Anwendungsklasse mit dem voreingestellten Namen App, die von der Klasse **Application** im Namensraum **Avalonia** abstammt, z. B.:[1]

```
using Avalonia;
using Avalonia.Controls.ApplicationLifetimes;
using Avalonia.Markup.Xaml;

namespace CodeBehind;

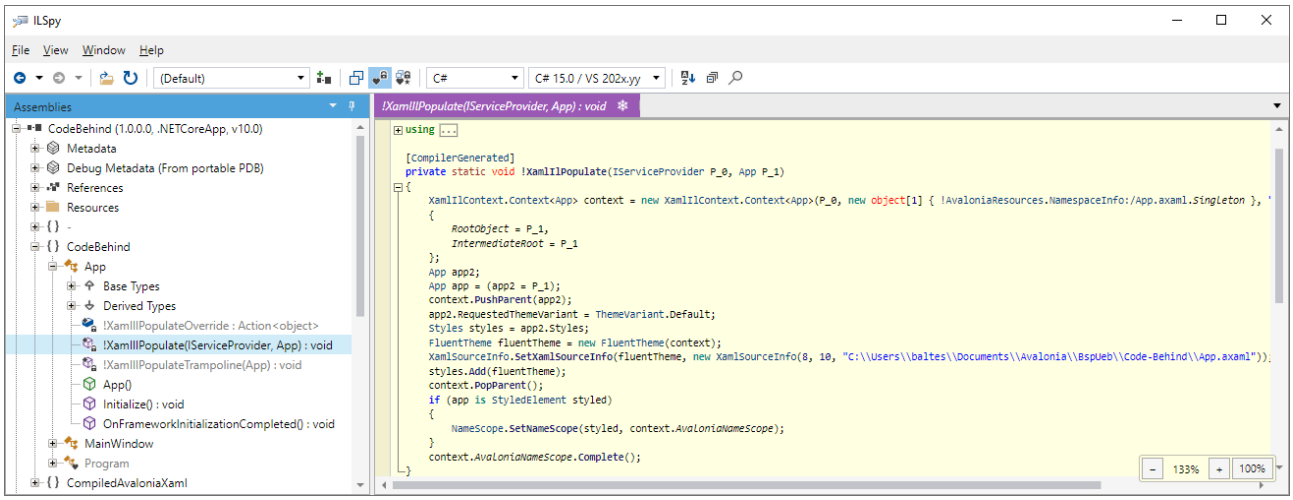
public partial class App : Application {
    public override void Initialize() {
        AvaloniaXamlLoader.Load(this);
    }
```

[1] Ein Visual Studio - Projekt mit dem Demonstrationsprogramm des Kapitels 6 ist hier zu finden:
...\BspUeb\Code-Behind

```
        public override void OnFrameworkInitializationCompleted() {
            if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
                desktop.MainWindow = new MainWindow();
            base.OnFrameworkInitializationCompleted();
        }
    }
```

Das Schlüsselwort **partial** signalisiert dem Compiler, dass zu der Klassendefinition noch (mindestens) ein Ergänzungsstück in einer anderen Datei gehört, wobei die partiellen Definitionen gleichberechtigt und voneinander abhängig ein Ganzes ergeben. Zu der aus WPF bekannten C# - Quellcodedatei **App.g.cs** mit dem von MSBuild erstellten Ergänzungsstück zur partiellen Anwendungsklassendefinition in der Code Behind - Datei existiert in Avalonia keine Entsprechung.

Um die in **App.axaml** deklarierten Instanzen und Eigenschaften kümmert sich die vom XamlX-Compiler von Avalonia im IL-Code des Assemblies **CodeBehind.dll** erstellte private `App`-Methode **!XamlIlPopulate**. Dieser in C# unzulässige Methodenname ist in der IL erlaubt. Der vom ILSpy angezeigte C# - Code ist durch Rückübersetzung aus dem IL-Code entstanden:



Die beiden in **App.axaml.cs** definierten Methoden der Klasse `App` kennen wir schon aus dem Abschnitt 4.7.1:

- In der Methode **Initialize()** dient ein Aufruf der statischen Methode **Load()** der Klasse **AvaloniaXamlLoader** dazu, per XAML deklarierte Objekte zu instanziieren und Eigenschaftswerte zuzuweisen. Das erledigt der vom XamlX-Compiler erzeugte IL-Code, sodass zur Laufzeit keine XAML-Interpretation erforderlich ist.
- Die Methode **OnFrameworkInitializationCompleted()** ist in einer Desktop-Anwendung u. a. dafür zuständig, das Fensterobjekt zu erstellen.

6.2 Definition der Hauptfensterklasse

In der Datei **MainWindow.axaml.cs** findet sich eine partielle Definition der Fensterklasse `MainWindow`, die von der Klasse **Window** im Namensraum **Avalonia.Controls** abstammt, z. B.:

```
using Avalonia.Controls;

namespace CodeBehind;

public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
    }
}
```

Initial enthält die Fensterklasse nur einen Konstruktor mit einem Aufruf der Methode **InitalizeComponent()**. Sie sorgt direkt bzw. indirekt für die folgenden Initialisierungen (siehe Abschnitt 6.2.2):

- Per XAML deklarierte Objekte werden instanziiert und mit Eigenschaftswerten versorgt.
- Instanzvariablen der Hauptfensterklasse erhalten Referenzen auf die per XAML deklarierten und benannten Steuerelemente.

Das Schlüsselwort **partial** signalisiert dem Compiler, dass zu der Klassendefinition noch (mindestens) ein Ergänzungsstück in einer anderen Datei gehört, wobei die partiellen Definitionen gleichberechtigt und voneinander abhängig ein Ganzes ergeben. Während zur Code Behind – Datei **App.axaml.cs** keine *C# - Quellcodedatei* als Ergänzungsstück vorhanden ist, existiert (allerdings nur im Hauptspeicher) die Datei **MainWindow.g.cs** mit einer von MSBuild und Avalonia erstellten partiellen Definition der Hauptfensterklasse (siehe Abschnitt 6.2.2.1). Hier ist u. a. die Implementation der Methode **InitializeComponent()** zu finden.

6.2.1 Beiträge des Entwicklers

Wir deklarieren im XAML-Code des Hauptfensters als Inhalt einen **StackPanel**-Container (siehe Abschnitt 10.4) mit einem **Button**- und einem **Label**-Steuerelement:

```
<StackPanel>
  <Button Name="button" Content="Click" Margin="0, 40, 0, 0" VerticalAlignment="Top"
          HorizontalAlignment="Center" Click="Button_Click" />
  <Label Name="label" Margin="20, 60, 0, 0" VerticalAlignment="Top"
         HorizontalAlignment="Center" />
</StackPanel>
```

Dem **Click**-Ereignis des **Button**-Steuerelements weisen wir die noch nicht existente Behandlungsmethode `Button_Click()` zu, was eine durch Unterschlängeln ausgedrückte Kritik veranlasst. Zur Lösung des Problems verfahren wir wie im Abschnitt 2.4.4.8:

- Befindet sich die Einfügemarke im rot unterschlängelten Bereich, dann erscheint eine Glühbirne am linken Rand.
- Nach einem Mausklick auf die Glühbirne erscheint der Vorschlag **Generate Button_Click**. Befindet sich der Mauszeiger über diesem Vorschlag, dann erscheint eine Vorschau auf die Ausführung, die ein Klick auf den Vorschlag veranlasst. Dann legt das Avalonia-Plugin in der Code Behind - Datei **MainWindow.axaml.cs** die Behandlungsmethode mit einem leeren Rumpf an.

Wir fügen in die Behandlungsmethode eine Anweisung zum Ändern der **Label**-Beschriftung ein und erhalten erneut eine rote Unterschlängelung, weil das Steuerelement in der Hauptfensterklasse noch nicht bekannt ist:

```
private void Button_Click(object? sender, Avalonia.Interactivity.RoutedEventArgs e) {
    label.Content = "Clickereignis behandelt";
}
```

Eine Neuerstellung des Projektmappe über den Menübefehl

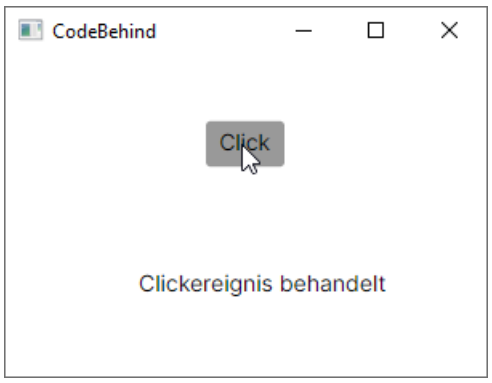
Erstellen > Projektmappe neu erstellen

löst das Problem.

Das Hauptfenster erhält im XAML-Code noch passende Maße,

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="CodeBehind.MainWindow"
        Title="CodeBehind" Width="300" Height="200">
    . . .
</Window>
```

sodass ein bescheidenes Programm resultiert, das zur Demonstration der Code-Behind – Funktionalität genügt:



In einer kleinen Avalonia-Anwendung enthalten die Code-Behind - Dateien zu den Fensterklassen die Präsentationslogik (z. B. Verwaltung der präsentierten Daten und des GUI-Zustands) *und* die Anwendungslogik (z. B. Algorithmen). Demgegenüber existiert in einer professionellen Anwendung zu jedem Fenster eine *Ansichtsmodellklasse*, die sich um die Präsentationslogik kümmert und von Steuerelementen initiierte Kommandos in Kooperation mit *Modellklassen* ausführt, die für die Anwendungs- bzw. Geschäftslogik zuständig sind (siehe Kapitel 12 über die MVVM-Anwendungsarchitektur). Als Ergebnis dieser Trennung der Zuständigkeiten sind die Code-Behind – Dateien einer professionellen Anwendung fast leer.

6.2.2 Ergänzungen durch das Erstellungssystem

In diesem Abschnitt ist zu erfahren, wo sich die von MSBuild bzw. Avalonia erstellten Ergänzungen zur Code Behind – Datei **MainWindow.axaml.cs** befinden:

- Wir lokalisieren eine partielle Klassendefinition in der „versteckten“ C# - Datei **MainWindow.g.cs**
- und werden bei der Suche nach der Registrierung der **Click**-Ereignismethode im IL-Code des Assemblies **CodeBehind.dll** fündig.

Wer weniger am technischen Hintergrund interessiert ist, kann das Kapitel 6 hier ohne Risiko für die weitere Lektüre verlassen.

6.2.2.1 Generierter C# - Quellcode

Aufgrund der XAML-Datei zur Hauptfensterklasse `MainWindow` produziert das Erstellungssystem eine C# - Quellcodedatei mit einer partiellen Klassendefinition, die aber nicht auf einem Festspeicher (SSD oder Festplatte) landet, sondern im Hauptspeicher. Um diesen C# - Quellcode zu sehen, kann man so vorgehen:

- in der Datei **MainWindow.axaml.cs** die Einfügemarke auf die Methode **InitalizeComponent()** setzen
- mit der Taste **F12** die Datei mit der Implementation dieser Methode öffnen.

Wir erhalten eine partielle Klassendefinition mit der Deklaration der Instanzvariablen zu den beiden Steuerelementen:

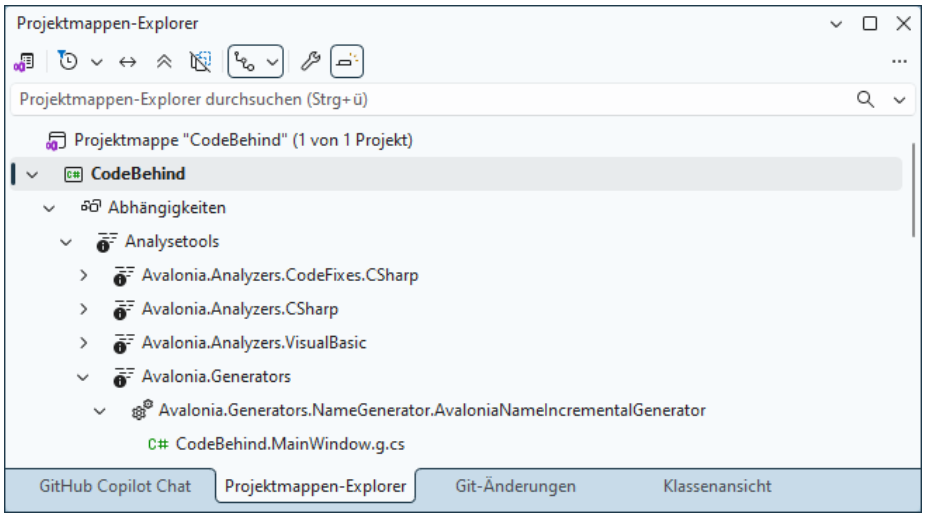
```
namespace CodeBehind {
  partial class MainWindow {
    [global::System.CodeDom.Compiler.GeneratedCode(
      "Avalonia.Generators.NameGenerator.InitializeComponentCodeGenerator", "12.1.1.0")]
    internal global::Avalonia.Controls.Button button;
    [global::System.CodeDom.Compiler.GeneratedCode(
      "Avalonia.Generators.NameGenerator.InitializeComponentCodeGenerator", "12.1.1.0")]
    internal global::Avalonia.Controls.Label label;

    [global::System.CodeDom.Compiler.GeneratedCode(
      "Avalonia.Generators.NameGenerator.InitializeComponentCodeGenerator", "12.1.1.0")]
    [global::System.Diagnostics.CodeAnalysis.ExcludeFromCodeCoverage]
    public void InitializeComponent(bool loadXaml = true) {
      if (loadXaml) {
        AvaloniaXamlLoader.Load(this);

      var __thisNameScope__ = this.FindNameScope();
      button = __thisNameScope__?.Find<global::Avalonia.Controls.Button>("button");
      label = __thisNameScope__?.Find<global::Avalonia.Controls.Label>("label");
    }
  }
}
```

In der ebenfalls enthaltenen Implementation der Methode **InitializeComponent()** befindet sich ein Aufruf der statischen Methode **Load()** der Klasse **AvaloniaXamlLoader**. Der von XamlX-Compiler zu dieser Methode erzeugte IL-Code führt die per XAML angeforderten Objektkreationen und Wertzuweisungen aus (siehe Abschnitt 6.2.2.2). Außerdem erhalten in **InitializeComponent()** die Instanzvariablen `button` und `label` jeweils eine Referenz auf das zugehörige Steuerelement. Wo Avalonia die **Click**-Ereignisbehandlungsmethode zum **Button**-Steuerelement registriert, ist im Abschnitt 6.2.2.2 zu sehen.

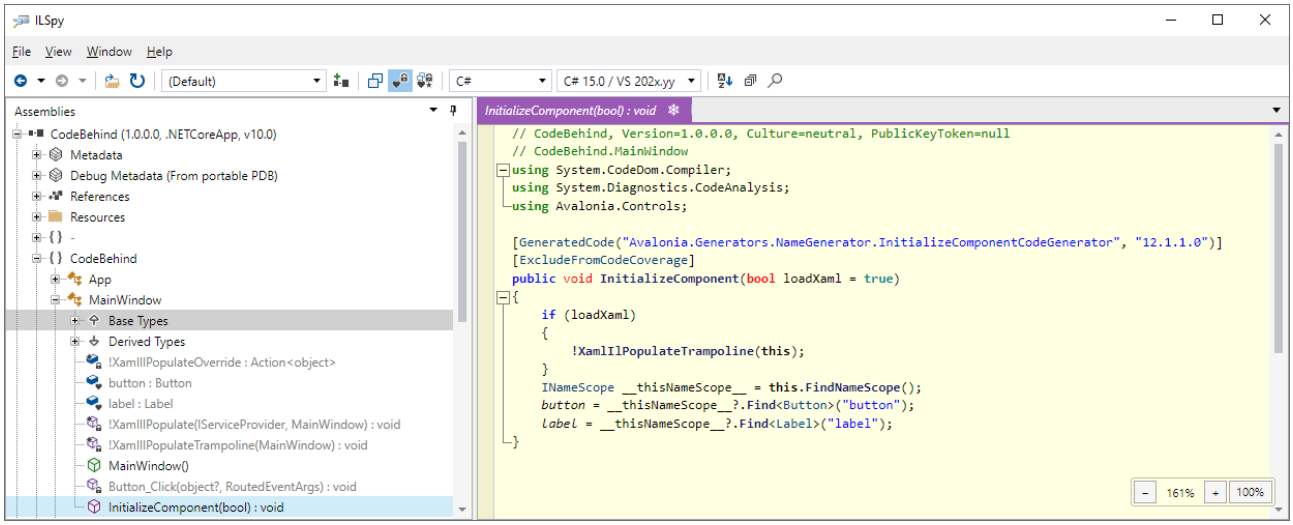
Für den „versteckten“ `MainWindow`-Quellcode ist ein Codegenerator verantwortlich, den der **Projektmappen-Explorer** im Projektknoten **Abhängigkeiten > Analysetools** auflistet:



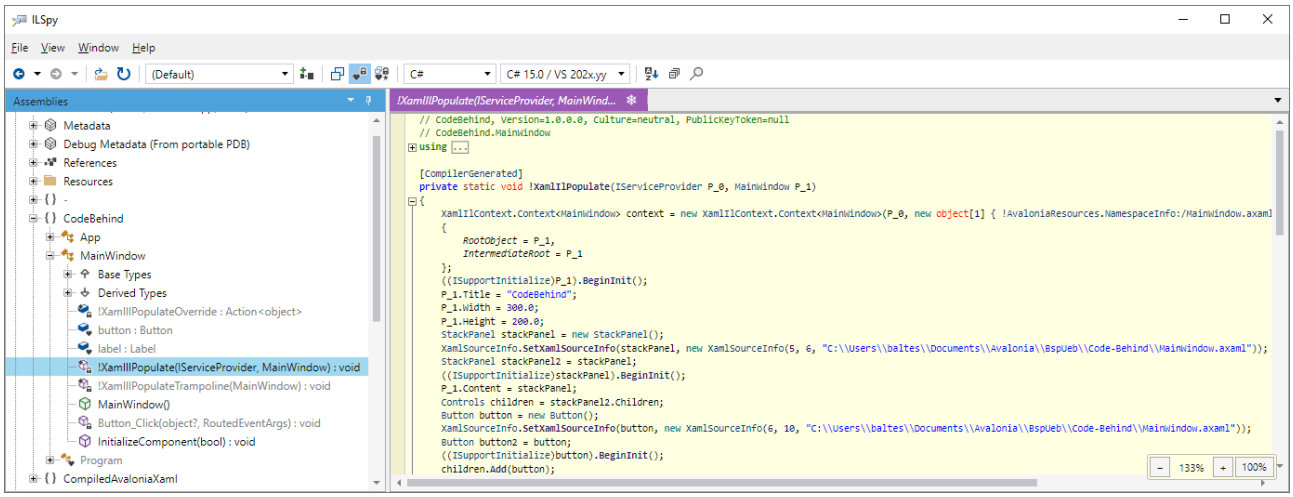
Hier erscheint auch der Name der versteckten Datei: **MainWindow.g.cs**

6.2.2.2 Vom XamlX-Compiler erstellter IL-Code

Was die von **InitializeComponent()** aufgerufene Methode **Load()** der Klasse **AvaloniaXamlLoader** tut, lässt sich mit ILSpy trotz des unerwarteten IL-Namens **!XamlIlPopulateTrampoline()** der Methode aufklären:[1]



Die IL-Methode **!XamlIlPopulateTrampoline()** ruft die Methode **!XamlIlPopulate()** auf, wo die per XAML angeforderten Instanzkreationen und Wertzuweisungen vorgenommen werden:



Auch die Registrierung der **Click**-Behandlungsmethode zum **Button**-Objekt des Demonstrationsprogramms findet im IL-Code der vom XamlX-Compiler generierten Lademethode **!XamlIlPopulate()** statt. Dort wird ein Delegatenobjekt auf Basis der Methode `Button_Click()` erstellt und über die Methode **AddHandler()** beim Avalonia-Ereignissystem registriert:



[1] Der Methodenname **!XamlIlPopulateTrampoline** ist in C# unzulässig, in der IL aber erlaubt. Der vom ILSpy angezeigte C# - Code ist durch Rückübersetzung aus dem IL-Code entstanden.

Das **Click**-Ereignis der Klasse **Button** gehört zu den sogenannten *Routingereignissen*, und mit dieser Ergänzung zu den CLR – Ereignissen (*Common Language Runtime*) beschäftigt sich das nächste Kapitel.

7 Routingereignisse

Bei der GUI-Programmierung spielt die Kommunikation über Ereignisse eine zentrale Rolle. Steuerelemente erfahren durch die Vermittlung von Betriebs- und Laufzeitsystem von sie betreffenden Eingaben (z. B. per Maus oder Tastatur) und lösen die von ihnen registrierten Ereignisse aus, was zum Aufruf von angemeldeten Behandlungsmethoden führt. Manchmal kombinieren Steuerelemente elementare Ereignisse (z. B. **PointerPressed** und **PointerReleased**) zu höherwertigen Ereignissen (z. B. **Click**).

Avalonia besitzt mit den sogenannten *Routingereignissen* eine flexible Kommunikationstechnik, die eine starke Verwandtschaft mit dem WPF-Ereignissystem aufweist. Im Vergleich zu den gewöhnlichen CLR-Ereignissen (*Common Language Runtime*) bestehen bei den Routingereignissen zwei Besonderheiten:

- Funktionsweise
  Man kann eine Behandlungsmethode (z. B. zum **Click**-Ereignis eines **Button**-Steuerelements) nicht nur bei der Ereignisquelle anmelden, sondern auch bei den übergeordneten GUI-Elementen bis hinauf zum Fensterobjekt. Tritt ein Ereignis auf, dann kommen nicht nur bei der Ereignisquelle angemeldete Behandlungsmethoden zum Zug, sondern nacheinander auch bei übergeordneten Elementen angemeldete Behandlungsmethoden. Dabei kann die Weiterleitung von der Quelle bis zur Fensterebene erfolgen oder auch umgekehrt.
- Implementierung
  Ein Routingereignis basiert auf einem Objekt der Klasse **RoutedEvent**, und für die Verarbeitung ist das Avalonia-Ereignissystem zuständig.

In einer Avalonia-Anwendung sind keinesfalls alle Ereignisse vom Routingtyp. Wo keine Weiterleitung und keine sonstige Funktion des Avalonia-Ereignissystems erforderlich ist, kommen die einfacheren CLR-Ereignisse zum Einsatz.

Wenn man für ein Routingereignis nur bei der *Ereignisquelle* eine Behandlungsmethode anmeldet, dann bestehen keine relevanten Unterschiede gegenüber einem CLR-Ereignis. Wir haben z. B. für den im Abschnitt 2.4 entwickelten Währungskonverter in der Code-Behind - Datei zur Hauptfensterklasse `MainWindow` eine Behandlungsmethode zum **Click**-Ereignis des **Button**-Steuerelements erstellt

```
private void Button_Click(object? sender, RoutedEventArgs e) {
    . . .
}
```

und diese Behandlungsmethode per XAML beim **Button** angemeldet:

```
<Button Content="Konvertieren" . . . Click="Button_Click" />
```

Um im Quellcode das Anmelden von Behandlungsmethoden zu Routingereignissen bei der Ereignisquelle mit der vertrauten C# - Syntax für CLR-Ereignisse zu ermöglichen, bieten die meisten Routingereignisse ein CLR-Ereignis als Wrapper an (dt.: *Verpackung*, siehe Abschnitt 7.5), z. B.:

```
konvertieren.Click += Button_Click;
```

Um eine Behandlungsmethode zu einem Routingereignis bei übergeordneten Elementen in der GUI-Hierarchie anzumelden, eignen sich:

- Attribute im XAML-Code
- die **Interactive**-Methode **AddHandler()** im Quellcode (siehe z. B. Abschnitt 7.3).

7.1 Routingstrategien

Bei einem Routingereignis sind drei Weiterleitungsstrategien möglich:

- **Tunnel**-Weiterleitung
  Auf das von einem GUI-Element (z. B. einem **Image**-Objekt als Inhaltsbestandteil einer **Button**-Oberfläche) ausgelöste Ereignis hat das oberste Element in der Hierarchie (also das **Window**-Objekt) den ersten Zugriff, d. h. die dort vereinbarten Behandlungsmethoden werden ggf. aufgerufen. Anschließend durchläuft das Ereignis die hierarchisch *absteigende* Route bis zur Ereignisquelle. Übergeordnete Elemente sehen das Ereignis *vor* dem unmittelbar betroffenen Element und können es abfangen.

  Im Unterschied zu WPF kennt Avalonia keine dezidierten Tunnelereignisse mit dem Namenspräfix **Preview**. Stattdessen sind z. B. in der Klasse **InputElement** für das Ereignis **PointerPressed** die **Tunnel**- *und* die **Bubble**-Route vorgesehen. Hier ist die **PointerPressed**-Registrierung beim Ereignissystem zu sehen:[1]

```
public static readonly RoutedEvent<PointerPressedEventArgs> PointerPressedEvent =
            RoutedEvent.Register<InputElement, PointerPressedEventArgs>(
                nameof(PointerPressed),
                RoutingStrategies.Tunnel | RoutingStrategies.Bubble);
```

  Bei der Zuweisung einer Behandlungsmethode an ein **Interactive**-Objekt (z. B. ein Steuerelement) per **AddHandler()** legt man fest, ob die Methode auf der absteigenden oder auf der aufsteigenden Ereignispassage aufzurufen ist:

```
control.AddHandler(PointerPressedEvent, ControlPointerPressedTunnel,
                   RoutingStrategies.Tunnel);
```

- **Bubble**-Weiterleitung
  Zuerst treten die der Ereignisquelle zugewiesenen Behandlungsmethoden in Aktion. Danach kommen sukzessive die übergeordneten Elementen in der Hierarchie (bis hinauf zum **Window**-Objekt) zugewiesenen Behandlungsmethoden an die Reihe. Wie das von der Klasse **Button** registrierte Ereignis **Click** zeigt, kennt Avalonia reine **Bubble**-Ereignisse:[2]

```
public static readonly RoutedEvent<RoutedEventArgs> ClickEvent =
    RoutedEvent.Register<Button, RoutedEventArgs>(
        nameof(Click), RoutingStrategies.Bubble);
```

  Diese ausschließlich aufperlende Bewegungsart ist bei den Avalonia-Routingereignissen am häufigsten anzutreffen.

- **Direktereignisse**
  Es kommen nur der Ereignisquelle zugewiesene Behandlungsmethoden zum Einsatz, was der Verarbeitung von normalen CLR-Ereignissen entspricht. Allerdings sind einige Techniken des Avalonia-Ereignissystems realisiert, z. B. klassenbezogene Behandlungsmethoden, die für alle Objekte ihrer Klasse zuständig sind und *vor* den Instanz-Behandlungsmethoden ablaufen (siehe Abschnitt 7.4).[3]

  Ein Beispiel ist das von der Klasse **InputElement** registrierte Ereignis **PointerEntered**:

```
public static readonly RoutedEvent<PointerEventArgs> PointerEnteredEvent =
    RoutedEvent.Register<InputElement, PointerEventArgs>(
        nameof(PointerEntered), RoutingStrategies.Direct);
```

Ein Routingereignis mit **Tunnel**- und/oder **Bubble**-Weiterleitung ist nach der Ausführung einer Behandlungsmethode *nicht* erledigt, sondern wandert entlang der Hierarchie weiter zu potenziellen Interessenten, bis es seine Endstation erreicht hat. Wenn eine Behandlungsmethode die Eigenschaft **Handled** des übergebenen Beschreibungsobjekts auf den Wert **true** setzt, dann kommt eine auf

---

1 https://github.com/AvaloniaUI/Avalonia/blob/main/src/Avalonia.Base/Input/InputElement.cs

2 https://github.com/AvaloniaUI/Avalonia/blob/main/src/Avalonia.Controls/Button.cs

3 https://github.com/AvaloniaUI/Avalonia/blob/main/src/Avalonia.Base/Input/InputElement.cs

typische Weise zugewiesene Behandlungsmethode *nicht* mehr ins Spiel. Aber auch ein als behandelt markiertes Ereignis setzt seine Wanderung fort zur potenziellen Verwendung durch Behandlungsmethoden, die auf besondere Weise über die **Interactive**-Methode **AddHandler()** zugewiesen wurden (mit dem Wert **true** für den booleschen Parameter *handledEventsToo*).

Ist ein Ereignis wie z. B. **PointerPressed** für die **Tunnel**- und für die **Bubble**-Route registriert (siehe oben), dann kommt es zum folgenden Ablauf:

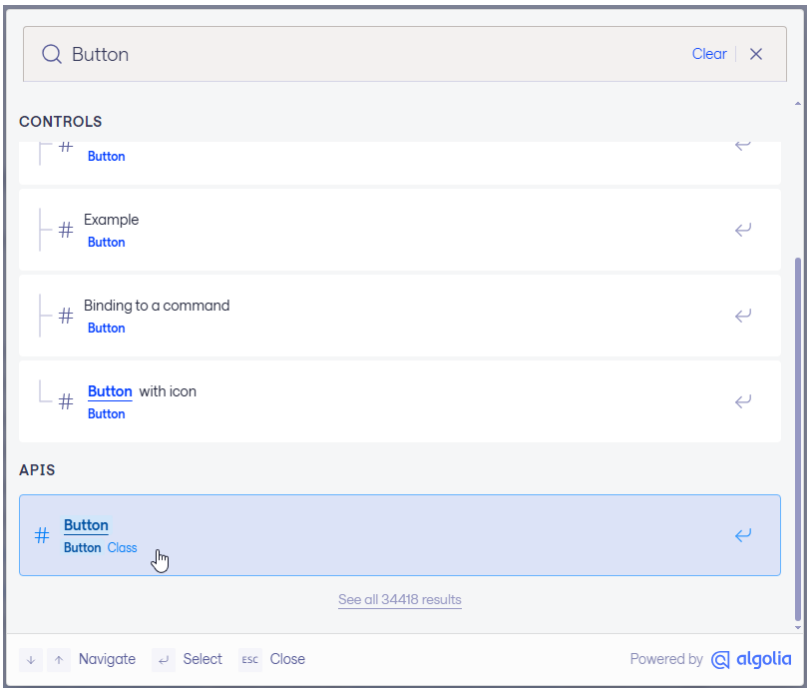
- Zunächst wird die **Tunnel**-Route absolviert. Auf dem Weg von der Wurzel der Elementenhierarchie (vom Fensterobjekt) bis zur Ereignisquelle kommt es zum Aufruf zugewiesener Behandlungsmethoden.
- Ist die **Tunnel**-Route beendet, dann startet die **Bubble**-Route von der Ereignisquelle hinauf zum Fensterobjekt.
- Ein Ereignis auf der **Tunnel**-Route als behandelt zu markieren, wirkt sich auch auf die **Bubble**-Route aus, weil beide Routen dasselbe Ereignisbeschreibungsobjekt verwenden. Die **Tunnel**-Route dient meist dazu, eine Ereignisbehandlung zu blockieren oder vorzubereiten.

Das Ereignis **Click** der Klasse **Button** absolviert hingegen nur die **Bubble**-Route.

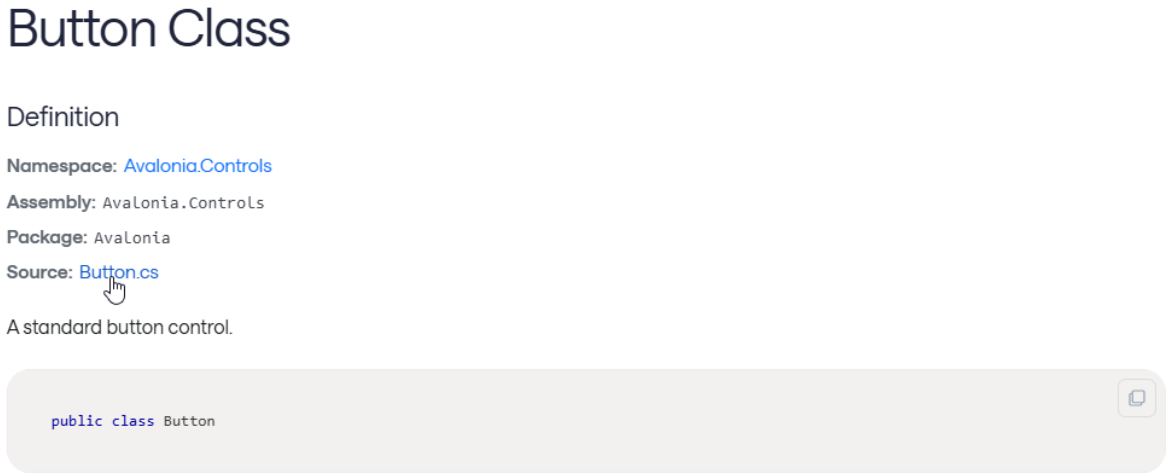
7.1.1 Registrierte Ereignisrouten ermitteln

Gelegentlich möchte man wissen, für welche Routen ein Ereignis registriert ist. Um z. B. für das **Click**-Ereignis der Klasse **Button** die beim Registrieren festgelegten Routingstrategien in Erfahrung zu bringen, geht man so vor:

- API-Webseite https://docs.avaloniaui.net/api/ aufsuchen
- Nach **Button** suchen und die API-Dokumentation zur Klasse wählen:



- Auf der API-Dokumentationsseite den Quellcode in der Datei **Button.cs** aufrufen



- Auf der GitHub-Seite mit dem Quellcode nach **ClickEvent** suchen und dem **Register()** – Methodenaufruf die Routingstrategie(en) entnehmen:[1]

```
public static readonly RoutedEvent<RoutedEventArgs> ClickEvent =
    RoutedEvent.Register<Button, RoutedEventArgs>(nameof(Click),
    RoutingStrategies.Bubble);
```

7.1.2 Delegatensignaturen der Behandlungsmethoden

Wenn zu einem Routingereignis (wie z. B. zu **Click**) über die allen Routingereignissen gemeinsamen Informationen hinaus keine spezifischen Daten existieren, dann genügt für eine Behandlungsmethode im zu erfüllenden Delegatentyp die allgemeine Klasse **RoutedEventArgs**:

public delegate void EventHandler<RoutedEventArgs>(Object? *sender*, **RoutedEventArgs** *e*)

Die Methode erhält dann beim Aufruf die folgenden Daten:

- **Object** *sender*
  Dieser Parameter zeigt auf das Objekt, ...
  - bei dem das Ereignis auf seiner Route gerade angekommen ist,
  - bzw. dem die Behandlungsmethode zugewiesen wurde.

  Das ist i. A. *nicht* die Ereignisquelle.
- **RoutedEventArgs** *e*[2]
  Das Ereignisbeschreibungsobjekt besitzt u. a. die folgenden Eigenschaften:
  - **Source**
    Man erfährt die Ereignisquelle.
  - **Handled**
    Über diese boolesche Eigenschaft kann man den Ereignisbehandlungsstatus einsehen oder setzen.

---

1 https://github.com/AvaloniaUI/Avalonia/blob/main/src/Avalonia.Controls/Button.cs

2 https://docs.avaloniaui.net/api/avalonia/interactivity/routedeventargs

- **Route**
  Über diese Eigenschaft vom Typ **RoutingStrategies** kann man die Routingstrategie einsehen oder setzen. Eine Änderung der **Route**-Eigenschaft des **RoutedEventArgs**-Objekts durch eine Behandlungsmethode ...
  - hat keinen Effekt auf die Bewegungsrichtung und die noch anzusteuernden Stationen,
  - beeinflusst aber die auf dem weiteren Weg ausgeführten Behandlungsmethoden, die zur geänderten **Route**-Einstellung passen müssen.
- **RoutedEvent**
  Über diese Eigenschaft vom Typ **RoutedEvent** kann man das zum **RoutedEventArgs**-Objekt gehörende Routingereignis einsehen oder setzen. Eine Änderung hat keinen Effekt auf die bereits feststehende Reiseplanung und ist während einer Ereignispassage kaum sinnvoll.

Ein Beispiel für ein Routingereignis *mit* spezifischen Ereignisinformationen ist **PointerPressed**. Eine Behandlungsmethode für diese Ereignisse muss den folgenden Delegatentyp erfüllen:

public delegate void EventHandler<PointerPressedEventArgs>(
Object? *sender*, **PointerPressedEventArgs** *e*)

Während sich beim **Object**-Parameter im Vergleich zur obigen Beschreibung nichts ändert, besitzt das Ereignisbeschreibungsobjekt vom Typ **PointerPressedEventArgs** zusätzliche Eigenschaften, z. B.:

- **Pointer**
  Diese Eigenschaft zeigt auf einen die Schnittstelle **IPointer** implementierenden Typ, sodass u. a. die Eigenschaft **Type** vom Enumerationstyp **PointerType** mit den Werten **Mouse**, **Pen** und **Touch** vorhanden ist.
- **Properties**
  Die Eigenschaft **Properties** vom Typ **PointerPointProperties** zeigt auf eine Struktur mit diversen Eigenschaften, z. B. **IsLeftButtonPressed** (beim Zeigertyp **Mouse**), **Pressure** (die von einem Stift auf die Oberfläche ausgeübte Kraft).

7.2 Einsatzfälle für Routingereignisse

Aufperlende Ereignisse (auf der **Bubble**-Route) bewähren sich z. B. in den folgenden Situationen:

- Gemeinsame Ereignisbehandlung für mehrere Steuerelemente
  Befinden sich in einem Layoutcontainer mehrere **Button**-Objekte, und sollen deren **Click**-Ereignisse durch eine gemeinsame Methode behandelt werden, dann genügt eine einmalige Zuweisung dieser Methode an den Layoutcontainer. Wäre **Click** *kein* aufperlendes Ereignis, dann müsste man jedem Schalter die Methode zuweisen.
- Höherwertige Ereignisse für Steuerelemente mit mehreren Bestandteilen
  Ein **Button**-Objekt kann Inhalte aufnehmen, z. B. ein **Image**- und ein **TextBlock**-Objekt:

```
<Button Name="button" . . . Click="Handle_Click">
  <StackPanel Orientation="Horizontal">
    <Image Source="avares://Beteiligung/Assets/machmit.png" />
    <TextBlock Text="Einverstanden" />
  </StackPanel>
</Button>
```

  Die ein **Click**-Ereignis konstituierenden elementaren Ereignisse **PointerPressed** und **PointerReleased** können jeweils bei einem anderen Inhaltsbestandteil auftreten, z. B. ...
  - beim **Image**-Objekt das Routingereignis **PointerPressed**
  - und wegen einer Mausbewegung beim **TextBlock**-Objekt das Routingereignis **PointerReleased**.

Weil das **Button**-Objekt von beiden Ereignissen erfährt, kann es den **Click** erkennen und als neues, höherwertiges Routingereignis feuern.

Um **Tunnel**-Ereignisrouten kümmern sich Anwendungsprogramme seltener als um als **Bubble**-Routen. Manchmal ist es aber für ein elterliches GUI-Element relevant, *vor* seinen Kindern von einem Ereignis zu erfahren. Das macht es möglich, ...

- ein Ereignis zu blockieren
  Dazu erhält in der Behandlungsmethode zur **Tunnel**-Route die **Handled**-Eigenschaft des übergebenen Beschreibungsobjekts den Wert **true**. Soll z. B. ein **TextBox**-Steuerelement nur Ziffern erhalten, dann kann man ...
  - für das Ereignis **TextInput** auf der **Tunnel**-Route eine Validierung vornehmen
  - und beim Auftreten irregulärer Zeichen die **Bubble**-Route verhindern.
- oder eine Ereignisbehandlung vorzubereiten.

Wir beschränken uns im Kapitel 7 auf generelle Informationen über das Avalonia-Ereignissystem. Über spezielle Ereignisgruppen (z. B. Maus- und Tastaturereignisse) und die aus Zeigerereignissen komponierten **Gesten** (z. B. Antippen von beliebigen Elementen, Zoomen mit zwei Fingern, Scrollen zum Verschieben des Inhalts) informiert die Online-Dokumentation.[1]

7.3 Beobachtungsstudie

Um die Ereignisrouten beobachten zu können, erstellen wir eine Anwendung mit der folgenden Hierarchie von GUI-Elementen:[2]

Ein Wurzelelement aus der von **Window** abstammenden Klasse `MainWindow`,
darin ein Layoutcontainer aus der Klasse **Panel** (siehe Abschnitt 10.1),
darin u. a. ein Objekt der Klasse **Button** (siehe Abschnitt 11.3.1),
darin ein Layoutcontainer aus der Klasse **StackPanel** (siehe Abschnitt 10.3),
darin ein Objekt der Klasse **Image**
und ein Objekt der Klasse **TextBlock**

Im **Panel**-Container befindet sich außerdem ein **ListBox**-Objekt, das zum Protokollieren von Ereignissen dient (siehe Abschnitt 11.3.5).

Für die folgenden Routingereignisse sind allen betroffenen GUI-Elementen Behandlungsmethoden zugewiesen (siehe unten):

- **PointerPressed** (Klasse **InputElement**)
- **Click** (Klasse **Button**)

Zum Ereignis **PointerPressed** werden Behandlungsmethoden für die **Tunnel**- und für die **Bubble**-Route vereinbart. Während die Vereinbarung einer **Bubble**-Behandlungsmethode (bei **PointerPressed** und bei **Click**) per XAML erfolgen kann,

```
<Panel Name="panel" PointerPressed="Handle_PointerPressedBubble"
       Button.Click="Handle_Click">
```

ist für die Vereinbarung einer **Tunnel**-Behandlungsmethode ein Aufruf der **Interactive**-Methode **AddHandler()** erforderlich, z. B.:

```
panel.AddHandler(PointerPressedEvent, Handle_PointerPressedTunnel,
                 RoutingStrategies.Tunnel);
```

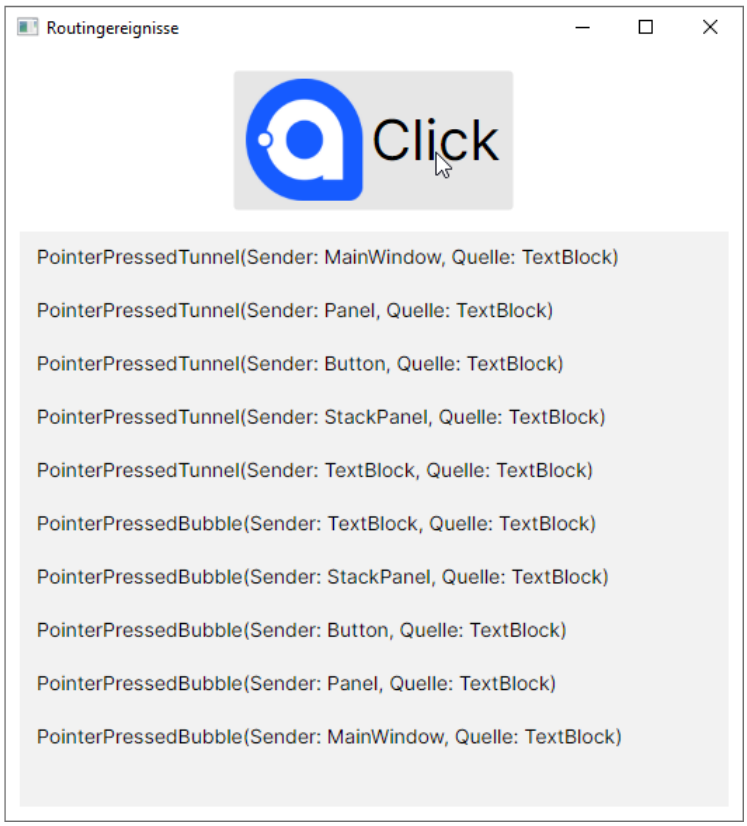
---

[1] https://docs.avaloniaui.net/docs/events/input-events, https://docs.avaloniaui.net/docs/input-interaction/gestures

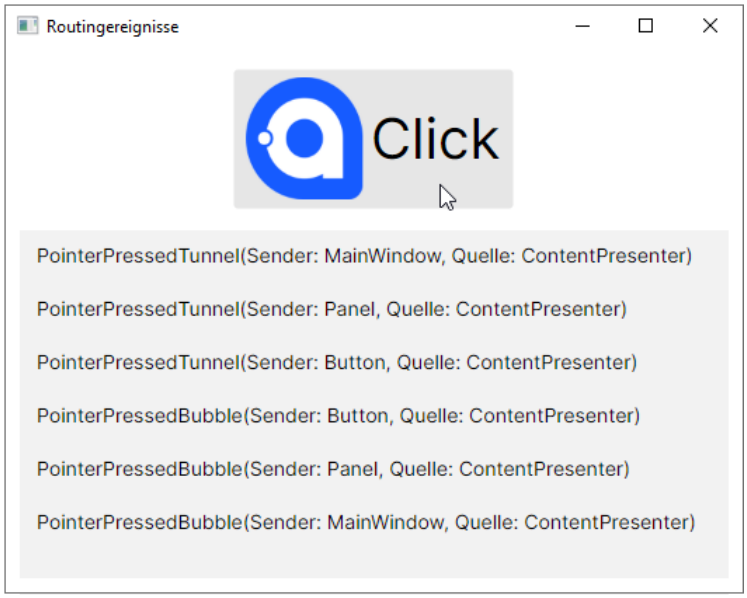
[2] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
...\BspUeb\Routingereignisse\Beobachtungsstudie

Warum man dem **Window**-Objekt eine Behandlungsmethode zum **Button**-Ereignis **Click** (unter Verwendung des vollqualifizierten Namens **Button.Click**) zuweisen darf, ist im Abschnitt 7.6 zu erfahren.

Beim folgenden Einsatz ist das **TextBlock**-Objekt wegen Drückens der ***rechten*** Maustaste zur Quelle für das Ereignis **PointerPressed** geworden. Das Ereignis hat gemäß Abschnitt 7.1 während seiner **Tunnel**-Route bei allen GUI-Elementen auf dem Weg von der Wurzel bis zur Quelle zum Aufruf der zugewiesenen Behandlungsmethode geführt. Danach hat das Ereignis seine **Bubble**-Route begonnen und den umgekehrten Weg von der Quelle bis zur Wurzel zurückgelegt.

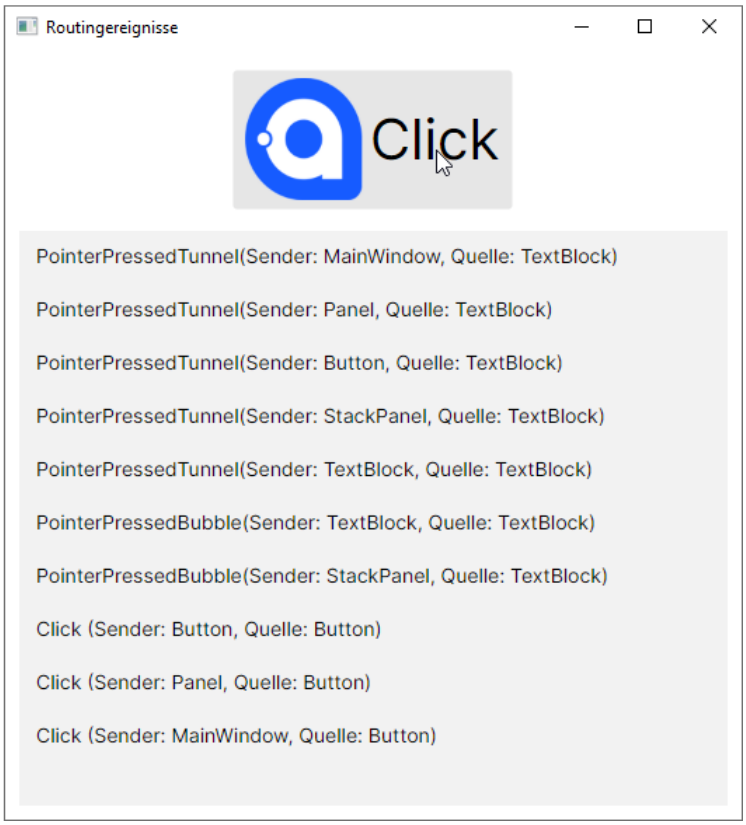


Wenn man ungenau zielt, dann protokolliert das Programm die unerwartete Ereignisquelle **ContentPresenter**:



Weil die Diskussion dieses Phänomens momentan didaktisch schlecht passt, verschieben wir sie in den Anhang B.

Nach einem ***linken*** Mausklick (Maustaste drücken und loslassen) auf das **TextBlock**-Objekt ist eine andere Ereignishistorie zu beobachten:



Zu Beginn zeigt sich kein Unterschied zum Drücken der rechten Maustaste: Das Ereignis **PointerPressed** mit dem **TextBlock**-Objekt als Quelle führt mit der Wurzel beginnend auf allen Hierarchieebenen zum Aufruf der zugewiesenen **Tunnel**-Behandlungsmethode. Danach startet das Ereignis seine **Bubble**-Route an der Quelle, die allerdings beim **StackPanel**-Objekt endet. Das **Button**-Objekt verlängert nicht die Serie von **PointerPressed** - Bubble-Ereignisbehandlungen, sondern feuert stattdessen das Ereignis **Click**, das seinen aufperlenden Weg nach oben bis zur Wurzel der GUI-Elementenhierarchie nimmt.

Das Programm verwendet für jede von den drei möglichen Kombinationen aus einem Routingereignis und einer Bewegungsrichtung

- **PointerPressed**-Ereignis auf der **Tunnel**-Route
- **PointerPressed**-Ereignis auf der **Bubble**-Route
- **Click**-Ereignis auf der **Bubble**-Route

eine Behandlungsmethode, die *allen* GUI-Elementen in der beschriebenen Verschachtelung zugewiesen ist. Neben dem Ereignistyp protokollieren die drei Behandlungsmethoden auch ...

- den Aufrufer der Methode
  Sein Laufzeitdatentyp lässt sich über das erste Parameterobjekt (*sender*) feststellen.
- die Ereignisquelle
  Ihr Laufzeitdatentyp lässt sich über die **Source**-Eigenschaft des zweiten Parameterobjekts (Typ **PointerPressedEventArgs** bzw. **RoutedEventArgs**) feststellen.

Die Behandlungsmethoden befinden sich in der Code-Behind - Datei mit der partiellen Definition der Fensterklasse (vgl. Abschnitt 6.2):

```
using Avalonia.Controls;
using Avalonia.Input;
using Avalonia.Interactivity;

namespace Beobachtungsstudie;

public partial class MainWindow : Window {
  public MainWindow() {
    InitializeComponent();
    this.AddHandler(PointerPressedEvent, Handle_PointerPressedTunnel,
                    RoutingStrategies.Tunnel);
    panel.AddHandler(PointerPressedEvent, Handle_PointerPressedTunnel,
                    RoutingStrategies.Tunnel);
    button.AddHandler(PointerPressedEvent, Handle_PointerPressedTunnel,
                     RoutingStrategies.Tunnel);
    stackPanel.AddHandler(PointerPressedEvent, Handle_PointerPressedTunnel,
                     RoutingStrategies.Tunnel);
    image.AddHandler(PointerPressedEvent, Handle_PointerPressedTunnel,
                    RoutingStrategies.Tunnel);
    textBlock.AddHandler(PointerPressedEvent, Handle_PointerPressedTunnel,
                    RoutingStrategies.Tunnel);
  }

  // Tunnel-Route PointerPressed
  private void Handle_PointerPressedTunnel(object? sender, PointerPressedEventArgs e) {
    listBox.Items.Add($"PointerPressedTunnel(Sender: {sender.GetType().Name}, " +
                     $"Quelle: {e.Source.GetType().Name})");
  }
  // Bubble-Route PointerPressed
  private void Handle_PointerPressedBubble(object? sender, PointerPressedEventArgs e) {
    listBox.Items.Add($"PointerPressedBubble(Sender: {sender.GetType().Name}, " +
                     $"Quelle: {e.Source.GetType().Name})");
  }
  // Bubble-Route Click
  private void Handle_Click(object? sender, RoutedEventArgs e) {
    listBox.Items.Add($"Click (Sender: {sender.GetType().Name}, " +
                     $"Quelle: {e.Source.GetType().Name})");
  }
}
```

In den Namen der Ereignisbehandlungsmethoden gibt der erste Bestandteil nicht wie sonst üblich den Absender des Ereignisses an, weil die Methoden für verschiedene Absender zuständig sind.

Zur Anzeige durch das **ListBox**-Steuerelement fügen die Behandlungsmethoden Protokolleinträge in das per **Items**-Eigenschaft ansprechbare **ItemCollection**-Mitgliedsobjekt ein, das die Methode **Add()** beherrscht (vgl. Abschnitt 11.3.5.1).

Die Zuweisung der **Bubble**-Ereignisbehandlungsmethoden fällt per XAML-Code leichter als per C# - Code:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="Beobachtungsstudie.MainWindow"
        Title="Routingereignisse" Height="550" Width="525"
        PointerPressed="Handle_PointerPressedBubble"
        Button.Click="Handle_Click">
```

```
    <Panel Name="panel" PointerPressed="Handle_PointerPressedBubble"
           Button.Click="Handle_Click">
      <Button Name="button" Margin="0,15" Height="100" BorderBrush="Black"
              Background="White" VerticalAlignment="Top" HorizontalAlignment="Center"
              PointerPressed="Handle_PointerPressedBubble" Click="Handle_Click">
        <StackPanel Name="stackPanel" Orientation="Horizontal"
                    PointerPressed="Handle_PointerPressedBubble">
          <Image Name="image" Source="avares://Beobachtungsstudie/Assets/avalonia.png"
              RenderOptions.BitmapInterpolationMode="HighQuality"
              VerticalAlignment="Center" PointerPressed="Handle_PointerPressedBubble"/>
          <TextBlock Name="textBlock" FontSize="40" Text="Click"
                     VerticalAlignment="Center" Margin="5,0,0,0"
                     PointerPressed="Handle_PointerPressedBubble"/>
        </StackPanel>
      </Button>
      <ListBox Name="listBox" Margin="10,130,10,10" />
    </Panel>
  </Window>
```

Auf der **Button**-Oberfläche befinden sich nebeneinander ein **Image**- und ein **TextBlock**-Objekt, die ein **StackPanel**-Layoutcontainer verwaltet (siehe Abschnitt 10.4).

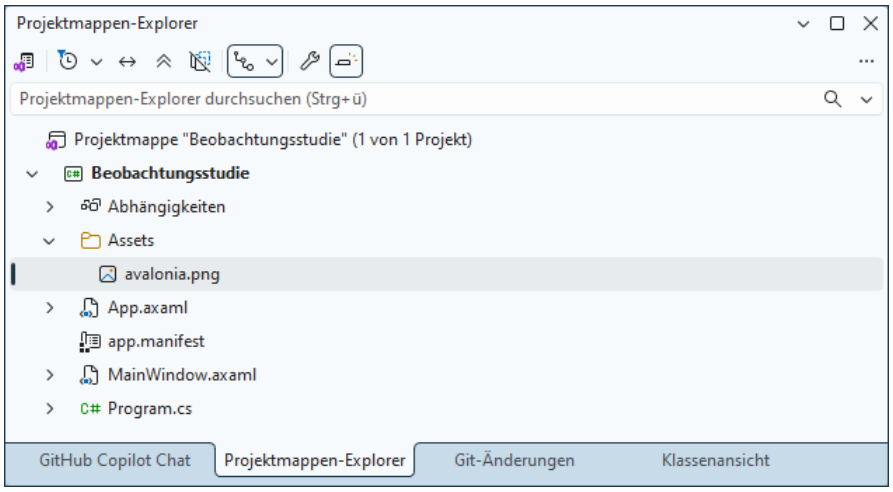
Wir haben das Bild  über die Webseite

https://commons.wikimedia.org/wiki/File:Avalonia_logo.svg

bezogen, mit dem Freeware-Programm **IrfanView**[1] eine Transparenzzone festgelegt und das Ergebnis im ***PNG***-Format (***P**ortable **N**etwork **G**raphics*) in der Datei **avalonia.png** gespeichert (zur IrfanView-Verwendung siehe Abschnitt 11.3.1.4).

Um der **Source**-Eigenschaft eines **Image**-Objekts ein Bild zuzuweisen, kann man so verfahren:

- im Projektordner den Unterordner **Assets** anlegen und die Bilddatei dorthin kopieren, z. B.:



- in die Projektdatei (Namenserweiterung **csproj**) das Element **AvaloniaResource** einfügen:

```
<ItemGroup>
  <AvaloniaResource Include="Assets\**" />
</ItemGroup>
```

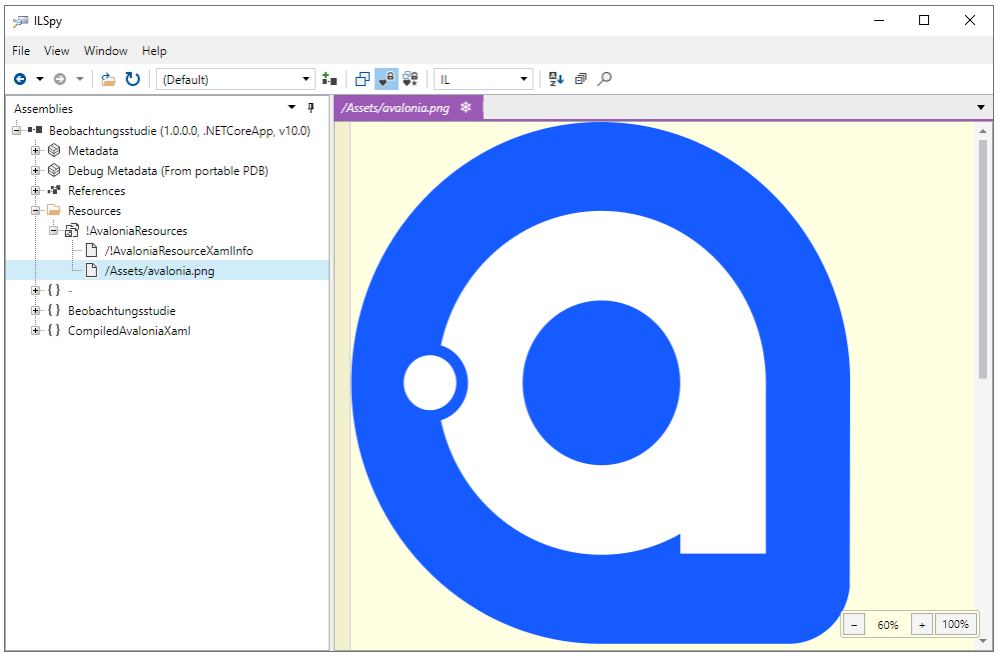
[1] Homepage: http://www.irfanview.de/

- im **Image**-Element der XAML-Datei die Bilddatei per URI-Syntax mit dem speziellen Protokoll **avres** als Wert der Eigenschaft **Source** eintragen, z. B.:

```
<Image Name="image" Source="avares://Beobachtungsstudie/Assets/avalonia.png"
  RenderOptions.BitmapInterpolationMode="HighQuality" . . ./>
```

  Die angefügte Eigenschaft (vgl. Abschnitt 8.4) **RenderOptions.BitmapInterpolationMode** sollte statt der Voreinstellung **LowQuality** die Alternative **HighQuality** als Wert erhalten.

Eine Inspektion mit ILSpy zeigt, dass sich die Datei **avalonia.png** als Ressource im Assembly befindet:



7.4 Klassenbezogene Behandlungsmethoden

Eine Steuerelementklasse kann für ein Routingereignis eine klassenbezogene Behandlungsmethode registrieren, die ...[1]

- für alle Objekte dieser Klasse zuständig ist
- und vor objektbezogenen Behandlungsmethode für dasselbe Routingereignis aufgerufen wird.

Eine das Routingereignis als behandelt deklarierende objektbezogene Behandlungsmethode kann also eine klassenbezogene Behandlungsmethode nicht ausblenden.[2]

Zur Demonstration leiten wir von **Button** eine Klasse namens `MaiButton` ab. Sie erbt u. a. das statische Feld **ClickEvent**, das aufgrund einer von **Button** vorgenommenen Registrierung das **Click**-Routingereignis repräsentiert (siehe Abschnitt 7.5 zur Beziehung von **ClickEvent** und **Click**):[3]

```
public static readonly RoutedEvent<RoutedEventArgs> ClickEvent =
    RoutedEvent.Register<Button, RoutedEventArgs>(nameof(Click),
                                                  RoutingStrategies.Bubble);
```

---

1 Bei der Umsetzung von klassenbezogenen Behandlungsmethoden unterscheidet sich Avalonia deutlich von WPF, siehe:
https://docs.avaloniaui.net/docs/migration/wpf/events#class-handlers

2 https://docs.avaloniaui.net/docs/custom-controls/defining-events

3 https://github.com/AvaloniaUI/Avalonia/blob/main/src/Avalonia.Controls/Button.cs

Über die **RoutedEvent<RoutedEventArgs>** - Methode **AddClassHandler<Target>()**

public IDisposable AddClassHandler<TTarget>(
Action<TTarget,TEventArgs> *handler*,
RoutingStrategies *routes* = **RoutingStrategies.Direct | RoutingStrategies.Bubble,**
bool *handledEventsToo* = **false) where TTarget : Interactive**

weist die Klasse `MaiButton` im statischen Konstruktor dem Feld **ClickEvent** eine klassenbezogenen Behandlungsmethode zu:[1]

```
using Avalonia.Controls;
using Avalonia.Interactivity;
using MsBox.Avalonia;
using MsBox.Avalonia.Enums;
using System;

namespace ClassHandler;

class MaiButton : Button {
    protected override Type StyleKeyOverride => typeof(Button);

    static MaiButton() {
        ClickEvent.AddClassHandler<MaiButton>((x, e) => x.MaiClickEventHandler(e));
    }

    protected async void MaiClickEventHandler(RoutedEventArgs e) {
        var box = MessageBoxManager.GetMessageBoxStandard("MaiButton",
                   "Click-Behandlung durch den ClassHandler der Klasse MaiButton",
                   ButtonEnum.Ok);
        await box.ShowWindowAsync();
    }
}
```

Durch die Überschreibung der **StyledElement**-Eigenschaft **StyleKeyOverride** übernehmen wir das **ControlTheme** der Basisklasse **Button**. `MaiButton` hätte als benutzerdefinierte Steuerelementklasse ohne eigene Layout-Deklaration ansonsten keinerlei visuelle Attribute (Hintergrund, Rand etc.), sodass von den Schaltern nur die Beschriftung zu sehen wäre.

Die XAML-Datei zum Fenster demonstriert die Verwendung einer benutzerdefinierten Steuerelementklasse:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="using:ClassHandler"
        x:Class="ClassHandler.MainWindow"
        Title="Class Handler" Height="140" Width="300">
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
        <local:MaiButton Content="Knopf 1" Margin="5" Click="Knopf1_Click" />
        <local:MaiButton Content="Knopf 2" />
    </StackPanel>
</Window>
```

Zum Projektnamensraum ist eine XML-Namensraumdefinition mit dem Präfix **local** vorzunehmen, und in Instanzelementen ist dem Klassennamen `MaiButton` das Präfix voranzustellen.

---

[1] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
...\BspUeb\Routingereignisse\ClassHandler
Wir ignorieren der üblichen Praxis folgende die **IDisposable**-Rückgabe von **AddClassHandler<TTarget>()**, obwohl dieser Datentyp generell als Aufforderung zu verstehen ist, die mit einem Objekt verbundenen Ressourcen baldmöglichst durch einen **Dispose()** – Aufruf freizugeben. Dieser Aufruf würde die Registrierung der klassenbezogenen Behandlungsmethode aufheben. Wenn es dazu keinen sinnvollen Zeitpunkt gibt, verzichtet man darauf.

In der Code Behind – Datei zum Fenster definieren wir die dem linken Schalter zugewiesene instanzbezogene **Click**-Behandlungsmethode:

```
using Avalonia.Controls;
using Avalonia.Interactivity;
using MsBox.Avalonia;
using MsBox.Avalonia.Enums;

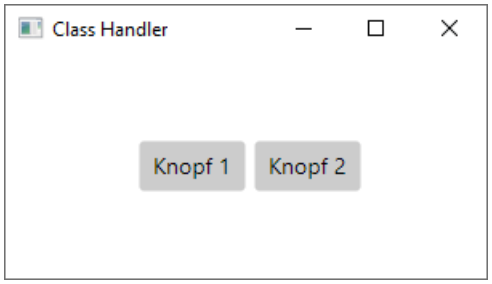
namespace ClassHandler;

public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
    }

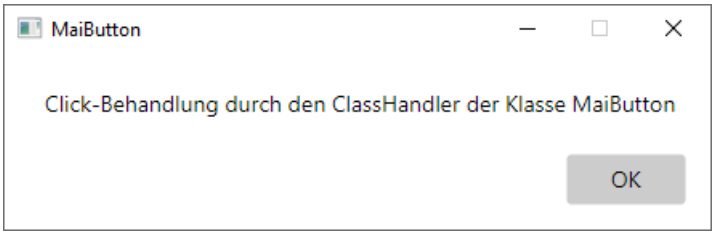
    private async void Knopf1_Click(object? sender, RoutedEventArgs e) {
        var box = MessageBoxManager.GetMessageBoxStandard("Button",
                                    "Click-Handler von Knopf 1", ButtonEnum.Ok);
        await box.ShowWindowAsync();
    }
}
```

Weil Avalonia 12 kein Analogon zur WPF-Klasse **MessageBox** bietet, verwenden wir erneut das NuGet-Paket **MessageBox.Avalonia** (siehe Abschnitt 4.9).

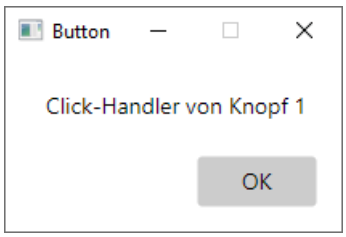
Das Programm präsentiert zwei Befehlsschalter aus der Klasse `MaiButton`,



wobei der linke Schalter über eine eigene **Click**-Behandlungsmethode verfügt. Beim Klick auf einen beliebigen Schalter findet zunächst die klassenbezogene Ereignisbehandlung statt:



Nach einem Klick auf den linken Schalter, folgt noch die objektbezogene Ereignisbehandlung:



7.5 Routingereignisse definieren

Wie schon im Abschnitt 7.4 zu sehen war, registriert die Klasse **Button** in der folgenden Deklarationsanweisung das **ClickEvent**-Routingereignis mit Hilfe der statischen und generischen Methode **Register<TOwner,TEventArgs>()** der Klasse **RoutedEvent** (Namensraum **Avalonia.Interactivity**) beim Avalonia-Ereignissystem:[1]

```
public static readonly RoutedEvent<RoutedEventArgs> ClickEvent =
    RoutedEvent.Register<Button,RoutedEventArgs>(nameof(Click),
                                                 RoutingStrategies.Bubble);
```

Es entsteht ein Objekt aus der Klasse **RoutedEvent<RoutedEventArgs>**, dessen Adresse in einem schreibgeschützten, öffentlichen und statischen Feld der Klasse **Button** landet. Der Feldname **ClickEvent** endet eine Konvention beachtend mit *Event*.

Beim Registrieren sind zu einem Routingereignis anzugeben:

- die Klasse, die das Routingereignis besitzt, festgelegt durch den ersten Typparameter mit dem Namen **TOwner**
  Besitzende (registrierende) Klasse im Beispiel: **Button**
- die von Behandlungsmethoden zu erfüllende Delegatenklasse, festgelegt durch den zweiten Typparameter mit dem Namen **TEventArgs**
  Delegatenklasse im Beispiel: **EventHandler<RoutedEventArgs>**
- der in traditioneller Ereignissyntax (mit Hilfe der Operatoren += und -=) zu verwendende Name (ohne den Bestandteil *Event*), festgelegt durch den ersten Parameter
  Ereignisname im Beispiel: **Click**
- die Routingstrategie(n) (siehe Abschnitt 7.1), festgelegt durch den zweiten Parameter
  Routingstrategie im Beispiel: **RoutingStrategies.Bubble**

Weil das **Click**-Routingereignis beim Avalonia-Ereignissystem registriert ist, kann man ...

- für ein Objekt vom Typ **Button** oder von einem abgeleiteten Typ
- sowie für ein in der GUI-Hierarchie übergeordnetes Objekt (siehe Abschnitt 7.6)

eine Aufrufliste mit **Click**-Behandlungsmethoden verwalten. Für die Zuweisung einer Behandlungsmethode per C# ist auf unserem jetzigen Wissensstand ein etwas umständlicher Aufruf der **Interactive**-Methode **AddHandler()** erforderlich, z. B.:

```
var knopf = new Button { Content = "Knopf"} ;
knopf.AddHandler(Button.ClickEvent, Knopf_Click, RoutingStrategies.Bubble);
```

Mit der Methode **RemoveHandler()** lässt sich eine Registrierung aufheben. Die Zugriffe auf das **RoutedEvent<TEventArgs>** - Objekt sind wegen der Zugriffsstufe **public** generell erlaubt.

Ein per Klassenvariable ansprechbares **RoutedEvent<TEventArgs>** - Objekt wird in der Regel durch ein instanzbezogenes CLR-Ereignis ergänzt, sodass die gewohnte Syntax zum Zuweisen von Behandlungsmethoden verfügbar ist. Die Klasse **Button** definiert das CLR-Ereignis **Click**, wobei die **add**- und **remove**-Zugriffe die **Interactive**-Methoden **AddHandler()** bzw. **RemoveHandler()** verwenden. Diese Methoden erweitern bzw. reduzieren die Aufrufliste des durch den ersten Parameter (das statische **RoutedEvent**-Objekt) identifizierten Routingereignisses:

```
public event EventHandler<RoutedEventArgs>? Click {
    add => AddHandler(ClickEvent, value);
    remove => RemoveHandler(ClickEvent, value);
}
```

Damit lässt sich die obige Behandlungsmethodenregistrierung einfacher erledigen:

```
knopf.Click += knopf_Click;
```

---

[1] https://github.com/AvaloniaUI/Avalonia/blob/main/src/Avalonia.Controls/Button.cs

Im XAML-Code kann die Registrierung per Attribut-Syntax erfolgen, z. B.:

```
<Button Name="knopf" . . . Click="Knopf_Click">
```

Während beim **Click**-Ereignis nur die **Bubble**-Route stattfindet, ist für das in der Klasse **InputElement** definierte Ereignis **PointerPressed** auch die **Tunnel**-Route geplant:

```
public static readonly RoutedEvent<PointerPressedEventArgs> PointerPressedEvent =
    RoutedEvent.Register<InputElement, PointerPressedEventArgs>(
        nameof(PointerPressed),
        RoutingStrategies.Tunnel | RoutingStrategies.Bubble);
```

Die Zuweisung einer Behandlungsmethode für die **Tunnel**-Route ist weder mit der CLR-Ereignissyntax noch per XAML - Attribut-Syntax möglich, sondern nur per **AddHandler()** – Aufruf.

Die Klasse **Button** feuert das **Click**-Routingereignis in der Methode **OnClick()**. Das geschieht *nicht* (wie bei einem CLR-Ereignis) durch den Aufruf eines Delegatenobjekts, das zu einem bestimmten Objekt gehört. Stattdessen ruft **OnClick()** die **UIElement**-Methode **RaiseEvent()** auf:

```
protected virtual void OnClick(){
    if (IsEffectivelyEnabled) {
        . . .
        var e = new RoutedEventArgs(ClickEvent);
        RaiseEvent(e);
        . . .
    }
}
```

Von einem Routingereignis sind mehrere GUI-Elemente betroffen, und das Avalonia-Ereignissystem ruft die dort angemeldeten Behandlungsmethoden in der richtigen Reihenfolge aufruft.

7.6 Angefügte Ereignisse

Im Abschnitt 7.3 war zu sehen, dass man bei einem Objekt der (indirekt von **InputElement** abstammenden) Layoutmanager-Klasse **Panel** per Attributsyntax nicht nur zum **PointerPressed**-Routingereignis der Klasse **InputElement**, sondern auch zum **Click**-Routingereignis der Klasse **Button**, die sich *nicht* in der **Panel**-Ahnenreihe befindet, eine Behandlungsmethode anmelden kann:

```
<Panel Name="panel" PointerPressed="Handle_PointerPressedBubble"
                    Button.Click="Handle_Click">
```

Weil es sich bei **Click** um ein Routingereignis handelt, kann man dem **Panel**-Container eine Methode zur Behandlung der **Click**-Ereignisse zuweisen, die im Container enthaltene Steuerelemente aus der **Button**-Hierarchie auslösen. Dabei ist in der Attributsyntax zum Ereignis auch der Name der besitzenden Klasse anzugeben.

Das ermöglicht die im Abschnitt 7.2 beschriebene rationelle Behandlung von aufperlenden Routingereignissen: Befinden sich in einem Layoutcontainer mehrere Steuerelemente vom selben Typ, dann lässt sich über eine Zuweisung auf Container-Ebene eine gemeinsame Ereignisbehandlung durch eine einzige Methode realisieren. In dieser Behandlungsmethode kann man die Ereignisquelle über die **Source**-Eigenschaft des übergebenen Objekts mit Ereigniseigenschaften feststellen.

Dieselbe qualifizierte Ereignisbenennung findet sich auch bei den sogenannten *angefügten Ereignissen* (engl.: *attached events*), bei denen es sich um spezielle Routingereignisse handelt.[1] Ihre Besonderheit besteht darin, dass man einem beliebigen Element eine Behandlungsmethode zuweisen kann:

---

[1] https://docs.avaloniaui.net/docs/input-interaction/routed-events

- Das behandelnde (die Ereignismethode ausführende) Element muss *nicht* zu einer Klasse gehören, die das angefügte Ereignis besitzt (registriert) oder erbt.
- Dasselbe gibt für das auslösende Element. Hier liegt der Unterschied zur Behandlung der von enthaltenen **Button**-Objekten ausgelösten **Click**-Ereignissen durch eine dem Container zugewiesene Methode.

Wir betrachten exemplarisch die von der Klasse **DragDrop** registrierten Routingereignisse **DragEnter**, **DragLeave**, **DragOver** und **Drop**. Die Klasse **DragDrop** stammt direkt von **Object** ab und gehört damit nicht zu Avalonias Steuerelement-Hierarchie (siehe Abschnitt 4.2). Als Besitzer bzw. Anbieter von angefügten Routingereignissen sind beliebige Klassen erlaubt.

Ein angefügtes Ereignis wird durch den Aufruf einer Überladung der statischen **RoutedEvent**-Methode **Register<TEventArgs>()** registriert, z. B. das Ereignis **DragEnterEvent**:[1]

```
public static readonly RoutedEvent<DragEventArgs> DragEnterEvent =
    RoutedEvent.Register<DragEventArgs>("DragEnter",
                                        RoutingStrategies.Bubble, typeof(DragDrop));
```

Weil es kein **DragDrop**-Steuerelement gibt, fehlt eine ergänzende CLR-Ereignisdefinition ebenso wie die Registrierung einer klassenbezogenen Behandlungsmethode. Stattdessen enthält die Klasse **DragDrop** die statischen Methoden **AddDragEnterHandler()** und **RemoveDragEnterHandler()**:

```
public static void AddDragEnterHandler(Interactive element,
                                       EventHandler<DragEventArgs> handler) {
    element.AddHandler(DragEnterEvent, handler);
}
public static void RemoveDragEnterHandler(Interactive element,
                                          EventHandler<DragEventArgs> handler) {
    element.RemoveHandler(DragEnterEvent, handler);
}
```

Diese Methoden ermöglichen es einer beliebigen Klasse, für ein **Interactive**-Objekt die Zuweisung einer Methode zur Behandlung eines angefügten Ereignisses vorzunehmen oder aufzuheben, z. B.:

```
DragDrop.AddDragEnterHandler(border, OnDragEnter);
```

Über die **Interactive**-Instanzmethode **AddHandler()** kann ein Objekt auch selbst eine `DragEnterEvent`-Behandlungsmethode vereinbaren, z. B.:

```
border.AddHandler(DragDrop.DragEnterEvent, OnDragEnter);
```

Wie gleich zu sehen ist, lassen sich angefügte Ereignisse auch per XAML – Attributsyntax zuweisen, wobei der XamlX-Compiler in Avalonia die statischen **Add**- bzw. **Remove**-Methoden benötigt.[2]

Im folgenden Fenster dient zur Drag & Drop – Demonstration ein **Border**-Element,, dessen Aufgabe generell darin besteht, ein einzelnes Kindelement mit einem Rahmen und einem Hintergrund auszustatten. Im Beispiel signalisiert das **Border**-Element durch die angefügte *Eigenschaft* (siehe Abschnitt 8.4) **DragDrop.AllowDrop** seine Bereitschaft zur Teilnahme am Drag & Drop – Verfahren. Außerdem sind per Attributsyntax Methoden zur Behandlung der oben erwähnten angefügten Routingereignisse registriert:

---

[1] https://github.com/AvaloniaUI/Avalonia/blob/main/src/Avalonia.Base/Input/DragDrop.cs

[2] https://github.com/AvaloniaUI/Avalonia/issues/9507

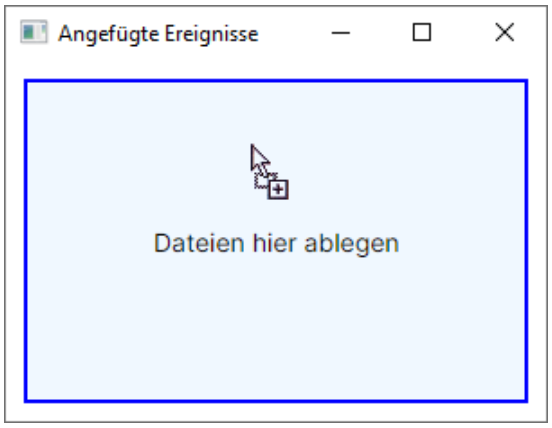
```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="AttatchedProperties.MainWindow"
        Title="Angefügte Ereignisse" Width="300" Height="200">

    <Border Name="border" DragDrop.AllowDrop="True" Margin="10"
        Background="AliceBlue"
        DragDrop.DragEnter="OnDragEnter"
        DragDrop.DragLeave="OnDragLeave"
        DragDrop.DragOver="OnDragOver"
        DragDrop.Drop="OnDrop">
        <TextBlock Name="textBlock" Text="Dateien hier ablegen"
                   HorizontalAlignment="Center" VerticalAlignment="Center"/>
    </Border>
</Window>
```

Das Fenster gehört zu einer weitgehend von der Avalonia-Webseite

https://docs.avaloniaui.net/docs/input-interaction/drag-and-drop

übernommenen Anwendung:[1]



Wenn sich in der Zwischenablage eine Liste mit Dateien befindet, denn ...

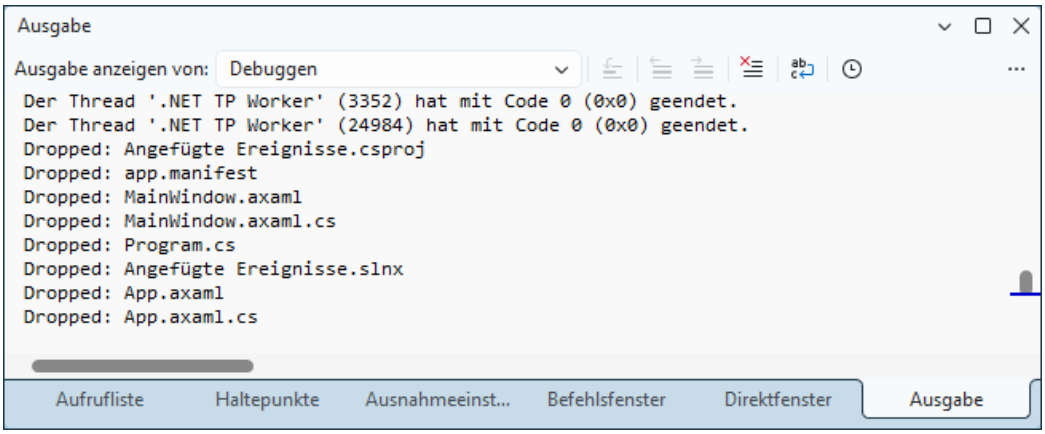
- reagiert die Anwendung mit einer geänderten Rahmenfarbe, wenn sich der Mauszeiger über der Trefferzone befindet,
- verarbeitet die Anwendung abgelegte Dateilisten durch eine Protokollausgabe:

```
private void OnDrop(object? sender, DragEventArgs e) {
    if (e.DataTransfer.Formats.Contains(DataFormat.File)) {
        var files = e.DataTransfer.TryGetFiles();
        if (files != null) {
            foreach (var file in files)
                Debug.WriteLine($"Dropped: {file.Name}");
        }
    }
    border.BorderBrush = null;
    border.BorderThickness = new Thickness(0);
}
```

[1] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
...\BspUeb\Routingereignisse\Angefügte Ereignisse (Drag & Drop)

Dieser Quellcode gibt Gelegenheit, die Funktionsweise der **Debug**-Methode **WriteLine()** im Visual Studio zu erläutern:

- Die Methode wird nur im Debug-Modus ausgeführt (nach einem Anwendungsstart mit **F5** statt **Strg-F5**)
- und artikuliert sich im **Ausgabe**-Fenster des Debuggers, z. B.:



8 Abhängigkeitseigenschaften

Die Avalonia-Designer haben dem Vorbild der WPF-Designer folgend nicht nur die CLR-Ereignisse durch die leistungsfähigeren Routingereignisse ergänzt, sondern auch zu den CLR-Eigenschaften, die meist aus einer Instanzvariablen und einem Methodenpaar für den lesenden bzw. schreibenden Zugriff bestehen, mit den sogenannten *Abhängigkeitseigenschaften* (engl.: *dependency properties*) eine funktional erheblich erweiterte Variante geschaffen.

Wie es der Name vermuten lässt, wirken sich auf die Ausprägung einer Abhängigkeitseigenschaft verschiedene Einflussquellen aus (übergeordnete Steuerelemente, eine Datenbindung, Stile und Animationen).

Die allgemeine Technik und Funktionsvielfalt der Abhängigkeitseigenschaften darzustellen, würde den Rahmen des Manuskripts sprengen. Wir konzentrieren uns auf drei schon beim Avalonia-Einstieg unvermeidliche bzw. nützliche Funktionen:

- Eigenschaftsübertragung auf enthaltene GUI-Elemente
  Bei Abhängigkeitseigenschaften mit aktivierter Vererbung (**Inherits**-Ausprägung gleich **true**) überträgt sich die einem GUI-Element zugewiesene Ausprägung (z. B. die Schriftgröße) auf enthaltene GUI-Elemente, falls keine andere Eigenschaftsquelle dominiert.
- Angefügte Eigenschaften von Layoutcontainern
  Bei den sogenannten *angefügten Eigenschaften* handelt es sich um spezielle Abhängigkeitseigenschaften, die zur Positionierung der in Layoutcontainern enthaltenen Steuerelemente unverzichtbar und auch für andere Zwecke nützlich sind (siehe Abschnitt 8.4).
- Datenbindung
  Eine Abhängigkeitseigenschaft lässt sich an ein beliebiges Datenobjekt der Anwendung oder an ein Steuerelement binden, um stets den aktuellen Zustand der Quelle anzuzeigen (siehe Kapitel 9).

Zu den mit Hilfe von Abhängigkeitseigenschaften realisierten Avalonia-Optionen, die im Manuskript aus Zeitgründen zu wenig Beachtung finden, gehören Stile und Animationen.

Wir konzentrieren uns auf die *Anwendung* der in Avalonia-Typen zahlreich vorhandenen Abhängigkeitseigenschaften, liefern jedoch im Abschnitt 8.2 auch einige Informationen für Entwickler, die Abhängigkeitseigenschaften definieren wollen.

8.1 AvaloniaProperty - Varianten

In Avalonia spielt die leistungsfähige, von der Internet-Technik ***CSS*** (***C**ascading **S**tyle **S**heets*) inspirierte Verwaltung von Steuerelementstilen eine große Rolle. Statt von *Abhängigkeitseigenschaften* (wie in WPF) spricht man daher bevorzugt von *gestalteten Eigenschaften* (engl.: *styled properties*). Wie z. B. die Registrierung der **Border**-Eigenschaft **Background** zeigt, haben die Abhängigkeitseigenschaften als Datentyp eine Konkretisierung der generischen Klasse **StyledProperty<TValue>**:[1]

```
public static readonly StyledProperty<IBrush?> BackgroundProperty =
    AvaloniaProperty.Register<Border, IBrush?>(nameof(Background));
```

Wir bevorzugen den Begriff *Abhängigkeitseigenschaften*, weil im Manuskript z. B. Datenbindungen und übergeordnete Steuerelemente als Einflussquellen eine größere Rolle spielen als Stile.

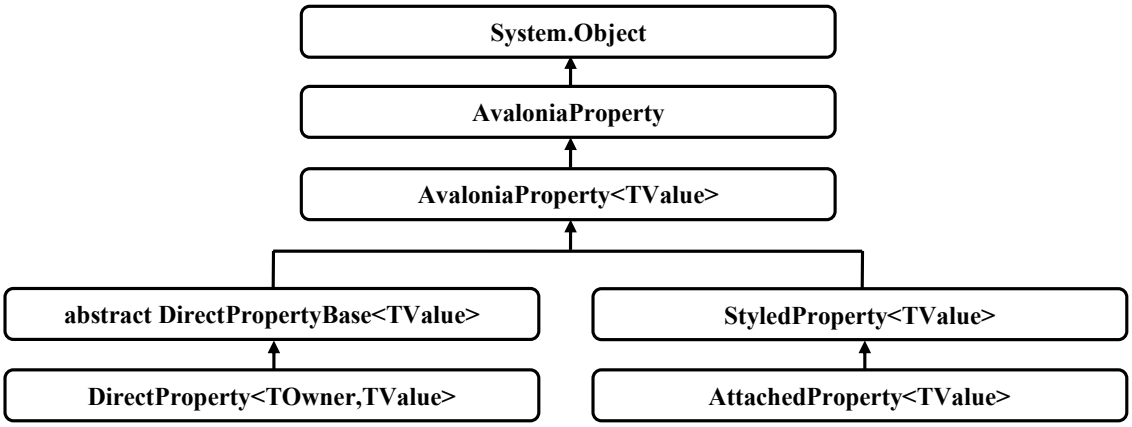
Zu den Abhängigkeitseigenschaften (Datentyp **StyledProperty<TValue>**) zählen als spezialisierte Variante auch die *angefügten Eigenschaften* (engl.: *attached properties*, Datentyp

---

[1] https://github.com/AvaloniaUI/Avalonia/blob/main/src/Avalonia.Controls/ContentControl.cs

AttachedProperty<TValue>), die eine Besonderheit beim Besitzer aufweisen (siehe Abschnitt 8.4).

Die Abhängigkeitseigenschaften gehören zusammen mit den *Direkteigenschaften* (Datentyp **DirectProperty<TOwner,TValue>**) zu den *Avalonia-Eigenschaften* (Datentyp **AvaloniaProperty<TValue>**). Hier ist die Klassenhierarchie der Avalonia-Eigenschaften zu sehen:



Im Vergleich zu den Abhängigkeitseigenschaften (alias: *gestalteten Eigenschaften*) haben die Direkteigenschaften eine geringere Bedeutung. Sie sparen Aufwand durch den Verzicht auf viele Leistungen des Avalonia-Eigenschaftssystems:

- keine Eigenschaftsübertragung auf Kindelemente
- keine Stilverwaltung und Animationen

Als Ziel einer Datenbindung ist eine Direkteigenschaft aber zugelassen. Der Wert einer Direkteigenschaft ist wie bei einer traditionellen CLR-Eigenschaft in einem Feld gespeichert, während sich der Wert einer Abhängigkeitseigenschaft im sogenannten *Wertespeicher* befindet, den das Avalonia Eigenschaftssystem verwaltet (siehe Abschnitt 8.2). Weitere Informationen zu den Direkteigenschaften bietet die Online-Dokumentation.[1]

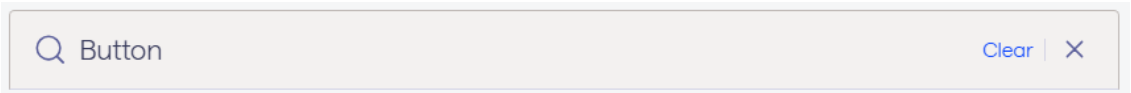
Jede von **AvaloniaObject** abstammende Klasse (siehe Abschnitt 4.2) kann Avalonia-Eigenschaften anbieten. Bei den Eigenschaften von Steuerelementen handelt es sich meist um Avalonia-Eigenschaften, die aber in aller Regel auch wie gewöhnliche CLR-Eigenschaften verwendbar sind (dazu mehr im Abschnitt 8.2).

Ob es sich bei einer Eigenschaft einer Steuerelementklasse um

- eine **StyledProperty<TValue>**,
- eine **AttachedProperty<TValue>**
- oder um eine **DirectProperty<TOwner,TValue>**

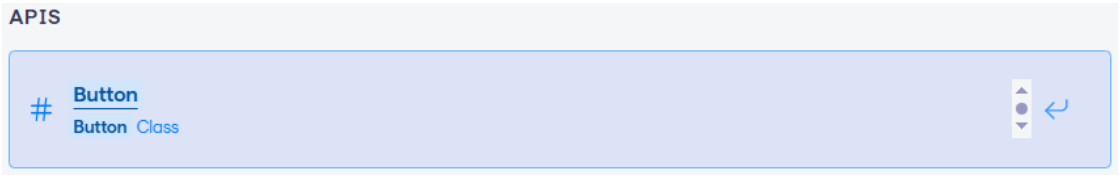
handelt, lässt sich mit Hilfe der API-Dokumentation ohne Quellcode-Studium ermitteln, z. B. bei der **Background**-Eigenschaft der Klasse **Button**:

- Über die Webseite https://docs.avaloniaui.net/api/ nach **Button** suchen:

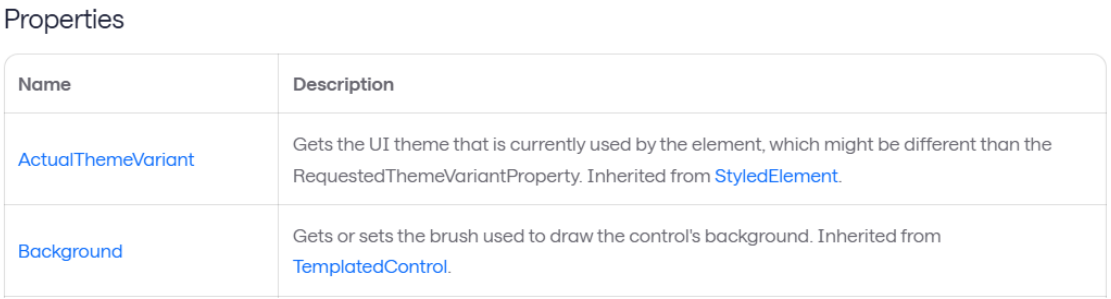


[1] https://docs.avaloniaui.net/docs/properties/

- Klick auf die API-Information zur Klasse **Button**:



- Klick auf die Eigenschaft **Background** in der **Properties**-Liste:

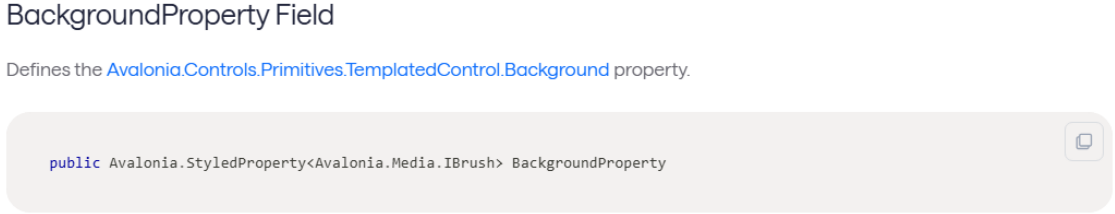


  Man gelangt zur API-Webseite der Klasse **TemplatedControl**, die die Eigenschaft **Background** definiert.

- Klick auf **BackgroundProperty** in der **Fields**-Liste:



- Nun gibt der Datentyp **StyledProperty<IBrush>** eine eindeutige Auskunft:



8.2 Registrierung und CLR-Verpackung

Analog zur Realisation der Routingereignisse (siehe Abschnitt 7.5) wird eine Abhängigkeitseigenschaft durch einen Aufruf der statischen und generischen **AvaloniaProperty**-Methode **Register<TOwner,TValue>** registriert. Im Abschnitt 8.1 war schon die Registrierung der **Border**-Eigenschaft **Background** zu sehen:[1]

```
public static readonly StyledProperty<IBrush?> BackgroundProperty =
    AvaloniaProperty.Register<Border, IBrush?>(nameof(Background));
```

Die Methode **Register<TOwner,TValue>()** liefert ein Objekt einer Konkretisierung der generischen Klasse **StyledProperty<TValue>**, auf das ein statisches Feld der registrierenden und besitzenden Klasse (im Beispiel **ContentControl**) mit den Modifikatoren **public** und **readonly** sowie *Property* als terminalem Namensbestandteil zeigt.

Die Methode besitzt zahlreiche Parameter zur Konfiguration einer Abhängigkeitseigenschaft, die mit Ausnahme des ersten Parameters optional sind, also Voreinstellungswerte besitzen:

---

[1] https://github.com/AvaloniaUI/Avalonia/blob/main/src/Avalonia.Controls/Border.cs

```
public static StyledProperty<TValue> Register<TOwner, TValue>(
            string name,
            TValue defaultValue = default!,
            bool inherits = false,
            BindingMode defaultBindingMode = BindingMode.OneWay,
            Func<TValue, bool>? validate = null,
            Func<AvaloniaObject, TValue, TValue>? coerce = null,
            bool enableDataValidation = false)
                where TOwner : AvaloniaObject
        { . . . }
```

Es folgen kurze Erläuterungen zu den Parametern:

- *name*
  Hier ist der Name der *CLR-Eigenschaftsverpackung* für die Abhängigkeitseigenschaft anzugeben (siehe unten).
- *defaultValue*
  Das ist die Voreinstellung für den Fall, dass keine Einflussquelle einen Wert liefert.
- *inherits*
  Soll eine Eigenschaftsübertragung auf Kindelemente stattfinden?
- *defaultBindingMode*
  Das ist der voreingestellte Modus für den Informationstransfer bei der Datenbindung (siehe Kapitel 9).
- *validate*
  Hier kann man einen Delegaten für die Validierung von zugewiesenen Eigenschaftsausprägungen vereinbaren.
- *coerce*
  Hier kann man einen Delegaten vereinbaren, der nach der Ermittlung der effektiven Eigenschaftsausprägung aus den konkurrierenden Einflussgrößen eine Endkontrolle vornimmt.
- *enableDataValidation*
  Soll für zugewiesene Werte eine Validierung stattfinden (siehe Abschnitt 9.7)?

Aus den Parametern resultiert ein (über die **StyledProperty<TValue>** – Methode **GetMetadata()** abrufbares) **AvaloniaPropertyMetadata**-Objekt. Das **StyledProperty<TValue>** – Objekt enthält also wichtige Informationen über die Abhängigkeitseigenschaft, aber keinesfalls die Eigenschaftsausprägungen von konkreten Steuerelementen.

Obwohl sich eine **AvaloniaProperty<TValue>** zur Realisation der Zusatzfunktionen (Eigenschaftsübertragung, Datenbindung, Stile, Animationen) erheblich von einer CLR-Eigenschaft unterscheidet, muss sie doch auf vertraute Weise verwendbar sein. Daher definiert eine registrierende Klasse stets eine ergänzende instanzbezogene Eigenschaft mit **get**- und **set**-Zugriff, z. B.:

```
public IBrush? Background  {
    get => GetValue(BackgroundProperty);
    set => SetValue(BackgroundProperty, value);
}
```

Man spricht hier von einer *CLR-Eigenschaftsverpackung* für die Abhängigkeitseigenschaft (engl.: *property wrapper*). Im **get**- bzw. **set**-Zugriff kommen die **AvaloniaObject**-Methoden **GetValue()** und **SetValue()** zum Einsatz.

Die Abhängigkeitseigenschafts-Ausprägungen eines Steuerelements werden vom Avalonia-Eigenschaftssystem verwaltet. Sie befinden sich im sogenannten *Wertespeicher* (engl.: *value store*) des Steuerelements. Für eine Eigenschaft können mehrere Werte in der folgenden absteigenden Dominanzordnung vorliegen:[1]

[1] https://docs.avaloniaui.net/docs/properties/value-precedence

- Animation
- lokaler Wert
  Er wird per XAML oder C# - Code gesetzt. Es kann sich um einen festen Wert oder um eine Datenbindung handeln.
- Wert aus einem Stil mit einem konditional zutreffenden Selektor
- in einer Steuerelementvorlage gesetzter Wert
- Wert aus einem Stil mit einem generell zutreffenden Selektor
- Übernahme von einem übergeordneten UI-Element
- Voreinstellungswert der Abhängigkeitseigenschaft

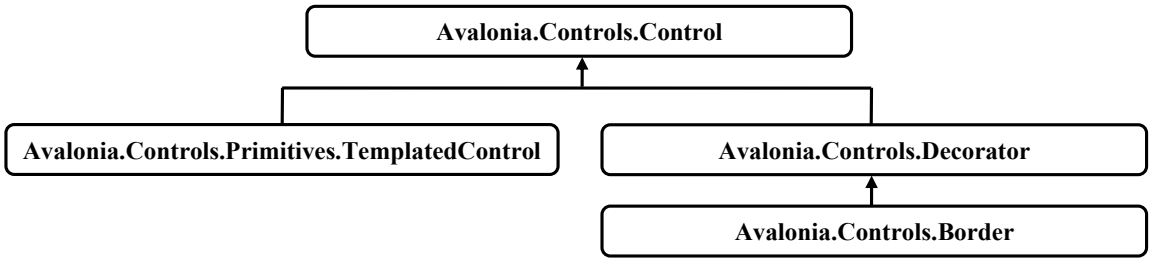
Das Eigenschaftssystem verwendet den Wertespeicher rationell und belegt z. B. für die zahlreich vorhandenen Abhängigkeitseigenschaften eines **Button**-Objekts *keinen* Speicherplatz, wenn keine lokalen Werte vorliegen.

Der Rest des Abschnitts ist nur relevant für Entwickler, die Abhängigkeitseigenschaften in eigenen Steuerelementklassen definieren wollen.

Will eine Klasse eine Abhängigkeitseigenschaft anbieten, die mit derselben Funktionalität in einer anderen, nicht im eigenen Stammbaum befindlichen Klasse bereits vorhanden ist, dann sollte sie gegenüber dem Avalonia-Eigenschaftssystem

- *nicht* per **Register()** eine neue Abhängigkeitseigenschaft anlegen,
- sondern per **AddOwner()** die vorhandene Abhängigkeitseigenschaft übernehmen.

So verhält sich z. B. die Klasse **TemplatedControl** bei der von **Border**



registrierten Abhängigkeitseigenschaft **Background**:

```
public static readonly StyledProperty<IBrush?> BackgroundProperty =
    AvaloniaProperty.Register<Border,IBrush?>(nameof(Background));
```

Die Klasse **TemplatedControl** registriert die Abhängigkeitseigenschaft **Background** nicht neu, sondern meldet ihren Beitritt:[1]

```
public static readonly StyledProperty<IBrush?> BackgroundProperty =
    Border.BackgroundProperty.AddOwner<TemplatedControl>();
```

Aus der Verwendung von **AddOwner()** ergeben sich im Vergleich zu **Register()** u. a. die folgenden Vorteile:

- Bei aktivierter Eigenschaftsvererbung übernehmen Elemente der beigetretenen Klasse den Wert eines im GUI-Baum übergeordneten Objekts der registrierenden Klasse als Voreinstellung.
- Die Verwendung der vorhandenen Abhängigkeitseigenschaft spart Ressourcen im Vergleich zur Erstellung einer neuen.

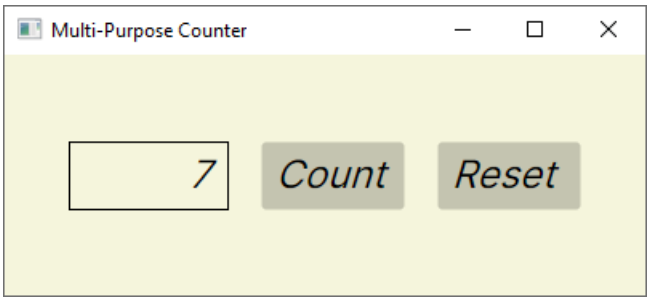
[1] https://github.com/AvaloniaUI/Avalonia/blob/main/src/Avalonia.Controls/Primitives/TemplatedControl.cs

8.3 Eigenschaftsübertragung auf enthaltene Elemente

Bei vielen Abhängigkeitseigenschaften (z. B. **FontSize**) findet die sogenannte *Eigenschaftswertvererbung* (engl. *property value inheritance*) statt,, wobei sich die an ein GUI-Element (z. B. an ein **Window**-Element) vergebene Ausprägung auf eingeschachtelte Elemente überträgt.[1] Allerdings hängt die Eigenschaftsausprägung bei einem konkreten Kindelement eventuell noch von anderen Quellen ab.

Man spricht zwar von der *Eigenschaftswertvererbung*, doch hat diese Spezialität des Avalonia-Eigenschaftssystems *nichts* mit der Ableitung von Klassen zu tun.

Im folgenden Beispiel mit einem **Label**-Objekt und zwei **Button**-Objekten in einem **Grid**-Layoutcontainer mit drei Spalten



sind für das XAML-Wurzelelement **Window** eine Hintergrundfarbe, eine Schriftgröße und ein Schriftstil durch die Abhängigkeitseigenschaften **Background**, **FontSize** und **FontStyle** (alle definiert in der Klasse **TemplatedControl** aus dem Namensraum **System.Windows.Controls**) vereinbart:[2]

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="Eigenschaftsübertragung.MainWindow"
        Title="Multi-Purpose Counter"
        Background="Beige" FontSize="24" FontStyle="Italic"
        CanResize="False" Height="150" Width="400" Loaded="Window_Loaded">

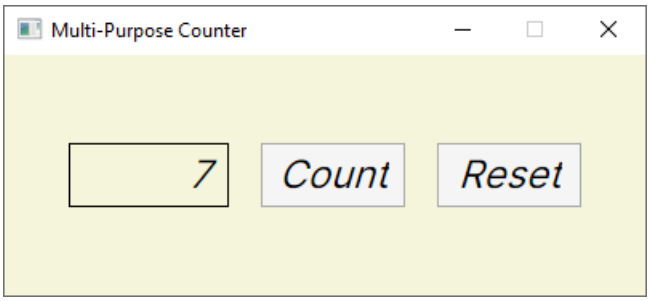
  <Grid ColumnDefinitions="*,*,*" Button.Click="Button_Click"
        VerticalAlignment="Center" HorizontalAlignment="Center" ColumnSpacing="20">
    <Label Name="label" Content="0 " Width="100" BorderThickness="1" BorderBrush="Black"
           HorizontalContentAlignment="Right" VerticalContentAlignment="Center"/>
    <Button Name="count" Content="_Count" Width="90" Grid.Column="1" IsDefault="True" />
    <Button Name="reset" Content="_Reset" Width="90" Grid.Column="2" />
  </Grid>
</Window>
```

Die enthaltenen Steuerelemente übernehmen die beiden Schriftattribute, die Hintergrundfarbe hingegen nicht. Weil das **Label**-Objekt einen transparenten Hintergrund besitzt, sieht es so aus, als würde es die Hintergrundfarbe des Fensters übernehmen. Weil die **Button**-Objekte im voreingestellten **FluentTheme** einen semi-transparenten Hintergrund haben, wirkt sich auch bei ihnen die **Window**-Farbe aus. Dieser Effekt ist aber nicht als Eigenschaftsvererbung zu deuten und verschwindet nach dem Wechsel zum **SimpleTheme** (zur Theme-Einstellung siehe Abschnitt 4.8):

---

[1] https://docs.avaloniaui.net/docs/properties/property-value-inheritance

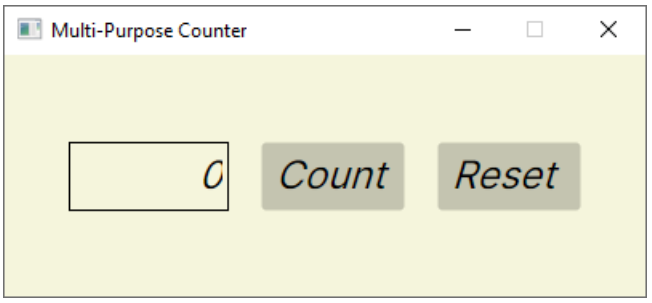
[2] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Abhängigkeitseigenschaften\Eigenschaftsübertragung



Der Text im **Label**-Steuerelement endet mit einem Leerzeichen,

```
Content="0 "
```

weil die zur Demonstration der Eigenschaftsübertragung gewählte kursive Schrift zusammen mit der zur Zahlenausgabe passenden Rechtsbündigkeit anderenfalls eine fehlerhafte Anzeige verursacht:



Bei den **Button**-Beschriftungen tritt im **SimpleTheme** dasselbe Problem auf.

Zu einer Abhängigkeitseigenschaft, also zu einer Konkretisierung der generischen Klasse **StyledProperty<TValue>**, erfährt man über die **Inherits**-Eigenschaft, ob eine Eigenschaftsübertragung auf Kindelemente stattfindet. Der folgende Konstruktor des aktuellen Demonstrationsprogramms

```
public MainWindow() {
    InitializeComponent();
    Console.WriteLine("BackgroundProperty: " + BackgroundProperty.Inherits);
    Console.WriteLine("FontSizeProperty:   " + FontSizeProperty.Inherits);
    Console.WriteLine("FontStyleProperty:  " + FontStyleProperty.Inherits);
}
```

liefert erwartungsgemäß die Ausgabe:

```
BackgroundProperty: False
FontSizeProperty:   True
FontStyleProperty:  True
```

Es ist für die Übertragung der Eigenschaftswerte vom **Window**-Objekt auf die Steuerelemente *kein* Problem, dass der zwischengeschaltete **Grid**-Container die Eigenschaften **FontSize** und **FontStyle** nicht kennt.

8.4 Angefügte Eigenschaften

Zur Verwaltung der Steuerelemente eines Fensters mit dynamischer Anpassung von Positionen und Dimensionen bei einer Änderung der Fenstergröße kommt oft der **Grid**-Layoutcontainer zum Einsatz, der im Allgemeinen mehrere Zeilen und/oder Spalten zur Platzierung der enthaltenen Steuerelemente verwendet. Im Abschnitt 10.2.1 ist zu erfahren, wie man über die Eigenschaften **ColumnDefinitions** und **RowDefinitions** die Struktur und die Platzaufteilung eines **Grid**-Containers flexibel festlegt. Fügt man ein Steuerelement *ohne* Ortsangabe in einen **Grid**-Container ein, dann landet es in der Zelle (0, 0), also oben links. Um für ein Steuerelement eine alternative Zelle

festzulegen, sind im zugehörigen XAML-Element die Attribute **Grid.Row** bzw. **Grid.Column** mit der 0-basierten Nummer der Zeile bzw. Spalte zu versorgen. Der folgende XAML-Code setzt ein **Button**-Objekt in die Zelle (1, 1) einer **Grid**-Matrix mit drei Spalten und zwei Zeilen:

```
<Grid ColumnDefinitions="*,*,*" RowDefinitions="*,*">
    <Button Content="Laden" Click="Button_Click"
            HorizontalAlignment="Center" VerticalAlignment="Center"
            Grid.Row="1" Grid.Column="1" />
    . . .
</Grid>
```

Während in der Steuerelementklasse **Button** zu den XAML-Attributen **Content**, **HorizontalAlignment** und **VerticalAlignment** jeweils eine korrespondierende Eigenschaft vorhanden ist, sucht man in der API-Klassenbeschreibung die Eigenschaften **Grid.Row** und **Grid.Column** oder auch **Row** und **Column** vergeblich. **Grid.Row** und **Grid.Column** sind sogenannte *angefügte Eigenschaften* (engl.: *attached properties*), die die Klasse **Grid** zur Verfügung stellt, damit die im **Grid**-Layoutcontainer verwalteten GUI-Elemente ihre Positionswünsche formulieren können.

Bei einer angefügten Eigenschaft handelt es sich um eine Abhängigkeitseigenschaft mit den folgenden Besonderheiten:

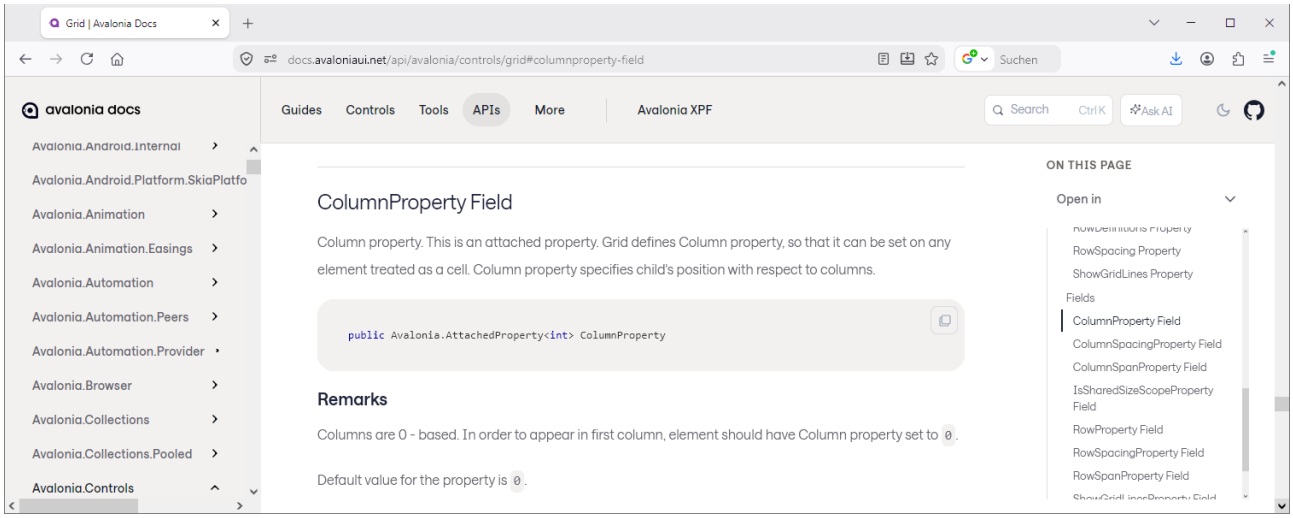
- Bei der Eigenschaftsvergabe ist kein Objekt der definierenden Klasse (im Beispiel: **Grid**) betroffen, sondern ein Objekt einer anderen Klasse (im Beispiel: **Button**).
- Man kann eine angefügte Eigenschaft im C# - Quellcode *nicht* wie eine normale CLR-Eigenschaft verwenden. Stattdessen sind in der anbietenden Klasse statische **Set**- und **Get**-Methoden vorhanden. Die Klasse **Grid** enthält z. B. die Methode **SetColumn()**, die als Parameter ein Objekt der Klasse **Control** (Namensraum **Avalonia.Controls**) und eine Spaltennummer erwartet:

  public void SetColumn(Control *element***, int** *value***)**

  Als Gegenstück zur schreibenden Methode **Grid.SetClumn()** ist die lesende Methode **Grid.GetClumn()** vorhanden:

  public int GetColumn(Control *element***)**

In der API-Dokumentation sind die von einer Klasse angebotenen angefügten Eigenschaften am Datentyp des zugehörigen Felds zu erkennen. Klickt man z. B. in der **Grid**-Dokumentation auf den **Fields**-Eintrag **ColumnProperty Field**,



dann belegen die Erläuterung und der Datentyp **AttachedProperty<int>**,

public Avalonia.AttachedProperty<int> ColumnProperty

dass es sich um eine angefügte Eigenschaft handelt. Die Klasse **AttachedProperty<TValue>** stammt von **StyledProperty<TValue>** ab (siehe Vererbungshierarchie im Abschnitt 8.1).

Das folgende Fenster enthält den oben vorgestellten **Grid**-Container mit drei Spalten und zwei Zeilen sowie einem **Button** in der Zelle (1, 1):[1]

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="AttachedProperties.MainWindow"
        Title="Angefügte Eigenschaften" Height="200" Width="400">
    <Grid ColumnDefinitions="*,*,*" RowDefinitions="*,*" ShowGridLines="True">
        <Button Content="Laden" Click="Button_Click"
                HorizontalAlignment="Center" VerticalAlignment="Center"
                Grid.Row="1" Grid.Column="1" />
    </Grid>
</Window>
```

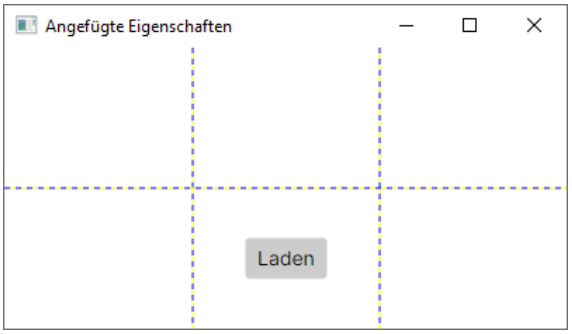
Die **Click**-Behandlungsmethode ist in der Code Behind - Datei definiert:

```
public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
    }

    async void Button_Click(object sender, RoutedEventArgs e) {
        var box = MessageBoxManager.GetMessageBoxStandard("Information",
                  "Das Laden ist noch nicht implementiert.", ButtonEnum.Ok,
                  MsBox.Avalonia.Enums.Icon.Info);
        await box.ShowWindowAsync();
    }
}
```

Weil Avalonia 12 kein Analogon zur WPF-Klasse **MessageBox** bietet, verwenden wir erneut das NuGet-Paket **MessageBox.Avalonia** (siehe Abschnitt 4.9).

Das **Button**-Objekt erscheint in der gewünschten Zelle des **Grid**-Layoutcontainers:



Damit im Programm die korrekte Positionierung des Schalters gut erkennbar ist, hat die Eigenschaft **ShowGridLines** des Layoutcontainers per Attributsyntax den Wert **true** erhalten:

```
<Grid ColumnDefinitions="*,*,*" RowDefinitions="*,*" ShowGridLines="True">
```

Eine Beschäftigung mit den C# - Methoden zum lesenden oder schreibenden Zugriff auf eine angefügten Eigenschaft scheint überflüssig, weil bei der Definition der Bedienoberfläche das Lesen nicht relevant und das Schreiben per XAML bequem zu erledigen ist. Allerdings sind per C# - Syntax auch Eingriffe zur Laufzeit möglich, sodass wir einen Blick auf diese Option werfen. Die zu einer angefügten Eigenschaft gehörige statische Schreib- bzw. Lesemethode (z. B. **Grid.SetRow()** bzw. **Grid.GetRow()**) ruft ihrerseits beim betroffenen Objekt die von der Klasse **AvaloniaObject** geerbte Instanzmethode **SetValue()** bzw. **GetValue()** auf und übergibt dabei als ersten Parameter

---

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Abhängigkeitseigenschaften\Angefügte Eigenschaften

das **AttachedProperty<TValue>** - Objekt. Hier ist die statische **Grid**-Methode **SetRow()** zu sehen:

```
public static void SetRow(Control element, int value) {
    _ = element ?? throw new ArgumentNullException(nameof(element));
    element.SetValue(RowProperty, value);
}
```

Weil die **AvaloniaObject**-Methoden **SetValue()** und **GetValue()** öffentlich sind, kann man sie auch direkt aufrufen und somit die zur angefügten Eigenschaft gehörenden statischen Methoden übergehen. Im Beispiel ist für einen Schalter namens `button` die Anweisung

```
Grid.SetRow(button, 1);
```

äquivalent zu:

```
button.SetValue(Grid.RowProperty, 1);
```

Die folgende **Click**-Behandlungsmethode sorgt im Beispielprogramm für einen hüpfenden Schalter, der bei Betätigung zwischen den **Grid**-Zellen (0, 0) und (1, 1) wechselt:

```
void Button_Click(object sender, RoutedEventArgs e) {
    if (button.GetValue(Grid.RowProperty) == 1) {
        button.SetValue(Grid.RowProperty, 0);
        button.SetValue(Grid.ColumnProperty, 0);
    } else {
        button.SetValue(Grid.RowProperty, 1);
        button.SetValue(Grid.ColumnProperty, 1);
    }
}
```

Es gibt vermutlich auch ernsthafte Anwendungen für die Modifikation von angefügten Eigenschaften zur Laufzeit.

9 Datenbindung

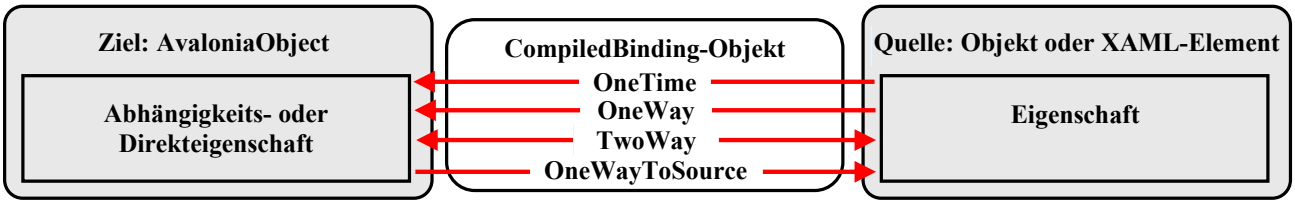
In sehr vielen Avalonia-Anwendungen spielt die zuverlässige und unaufwändige Kopplung zwischen

- Datenbeständen in Objekten
- und GUI-Elementen zur Anzeige und Modifikation der Daten

eine zentrale Rolle. Bei der dafür zuständigen Datenbindung sind beteiligt:

- Ziel
  Dazu kommt eine Abhängigkeitseigenschaft oder eine Direkteigenschaft in Frage (siehe Kapitel 8).
- Quelle
  Die für uns relevanten Quellen sind Objekte und XAML-Elemente (Steuerelemente).
- **CompiledBinding**-Objekt
  Ein Objekt der Klasse **CompiledBinding** (Namensraum **Avalonia.Data**) verwaltet die Datenbindung. Über seine Eigenschaft **Path** kann man den Pfad zur beteiligten Eigenschaft in der Quelle angeben oder abrufen. Seit Avalonia 12 kommt bei Verwendung der Markup-Erweiterung **Binding** (siehe Abschnitt 5.2.3.6) per Voreinstellung die empfehlenswerte *übersetzte* Datenbindung unter Verwendung der Klasse **CompiledBinding** zum Einsatz (siehe Abschnitt 9.2).

Die folgende Abbildung zeigt die für uns wichtigen Varianten der Datenbindung zusammen mit den Transportrichtungen (Werten der Enumeration **BindingMode** im Namensraum **Avalonia.Data**):[1]



Man kann die Transportrichtungen über die **Mode**-Eigenschaft des **CompiledBinding**-Objekts einstellen. Der voreingestellte Bindungsmodus gehört zu den Eigenschaften des **AvaloniaPropertyMetadata**-Objekts, das beim Registrieren einer Abhängigkeitseigenschaft entsteht (siehe Abschnitt 8.2). Die meistverwendete Voreinstellung **OneWay** ist bei benutzerveränderlichen Eigenschaften durch **TwoWay** ersetzt.

Von einem als Quelle fungierenden Objekt erfährt ein Datenbindungsziel über die **StyledElement**-Eigenschaft **DataContext**.

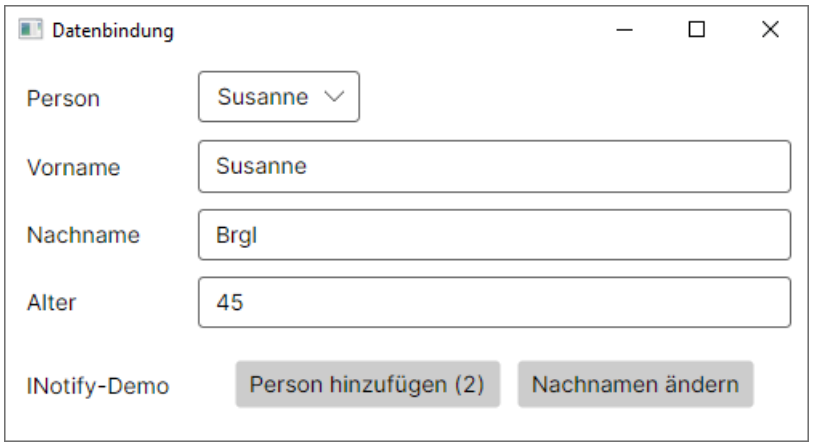
9.1 Personenverwaltungsbeispiel

Ein im weiteren Verlauf des Manuskripts mehrfach zur Demonstration verwendetes Programm, in das Ideen aus Troelsen & Japikse (2022, Kapitel 29) eingeflossen sind, präsentiert eine Liste mit Personen und zeigt den Listenumfang als Teil einer **Button**-Beschriftung an:[2]

---

[1] https://docs.avaloniaui.net/api/avalonia/data/bindingmode

[2] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
...\BspUeb\Datenbindung\Personenverwaltung



Die Klasse `Person` enthält die Eigenschaften `FirstName`, `LastName` und `Age`:

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

    public string FirstName { get; set { . . . } } = "NN";
    public string LastName { get; set { . . . } } = "NN";
    public int Age { get; set { . . . } } = 0;

    protected void OnPropertyChanged([CallerMemberName] string propertyName = "") {
        . . .
    }
}
```

Zur Benachrichtigung von Datenbindungszielen über Wertveränderungen implementiert die Klasse `Person` die Schnittstelle **INotifyPropertyChanged**. Im Zusammenhang mit der Erläuterung dieser Schnittstelle präsentieren wir im Abschnitt 9.6.1 den komplette Quellcode der Klasse `Person`.

Als Datenbindungsquelle fungiert ein Objekt der Klasse `Daten`, die eine Eigenschaft namens `Personen` mit einem Kollektionstyp enthält:

```
public class Daten {
  public ObservableCollection<Person> Personen { get; } = [];

  public Daten() {
    Personen.Add(new Person() { FirstName = "Susanne", LastName = "Brgl", Age = 45 });
    Personen.Add(new Person() { FirstName = "Randolf", LastName = "Schmitt", Age = 34 });
  }
}
```

Warum wir zur Verwaltung der Personen eine Liste mit dem konkretisierten generischen Typ **ObservableCollection<Person>** verwenden, ist im Abschnitt 9.6.2 zu erfahren.

Aus didaktischen Gründen verschieben wir den Erstkontakt mit dem MVVM-Anwendungsdesign auf das Kapitel 12 und verwenden eine möglichst einfache Anwendungsstruktur zur Einarbeitung in die Datenbindung, die Änderungsbenachrichtigung und die Validierung. Zur Datenverwaltung wählen wir aber eine Lösung, die sich später leicht in eine MVVM-Anwendung übernehmen lässt. Wir gehen also einen kleinen Schritt in Richtung auf das in der Praxis dominierende MVVM-Anwendungsdesign, wo sich die Fensterklassen auf die GUI-Logik beschränken.

In der **Application**-Methode **OnFrameworkInitializationCompleted()** (vgl. Abschnitt 4.7.1) erhält die **DataContext**-Eigenschaft des Fensterobjekts eine Referenz auf ein Objekt der Klasse `Daten`, sodass alle Steuerelemente diese Datenbindungsquelle nutzen können:

```
public override void OnFrameworkInitializationCompleted() {
    if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
        desktop.MainWindow = new MainWindow {
            DataContext = new Daten()
        };
    base.OnFrameworkInitializationCompleted();
}
```

9.2 Übersetzte Datenbindung

Seit der Version 12 verwendet Avalonia per Voreinstellung *übersetzte* Datenbindungen statt der vorher üblichen Auflösung von Datenbindungen durch Reflexion zur Laufzeit. Wegen der erheblichen Vorteile sollte man die neue Voreinstellung nicht ändern:

- Fehlerhafte Bindungsdeklarationen fallen beim Erstellen auf und sind leicht zu korrigieren.
- Weil die Reflexion eine zeitaufwändige Technik ist, verbessern übersetzte Datenbindungen die Leistung.

Um die Kontrolle und Realisation von Bindungen durch den Compiler zu ermöglichen, muss man den konkreten Typ des **DataContext**-Objekts, das den deklarierten Typ **Object?** besitzt, per **x:Data** – Attribut angeben, was oft im XAML-Wurzelelement eines Fensters geschieht, z. B.:

```
<Window xmlns="https://github.com/avaloniaui"
         . . .
        xmlns:local="using:Datenbindung"
        x:Class="Datenbindung.MainWindow"
        x:DataType="local:Daten"
        Title="Datenbindung" Width="500" SizeToContent="Height" MinHeight="240">
 . . .
</Window>
```

9.3 Bindung an eine Objekteigenschaft

Die beiden **Button**-Objekte des Personenverwaltungsprogramms stecken in einem Layoutcontainer vom Typ **StackPanel** (siehe Abschnitt 10.4), der sich zusammen mit einem **Label**-Objekt in einem inneren **Grid**-Container befindet:

```
<Grid ColumnDefinitions="110,*">
  <Label Grid.Column="0" Content="INotify-Demo" VerticalContentAlignment="Center" />
  <StackPanel Grid.Column="1" HorizontalAlignment="Center" Orientation="Horizontal" >
    <Button Click="BtnAddPerson_Click">
      <StackPanel Orientation="Horizontal">
        <TextBlock Text="Person hinzufügen (" />
        <TextBlock Text="{Binding Path=Personen.Count}" />
        <TextBlock Text=")" />
      </StackPanel>
    </Button>
    <Button Content="Nachnamen ändern" Margin="10,0,0,0"
            Click="BtnChangeLastName_Click" />
  </StackPanel>
</Grid>
```

Weil die Beschriftung des linken Schalters neben dem durch Datenbindung ermittelten Listenumfang auch Text anzeigen soll, weisen wir der **Button**-Inhaltseigenschaft **Content** einen **StackPanel**-Container mit Inhalt zu, und die Abhängigkeitseigenschaft **Text** eines **TextBlock**-Elements ist schließlich das Datenbindungsziel (vgl. Abschnitt 5.2.3.6 zur Markup-Syntax):

```
<TextBlock Text="{Binding Path=Personen.Count}" />
```

Die automatische, in der Container-Verschachtelung nach außen voranschreitende Suche nach einer versorgten **DataContext**-Eigenschaft endet beim Fensterobjekt (siehe Abschnitt 9.1).

In der **Binding**-Syntax hat das **Path**-Attribut, das die zu verwendende Eigenschaft des **DataContext**-Objekts angibt, die mit Abstand häufigsten Auftritte. Der zwei-segmentige Pfad des Beispiels zeigt auf die **Count**-Eigenschaft des Kollektionsobjekts `Personen`. Ist der Pfad das erste **Binding**-Attribut, dann kann man das Schlüsselwort weglassen, z. B.:

```
<TextBlock Text="{Binding Personen.Count}" />
```

Alternativ zur kompakten, allgemein bevorzugten und im Manuskript ausschließlich verwendeten Markup-Syntax (vgl. Abschnitt 5.2.3.6) kann man zur **Binding**-Deklaration auch die Eigenschafts-elementsyntax verwenden. Das **TextBlock**-Element

```
<TextBlock Text="{Binding Personen.Count}" />
```

lässt sich auch so deklarieren:

```
<TextBlock>
    <TextBlock.Text>
        <Binding Path="Personen.Count" />
    </TextBlock.Text>
</TextBlock>
```

Der oben beschriebene innere **Grid**-Container mit den **Button**-Objekten bildet die dritte Schicht des Wurzel-Containers vom Typ **StackPanel**:

```
<Window xmlns=https://github.com/avaloniaui . . . >

    <StackPanel Margin="10">

      <Grid ColumnDefinitions="110,*">
          . . .
      </Grid>

      <Grid ColumnDefinitions="110,*" RowDefinitions="Auto,Auto,Auto" . . . >
          . . .
      </Grid>

      <Grid ColumnDefinitions="110,*">
          . . .
      </Grid>
    </StackPanel>

</Window>
```

An der Flexibilität einer XAML-Fensterdeklaration hat die Möglichkeit zum Verschachteln von Layout-Managern einen wesentlichen Anteil.

9.4 Bindung an eine Kollektion mit komplexem Elementtyp

Um ein **ItemsControl** – Steuerelement (z. B. **ListBox**, **ComboBox**) an eine Kollektion zu binden, weist man die Kollektion der **ItemsSource** - Eigenschaft zu, was im aktuellen Personenverwaltungsprogramm mit dem **ComboBox**-Steuerelement im XAML-Code per **Binding**-Objekt geschieht:

```
<Grid ColumnDefinitions="110,*">
    <Label Grid.Column="0" Content="Person" VerticalContentAlignment="Center" />
    <ComboBox Name="comboBox" ItemsSource="{Binding Personen}" Grid.Column="1"
              DisplayMemberBinding="{Binding FirstName}"
              PlaceholderText="Person wählen ..." Margin="0,0,0,10"/>
</Grid>
```

Das **Binding**-Objekt benötigt eine versorgte **DataContext**-Eigenschaft mit einer Kollektion als Referenzziel. Die automatische, in der Container-Verschachtelung nach außen voranschreitende Suche

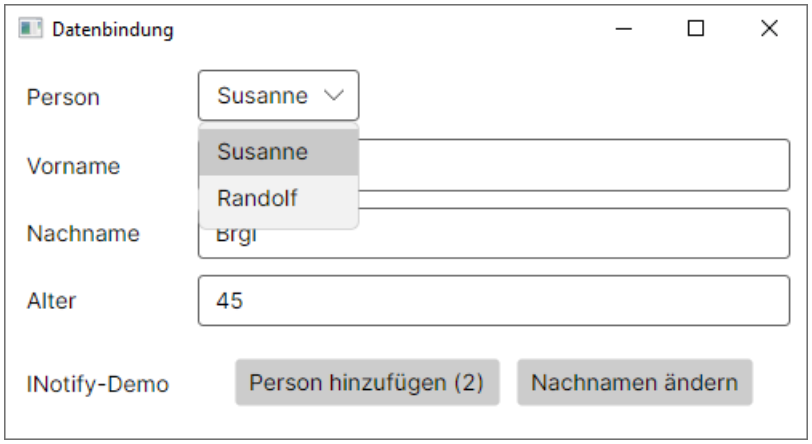
endet beim Fensterobjekt. Dessen **DataContext**-Eigenschaft hat in der **Application**-Methode **OnFrameworkInitializationCompleted()** eine Referenz auf ein Objekt der Klasse `Daten`, mit der Eigenschaft `Personen` vom Typ `ObservableCollection<Person>` erhalten (siehe Abschnitt 9.1).

Es kommt eine *unidirektionale* Verbindung zustande, sodass sich die Zusammensetzung der Kollektion *nicht* über das **ComboBox**-Steuerelement ändern lässt. Andererseits informiert eine beobachtbare Kollektion vom Typ **ObservableCollection<T>** ein angebundenes Steuerelement über eine Änderung der Zusammensetzung (siehe Abschnitt 9.6.2).

Die **ComboBox** zeigt aufgrund des Attributs

```
DisplayMemberBinding="{Binding FirstName}"
```

zu den Kollektionselementen die Vornamen an:



Das Programm präsentiert in einem **Master-Details – Design** die Eigenschaften der im **ComboBox**-Steuerelement ausgewählten Person durch **TextBox**-Objekte, die sich zusammen mit **Label**-Objekten in einem **Grid**-Container befinden, der die zweite Schicht des Wurzel-Containers vom Typ **StackPanel** belegt:

```
<Grid ColumnDefinitions="110,*" RowDefinitions="Auto,Auto,Auto"
      VerticalAlignment="Center" RowSpacing="10" Margin="0,0,0,20"
      DataContext="{Binding #comboBox.SelectedItem}" x:DataType="local:Person" >
    <Label Grid.Column="0" Grid.Row="0" Content="Vorname"
           VerticalContentAlignment="Center" />
    <TextBox Grid.Column="1" Grid.Row="0" Text="{Binding FirstName}" />
    <Label Grid.Column="0" Grid.Row="1" Content="Nachname"
           VerticalContentAlignment="Center" />
    <TextBox Grid.Column="1" Grid.Row="1" Text="{Binding LastName}" />
    <Label Grid.Column="0" Grid.Row="2" Content="Alter"
           VerticalContentAlignment="Center" />
    <TextBox Grid.Column="1" Grid.Row="2" Text="{Binding Age}" />
</Grid>
```

Zur Versorgung der **TextBox**-Objekte per Datenbindung definieren wir als Eigenschaft des Layoutcontainers einen neuen **DataContext** mit dem **ComboBox**-Steuerelement als Quelle und einem Pfad zur gewählten Person:

```
DataContext="{Binding #comboBox.SelectedItem}"
```

Zu dieser **Binding**-Syntax ist anzumerken:[1]

[1] https://github.com/AvaloniaUI/Avalonia/blob/main/src/Avalonia.Base/AvaloniaObject.cs

- Das als Quelle fungierende Steuerelement benötigt einen Namen.
- Das Doppelkreuz vor dem Namen steht für das Attribut **ElementName** und spricht ein XAML-Element an. Es ermöglicht eine Vereinfachung der folgenden Syntax:

  ```
  {Binding SelectedItem, ElementName=comboBox}
  ```

  In WPF dient das Attribut **RelativeSource** zur Ansprache eines anderen Steuerelements als Datenbindungsquelle. Dieses Attribut ist auch in Avalonia zulässig, hat aber im Vergleich zu **ElementName** (bzw. #) eine leicht abweichende Bedeutung: **ElementName** arbeitet mit dem *logischen* Baum des Fensters bestehend aus den im XAML-Code deklarierten Elementen, während **RealtiveSource** den feiner aufgelösten *visuellen* Baum mit den darzustellenden Elementen verwendet.[1] Zu einem **Button**-Objekt des logischen Baums gehören z. B. im visuellen Baum ein **Border**- und ein **TextBlock**-Objekt. Im Manuskript können wir uns auf den logischen Baum beschränken.
- Wie die Langform der Bindung an ein Steuerelement zeigt, sind **Binding**-Attribute im Unterschied zu den Attributen eines XAML-Startelements durch Kommata zu trennen.
- **Binding**-Objekte im Details-Bereich des Fensters sind relativ zum spezifischen **DataContext** des **Grid**-Containers zu verstehen und müssen nur den Pfad zur jeweiligen Eigenschaft enthalten, z. B.:

  ```
  {Binding FirstName}
  ```

Weil ...

- in Avalonia12 die übersetzte Datenbindung voreingestellt ist (vgl. Abschnitt 9.2),
- und die **ComboBox**-Eigenschaft **SelectedItem** den deklarierten Datentyp **Object?** besitzt,

ist auch zum **DataContext** des **Grid**-Containers ein Datentyp anzugeben:

```
DataContext="{Binding #comboBox.SelectedItem}" x:DataType="local:Person"
```

9.5 Bindung an ein anderes Steuerelement

Im Abschnitt 9.4 haben wir bereits **TextBox**-Eigenschaften an **ComboBox**-Eigenschaften gebunden. Das geschah in einem Fenster mit Master-Details – Darstellung einer Kollektion, und im Vergleich zu dieser speziellen Konstellation präsentieren wir nun eine typische Element-zu-Element – Verbindung, wobei ...

- kein **DataContext**-Objekt erforderlich ist,
- ohne Aufwand unsererseits eine Änderungsbenachrichtigung stattfindet,
- und eine die Schnittstelle **IValueConverter** implementierende Klasse für Typ- und Formatanpassungen sorgt.

Zur Realisation der Änderungsbenachrichtigung muss eine Klasse die Schnittstelle **INotifyPropertyChanged** implementieren (siehe Abschnitt 9.6.1). Dass diese Kompetenz bei Steuerelementen generell verfügbar ist, liegt an der Implementation der Schnittstelle **INotifyPropertyChanged** durch die Basisklasse **AvaloniaObject** (siehe Abschnitt 4.2 zur Klassenhierarchie der Avalonia-Steuerelemente):[2]

---

[1] https://docs.avaloniaui.net/docs/fundamentals/visual-and-logical-trees

[2] Über weitere Details zur **Binding**-Syntax (insbesondere zum Zugriff auf Steuerelemente) informiert die Webseite: https://docs.avaloniaui.net/docs/data-binding/data-binding-syntax

```
public class AvaloniaObject : IAvaloniaObjectDebug, INotifyPropertyChanged {
    private readonly ValueStore _values;
     . . .
}
```

Das folgende Programm konvertiert den in eine **TextBox** eingetragenen DM-Betrag *spontan* in einen Euro-Betrag, den ein **Label** anzeigt.[1] Im Unterschied zu dem im Abschnitt 2.4 erstellten Währungskonverter ist kein Schalter zur Anforderung der Konvertierung erforderlich:



Für die Struktur des Fensters sorgen:

- ein **Grid**-Container mit zwei Spalten
- die für jeweils eine **Grid**-Spalte zuständigen **StackPanel**-Container mit zwei nebeneinander angeordneten Steuerelementen

Von einem **DataContext**-Objekt ist im XAML-Code nichts zu sehen, weil ein solches Objekt bei der Bindung an ein anderes Steuerelement in der Regel nicht erforderlich ist.

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="DmToEuroBinding.MainWindow"
        Title="DM &#x27a0; Euro" Height="150" Width="400" Loaded="Window_Loaded">
    <Grid Margin="10" ColumnDefinitions="*,*">
        <StackPanel Orientation="Horizontal" VerticalAlignment="Center">
            <Label Content="DM:" VerticalContentAlignment="Center" />
            <TextBox Name="tbEingabe" VerticalContentAlignment="Center"
                     MinWidth="120" TextAlignment="Right" />
        </StackPanel>
        <StackPanel Orientation="Horizontal" Grid.Column="1"
                    VerticalAlignment="Center">
            <Label Content="Euro:" VerticalContentAlignment="Center" />
            <Label BorderBrush="Black" BorderThickness="1" MinWidth="120"
                   HorizontalContentAlignment="Right"
                   Content="{Binding #tbEingabe.Text,
                             Converter={StaticResource DmToEuroConverter}}" />
        </StackPanel>
    </Grid>
</Window>
```

Relevant ist vor allem das **Label**-Steuerelement auf der rechten Seite des horizontalen **StackPanel**-Containers in der zweiten Spalte des **Grid**-Containers. Seine **Content**-Eigenschaft wird über ein **Binding**-Objekt mit einer Zeichenfolge versorgt:

- Bei Verwendung einer expliziten **Binding**-Syntax
  `Binding ElementName=tbEingabe, Path=Text`
  benennt das Attribut **ElementName** als Quelle das **TextBox**-Element mit dem Namen `tbEingabe`, und das Attribut **Path** benennt die zu verwendende Eigenschaft der Quelle.

---

[1] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
...\BspUeb\Datenbindung\DmToEuroBinding

- Statt die Attributnamen anzugeben, verwenden wir (wie im Abschnitt 9.4 bei der Master-Details – Lösung zur Bindung von Steuerelementen an eine Kollektion) eine kompakte Syntax: Das Doppelkreuz vor dem Namen des Steuerelements steht für das Attribut **ElementName**, und nach einem Punkt kann man das **Path**-Attribut ohne Nennung seines Namens versorgen. Damit vereinfacht sich die **Binding**-Syntax zu:

```
Binding #tbEingabe.Text
```

In der **Content**-Eigenschaft des Ziels landet aber keine Kopie der **Text**-Eigenschaft des Quellelements, sondern das Ergebnis einer Konvertierung durch die Klasse `DmToEuroConverter`, die das Interface **IValueConverter** implementiert, das die Methoden **Convert()** und **ConvertBack()** vorschreibt:

```
using Avalonia.Data.Converters;
using System;
using System.Globalization;

namespace DmToEuroBinding;

public class DmToEuroConverter : IValueConverter {
    const double factor = 1.95583;
    public object Convert(object? value, Type targetType,
                          object? parameter, CultureInfo culture) {
        if (value is string instr) {
            if (Decimal.TryParse(instr, NumberStyles.Any, culture, out decimal dm)) {
                decimal euro = dm / (decimal)factor;
                return euro.ToString("0.00", culture);
            } else
                return "";
        } else
            return "";
    }

    public object ConvertBack(object? value, Type targetType,
                              object? parameter, CultureInfo culture) {
        throw new NotImplementedException();
    }
}
```

Die `DmToEuroConverter`-Methode **Convert()** liefert ...

- das Ergebnis einer erfolgreichen Konvertierung mit zwei Nachkommastellen
- oder eine leere Zeichenfolge.

Wir deklarieren in der Datei **App.axaml** ein Objekt der Klasse `DmToEuroConverter` als Ressource

```
<Application xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="DmToEuroBinding.App"
             xmlns:local="using:DmToEuroBinding"
             RequestedThemeVariant="Default">

    <Application.Resources>
        <local:DmToEuroConverter x:Key="DmToEuroConverter"/>
    </Application.Resources>

    <Application.Styles>
        <FluentTheme />
    </Application.Styles>

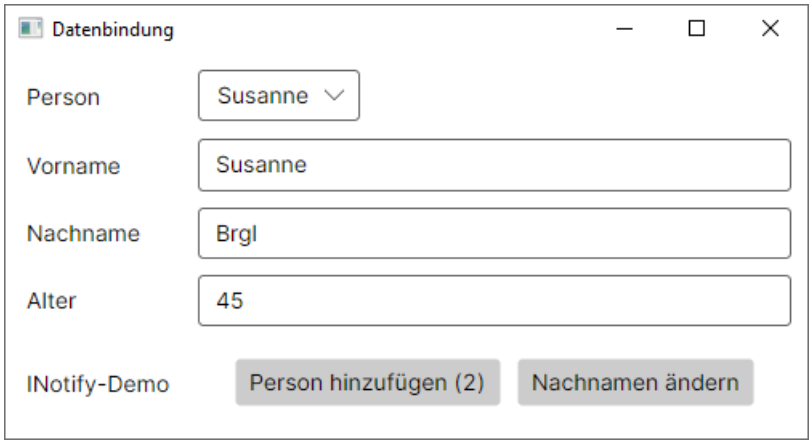
</Application>
```

und weisen diese statische Ressource im XAML-Code zum Anwendungsfenster dem **Binding**-Attribut **Converter** zu:

```
<Label BorderBrush="Black" BorderThickness="1" MinWidth="120"
       HorizontalContentAlignment="Right"
       Content="{Binding #tbEingabe.Text,
                 Converter={StaticResource DmToEuroConverter}}" />
```

9.6 Änderungsbenachrichtigungen

Das im Abschnitt 9.1 vorgestellte Personenverwaltungsprogramm enthält zwei Schalter zum Ändern von verbundenen Daten, um die Effekte von Benachrichtigungen über geänderte Objekteigenschaften bzw. Kollektionsinhalte zu demonstrieren (siehe Abschnitte 9.6.1 bzw. 9.6.2):



Die **Click**-Behandlungsmethoden sind in der Code Behind – Datei implementiert:[1]

```
using Avalonia.Controls;
using Avalonia.Interactivity;

namespace Datenbindung;

public partial class MainWindow : Window {

  public MainWindow() {
    InitializeComponent();
  }

  private void BtnChangeLastName_Click(object sender, RoutedEventArgs e) {
    if (DataContext is Daten daten)
       daten.Personen[0].LastName = "Rempremerding";
  }

  private void BtnAddPerson_Click(object sender, RoutedEventArgs e) {
   if (DataContext is Daten daten)
      daten.Personen.Add(new Person() { LastName = "NN",FirstName = "NN",Age = 0 });
  }
}
```

In einer kleinen Anwendung ist es akzeptabel, die Anwendungslogik in der Code-Behind – Datei zur Hauptfensterklasse unterzubringen. Demgegenüber sind in einer professionellen Anwendung die Code-Behind – Dateien fast leer (siehe Kapitel 12 über die MVVM-Anwendungsarchitektur).

---

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Datenbindung\Personenverwaltung

9.6.1 INotifyPropertyChanged

Die rechte Schaltfläche im Personenverwaltungsprogramm bewirkt eine Änderung des Nachnamens von Susanne. Wenn diese `Person`-Eigenschaft während der „externen“ Änderung in einem **TextBox**-Steuerelement sichtbar ist, dann sollte sich die Anzeige ändern und somit aktuell bleiben. Dazu muss der Datentyp `Person` die Schnittstelle **INotifyPropertyChanged** (Namensraum **System.ComponentModel**) implementieren, die das im Fall einer Eigenschaftsänderung zu feuernde CLR-Ereignis **PropertyChanged** diktiert:

```
public interface INotifyPropertyChanged {
   event PropertyChangedEventHandler? PropertyChanged;
}
```

Der Delegatentyp **PropertyChangedEventHandler** beschreibt die Anforderungen an eine **PropertyChanged**-Behandlungsmethode:

```
public delegate void PropertyChangedEventHandler(Object? sender,
                                                  PropertyChangedEventArgs e)
```

Dem Konstruktor eines Ereignisbeschreibungsobjekts vom Typ **PropertyChangedEventArgs**

```
public PropertyChangedEventArgs(String? propertyName)
```

ist der Name der betroffenen Eigenschaft zu übergeben.

Für die Verarbeitung der **PropertyChanged**-Ereignisse sorgt die Avalonia-Datenbindungstechnik. Unser Beitrag besteht in der Implementation der Schnittstelle **INotifyPropertyChanged** in der Klasse `Person`, die nun endlich zu sehen ist:[1]

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace Datenbindung;

public class Person : INotifyPropertyChanged {
    public event PropertyChangedEventHandler? PropertyChanged;

    public string FirstName {
        get;
        set {
            if (value == field)
                return;
            field = value;
            OnPropertyChanged();
        }
    } = "NN";

    public string LastName {
        get;
        set {
            if (value == field)
                return;
            field = value;
            OnPropertyChanged();
        }
    } = "NN";
```

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Datenbindung\Personenverwaltung

```
        public int Age {
            get;
            set {
                if (value == field)
                    return;
                field = value;
                OnPropertyChanged();
            }
        } = 0;

        protected void OnPropertyChanged([CallerMemberName] string propertyName = "") {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }
    }
```

Die in den **set**-Zugriffen der Eigenschaften aufgerufene Methode `OnPropertyChanged()` macht es einfach, bei einer Eigenschaftsänderung per **Invoke()** – Aufruf das **PropertyChanged**-Ereignis zu feuern.[1] Wir verwenden die übliche Lösung zur vereinfachten Übergabe der Eigenschaftsnamen an die Methode `OnPropertyChanged()`. Der Parameter der Methode ist durch das eckig eingeklammerte Attribut **CallerMemberName** dekoriert. Er erhält daher den Namen der Eigenschaft, in deren Definition sich der Methodenaufruf befindet.[2] Die Eigenschaftsnamen als Zeichenfolgenliterale zu übergeben, ist wegen der Gefahr von Tippfehlern nicht zu empfehlen. Man kann ihre Verwendung auch durch den **nameof**-Operator vermeiden, z. B.

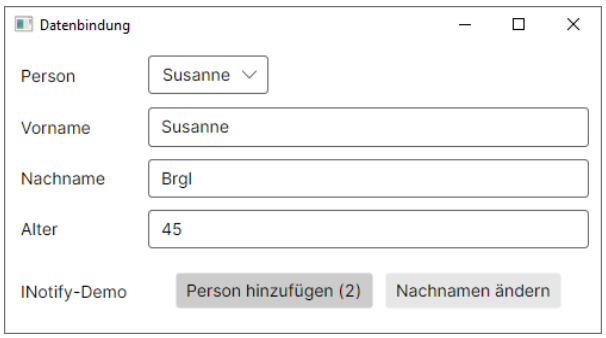
```
OnPropertyChanged(nameof(FirstName));
```

Ein Klick auf den Schalter **Nachnamen ändern** beeinflusst nicht direkt das **TextBox**-Steuerelement, sondern die `LastName`-Eigenschaft des `Personen`-Elements mit der Indexnummer 0:

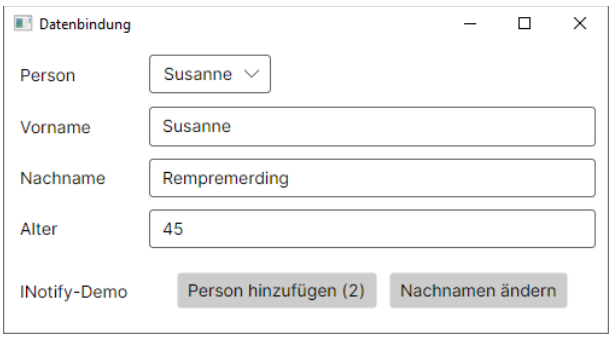
```
daten.Personen[0].LastName = "Rempremerding";
```

Weil Avalonia die **PropertyChanged**-Ereignisse der `Person`-Instanzen abonniert, kann das Framework die **Text**-Abhängigkeitseigenschaften der **TextBox**-Steuerelemente aktualisieren:

Vor dem Klick auf **Nachnamen ändern**



Nach dem Klick auf **Nachnamen ändern**



---

[1] Man darf ein Ereignis nur dann mit **Invoke()** feuern, wenn mindestens eine Behandlungsmethode registriert, also das zum Ereignis gehörende Delegatenobjekt von **null** verschieden ist. Der **null**-bedingte Operator am Ende des Eigenschaftsnamens `PropertyChanged` vereinfacht die **null**-Überprüfung im Vergleich zur einer **if**-Anweisung (siehe z. B. Abschnitt 15.1.5.2 in Baltes-Götz & Frischknecht 2026):

```
if (PropertyChanged != null)
    PropertyChanged.Invoke(this, new PropertyChangedEventArgs(propertyName));
```

[2] https://learn.microsoft.com/en-us/dotnet/api/system.runtime.compilerservices.callermembernameattribute

9.6.2 INotifyCollectionChanged

Die linke Schaltfläche im Personenverwaltungsprogramm fügt eine Instanz in die vom **ComboBox**-Steuerelement dargestellte `Personen`-Liste ein. Außerdem zeigt die Beschriftung dieses Schalters aufgrund einer Datenbindung (siehe Abschnitt 9.1)

```
<TextBlock Text="{Binding Personen.Count}" />
```

die Anzahl der Listenelemente an. Damit diese Anzeige bei einer Listenerweiterung aktuell bleibt, muss die Kollektion die Schnittstelle **INotifyCollectionChanged** (Namensraum **System.Collections.Specialized**) implementieren, die das im Fall einer veränderten Listenzusammensetzung zu feuernde Ereignis **CollectionChanged** verlangt:

```
public interface INotifyCollectionChanged {
   event NotifyCollectionChangedEventHandler? CollectionChanged;
}
```

Ein **NotifyCollectionChangedEventHandler** zur Behandlung von **CollectionChanged**-Ereignissen

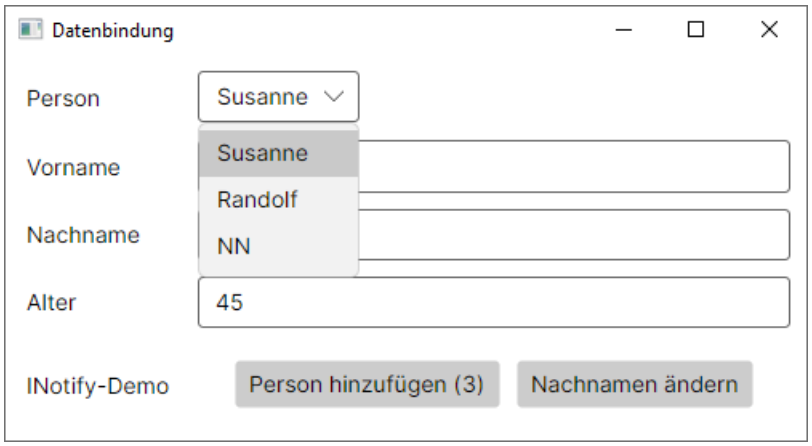
public delegate void NotifyCollectionChangedEventHandler(Object? *sender*, **NotifyCollectionChangedEventArgs** *e*)

erhält als zweiten Parameter ein **NotifyCollectionChangedEventArgs**-Objekt, das die eingetretene Änderung in der Listenzusammensetzung beschreibt. Es sind etliche Änderungsarten möglich (Ergänzung, Entfernung, Ersetzung, Verschiebung), die zudem ein einzelnes Element oder mehrere Elemente betreffen können. Um alle Fälle abzudecken, benötigt die Ereignisbeschreibungsklasse 11 Konstruktorüberladungen.

Man kann sich in vielen Fällen den Aufwand der Ereignisbeschreibung und -auslösung ersparen durch die Verwendung der Klasse **ObservableCollection<T>** (Namensraum **System.Collections.ObjectModel**), die das Interface **INotifyCollectionChanged** implementiert. Diese Klasse kommt auch im Personenverwaltungsprogramm zum Einsatz:

```
public ObservableCollection<Person> Personen { get; } = [];
```

Sie sorgt zusammen mit der Avalonia-Datenbindung und -Änderungsbenachrichtigung dafür, dass nach einem Klick auf den linken Schalter des Personenverwaltungsprogramms seine Beschriftung die neue Listenstärke anzeigt. Außerdem ist der neue Fall in der aufgeklappten **ComboBox**-Liste zu sehen:



9.7 Datenvalidierung

Die Avalonia-Datenbindung unterstützt auch die Eingabevalidierung inklusive der Benachrichtigung der Benutzer über die beim Aktualisieren der Datenbindungsquelle aufgetretenen Fehler (möglich bei den **BindingMode**-Werten **TwoWay** und **OneWayToSource**). Durch das Implementieren der Schnittstelle **INotifyDataErrorInfo** (Namensraum **System.ComponentModel**) kann eine Klasse Validierungsfehler durch Ereignisse signalisieren:

```
public interface INotifyDataErrorInfo {
   event EventHandler<DataErrorsChangedEventArgs>? ErrorsChanged;
   bool HasErrors { get; }
   IEnumerable GetErrors(String? propertyName);
}
```

Die Schnittstelle verlangt ...

- das bei einer Änderung im Fehlerstatus irgendeiner Quelleigenschaft zu feuernde Ereignis **ErrorsChanged**,
- die **bool**-Eigenschaft **HasErrors** mit der Information darüber, ob bei irgendeiner Quelleigenschaft ein Fehler vorliegt,
- die Methode **GetErrors()**, die eine Liste aller Fehler zu der per Parameter bestimmten Quelleigenschaft liefert.

Wir implementieren in der seit dem Abschnitt 9.1 zur Demonstration der Datenbindung dienenden Klasse `Person` zusätzlich zu **INotifyPropertyChanged** auch die Schnittstelle **INotifyDataErrorInfo**:[1]

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Runtime.CompilerServices;

namespace Validierung;

public class Person : INotifyPropertyChanged, INotifyDataErrorInfo {
    public event PropertyChangedEventHandler? PropertyChanged;
    public event EventHandler<DataErrorsChangedEventArgs>? ErrorsChanged;

    readonly Dictionary<string,List<string>> errors = [];

    public string FirstName {
        get; set {
            if (value == field)
                return;
            field = value;
            ValidateName(value);
            OnPropertyChanged();
        }
    } = "NN";
```

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Validierung
In die **INotifyDataErrorInfo**-Implementierung sind Ideen von Kazimierz Matyaszek eingeflossen, siehe:
https://kmatyaszek.github.io/wpf%20validation/2019/03/13/wpf-validation-using-inotifydataerrorinfo.html

```
    public string LastName {
        get; set {
            if (value == field)
                return;
            field = value;
            ValidateName(value);
            OnPropertyChanged();
        }
    } = "NN";

    public int Age {
        get; set {
            if (value == field)
                return;
            field = value;
            ValidateAge(nameof(Age), value);
            OnPropertyChanged();
        }
    } = 0;

    public bool HasErrors => errors.Count > 0;

    public IEnumerable GetErrors(string? propertyName) {
       return errors.TryGetValue(propertyName,out List<string>? errorList) ? errorList : null;
    }

    void OnErrorsChanged(string propertyName) {
        ErrorsChanged?.Invoke(this, new DataErrorsChangedEventArgs(propertyName));
    }

    void ClearErrors(string propertyName) {
        if (!errors.ContainsKey(propertyName)) {
            return;
        }
        errors.Remove(propertyName);
        OnErrorsChanged(propertyName);
    }

    private void AddError(string propertyName, string error) {
        if (!errors.ContainsKey(propertyName))
            errors[propertyName] = new List<string>();

        if (!errors[propertyName].Contains(error)) {
            errors[propertyName].Add(error);
            OnErrorsChanged(propertyName);
        }
    }

    void ValidateName(string value, [CallerMemberName] string propertyName = "") {
        ClearErrors(propertyName);
        if (string.IsNullOrWhiteSpace(value))
            AddError(propertyName, "Der Name darf nicht leer sein.");
        if (!value.All(c => char.IsLetter(c)))
            AddError(propertyName, "Der Name darf nur Buchstaben enthalten.");
    }

    void ValidateAge(string propertyName, int value) {
        ClearErrors(propertyName);
        if (value < 0 || value > 120)
            AddError(propertyName, "Zulässige Altersangaben: 0-120");
    }

    protected void OnPropertyChanged([CallerMemberName] string propertyName = "") {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Zur Aufbewahrung von Validierungsfehlermeldungen zu den Quelleigenschaften `FirstName`, `LastName` und `Age` dient die **Dictionary<String,List<String>>** – Kollektion `errors`:

```
readonly Dictionary<string,List<string>> errors = [];
```

Weil prinzipiell pro Quelleigenschaft *mehrere* Fehlermeldungen erlaubt sind, ist der **TValue**-Typformalparameter der generischen Kollektion **Dictionary<TKey,TValue>** durch **List<String>** konkretisiert. Im Beispiel kann dieser Fall aber nicht auftreten.

Die **set**-Zugriffsblöcke der Eigenschaften rufen eine Methode zur Durchführung der Validierung auf (z. B. `ValidateName()`, zum Attribut **CallerMemberName** für den Parameter `propertyName` siehe Abschnitt 9.6.1). Zu Beginn der Validierung prüft die Methode `ClearErrors()`, ob in der `errors`-Kollektion Fehlermeldungen zur betroffenen Quelleigenschaft vorhanden sind. In diesem Fall ...

- löscht `ClearErrors()` diese Fehlermeldungen
- und löst mit Hilfe der Methode `OnErrorsChanged()` das Ereignis **ErrorsChanged** aus.

Anschließend prüfen die Validierungsmethoden die Einhaltung der Regeln und fügen bei einem festgestellten Verstoß eine Fehlermeldung in den `errors`-Container ein, z. B.:

```
void ValidateName(string value, [CallerMemberName] string propertyName = "") {
    ClearErrors(propertyName);
    if (string.IsNullOrWhiteSpace(value))
        AddError(propertyName, "Der Name darf nicht leer sein.");
    if (!value.All(c => char.IsLetter(c)))
        AddError(propertyName, "Der Name darf nur Buchstaben enthalten.");
}
```

Dabei kommt die Methode `AddError()` zum Einsatz:

- Die Methode erstellt nötigenfalls zur Quelleigenschaft ein neues Element in `errors`.
- Sofern die Fehlermeldung in der Liste zur Quelleigenschaft noch nicht vorhanden ist, ...
  - fügt `AddErrors()` die Fehlermeldung dort ein
  - und löst mit Hilfe der Methode `OnErrorsChanged()` das Ereignis **ErrorsChanged** aus.

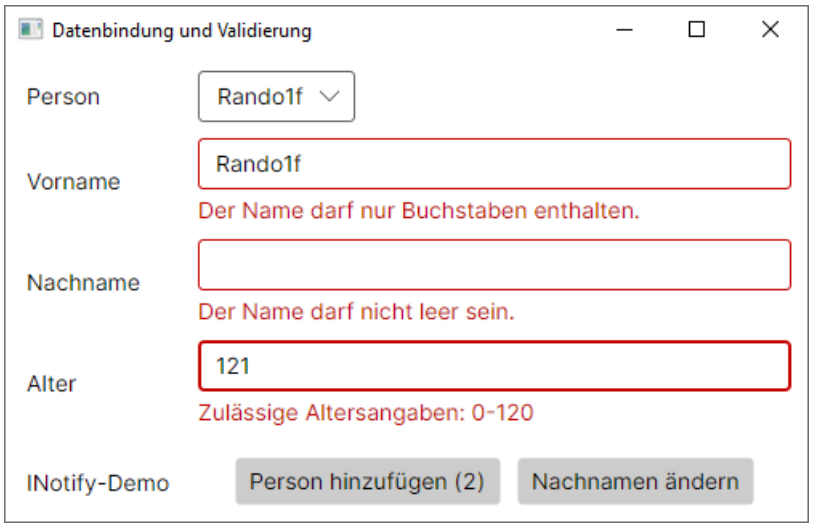
Die restlichen **INotifyPropertyChanged**-Implementierungspflichten sind leicht zu erfüllen:

- Um den Wert der **get-only** - Eigenschaft **HasErrors** zu ermitteln, muss man lediglich prüfen, ob der `errors`-Container mindestens einen Eintrag hat.
- Die Methode **GetErrors()** liefert zur angefragten Quelleigenschaft die Fehlerliste oder **null**, wenn keine Fehlermeldungen vorhanden sind.

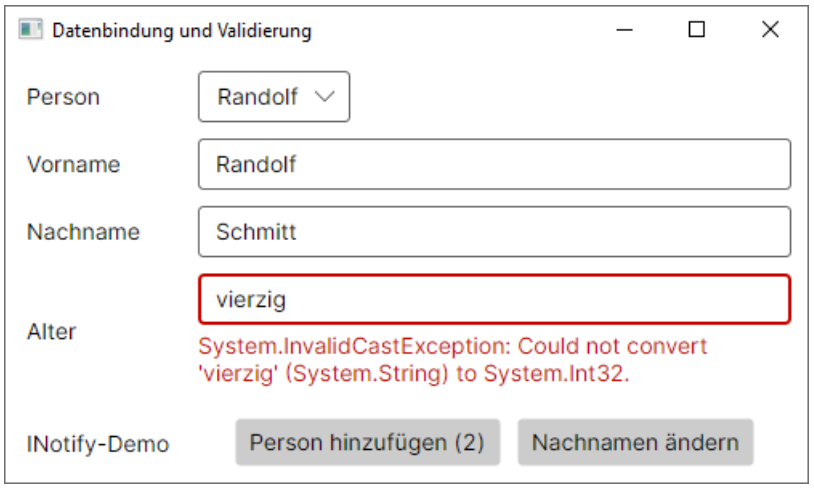
In der XAML-Deklaration des Anwendungsfensters sind zur Unterstützung der Validierung fast keine Änderungen erforderlich. Man sollte allerdings per **SizeToContent**-Attribut die Anpassung der Fensterhöhe an den Inhalt erlauben,

```
<Window xmlns="https://github.com/avaloniaui"
        . . .
      Title="Datenbindung und Validierung" Width="500"
      SizeToContent="Height">
 . . .
</Window>
```

damit Avalonia nötigenfalls Fehlermeldungen einfügen kann, z. B.:



Beim Validieren auftretende Ausnahmefehler zeigt Avalonia ungefiltert an:



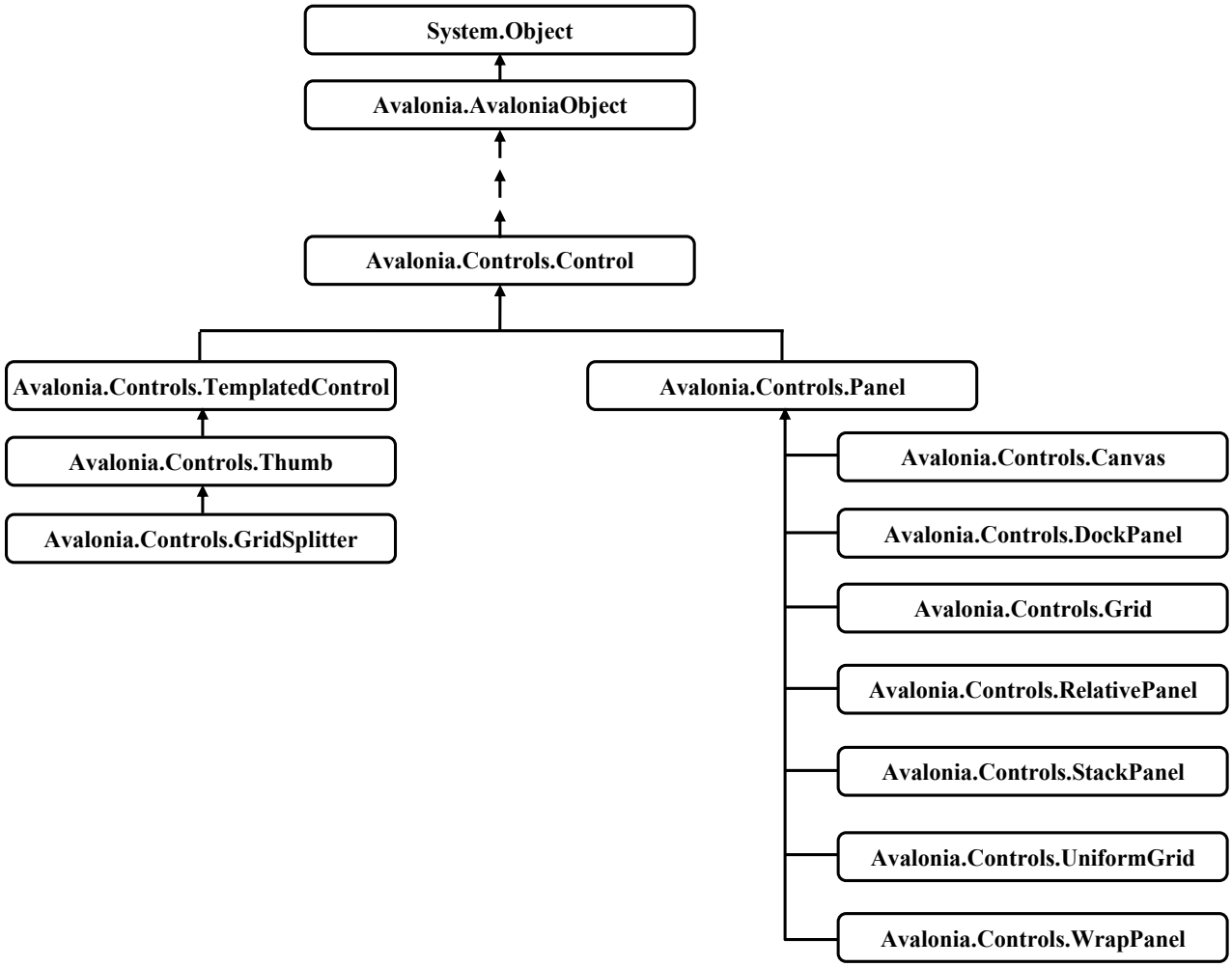
Wie man für Benutzer besser verständliche Meldungen produziert, erläutert der Abschnitt 12.4.4.6.

Es ist zu beachten, dass die beschriebene Avalonia-Validierungstechnik erst *nach* dem Speichern eine Überprüfung vornimmt, sofern keine Typunverträglichkeit das Speichern verhindert (z. B. bei der Altersangabe *dreizehn*). Im Abschnitt 12.4.2 ist eine *abweisende* Validierung zu sehen (ohne Speicherung ungeprüfter Werte).

10 Layoutcontainer

Zur Verwaltung der mehr oder weniger zahlreichen Steuerelemente in einem Fenster dienen Layoutcontainer mit diversen Optionen zur festen oder dynamischen Bestimmung der Positionen und Größen der Elemente.

Alle Avalonia-Layoutcontainer stammen von der Klasse **Avalonia.Controls.Panel** ab. Die folgende Abbildung zeigt die meistbenutzten Klassen mit ihren Abstammungsverhältnissen:



Von den Mitgliedern, die ein Layoutcontainer von seiner Basisklasse **Panel** erbt, ist besonders die Eigenschaft **Children** zu erwähnen, die auf ein Objekt der von **AvaloniaList<Control>** abstammenden Klasse **Controls** zeigt, das die im Container enthaltenen **Control**-Objekte verwaltet. Es ist die Inhaltseigenschaft der Klasse **Panel** (vgl. Abschnitt 5.2.3.3):

```
public class Panel : Control, IChildIndexProvider     {
    . . .
    [Content]
    public Controls Children { get; } = new Controls();
    . . .
}
```

10.1 Panel

Im Beispiel aus dem Abschnitt 2.4



sind alle Positionen und Größen fixiert. Es kommt ein *Wurzel-Layoutcontainer* (engl.: *root layout container*) aus Klasse **Panel** (Namensraum **Avalonia.Controls**) zum Einsatz. Die Größen der Steuerelemente sind durch Attribute (**Width**, **Height**) oder durch ihre Inhalte festgelegt. Zur Bestimmung der Positionen dienen die Abstände der Steuerelemente vom linken und oberen Container-Rand:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="DmToEuro.MainWindow"
        Title="DM → Euro" Width="500" Height="180" CanResize="False"
        Loaded="Window_Loaded">
  <Panel>
    <Label Content="DM:" HorizontalAlignment="Left" VerticalAlignment="Top"
           Margin="40,40,0,0" Width="60" Height="30" />
    <TextBox Name="eingabe" HorizontalAlignment="Left" VerticalAlignment="Top"
             Margin="130,36,0,0" . . .  />
    <Label Content="Euro:" HorizontalAlignment="Left" VerticalAlignment="Top"
           Margin="260,40,0,0" Width="60" Height="30" />
    <Label Name="ausgabe" HorizontalAlignment="Left" VerticalAlignment="Top"
           Margin="350,36,0,0" . . ./>
    <Button Content="Konvertieren" HorizontalAlignment="Left" VerticalAlignment="Top"
            Margin="130,110,0,0" . . ./>
  </Panel>
</Window>
```

Auf die Positionen und Größen der in einem **Panel**-Container enthaltenen Steuerelemente wirken sich die folgenden Attribute bzw. Eigenschaften der Elemente aus:

- **HorizontalAlignment**
  Mögliche Werte:
  - **Left**
    Verankerung am linken Rand
  - **Right**
    Verankerung am rechten Rand
  - **Center**
    Horizontale Zentrierung
  - **Stretch**
    Verankerung am linken und am rechten Rand mit dem Ergebnis einer dynamischen Anpassung der Breite an die Fensterbreite

- **VerticalAlignment**
  Mögliche Werte:
  - **Top**
    Verankerung am oberen Rand
  - **Bottom**
    Verankerung am unteren Rand
  - **Center**
    Vertikale Zentrierung
  - **Stretch**
    Verankerung am oberen und am unteren Rand mit dem Ergebnis einer dynamischen Anpassung der Höhe an die Fensterhöhe
- **Margin**
  Man gibt für die Randbreite per **double**-Wert eine Anzahl von geräteunabhängigen Pixeln an, wobei unterschiedlich detaillierte Angaben möglich sind:
  - Mit *einer* Zahl legt man für alle Seiten denselben Rand fest, z. B.:
    ```
    Margin="50"
    ```
  - Von *zwei* Zahlen legt die erste Zahl die beiden horizontalen und die zweite Zahl die beiden vertikalen Ränder fest, z. B.:
    ```
    Margin="50,100"
    ```
  - *Vier* Zahlen beziehen sich (in dieser Reihenfolge) auf den linken, oberen, rechten und den unteren Rand, z. B.:
    ```
    Margin="20,30,40,50"
    ```

Bei überlappenden Elementen hängt das Ergebnis von der Ausgabereihenfolge ab:

- Die orientiert sich per Voreinstellung an der Reihenfolge im XAML-Code.
- Über die (von **Visual** geerbte) Eigenschaft **ZIndex** lässt sich das Ergebnis beeinflussen, wobei ein höherer Index zu einer späteren Ausgabe führt.

Durch geschickt gewählte Größen, Abstände und Verankerungen lässt sich für eine kleine Anzahl von Steuerelementen die gewünschte Anordnung erzielen. Für ein komplexeres Steuerelement-Ensemble benötigt man aber einen flexibleren Layoutcontainer (z. B. aus der Klasse **Grid**) oder eine Verschachtelung von mehreren Containern.

10.2 Grid

Der **Grid**-Container ist als Wurzel-Layoutmanager und auch als untergeordneter Container wegen seiner Flexibilität oft anzutreffen.

10.2.1 Zeilen und Spalten definieren

Die Zeilen bzw. Spalten eines **Grid**-Containers legt man bzgl. Anzahl und Ausdehnung über die Eigenschaft **RowDefinitions** bzw. **ColumnDefinitions** fest:

- **RowDefinitions** zeigt auf eine Kollektion vom Typ **DefinitionList<RowDefinition>**.
- **ColumnDefinitions** zeigt auf eine Kollektion vom Typ **DefinitionList<ColumnDefinition>**.

Zur Festlegung der Höhe einer Zeile (**Height**-Eigenschaft eines **RowDefinition**-Objekts) bzw. der Breite einer Spalte (**Width**-Eigenschaft eines **ColumnDefinition**-Objekts) bestehen drei Optionen:

- Fixierung auf eine feste Anzahl von geräteunabhängigen Pixeln (1/96 Zoll pro Einheit)
- Automatische Anpassung an den größten Platzbedarf eines Kindelements
  Diese Option wählt man über das Schlüsselwort **Auto**.
- Anteil vom Rest nach Abzug der festen oder automatisch berechneten Ausdehnungen
  Diese Option wählt über einen Faktor vor einem Stern, z. B. 3*. Der Anteil einer Zeile bzw. Spalt am Rest ist der Quotient aus ihrem Faktor und der Summe aller Faktoren. Wählt man Faktoren mit der Summe 100, dann steht ein Faktor gerade für den Prozentanteil.

Um die **RowDefinition**- bzw. **ColumnDefinition**-Objekte samt Ausdehnungen festzulegen, kann man im XAML-Code ...

- der **RowDefinitions**- bzw. **ColumnDefinitions**-Eigenschaft des **Grid**-Elements eine Zeichenfolge mit Komma-getrennten Größenangaben zuweisen, z. B.:
  - `RowDefinitions="Auto,Auto,Auto"`
  - `ColumnDefinitions="100,*,2*"`
- dem **Grid**-Element Eigenschaftselemente unterordnen, z. B.:

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="100" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="2*" />
</Grid.ColumnDefinitions>
```

Die aufwändigere Syntax bietet zusätzliche Optionen. Über **SharedSizeGroup**-Attribute ist es z. B. möglich, für mehrere, in einem gemeinsamen übergeordneten Container befindliche **Grid**-Layoutmanager die Breiten korrespondierender Spalten gleichzusetzen.[1]

Im Vergleich zum **Grid**-Layoutmanager in WPF hat die Avalonia-Variante neben der kompakten **RowDefinitions**- bzw. **ColumnDefinitions**-Definition durch eine Zeichenfolge mit Komma-getrennten Größenangaben auch noch die Eigenschaften **RowSpacing** und **ColumnSpacing** zu bieten. Man kann z. B. die vertikalen Abstände zwischen allen Zeilen in einem **Grid**-Container durch eine einzelne Eigenschaftszuweisung festlegen, z. B.:

```
<Grid VerticalAlignment="Center" RowSpacing="20" RowDefinitions="*,*,*">
    <Label Grid.Row="0" Content="Zeile 0" />
    <Label Grid.Row="1" Content="Zeile 1" />
    <Label Grid.Row="2" Content="Zeile 2" />
</Grid>
```

Das macht es leichter, mit verschiedenen Abständen zu experimentieren:



[1] https://docs.avaloniaui.net/controls/layout/panels/grid

Es sind auch **Grid**-Container ohne Zeilen- und/oder Spaltendefinition, also mit einer einzigen Zelle (0, 0) möglich. Das Visual Studio legt bei Verwendung der Projektvorlage **WPF-Anwendung** einen solchen Wurzel-Container an:

```
<Window x:Class="WpfApp1.MainWindow" ... Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

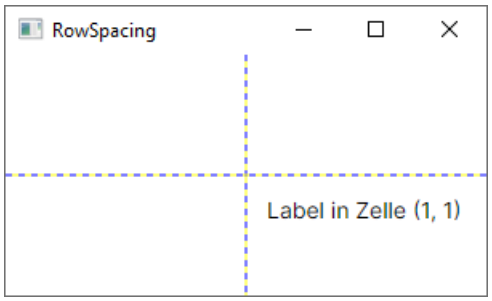
Statt eines **Grid**-Containers mit einer einzigen Zelle sollte man den sparsameren **Panel**-Container verwenden (siehe Abschnitt 10.1).

10.2.2 Ortsangaben

Besitzt ein per **Grid**-Container verwaltetes Steuerelement keine Ortsangabe, dann landet es in der **Grid**-Zelle (0, 0), also links oben. Um für ein Steuerelement eine alternative Zelle festzulegen, sind in seinem XAML-Element die Attribute bzw. die angefügten Eigenschaften **Grid.Row** und **Grid.Column** mit der 0-basierten Nummer der Zeile bzw. Spalte zu versorgen (zu angefügten Eigenschaften siehe Abschnitt 8.4). Das **Label**-Element im folgenden Beispiel erhält die Zelle (1, 1) als Einsatzort:

```
<Grid RowDefinitions="*,*" ColumnDefinitions="*,*" ShowGridLines="True">
    <Label Content="Label in Zelle (1, 1)" Grid.Column="1" Grid.Row="1" Margin="10"/>
</Grid>
```

Der Wert **True** für das **Grid**-Attribut **ShowGridLines** sorgt dafür, dass die (2 × 2) - Struktur des Containers optisch präsent ist:[1]



10.2.3 Mehrzellige Elemente

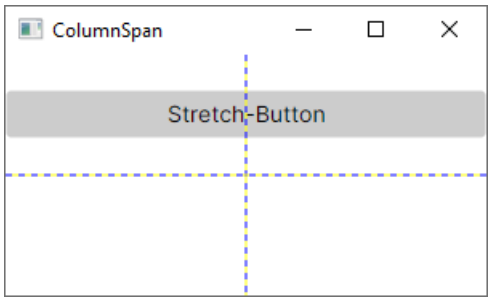
Es ist nicht vorgeschrieben, dass sich ein Steuerelement vollständig innerhalb *einer* **Grid**-Zelle befinden muss. Mit Hilfe der angefügten Eigenschaften **Grid.RowSpan** sowie **Grid.ColumnSpan** kann ein Steuerelement *mehrere* Zeilen und/oder Spalten belegen. Der folgende XAML-Code definiert einen (2 × 2) - **Grid** - Container mit einem **Button**-Objekt, das die *beiden* Zellen in der oberen Zeile belegt:

```
<Grid RowDefinitions="*,*" ColumnDefinitions="*,*" ShowGridLines="True">
    <Button Content="Stretch-Button" HorizontalContentAlignment="Center"
            HorizontalAlignment="Stretch" Grid.ColumnSpan="2" />
</Grid>
```

Wie im vorigen Abschnitt machen wir die (2 × 2) - Struktur des Containers sichtbar durch den Wert **true** für das **Grid**-Attribut **ShowGridLines**:

---

[1] Bei den Werten **true** und **false** für XAML-Attribute zu Eigenschaften vom Typ **bool** ist die Groß-/Kleinschreibung ausnahmsweise irrelevant.



Dass sich mehrere Steuerelemente *eine* **Grid**-Zelle teilen können, haben Sie schon wiederholt beobachtet. Sofern man die Größen, Abstände und Verankerungen der Elemente geschickt wählt, resultiert eine ergonomische Bedienoberfläche mit sinnvollen Anpassungen an Änderungen bei der Größe oder beim Seitenverhältnis des Fensters. Für ein komplexes Layout ist ein einzelliger **Grid**-Container aber keine gute Lösung.

10.2.4 GridSplitter

Ein **GridSplitter** mit dem voreingestellten **ResizeBehavior** ermöglicht den Benutzern zur Laufzeit die folgenden Anpassungen der Platzaufteilung:

- Eine als vertikaler Splitter dienende Spalte lässt sich horizontal verschieben, um den verfügbaren Platz zwischen den beiden Spalten links bzw. rechts neben dem Splitter neu aufzuteilen.
- Eine als horizontaler Splitter dienende Zeile lässt sich vertikal verschieben, um den verfügbaren Platz zwischen den beiden Zeilen über bzw. unter dem Splitter neu aufzuteilen.

Der **Grid**-Container im folgenden Beispiel

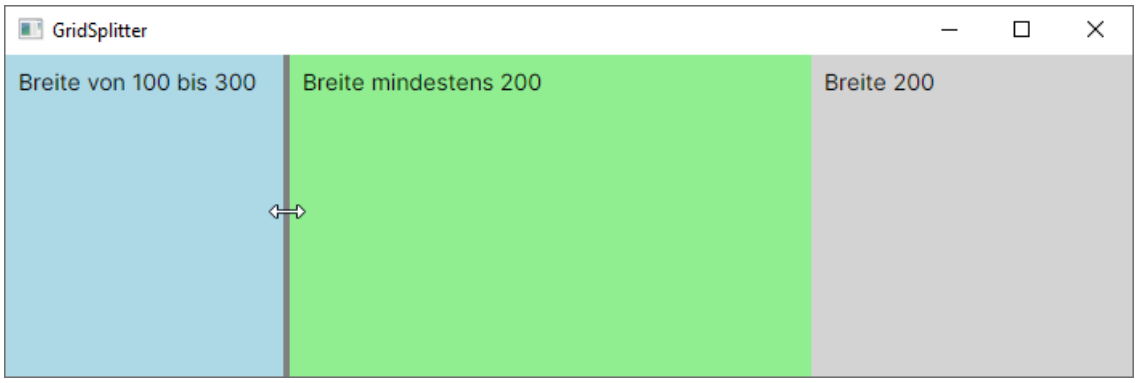
```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="GridSplitter.MainWindow"
        Title="GridSplitter" Width="702" Height="200">

    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="200" MinWidth="100" MaxWidth="300" />
            <ColumnDefinition Width="2" />
            <ColumnDefinition MinWidth="200" />
            <ColumnDefinition Width="200" />
        </Grid.ColumnDefinitions>
        <Border Grid.Column="0" Background="LightBlue">
            <TextBlock Text="Breite von 100 bis 300" Margin="8" />
        </Border>
        <GridSplitter Grid.Column="1" Background="Gray" ResizeDirection="Columns"/>
        <Border Grid.Column="2" Background="LightGreen">
            <TextBlock Text="Breite mindestens 200" Margin="8" />
        </Border>
        <Border Grid.Column="3" Background="LightGray">
            <TextBlock Text="Breite 200" Margin="8" />
        </Border>
    </Grid>
</Window>
```

besitzt eine **GridSplitter**-Spalte und drei Anzeigespalten:[1]

- Die Anzeigespalte links vom Splitter hat eine variable Breite von 100 bis 300 Pixeln.
- Die Anzeigespalte rechts vom Splitter hat eine variable Breite ab 200 Pixeln.
- Die dritte Anzeigespalte (ohne Kontakt mit dem Splitter) hat eine feste Breite von 200 Pixeln.

Für die beiden ersten Anzeigespalten sind insgesamt 500 Pixel verfügbar, die sich dank der **GridSplitter**-Spalte (mit der Breite 2) im Rahmen der Beschränkungen verteilen lassen:



10.3 DockPanel

Ein Layoutcontainer der Klasse **DockPanel** bietet den verwalteten Steuerelementen über die angefügte Eigenschaft **Dock** mit den Werten **Left**, **Top**, **Right**, **Bottom** die Möglichkeit, sich an einer Seite festzusetzen und diese (mit der eventuellen Ausnahme von bereits besetzten Ecken) komplett zu belegen. Wollen mehrere Elemente eine Seite belegen, dann werden sie dort in der Antragsreihenfolge gestapelt. In der zur Andock-Seitenlinie orthogonalen Richtung erhalten die Steuerelemente nach Möglichkeit ihre gewünschte Ausdehnung. Zuletzt muss ein Element *ohne* **Dock**-Angabe in den Layoutcontainer gelangen. Es sorgt für korrekte Layout-Berechnungen und belegt den restlichen Platz.

In der folgenden Fensterdeklaration[2]

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="DockPanel.MainWindow"
        Title="DockPanel" Height="250" Width="500">

  <Window.Styles>
    <Style Selector="Border">
      <Setter Property="BorderBrush" Value="Black"/>
      <Setter Property="BorderThickness" Value="1"/>
    </Style>
    <Style Selector="TextBlock">
      <Setter Property="FontSize" Value="14"/>
      <Setter Property="Foreground" Value="Black"/>
      <Setter Property="HorizontalAlignment" Value="Center"/>
      <Setter Property="VerticalAlignment" Value="Center"/>
    </Style>
  </Window.Styles>
```

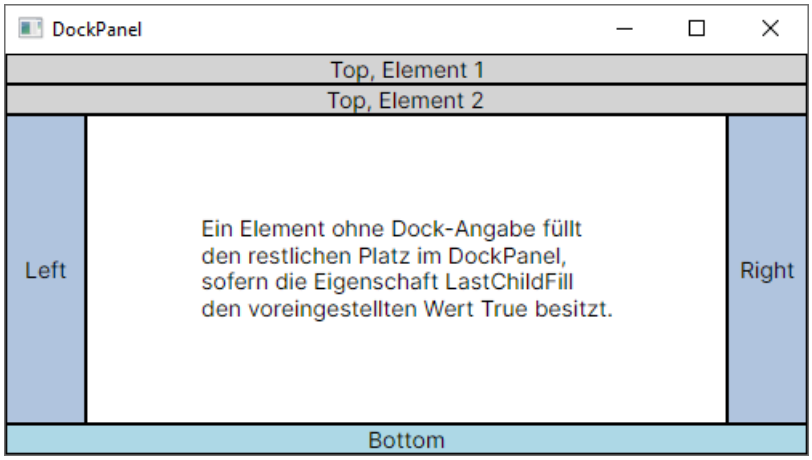
---

[1] Ein Visual Studio - Projekt mit dem an einem Vorbild auf der Webseite
https://docs.avaloniaui.net/controls/layout/panels/gridsplitter
orientierten Beispiel ist hier zu finden:
...\BspUeb\Layoutcontainer\GridSplitter

[2] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Layoutcontainer\DockPanel

```
    <DockPanel>
      <Border Background="LightGray" DockPanel.Dock="Top">
        <TextBlock>Top, Element 1</TextBlock>
      </Border>
      <Border Background="LightGray" DockPanel.Dock="Top">
        <TextBlock>Top, Element 2</TextBlock>
      </Border>
      <Border Background="LightBlue" DockPanel.Dock="Bottom">
        <TextBlock>Bottom</TextBlock>
      </Border>
      <Border Background="LightSteelBlue" DockPanel.Dock="Left" Width="50">
        <TextBlock>Left</TextBlock>
      </Border>
      <Border Background="LightSteelBlue" DockPanel.Dock="Right" Width="50">
        <TextBlock>Right</TextBlock>
      </Border>
      <Border Background="White">
        <TextBlock>
          Ein Element ohne Dock-Angabe füllt<LineBreak />
          den restlichen Platz im DockPanel,<LineBreak />
          sofern die Eigenschaft LastChildFill<LineBreak />
          den voreingestellten Wert True besitzt.
        </TextBlock>
      </Border>
    </DockPanel>
  </Window>
```

sind alle Seiten durch **Border**-Objekte mit **Dock**-Angabe besetzt, und das zuletzt, *ohne* **Dock**-Angabe eingefügte **Border**-Objekt belegt den frei gebliebenen Raum im Zentrum. Die **Border**-Objekte enthalten eine Beschriftung per **TextBlock**:



Um die **Border**- bzw. **TextBlock**-Attribute nicht wiederholen zu müssen, kommen **Style**-Elemente zum Einsatz mit ...

- dem Typ **Border**- bzw. **TextBlock** als **Selector**
- und per **Setter** festgelegten Eigenschaftsausprägungen.

Damit haben wir im Vorbeigehen eine wichtige Avalonia-Option an Beispielen kennengelernt.

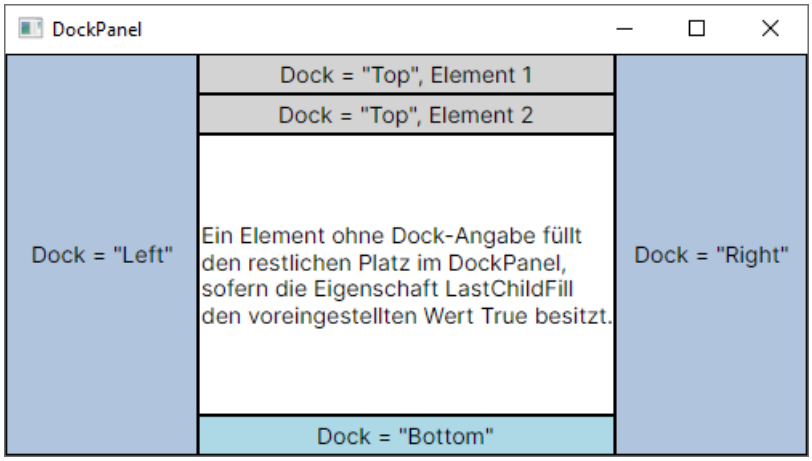
Der **DockPanel**-Layoutcontainer bietet ein für viele Programme geeignetes GUI-Gerüst, z. B. für einen Editor:

- Oben finden das Menü und eine Symbolleiste Platz.
- Links befindet sich eine Navigationszone.
- In der Mitte ist das Dokument editierbar.
- Rechts befindet sich eine Werkzeugsammlung.
- Unten befindet sich eine Statuszeile.

Ob die an einer horizontalen Seite (oben bzw. unten) oder die an einer vertikalen Seite (links bzw. rechts) angedockten Steuerelemente die Ecken belegen, hängt von der Aufnahmereihenfolge ab. Aus der folgenden Alternative zur obigen Layoutdefinition

```
<DockPanel>
  <Border Background="LightSteelBlue" DockPanel.Dock="Left" Width="50">
    <TextBlock>Left</TextBlock>
  </Border>
  <Border Background="LightSteelBlue" DockPanel.Dock="Right" Width="50">
    <TextBlock>Right</TextBlock>
  </Border>
  <Border Background="LightGray" DockPanel.Dock="Top">
    <TextBlock>Top, Element 1</TextBlock>
  </Border>
  <Border Background="LightGray" DockPanel.Dock="Top">
    <TextBlock>Top, Element 2</TextBlock>
  </Border>
  <Border Background="LightBlue" DockPanel.Dock="Bottom">
    <TextBlock>Bottom</TextBlock>
  </Border>
  <Border Background="White">
    <TextBlock>
       Ein Element ohne Dock-Angabe füllt<LineBreak />
       den restlichen Platz im DockPanel,<LineBreak />
       sofern die Eigenschaft LastChildFill<LineBreak />
       den voreingestellten Wert True besitzt.
    </TextBlock>
  </Border>
</DockPanel>
```

folgt diese Anordnung:



10.4 StackPanel

Mit einem Layoutcontainer der Klasse **StackPanel** realisiert man einen vertikalen oder horizontalen Stapel von Elementen. Bei einem vertikalen Stapel steht den enthaltenen Elementen die gesamte Container-Breite zur Verfügung. Bei einem horizontalen Stapel steht den enthaltenen Elementen die gesamte Container-Höhe zur Verfügung.

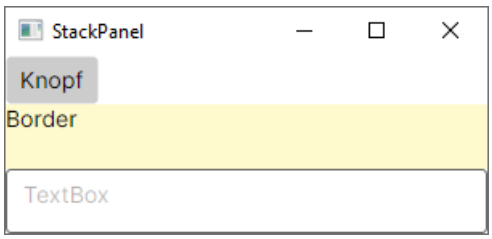
Die folgende XAML-Deklaration[1]

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Layoutcontainer\StackPanel

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="StackPanel.MainWindow"
        Title="StackPanel" SizeToContent="Height" Width="300">

  <StackPanel>
    <Button Content="Knopf" />
    <Border Background="LightSteelBlue" Height="40">
      <TextBlock>Border</TextBlock>
    </Border>
    <TextBox PlaceholderText="TextBox" Height="40" />
  </StackPanel>
</Window>
```

stapelt ein **Button**-, ein **Border**- und ein **TextBox**-Element übereinander:



Während die Klasse **Button** den **HorizontalAlignment**-Wert **Left** bevorzugt, nutzen die Klassen **Border** und **TextBox** aufgrund der **HorizontalAlignment**-Voreinstellung **Stretch** die gesamte Breite des **StackPanel**-Containers.

Weil die **Width**-Eigenschaften des Schalters keinen Wert erhalten hat, gilt die Voreinstellung **Double.NaN** (im XAML-Code: **Auto**, Groß-/Kleinschreibung beliebig), und zusammen mit dem voreingestellten **HorizontalAlignment**-Wert **Left** resultiert eine Anpassung an den Inhalt (vgl. Abschnitt 11.2.2).

Ein **StackPanel**-Container passt in der Stapelrichtung seine Größe an die enthaltenen Elemente an. Diese erhalten die gewünschten Ausdehnungen, solange der Platz reicht. Ist bei einem vertikalen Stapel die gewünschte Gesamthöhe bzw. bei einem horizontalen Stapel die gewünschte Gesamtbreite nicht realisierbar, dann bleiben die zuletzt eingefügten Elemente unterversorgt bzw. unsichtbar. Soll in diesem Situation ein Rollbalken erscheinen, packt man den **StackPanel** in einen **ScrollViewer**, z. B.:

```
<ScrollViewer>
    <StackPanel>
        <Button Name="button" Content="Knopf" />
        <Border Name="border" Background="LemonChiffon" Height="40">
          <TextBlock>Border</TextBlock>
        </Border>
        <TextBox Name="textBox" PlaceholderText="TextBox" Height="40" />
    </StackPanel>
</ScrollViewer>
```

Im **Window**-Element des Beispiels sorgt die Eigenschafts- bzw. Attributzuweisung

```
SizeToContent="Height"
```

für eine Anpassung der Fensterhöhe an den vom **StackPanel**-Container benötigten Platz.

Über die (in WPF fehlende) Eigenschaft **Spacing** erhält man einen gleichmäßigen Abstand zwischen den Elementen, z. B.:

```
<StackPanel Spacing="10">
    . . .
</StackPanel>
```

Um einen *horizontalen* Stapel zu erhalten, setzt man die **StackPanel**-Eigenschaft **Orientation** auf den Wert **Horizontal**:

```
<StackPanel Orientation="Horizontal">
  . . .
</StackPanel>
```

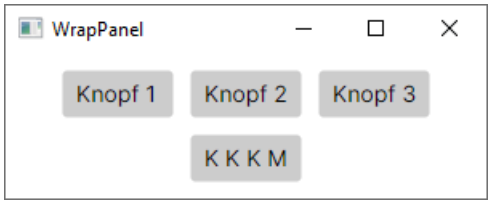
10.5 WrapPanel

Ein **WrapPanel** nimmt bei Platzmangel einen automatischer Spalten- bzw. Zeilenumbruch vor. Wegen der voreingestellten horizontalen Orientierung stapelt der folgende XAML-Code[1]

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="WrapPanel.MainWindow"
        Title="WrapPanel" SizeToContent="Height" Width="300">

   <WrapPanel Margin="10" ItemsAlignment="Center" ItemSpacing="10" LineSpacing="10">
        <Button Width="70">Knopf 1</Button>
        <Button Width="70">Knopf 2</Button>
        <Button Width="70">Knopf 3</Button>
        <Button Width="70">K K K M</Button>
    </WrapPanel>
</Window>
```

vier **Button**-Objekte nebeneinander, bis der Platz erschöpft und daher ein Zeilenumbruch erforderlich ist:



Weil die **Height**-Eigenschaften der Schalter keinen Wert erhalten haben, gilt die Voreinstellung **Double.NaN** (im XAML-Code: **Auto**, Groß-/Kleinschreibung beliebig), wobei im **WrapPanel** eine Anpassung an den Inhalt (die Beschriftung) resultiert (vgl. Abschnitt 11.2.2).

Als nützliche Erweiterungen von Avalonias **WrapPanel**-Container im Vergleich zur WPF-Variante sind die folgenden Einstellungen zu nennen:

- **ItemSpacing**, **LineSpacing**
  Man kann die Abstände zwischen den enthaltenen Elementen in horizontaler und vertikaler Richtung festlegen.
- **ItemsAlignment**
  Alternativ zu der im Beispiel durch den Wert **Center** gewählten Zentrierung ist über den Wert **Start** eine linksbündige

  und über den Wert **End** eine rechtsbündige Anordnung möglich:

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Layoutcontainer\WrapPanel



Um einen *vertikal* orientierten **WrapPanel**-Container zu erhalten, setzt man die Eigenschaft **Orientation** auf den Wert **Vertical**:

```
<WrapPanel Orientation="Vertical">
  . . .
</WrapPanel>
```

10.6 RelativePanel

Die Steuerelemente in einem **RelativePanel**-Container erhalten über angefügte Eigenschaften ihre Positionen in Relation zum Container oder zu einem benannten Steuerelement. Per Voreinstellung befindet sich ein Steuerelement oben links, d. h.:

- Sein linker Außenrand grenzt an den linken Rand des Containers.
- Sein oberer Außenrand grenzt an den oberen Rand des Containers.

Breite und Höhe eines Elements sind ...

- entweder über seine Eigenschaften **Width** und **Height**
- oder durch den Bezug auf Seiten des Containers
- oder durch Bezug auf Seiten eines andern Elements

festzulegen.

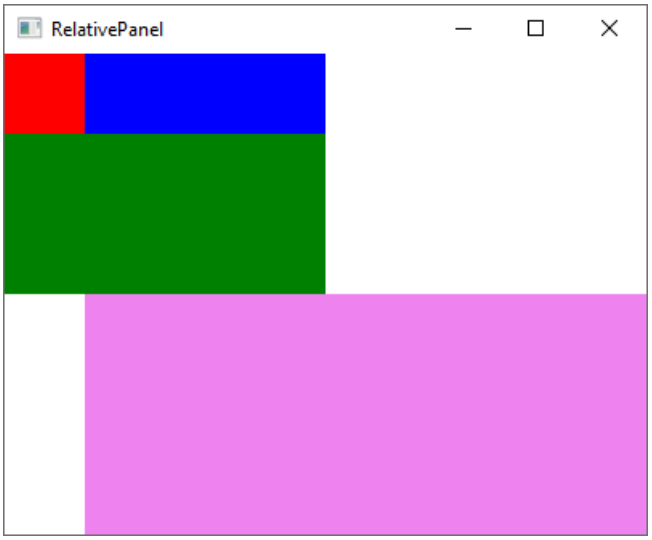
Über die angefügten **RelativePanel**-Eigenschaften in der folgenden Tabelle kann ein Steuerelement für seine Ränder einen Bezug zu den Rändern des Containers oder zu den Rändern anderer Elemente herstellen:

Angefügte Eigenschaft	Erklärung
AlignTopWithPanel	Oberen Rand des Containers übernehmen
AlignBottomWithPanel	Unteren Rand des Containers übernehmen
AlignLeftWithPanel	Linken Rand des Containers übernehmen
AlignRightWithPanel	Rechten Rand des Containers übernehmen
AlignHorizontalCenterWithPanel	Horizontal im Container zentrieren
AlignVerticalCenterWithPanel	Vertikal im Container zentrieren
AlignTopWith	Oberen Rand des benannten Elements übernehmen
AlignBottomWith	Unteren Rand des benannten Elements übernehmen
AlignLeftWith	Linken Rand des benannten Elements übernehmen
AlignRightWith	Rechten Rand des benannten Elements übernehmen
AlignHorizontalCenterWith	Horizontal in Bezug auf das benannte Element zentrieren
AlignVerticalCenterWith	Vertikal in Bezug auf das benannte Element zentrieren
Above	Über ein benanntes Element setzen
Below	Unter ein benanntes Element setzen
LeftOf	Den linken Rand des benannten Elements als den eigenen *rechten* Rand übernehmen
RightOf	Den rechten Rand des benannten Elements als den eigenen *linken* Rand übernehmen

Die folgende Fensterdeklaration stammt im Wesentlichen aus einem Beispiel der Avalonia-Dokumentation:[1]

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Width="400" Height="300"
        x:Class="RelativePanel.MainWindow"
        Title="RelativePanel">
    <RelativePanel>
        <Rectangle Name="Red" Fill="Red" Height="50" Width="50"/>
        <Rectangle Name="Blue" Fill="Blue" Height="50" Width="150"
                   RelativePanel.RightOf="Red" />
        <Rectangle Name="Green" Fill="Green" Height="100"
                   RelativePanel.Below="Red"
                   RelativePanel.AlignLeftWith="Red"
                   RelativePanel.AlignRightWith="Blue"/>
        <Rectangle Fill="Violet"
                   RelativePanel.Below="Green"
                   RelativePanel.AlignLeftWith="Blue"
                   RelativePanel.AlignRightWithPanel="True"
                   RelativePanel.AlignBottomWithPanel="True"/>
    </RelativePanel>
</Window>
```

Seine Bestandteile sind keine Steuer- sondern *Grafikelemente* (aus der Klasse **Rectangle**):



Das rote Rechteck ...

- nimmt die voreingestellte Position in der linken oberen Ecke des Containers ein
- und definiert seine Ausdehnungen selbst.

Das blaue Rechteck ...

- setzt sich rechts neben das rote Rechteck
- und definiert seine Ausdehnungen selbst.

---

[1] Ein Visual Studio - Projekt mit dem an einem Vorbild auf der Webseite
https://docs.avaloniaui.net/controls/layout/panels/relativepanel
orientierten Beispiel ist hier zu finden:
...\BspUeb\Layoutcontainer\RelativePanel

Das grüne Rechteck ...

- setzt sich unter das rote Rechteck,
- übernimmt seinen linken Rand vom roten Rechteck,
- übernimmt seinen rechten Rand vom blauen Rechteck
- und definiert seine Höhe selbst.

Das violette Rechteck ...

- setzt sich unter das grüne Rechteck,
- übernimmt seinen linken Rand vom blauen Rechteck,
- übernimmt seinen rechten Rand vom Container
- und übernimmt seinen unteren Rand vom Container.

10.7 UniformGrid

Von der flexiblen und oft verwendeten Klasse **Grid** unterscheidet sich die (z. B. für eine virtuelle Tastatur oder eine Farbpalette geeignete) Klasse **UniformGrid** trotz der Namensähnlichkeit erheblich:

- Man legt die Zahl der Zeilen und/oder Spalten über die Attribute bzw. Eigenschaften **Rows** und/oder **Columns** fest, z. B.:

```
<UniformGrid Rows="2" Columns="3">
      . . .
</UniformGrid>
```

- Verzichtet man auf die Definition der Matrixstruktur, dann entsteht eine *quadratische* Matrix mit einer automatisch ermittelten Anzahl von Zeilen bzw. Spalten. Ist die Anzahl der eingefügten Elemente z. B. größer als 4 und kleiner als 10, dann resultiert eine (3 × 3) - Matrix.
- Jede Zelle nimmt genau *ein* Element auf. Ein eingefügtes Element belegt die nächste freie Zelle, wobei die Belegung oben links startet und zeilenweise verläuft. Der Spaltenindex läuft also schneller als der Zeilenindex.
- Alle Zellen (ggf. auch die unbesetzten) sind gleich groß. Werden unterschiedlich dimensionierte Zellen benötigt, dann ist ein **Grid**-Container das Mittel der Wahl.

Aus der folgenden XAML-Deklaration[1]

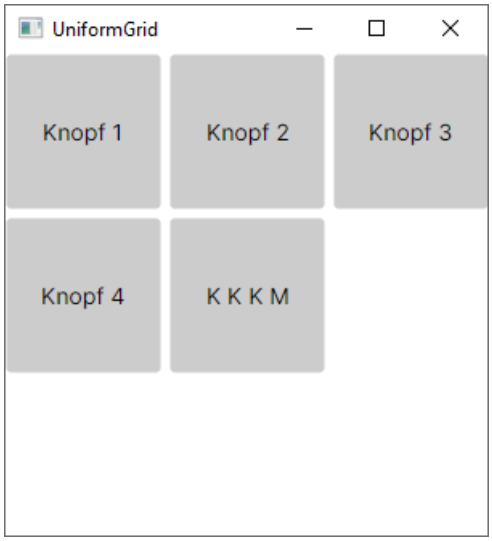
```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="UniformGrid.MainWindow"
        Title="UniformGrid" Height="300" Width="300">

    <Window.Styles>
      <Style Selector="Button">
          <Setter Property="HorizontalAlignment" Value="Stretch"/>
          <Setter Property="VerticalAlignment" Value="Stretch"/>
          <Setter Property="HorizontalContentAlignment" Value="Center"/>
          <Setter Property="VerticalContentAlignment" Value="Center"/>
      </Style>
    </Window.Styles>
```

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Layoutcontainer\UniformGrid

```
        <UniformGrid RowSpacing="5" ColumnSpacing="5">
            <Button>Knopf 1</Button>
            <Button>Knopf 2</Button>
            <Button>Knopf 3</Button>
            <Button>Knopf 4</Button>
            <Button>K K K M</Button>
        </UniformGrid>
    </Window>
```

resultiert ein Container mit (3 × 3) gleich großen Zellen (davon 4 unbesetzt):



Das Fenster verteilt den verfügbaren Platz auf die Matrixzellen.

Eine **Style**-Definition sorgt dafür, dass die **Button**-Objekte in den Zellen den verfügbaren Platz komplett belegen und die Beschriftung zentriert anzeigen.

Per **RowSpacing** bzw. **ColumnSpacing** legt man den Abstand zwischen den Zeilen bzw. Spalten fest.

10.8 Canvas

Die Klasse **Canvas** stellt eine Zeichenfläche mit fixierten Positionen und Größen von Bedienelementen, also ohne dynamische Anpassung an eine geänderte Fenstergröße oder an ein geändertes Seitenverhältnis zur Verfügung. Durch die angefügten Eigenschaften **Canvas.Left** und **Canvas.Top** legt man für die verwalteten Elemente den Abstand zum linken bzw. zum oberen Rand fest. So kann man Grafik- und Steuerelemente mit fester Größe und Position durch die sorgfältige Berechnung von Koordinaten auf eine Fensterfläche montieren.

Für die ebenfalls möglichen angefügten Eigenschaften **Canvas.Right** und **Canvas.Bottom** gilt:

- Sie vergrößern *nicht* den **Canvas**-Container.
- Sie unterliegen bei einer Konkurrenz zu den angefügten Eigenschaften **Canvas.Left** und **Canvas.Top**.

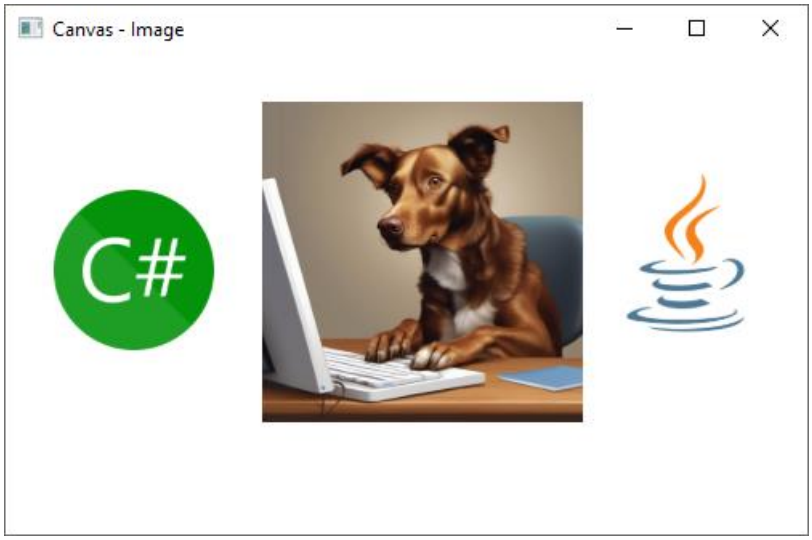
Ein **Canvas**-Container ist z. B. für Bilder oder andere Grafikelemente geeignet. In der folgenden Fensterdeklaration dienen **Image**-Objekte zur Anzeige von Bildern:[1]

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Layoutcontainer\Canvas\Bilder

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="Bilder.MainWindow"
        Title="Canvas - Image" Height="300" Width="500">

    <Canvas>
        <Image Source="avares://Bilder/Assets/Csharp.png"
               RenderOptions.BitmapInterpolationMode="HighQuality"
               Height="100" Width="100" Canvas.Left="30" Canvas.Top="85" />
        <Image Source="avares://Bilder/Assets/DogAtWork.png"
               RenderOptions.BitmapInterpolationMode="HighQuality"
               Height="200" Width="200" Canvas.Left="160" Canvas.Top="30" />
        <Image Source="avares://Bilder/Assets/Java.png"
               RenderOptions.BitmapInterpolationMode="HighQuality"
               Height="100" Width="100" Canvas.Left="380" Canvas.Top="75" />
    </Canvas>
</Window>
```

Wie man Bilder in ein Projekt aufnimmt, verrät der Abschnitt 7.3. Das Ergebnis zeigt einen Programmierer mit den Schwerpunkten C# und Java:[1]



Für Steuerelemente kommt ein **Canvas**-Container nur selten in Frage. Aus der folgenden XAML-Deklaration[2]

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="Schalter.MainWindow"
        Title="Canvas" Height="70" Width="250">

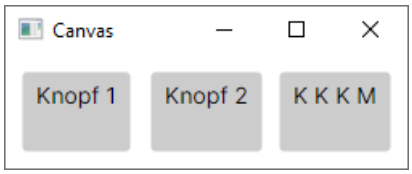
    <Canvas>
        <Button Height="50" Canvas.Left="10" Canvas.Top="10">Knopf 1</Button>
        <Button Height="50" Canvas.Left="90" Canvas.Top="10">Knopf 2</Button>
        <Button Height="50" Canvas.Left="170" Canvas.Top="10">K K K M</Button>
    </Canvas>
</Window>
```

resultiert das Fenster:

---

[1] Erzeugt mit Stable Diffusion (Creative ML OpenRAIL-M-Lizenz)

[2] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Layoutcontainer\Canvas\Schalter

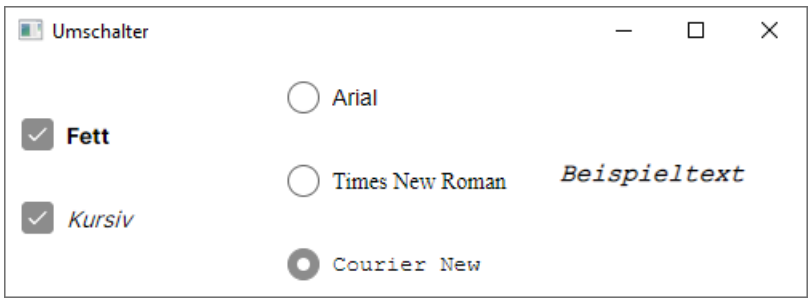


Bei überlappenden Elementen hängt das Ergebnis von der Ausgabereihenfolge ab:

- Die orientiert sich per Voreinstellung an der Definitionsreihenfolge.
- Über die (von **Visual** geerbte) Eigenschaft **ZIndex** lässt sich das Ergebnis beeinflussen, wobei ein höherer Index zu einer späteren Ausgabe führt.

10.9 Geschachtelte Layoutcontainer

Im folgenden Fenster, das uns im Abschnitt 11.3.2 erneut begegnen wird, sind zwei übereinanderstehende **CheckBox**-Elemente, drei übereinanderstehende **RadioButton**-Elemente und ein **Label**-Element jeweils in einer eigenen Spalte vertikal zentriert, und die drei Spalten stehen mit gleicher Breite nebeneinander:



Beim Fensterdesign per XAML-Code kommt ein dreispaltiger **Grid** - Wurzel-Container zum Einsatz. In den beiden ersten Spalten arbeitet jeweils ein vertikal orientierter **StackPanel**-Container mit den **CheckBox**- bzw. **RadioButton**-Objekten. In der dritten Spalte befindet sich das **Label**-Objekt:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="Umschalter.MainWindow"
        Title="Umschalter" Height="150" Width="500">
    <Grid ColumnDefinitions="*,*,*">
        <StackPanel  VerticalAlignment="Center" Button.Click="CheckBox_Click">
            <CheckBox Name="chkBold" Content="Fett" Margin="10"
                      FontWeight="Bold" />
            <CheckBox Name="chkItalic" Content="Kursiv " Margin="10"
                      FontStyle="Italic"/>
        </StackPanel>
        <StackPanel  Grid.Column="1" VerticalAlignment="Center"
                     Button.Click="RadioButton_Click">
            <RadioButton Name="rbArial" Content="Arial" Margin="10"
                         FontFamily="Arial" IsChecked="True"/>
            <RadioButton Name="rbTimesNewRoman" Content="Times New Roman" Margin="10"
                         FontFamily="Times New Roman"/>
            <RadioButton Name="rbCourierNew" Content="Courier New" Margin="10"
                         FontFamily="Courier New"/>
        </StackPanel>
        <Label  Name="lblText" Grid.Column="2" Content="Beispieltext"
                Margin="10" VerticalAlignment="Center"
                FontFamily="Arial" FontSize="16" />
    </Grid>
</Window>
```

Ohne die untergeordneten **StackPanel**-Container wäre die vertikal zentrierte Stapelung der **CheckBox**- bzw. **RadioButton**-Elemente nur schwer zu realisieren. Das Verschachteln von Layoutcontainern sollte aus Performanzgründen nicht über drei Ebenen hinausgehen.[1]

[1] https://docs.avaloniaui.net/docs/layout/choosing-a-layout-panel

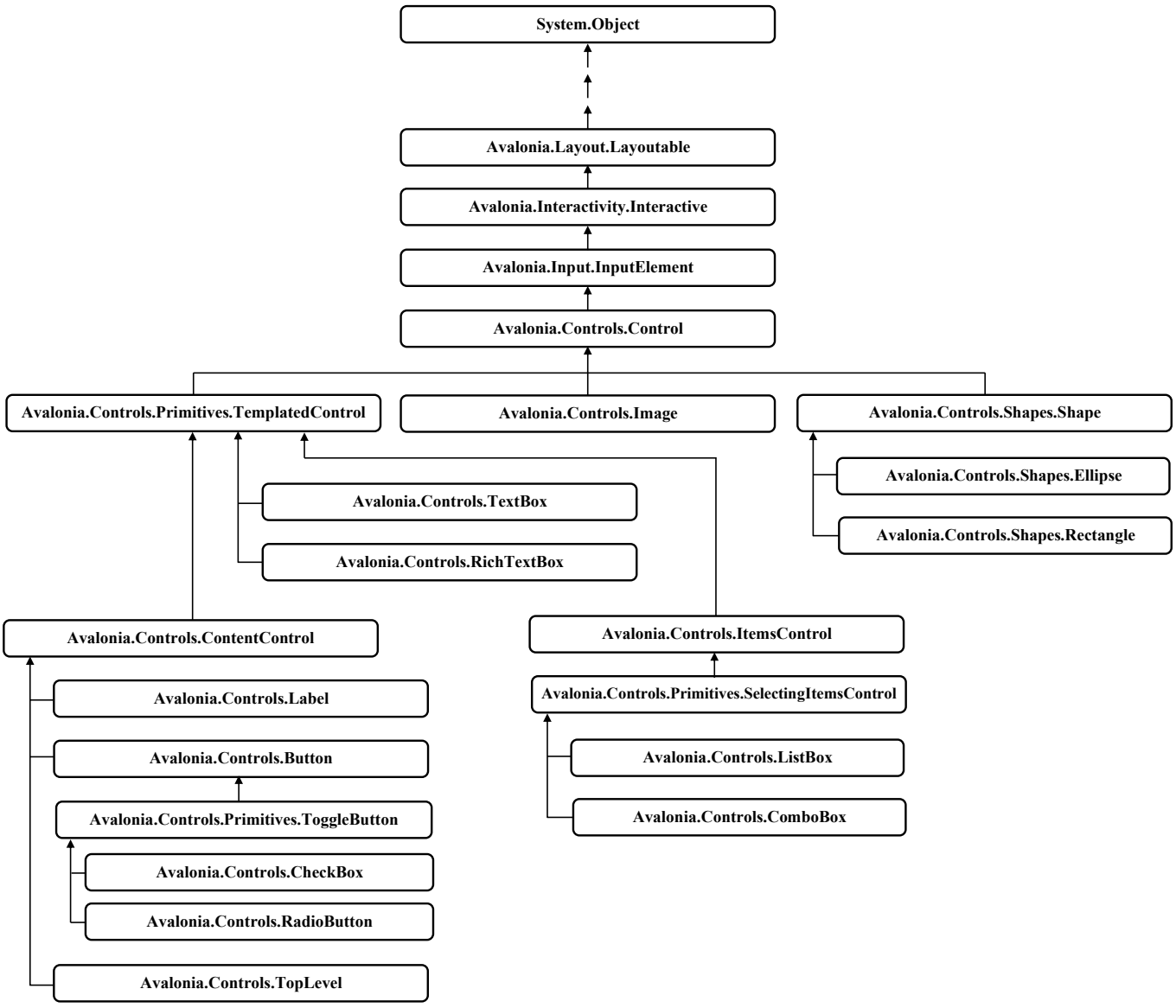
11 Steuerelemente

Avalonia ermöglicht durch zahlreiche Steuerelementklassen die Erstellung von ergonomischen und attraktiven Bedienoberflächen. Wir beschränken uns in diesem Kapitel auf elementare Klassen. Ein vollständiger Überblick ist hier zu finden:

https://docs.avaloniaui.net/controls/

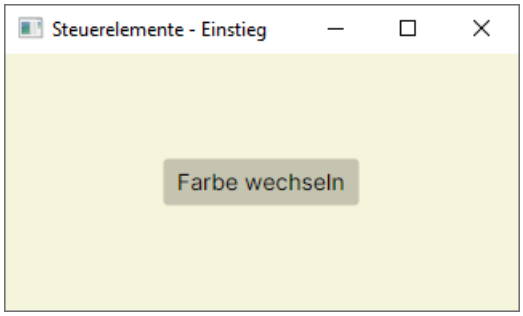
11.1 Abstammungsverhältnisse

Die zur Benutzerinteraktion dienenden Steuerelemente (**Button**, **TextBox** etc.) stammen von der Basisklasse **TemplatedControl** (Namensraum **Avalonia.Controls.Primitives**) ab, während die zur optischen Gestaltung eingesetzten *Grafikelemente* (**Ellipse**, **Rectangle** etc.) von der Basisklasse **Shape** (Namensraum **Avalonia.Controls.Shapes**) abgeleitet sind. Der folgende Stammbaum beschränkt sich auf die im Manuskript diskutierten Klassen:



11.2 Verwendung

Wir beschäftigen uns zunächst anhand eines einfachen Beispielprogramms



damit, ...

- wie aus den XAML-Elementen Member-Objekte der Fensterklasse entstehen,
- welche elementaren Eigenschaften die Steuerelemente besitzen,
- wie man die Ereignisse von Steuerelementen behandeln kann.

11.2.1 Instanziieren

Wir starten im Visual Studio ein Projekt mit der Vorlage **Avalonia .NET App** und nehmen im XAML-Code zum Fenster (in der Datei **MainWindow.axaml**) die folgenden Änderungen vor:

- Die folgenden Attribute in der Startkennung sind obsolet:
  ```
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
  ```
- Wir passen den **Title** an und wählen geeignete Werte für **Height** und **Width**.
- Das Fenster erhält einen **Panel**-Layoutcontainer.
- In den Container fügen wir ein horizontal und vertikal zentriertes **Button**-Steuerelement mit Beschriftung ein.

Hier ist der XAML-Code zum Fenster zu sehen:[1]

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="Einstieg.MainWindow"
        Title="Steuerelemente - Einstieg" Height="200" Width="320">
    <Panel>
        <Button HorizontalAlignment="Center" VerticalAlignment="Center"
                Content="Farbe wechseln" />
    </Panel>
</Window>
```

Weil mit dem **Button** nur ein einziges Steuerelement vorhanden ist, könnte man auf den **Panel**-Wurzelcontainer verzichten. Bei der geplanten Analyse des vom XamlX-Compiler erstellten IL-Codes soll aber ein typisches Bild zu sehen sein. Daher verwenden wir ein Fenstersdesign *mit* Wurzelcontainer.

Wie wir aus dem Abschnitt 4.7.1 wissen, erstellt des Anwendungsobjekt aus der Klasse `App` in der Methode **OnFrameworkInitializationCompleted()** das Fensterobjekt aus der Klasse `MainWindow`:

---

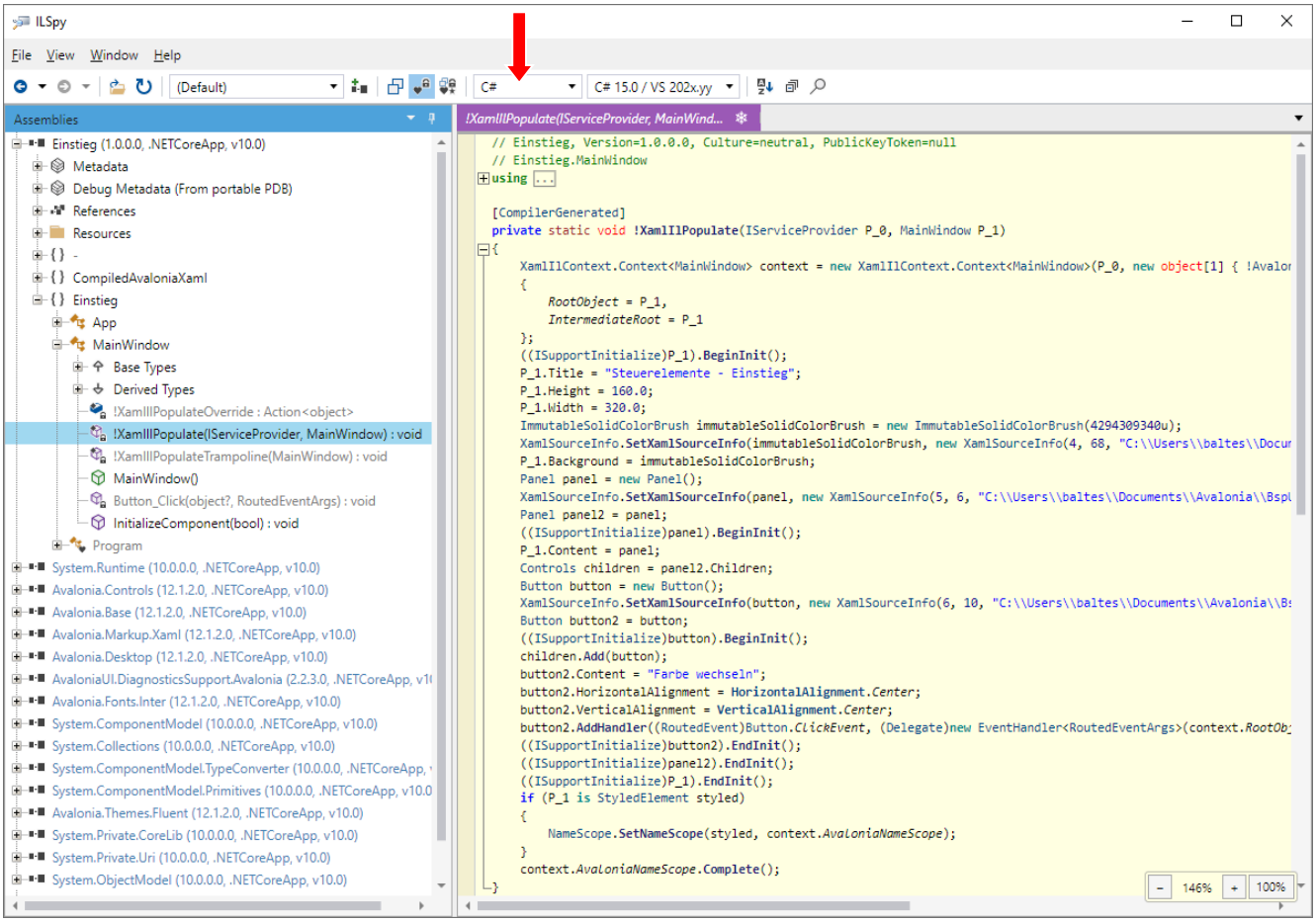
[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Steuerelemente\Einstieg

```
public partial class App : Application {
    public override void Initialize() {
        AvaloniaXamlLoader.Load(this);
    }

    public override void OnFrameworkInitializationCompleted() {
        if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop) {
            desktop.MainWindow = new MainWindow();
        }
        base.OnFrameworkInitializationCompleted();
    }
}
```

Das Instanziieren von `MainWindow`-Memberobjekten vom Typ **Panel** und **Button** lässt sich in den C# - Quellcodedateien des Projekts *nicht* beobachten. Der in Avalonia enthaltene XamlX-Compiler übersetzt den XAML-Code direkt in IL-Code (*Immediate Language*), und zur Laufzeit erstellt die vom Anwendungsobjekt aufgerufene statische Methode **Load()** der Klasse **AvaloniaXamlLoader** die zu XAML-Instanzelementen gehörenden Objekte.

Wenn man die beim Erstellen des Programms entstehende Assembly-Datei **Einstieg.dll** mit ILSpy (vgl. Abschnitt 2.6) öffnet und das Dekompilieren (die Übersetzung von IL-Code in C# - Code) anfordert, dann sieht man in der `MainWindow`-Methode **!XamlIlPopulate()**, deren Name in C# verboten, in IL aber erlaubt ist, vertraute C# - Anweisungen zum Erstellen und Konfigurieren von `MainWindow`-Memberobjekten aus den Klassen **Panel** und **Button**:



11.2.2 Elementare Eigenschaften

Die anschließend beschriebenen Eigenschaften eines Steuerelements legen seine Ausdehnung sowie seine Position im übergeordneten Element (Layoutcontainer(zelle) oder **ItemsControl**-Steuerelement) fest. Sie sind in der Klasse **Avalonia.Layout.Layoutable** definiert, also auch bei Layoutcontainern und Grafikelementen anwendbar. In der Regel handelt es sich um Abhängigkeitseigenschaften (siehe Abschnitt 7.6):

- **Width**, **Height**
  Mit den Eigenschaften **Width** und **Height** vom Typ **double** wählt man für die Breite bzw. Höhe eine Anzahl von geräteunabhängigen Pixeln (1/96 Zoll pro Einheit). Der voreingestellte Wert **NaN** (im XAML-Code: **Auto**, Groß-/Kleinschreibung beliebig) bewirkt eine automatische Berechnung, die sich am Inhalt orientiert.
- **MinWidth**, **MaxWidth, MinHeight, MaxHeight**
  Mit den Eigenschaften **MinWidth**, **MaxWidth, MinHeight** und **MaxHeight** vom Typ **double** wählt man für die Breite bzw. Höhe einen minimalen bzw. maximalen Wert. Im folgenden Beispiel resultiert ein Fenster mit der fixierten Höhe 200 und der initialen Breite 400, die der Benutzer zwischen 300 und 500 verändern kann:[1]

  ```
  <Window . . .
      Height="200" MinHeight="200" MaxHeight="200"
      Width="400" MinWidth="300" MaxWidth="500">
       . . .
  </Window>
  ```

  Konflikte zwischen **MinWidth**, **MaxWidth** und **Width** löst Avalonia so:
  - **MinWidth** dominiert **MaxWidth**
  - **MaxWidth** dominiert **Width**

  Eine analoge Aussage gilt für **MinHeight**, **MaxHeight** und **Height**.
- **Margin**
  Mit dieser Eigenschaft vom Typ **Avalonia.Thickness** legt man fest, wie viel Platz um das Element herum bis zu den Ankerseiten des übergeordneten Elements (Layoutcontainer(zelle) oder **ItemsControl**-Steuerelement) oder bis zu benachbarten Elementen freibleiben soll. Die Ankerseiten hängen von den anschließend beschriebenen Eigenschaften **HorizontalAlignment** und **VerticalAlignment** ab.

  Besteht die Möglichkeit der Überlappung von Steuerelementen, dann orientiert sich die **Margin**-Eigenschaft nur an den Ankerseiten (z. B. beim **Panel**- oder **Grid**-Container), anderenfalls auch an benachbarten Elementen (z. B. beim **StackPanel**-Container).

  Man gibt für die Randbreite per **double**-Wert eine Anzahl von geräteunabhängigen Pixeln an, wobei unterschiedlich detaillierte Angaben möglich sind:
  - Mit *einer* Zahl legt man für alle Seiten denselben Rand fest, z. B.:
    `Margin="50"`
  - Von *zwei* Zahlen legt die erste Zahl die beiden horizontalen und die zweite Zahl die beiden vertikalen Ränder fest, z. B.:
    `Margin="50,100"`
  - *Vier* Zahlen beziehen sich (in dieser Reihenfolge) auf den linken, oberen, rechten und den unteren Rand, z. B.:
    `Margin="20,30,40,50"`

[1] Die Klasse **Window** stammt von der Klasse **ContentControl** ab (siehe Abschnitt 4.2) und besitzt daher die aktuell beschriebenen Eigenschaften von Steuerelementen.

- **HorizontalAlignment**
  Diese Eigenschaft entscheidet darüber, ob das Steuerelement am linken und/oder rechten Rand des übergeordneten Elements unter Beachtung des gewünschte Randabstands (Eigenschaft **Margin**) verankert ist. Es sind vier Werte aus der Enumeration **HorizontalAlignment** im Namensraum **Avalonia.Layout** möglich, wobei deren Effekt auch vom Layoutcontainer abhängt. Die folgenden Aussagen gelten für **Panel**- und **Grid**-Container:
  - **Left**
    Das Element ist am linken Rand verankert.
  - **Right**
    Das Element ist am rechten Rand verankert.
  - **Stretch**
    Das Element ist am linken *und* am rechten Rand verankert. Seine horizontale Ausdehnung ändert sich zusammen mit dem übergeordneten Element, wenn das nicht durch die Eigenschaft **Width** verhindert wird.
  - **Center**
    Das Element ist horizontal zentriert. Ein per **Margin**-Eigenschaft eingestellter Randabstand wirkt sich auf die Zentrierung aus, sodass z. B. ein Unterschied zwischen dem linken und dem rechten Rand eine Verschiebung aus der Zentrallage bewirkt.

  Die Voreinstellung der Eigenschaft **HorizontalAlignment** hängt von der Steuerelementklasse ab und ist z. B. ...
  - gleich **Left** bei **Button**
  - und gleich **Stretch** bei **TextBox**.
- **VerticalAlignment**
  Diese Eigenschaft entscheidet darüber, ob das Steuerelement am oberen und/oder unteren Rand des übergeordneten Elements unter Beachtung des gewünschte Randabstands (Eigenschaft **Margin**) verankert ist. Es sind vier Werte aus der Enumeration **VerticalAlignment** im Namensraum **Avalonia.Layout** möglich, wobei deren Effekt auch vom Layoutcontainer abhängt. Die folgenden Aussagen gelten für **Panel**- und **Grid**-Container:
  - **Top**
    Das Element ist am oberen Rand verankert.
  - **Bottom**
    Das Element ist am unteren Rand verankert.
  - **Stretch**
    Das Element ist am oberen *und* am unteren Rand verankert, sodass sich seine vertikale Ausdehnung zusammen mit dem übergeordneten Element ändert, wobei aber ein expliziter **Height**-Wert dominiert.
  - **Center**
    Das Element ist vertikal zentriert. Ein per **Margin**-Eigenschaft eingestellter Randabstand wirkt sich auf die Zentrierung aus, sodass z. B. ein Unterschied zwischen dem oberen und dem unteren Rand eine Verschiebung aus der Zentrallage bewirkt.

  Die Voreinstellung der Eigenschaft **VerticalAlignment** hängt von der Steuerelementklasse ab und ist z. B. ...
  - gleich **Center** bei **Button**
  - und gleich **Stretch** bei **TextBox**.

Welche Werte für die Eigenschaften **HorizontalAlignment** und **VerticalAlignment** realisierbar sind, hängt vom Typ des Layoutcontainers ab, z. B. ...

- haben **Button**-Elemente in einem **Panel**- oder **Grid**-Container per Voreinstellung die horizontale Ausrichtung **Left** und können alternative horizontale Ausrichtungen erhalten,
- sind die Elemente in einem horizontal orientierten **StackPanel**-Container grundsätzlich von links nach rechts angeordnet, d. h. das **HorizontalAlignment**-Attribut der Elemente ist wirkungslos.

Bei der vertikalen Ausrichtung haben die Elemente der genannten Layoutcontainer die freie Wahl.

Es folgen Eigenschaften, die den *Inhalt* von XAML-Elementen betreffen:

- **Content**
  Diese in **Avalonia.Controls.ContentControl** definierte Eigenschaft vom Typ **Object** legt den Inhalt fest, z. B. bei einem **Button**-Objekt:
  ```
  <Button Content="Farbe wechseln"  . . . />
  ```
- **Padding**
  Diese in **Avalonia.Controls.Primitives.TemplatedControl** definierte Eigenschaft vom Typ **Avalonia.Thickness** legt eine freizuhaltende Randzone *innerhalb* der Elementfläche (einen Innenrand) fest. Die Angaben sind analog zur Eigenschaft **Margin** zu machen (siehe oben).
- **HorizontalContentAlignment**, **VerticalContentAlignment**
  Diese in **Avalonia.Controls.ContentControl** definierten Eigenschaften bestimmen die Ausrichtung des Inhalts innerhalb der Elementfläche. Es sind dieselben Werte möglich wie bei **HorizontalAlignment** bzw. **VerticalAlignment** (siehe oben).

11.2.3 Ereignisbehandlung

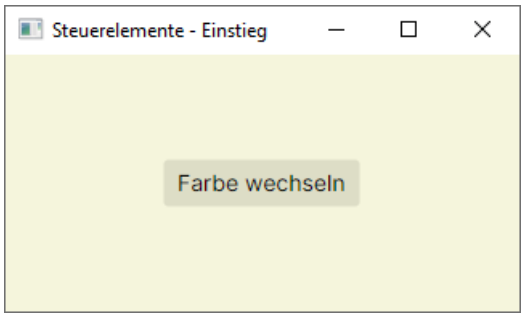
Zur Behandlung der von einem Steuerelement angebotenen Ereignisse kommen unter Verwendung von bislang im Manuskript erläuterten Techniken zwei Verfahren in Frage:

- Man kann zu einem Ereignis eine Behandlungsmethode mit passender Delegatensignatur erstellen und einem Steuerelement oder *mehreren* Steuerelementen zuweisen. Sofern es um ein Routingereignis geht, kann man die Methode nicht nur einer Ereignisquelle, sondern auch einem übergeordneten Element zuweisen. Meist wird die Behandlungsmethode in der Code Behind – Datei definiert und vom Fensterobjekt ausgeführt (siehe Abschnitt 11.2.3.1).
- Um ein Ereignis auszulösen, rufen Steuerelementklassen in der Regel eine Methode mit dem Namenspräfix **On** auf, die aufgrund der Modifikatoren **virtual** und **protected** in abgeleiteten Klassen überschreibbar ist. Eine abgeleitete Klasse kann in der überschreibenden Methode das Ereignis behandeln und/oder die Behandlungslogik ändern (siehe Abschnitt 11.2.3.2). Dieses Verfahren setzt die Definition einer abgeleiteten Steuerelementklasse voraus und kommt daher seltener zum Einsatz.

Im Zusammenhang mit der MVVM-Anwendungsarchitektur werden wir mit der sogenannten *Kommandobindung* eine weitere, speziell für professionelle Anwendungen relevante Option kennenlernen (siehe Abschnitt 12.3). Dabei führt ein Objekt der zum Fenster gehörenden *Ansichtsmodellklasse* die Behandlungsmethoden aus, und die Code Behind – Datei bleibt fast leer.

11.2.3.1 Behandlungsmethoden definieren und zuweisen

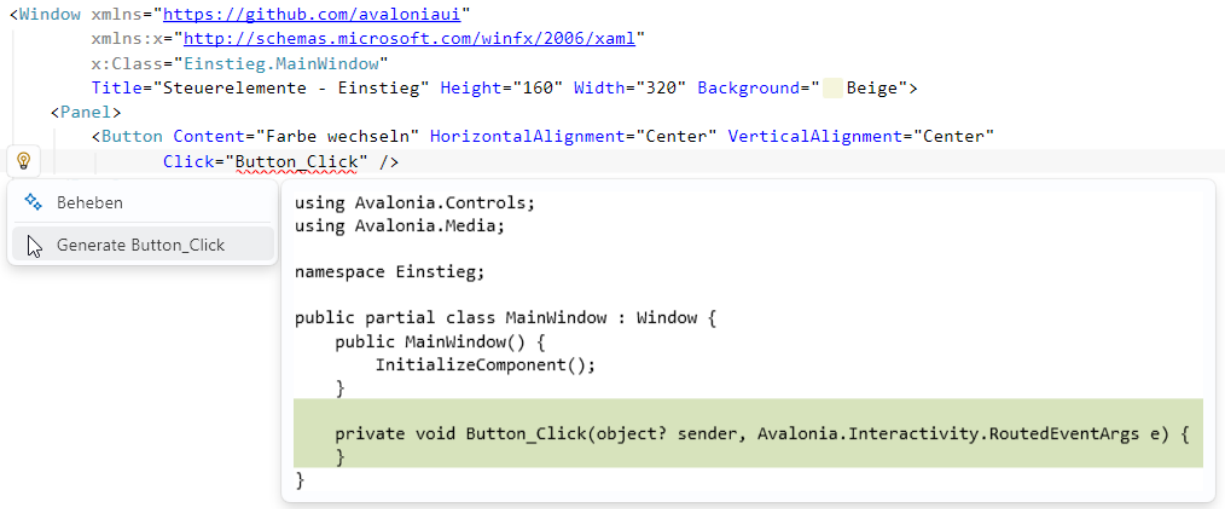
Um in einer einfachen Anwendung auf ein Fenster- oder Steuerelementereignis zu reagieren, definiert man in der Code-Behind - Datei eine Behandlungsmethode vom passenden Delegatentyp und weist diese Methode einem Steuerelement zu. Im aktuellen Beispiel



ist das **Click**-Ereignis des **Button**-Objekts zu behandeln. Dazu kann man ...

- mit der Zuweisung per XAML-Code starten, das Visual Studio per Glühbirnen-Schnellaktion zur Erstellung eines Behandlungsmethodenrohlings auffordern und diesen vervollständigen.
- Die Behandlungsmethode definieren und anschließend im XAML-Code zuweisen.

Wir wählen den ersten Weg und erhalten als Ergebnis der Schnellaktion



den folgenden Methodenrohling:

```
private void Button_Click(object? sender, Avalonia.Interactivity.RoutedEventArgs e) {
}
```

Durch das automatische Erstellen des Methodenkopfs spart man Aufwand und Fehlermöglichkeiten.

Im Beispiel soll bei jedem Mausklick auf den Schalter die Hintergrundgestaltung des Fensters zwischen **Brushes.Beige** (Pinsel mit Volltonfarbe **Beige**) und **Brushes.LightGray** (Pinsel mit Volltonfarbe **LightGray**) wechseln. Wir ergänzen eine Anweisung zur Änderung der **Background**-Eigenschaft des Fensters:

```
private void Button_Click(object? sender, Avalonia.Interactivity.RoutedEventArgs e) {
    Background = Background.Equals(Brushes.Beige)
                    ? Brushes.LightGray : Brushes.Beige;
}
```

Verwendet man statt der Methode **Equals()** den Identitätsoperator,

```
Background = Background == Brushes.Beige ? Brushes.LightGray : Brushes.Beige;
```

dann bleibt der *erste* Mausklick nach dem Programmstart folgenlos, weil offenbar in dieser Lage aufgrund der Eigenschaftsinitialisierung per XAML

```
<Window . . . Background="Beige">
```

verschiedene **IBrush**-Objekte mit derselben Farbe **Beige** im Spiel sind.

Wo aufgrund der Tätigkeit des XamlX-Compilers die Ereignisregistrierung im IL-Code stattfindet, verrät der Abschnitt 6.2.2.2.

11.2.3.2 On-Methoden überschreiben

Wenn man zu einer Steuerelementklasse eine Ableitung definiert, dann besteht eine Alternative zu der im Abschnitt 11.2.3.1 beschriebenen Technik der Ereignisbehandlung. Um ein Ereignis auszulösen, rufen Avalonia-Klassen bis auf wenige Ausnahmen eine Methode mit dem Namenspräfix **On** auf, z. B.

- **OnPointerEntered()**
- **OnPointerPressed()**

Weil die **On**-Methoden als **protected** sowie **virtual** definiert sind, kann man sie in abgeleiteten Klassen überschreiben, um …

- die gewünschte Ereignisbehandlung direkt in der **On**-Methode vorzunehmen
- und/oder die Behandlungslogik zu ändern.

Das Überschreiben der **On**-Methode ist bei den Ereignissen von Steuerelementen (z. B. **Button**) nicht unbedingt die bevorzugte Technik, weil man dazu eine abgeleitete Klasse definieren muss. Bei den Fensterereignissen ist die Technik hingegen leicht nutzbar, weil wir die Klasse **Window** regelmäßig beerben.

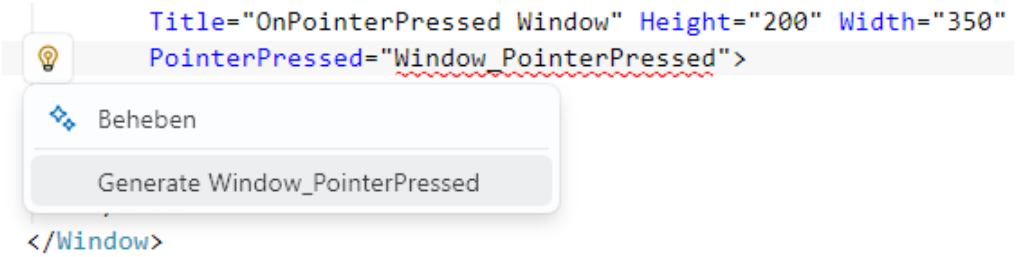
Wir betrachten das **PointerPressed**-Ereignis des Fensterobjekts und verwenden zunächst das im Abschnitt 11.2.3.1 beschriebenen Verfahren zur semiautomatischen Erstellung einer Behandlungsmethode:

- Durch das Registrierung der noch nicht vorhandenen Behandlungsmethode `Window_PointerPressed()` im XAML-Code

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="OnPointerPressedWindow.MainWindow"
        Title="OnPointerPressed Window" Height="200" Width="350"
        PointerPressed="Window_PointerPressed">
    <Label x:Name="label"/>
</Window>
```

  provozieren wir eine Fehlermeldung.

- Über die Glühbirnen-Schnellaktion zum „Fehler“



  erhalten wir einen Behandlungsmethodenrohling in der Code Behind - Datei.

Nach dem Erstellen des Programms (z. B. per **Erstellen > Projektmappe neu erstellen**) ist die im XAML-Code deklarierte `MainWindow`-Instanzvariable `label` in der Code Behind – Datei bekannt, sodass wir die Behandlungsmethode vervollständigen können:[1]

---

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Steuerelemente\On-Methoden\OnPointerPressedWindow

```
using Avalonia;
using Avalonia.Controls;

namespace OnPointerPressedWindow;

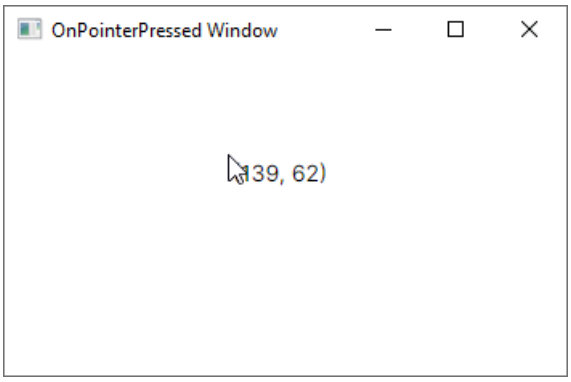
public partial class MainWindow : Window {
  public MainWindow() {
    InitializeComponent();
  }

  private void Window_PointerPressed(object? sender,
                                     Avalonia.Input.PointerPressedEventArgs e) {
    label.Margin = new Thickness(e.GetPosition(this).X, e.GetPosition(this).Y, 0, 0);
    label.Content = "(" + e.GetPosition(this).ToString() + ")";
  }
}
```

Die Methode `Window_MouseDown()` ...

- legt die **Label**-Position über die **Margin**-Eigenschaft (siehe Abschnitt 11.2.2) neu auf die Klickstelle fest, die vom Parameterobjekt vom Typ **PointerPressedEventArgs** über die Methode **GetPosition()** zu erfahren ist,
- und verarbeitet die Koordinaten der Klickstelle zur neuen **Content**-Eigenschaftsausprägung des **Label**-Objekts.

Somit kann das Programm die Koordinaten einer Klickstelle am Ort des Geschehens anzeigen, z. B.:



Da im Projekt die aus **Window** abgeleitete Klasse `MainWindow` entsteht, ist eine **PointerPressed** -Behandlung auch durch das Überschreiben der geerbten **Window**-Methode **OnPointerPressed()** möglich, z. B.:

```
protected override void OnMouseDown(MouseButtonEventArgs e) {
    base.OnMouseDown(e);
    label.Margin = new Thickness(e.GetPosition(this).X, e.GetPosition(this).Y, 0, 0);
    label.Content = "(" + e.GetPosition(this).ToString() + ")";
}
```

Es resultiert exakt dasselbe Programmverhalten.

Bei einer Ereignisbehandlung durch das Überschreiben der zuständigen **On**-Methode spart man sich die Zuweisung an ein Steuerelement. Man darf aber nicht vergessen, in der ersten Anweisung der Überschreibung die Basisklassenmethode aufzurufen. Eine Unterlassung dieses Aufrufs hat bei vielen Ereignissen das meist unerwünschte Übergehen von zugewiesenen Behandlungsmethoden zur Folge. Das trifft z. B. für die Methode **OnClick()** der Klasse **ButtonBase** zu, die das **Click**-Ereignis per **RaiseEvent()** auslöst:

```
    protected virtual void OnClick() {
        if (IsEffectivelyEnabled) {
            . . .
            var e = new RoutedEventArgs(ClickEvent);
            RaiseEvent(e);
            . . .
        }
    }
```

Ohne den **RaiseEvent()** – Aufruf kann das Avalonia-Ereignissystem keine Behandlungsmethode starten, die über den Operator += oder mit der Methode **AddHandler()** zugewiesen wurde. In der folgenden **Button**-Ableitung namens `MaiButton` passiert dieser Fehler beim Überschreiben von **OnClick()**, z. B.:[1]

```
using Avalonia.Controls;
using Avalonia.Interactivity;
using MsBox.Avalonia;
using MsBox.Avalonia.Enums;
using System;

namespace Vergessener_base_Aufruf;

public class MaiButton : Button {
    protected override Type StyleKeyOverride => typeof(Button);

    protected async override void OnClick() {
        //base.OnClick(); // Erforderlicher Aufruf der überschriebenen Methode
        var box = MessageBoxManager.GetMessageBoxStandard("MaiButton",
                  "Überschreibende On-Methode von MaiButton", ButtonEnum.Ok);
        var window = TopLevel.GetTopLevel(this) as Window;
        if (window is not null)
            await box.ShowWindowDialogAsync(window);
    }
}

public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
    }

    private async void Button_Click(object sender, RoutedEventArgs e) {
        var box = MessageBoxManager.GetMessageBoxStandard("Button",
                              "Zugewiesene Click-Behandlungsmethode", ButtonEnum.Ok);
        var window = GetTopLevel(this) as Window;
        if (window is not null)
            await box.ShowWindowDialogAsync(window);
    }
}
```

Die Verwendung einer selbst definierte Steuerelementklasse im XAML-Code beschreibt der Abschnitt 7.4:

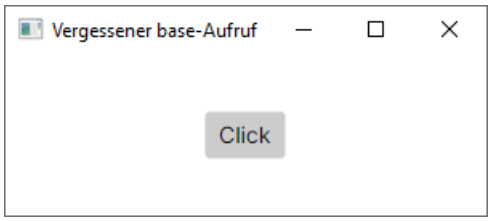
---

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Steuerelemente\On-Methoden\Vergessener base-Aufruf

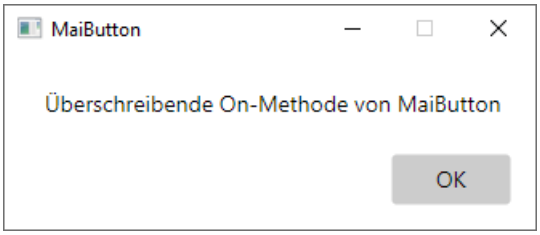
```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="using:Vergessener_base_Aufruf"
        x:Class="Vergessener_base_Aufruf.MainWindow"
        Title="Vergessener base-Aufruf" Height = "100" Width = "300">
   <local:MaiButton Content="Click" HorizontalAlignment="Center"
                    Click="Button_Click"/>
</Window>
```

Weil Avalonia 12 kein Analogon zur WPF-Klasse **MessageBox** bietet, verwenden wir erneut das NuGet-Paket **MessageBox.Avalonia** (siehe Abschnitt 4.9).

Aufgrund des Programmierfehlers wird bei einem Mausklick auf den Schalter



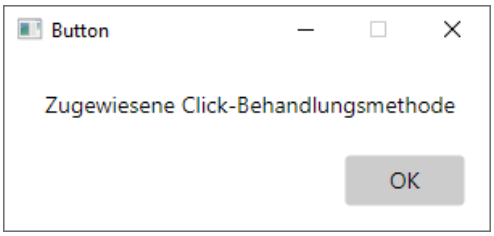
nur die **OnClick()** - Überschreibung ausgeführt:



Mit der korrekten Überschreibung

```
protected async override void OnClick() {
    base.OnClick(); // Erforderlicher Aufruf der überschriebenen Methode
    var box = MessageBoxManager.GetMessageBoxStandard("MaiButton",
              "Überschreibende On-Methode von MaiButton", ButtonEnum.Ok);
    var window = TopLevel.GetTopLevel(this) as Window;
    if (window is not null)
        await box.ShowWindowDialogAsync(window);
}
```

bleiben andere Ereignisbehandlungsmethoden ggf. im Spiel, z. B.:



Bei der **Window**-Methode **OnPointerPressed()** hat die Unterlassung des Basismethodenaufrufs keine Konsequenzen, weil die (von **InputElement** geerbte) Methode einen leeren Rumpf hat:

```
protected virtual void OnPointerPressed(PointerPressedEventArgs e) { }
```

11.3 Standardkomponenten

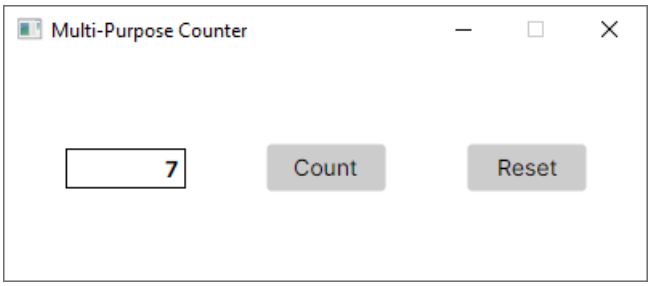
Dieser Abschnitt behandelt häufig benötigte Steuerelemente: Schaltflächen, Kontrollkästchen, Optionsfelder, Texteingabefelder, Listen- und Kombinationsfelder sowie Menü-Items.

11.3.1 Schaltflächen

Schaltflächen sind durch Objekte der Klasse **Button** realisiert.

11.3.1.1 Beschriftete Schaltflächen

Wir betrachten zunächst ein einfaches Zählprogramm mit zwei beschrifteten Schaltflächen:



Der folgende XAML-Code erledigt das GUI-Design und die Ereignismethodenvereinbarung:[1]

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="ButtonText.MainWindow"
        Width="400" Height="140" CanResize="False"
        Title="Multi-Purpose Counter" >

    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center"
                Spacing="50" Button.Click="Button_Click">
        <Label  Name="label" Content="0" Width="75" VerticalAlignment="Center"
                FontWeight="Bold" BorderThickness="1" BorderBrush="Black"
                HorizontalContentAlignment="Right" />
        <Button Name="count" Content="Count" Width="75" VerticalAlignment="Center"
                HorizontalContentAlignment="Center" IsDefault="True" />
        <Button Name="reset" Content="Reset" Width="75" VerticalAlignment="Center"
                HorizontalContentAlignment="Center" />
    </StackPanel>
</Window>
```

Weil eine Änderung der Fenstergröße nicht erwünscht ist, erhält im Wurzelelement das XAML-Attribut bzw. die **Window**-Eigenschaft **CanResize** den Wert **False**. Dadurch verschwindet erwartungsgemäß auch der Titelzeilenschalter zum Maximieren des Fensters.

Die **Height**-Eigenschaften der Steuerelemente ermittelt Avalonia automatisch unter Berücksichtigung von Schriftgröße und Innenrand.

Das **Label**-Steuerelement erhält die Schriftauszeichnung **fett**,

```
FontWeight="Bold"
```

einen Rahmen mit der Linienstärke 1 in schwarzer Farbe

```
BorderThickness="1" BorderBrush="Black"
```

und die (bei einer Zahlenanzeige übliche) rechtsbündige Textausrichtung:

```
HorizontalContentAlignment="Right"
```

---

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Steuerelemente\Button\Text

Die für das **Click**-Routingereignis *beider* **Button**-Steuerelemente zuständige Ereignisbehandlungsmethode `Buttons_Click()` ermittelt die für ihr Verhalten relevante Ereignisquelle über die **Source**-Eigenschaft des zweiten Parameters (vom Typ **RoutedEventArgs**):

```
public partial class MainWindow : Window {
    long anzahl;

    public MainWindow() {
        InitializeComponent();
    }

    private void Buttons_Click(object sender, RoutedEventArgs e) {
        if (e.Source == count)
            anzahl++;
        else
            anzahl = 0;
        label.Content = anzahl.ToString();
    }
}
```

Beim aufperlenden **Click**-Routingereignis empfiehlt es sich, die für beide **Button**-Objekte zuständige **Click**-Behandlungsmethode dem gemeinsamen **StackPanel**-Container zuzuweisen (vgl. Abschnitt 7.6):

```
<StackPanel Orientation="Horizontal" . . . Button.Click="Buttons_Click">
     . . .
</StackPanel>
```

Der beim Aufruf an `Buttons_Click()` als erster Parameter übergebene `sender` zeigt dann auf den **StackPanel**-Container, und die **Source**-Eigenschaft des zweiten Parameters (vom Typ **RoutedEventArgs**) zeigt auf die Ereignisquelle (den vom Klick betroffenen Schalter).

11.3.1.2 Standard- und Escape-Schalter

Damit im aktuellen Beispielprogramm auch die **Enter**-Taste das **Click**-Ereignis des Schalters zum Inkrementieren des Zählerstands auslöst, hat die **IsDefault**-Eigenschaft dieses Schalters den Wert **true** erhalten:

```
<Button Name="count" Content="Count" Width="75" VerticalAlignment="Center"
        HorizontalContentAlignment="Center" IsDefault="True" />
```

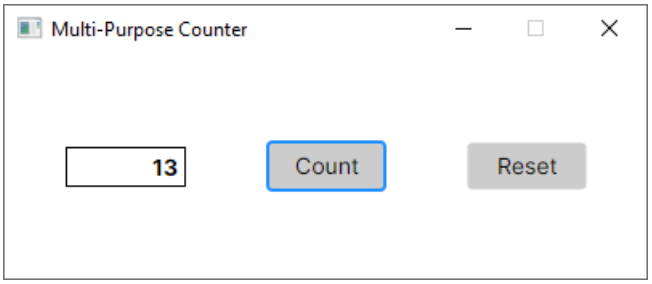
Das macht ihn zum **Standardschalter** des Fensters (engl.: *default button*), und ein Multi-Counter - Benutzer kann nach dem Start sofort die **Enter**-Taste zum Zählen verwenden, ohne zuvor den Tastaturfokus (z. B. per Tabulatortaste) auf den **Count**-Schalter setzen zu müssen (siehe Abschnitt 11.3.4).

Allerdings kommt die Rolle des Standardschalters optisch *nicht* zum Ausdruck. Ein Lösungsversuch über einen Stil für **Button**-Objekte mit dem Wert **true** für die Eigenschaft **IsDefault**

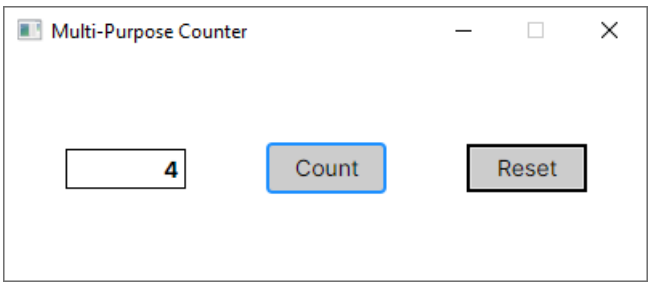
```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="ButtonText.MainWindow"
        Width="400" Height="140" CanResize="False"
        Title="Multi-Purpose Counter" >
    <Window.Styles>
        <Style Selector="Button[IsDefault=true]">
                <Setter Property="BorderThickness" Value="2" />
                <Setter Property="BorderBrush" Value="DodgerBlue" />
        </Style>
    </Window.Styles>
```

```
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center"
       . . .
    </StackPanel>
</Window>
```

erweist sich als kurzsichtig. Man erhält zwar ein an WPF erinnerndes Design mit einer blaue Umrandung des Schalters:



Wenn aber der `Reset`-Schalter den Tastaturfokus besitzt (vgl. Abschnitt 11.3.4),



dann löst die **Enter**-Taste das **Click**-Ereignis des **Reset**-Schalters aus mit dem Wert 0 als Folge. Die blaue Umrandung als Hinweis auf die **Enter**-Ansprechbarkeit vom Standardschalter zu entfernen, sobald ein anderes Steuerelement den Tastaturfokus besitzt, erfordert einen erheblichen Aufwand.[1] Daher kehren wir zur Avalonia-Voreinstellung *ohne* optische Auszeichnung des Standardschalters zurück.

Allerdings zeigt sich im Beispielprogramm immer noch ein gravierendes Bedienungsproblem, von dem auch WPF betroffen ist (siehe Abschnitt 12.12.4.1.2 in Baltes-Götz & Frischknecht 2026):

- Erhält ein Schalter den Tastaturfokus per Maus, dann erscheint *keine* schwarze Umrandung.
- Der per **Enter**- und Leertaste auslösbare Schalter ist also nicht erkennbar, was bei einem Fenster mit mehreren Schaltflächen ein erhebliches Fehlerrisiko verursacht.

Bei diesem Problem spielt die Standardschaltfläche keine Rolle. Die Ursache für den Mangel besteht darin, dass ein per Mausklick erworbener Tastaturfokus unsichtbar ist. Das gilt bei Avalonia auch für ein unter Linux ausgeführtes Programm.

Im Beispielprogramm ist kein nennenswerter Schaden zu befürchten:

- Per Mausklick auf den **Reset**-Schalter setzt der Benutzer den Zähler bewusst auf 0.
- Wenn er anschließend per `Enter`-Taste den Zählerstand erhöhen möchte und unerwartet den **Reset**-Schalter auslöst, dann bleibt es beim Zählerstand 0.

In Abhängigkeit von den Funktionen der in einem Fenster vorhandenen Schalter kann die Betätigung eines unerwarteten Schalters per `Enter` aber zu ärgerlichen Effekten führen.

Zur Lösung des Problems kann man über das Attribut **HotKey** die **Enter**-Taste unabhängig von der Standardrolle und vom Tastaturfokus einem Schalter definitiv zuordnen, z. B.:[2]

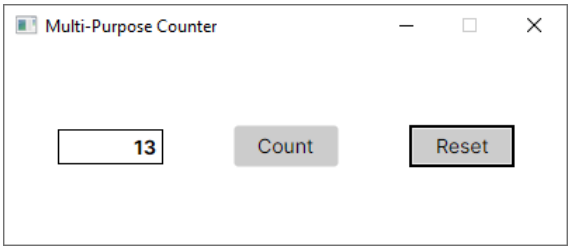
---

[1] https://github.com/AvaloniaUI/Avalonia/discussions/17095

[2] https://docs.avaloniaui.net/docs/input-interaction/keyboard-and-hotkeys

```
<Button Name="count" Content="_Count" Width="75" VerticalAlignment="Center"
        HorizontalContentAlignment="Center" HotKey="Enter" />
```

Aufgrund diese Einstellung löst in der folgenden Lage mit dem **Reset**-Schalter als Fokusinhaber die **Enter**-Taste das **Click**-Ereignis des **Count**-Schalters aus:



Nun ist das Zählergebnis vom dem unerwünschten Löschen durch das Drücken der **Enter**-Taste geschützt. Allerdings weicht das Programm von der Konvention ab, dass der Fokusschalter per **Enter** auslösbar ist.

Analog zur Standardschaltflächen-Ernennung kann man ein **Button**-Objekt über die Eigenschaft **IsCancel** als Escape-Schalter des Fensters festlegen. Sein **Click**-Ereignis lässt sich dann auch per **Esc**-Taste auslösen. Einen visuellen Indikator für den Escape-Schalter gibt es nicht.

11.3.1.3 Alt-Zugriffstaste

Um das **Click**-Ereignis eines Befehlsschalters auch über einen **Alt**-Tastenbefehl (über eine sogenannte *Zugriffstaste*) auslösen zu können, stellt man dem zugehörigen Buchstaben im **Content**-Attribut einen Unterstrich voran, z. B.:

```
<Button Name="reset" Content="_Reset" Width="75" VerticalAlignment="Center"
        HorizontalContentAlignment="Center" IsCancel="True" />
```

Im Beispiel ist **Click**-Ereignis des Schalters durch die Tastenkombination **Alt+R** auslösbar.

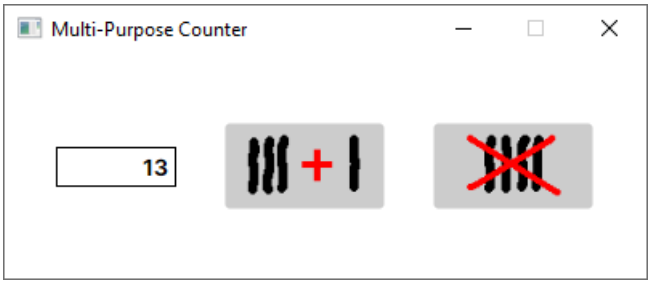
Definierte Zugriffstasten sind im laufenden Programm spätestens nach dem einmaligen Drücken der **Alt**-Taste durch eine Unterstreichung hervorgehoben, z. B.:



Mit Hilfe eines **AccessText**-Elements lässt sich eine Zugriffstaste auch für eine Schaltfläche realisieren, die ein Bild zusätzlich zu einer Beschriftung auf der Oberfläche zeigt (siehe Abschnitt 11.3.1.4).

11.3.1.4 Bilder auf Schaltflächen

Um dem oben vorgestellten Zählprogramm ein individuelles Design zu geben, ersetzen wir die Beschriftungen der Schaltflächen durch Bilder:

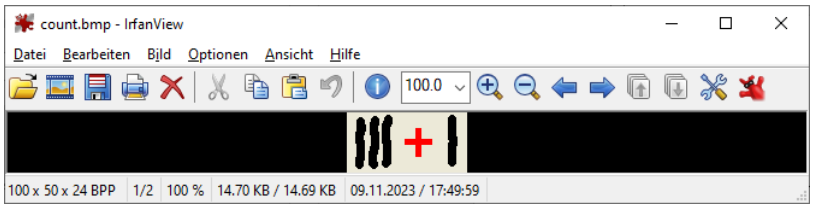


Für das Beispiel sind zunächst Bitmap-Dateien (Format **bmp** oder **png**) mit einer passenden Pixelmatrix zu erstellen (hier gewählt: 50 Zeilen und 100 Spalten), was z. B. mit der Windows-Zugabe *Paint* geschehen kann.

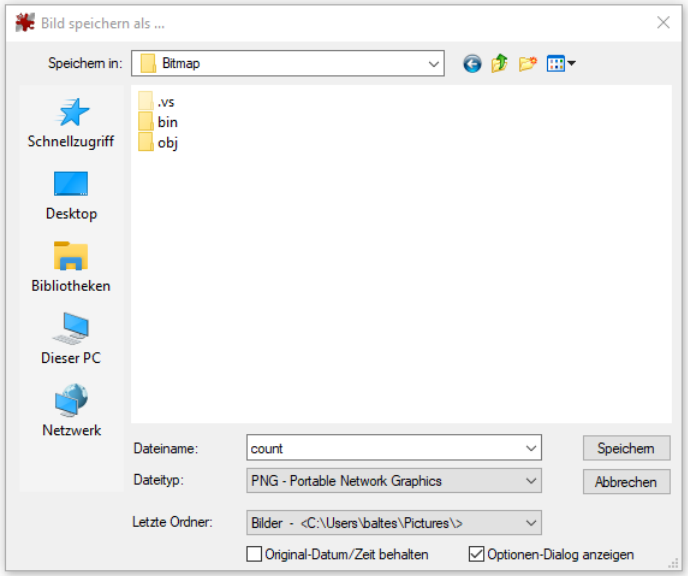
Wie man ein Bild in ein Projekt aufnimmt, beschreibt der Abschnitt 7.3. Um ein Bild auf eine Schaltfläche zu befördern, lässt man im XAML-Code zum **Button**-Startelement das **Content**-Attribut weg und ergänzt stattdessen ein Element vom Typ **Image** als Wert der Inhaltseigenschaft z. B.:[1]

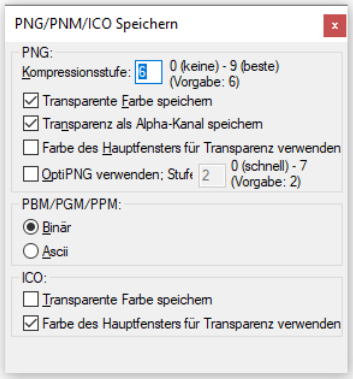
```
<Button Name="count" Width="75" IsDefault="True">
    <Image Source="avares://ButtonBitmap/Assets/count.png"
           RenderOptions.BitmapInterpolationMode="HighQuality" />
</Button>
```

Macht man die Hintergrundfarbe eines Bilds transparent, dann harmoniert der dekorierte Schalter unabhängig vom eingestellten Avalonia-Design optisch mit den restlichen Fensterbestandteilen. Eine ***PNG***-Datei (***P**ortable **N**etwork **G**raphics*) mit transparenter Hintergrundfarbe lässt sich z. B. mit der Freeware **IrfanView**[2]



durch eine Option im **Speichern unter** - Dialog erstellen:

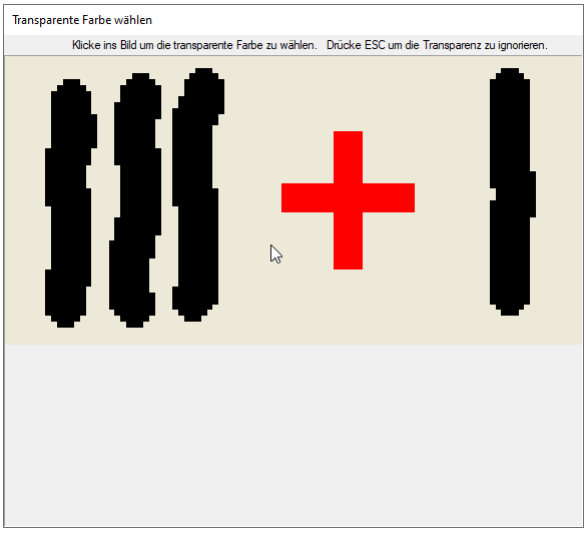




---

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Steuerelemente\Button\Bitmap

[2] Homepage: http://www.irfanview.de/



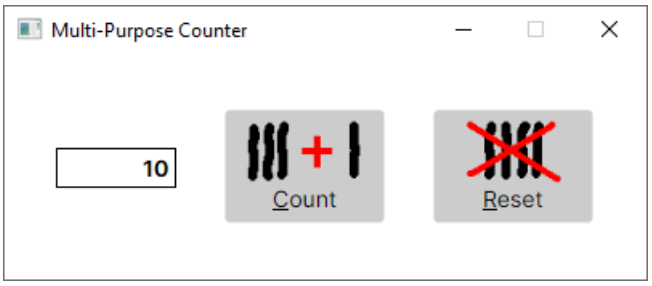
Eine **Button**-Oberfläche mit Bild *und* Beschriftung realisiert man mit Hilfe eines **StackPanel**-Containers zur Verwaltung der Oberfläche, z. B.:

```
<Button Name="count" Width="100" IsDefault="True">
    <StackPanel>
        <Image Source="avares://ButtonBitmap/Assets/count.png"
               RenderOptions.BitmapInterpolationMode="HighQuality" />
        <TextBlock HorizontalAlignment="Center">Reset</TextBlock>
    </StackPanel>
</Button>
```

Soll der Schalter eine **Alt**-Zugriffstaste erhalten (siehe Abschnitt 11.3.1.3), dann ersetzt man den **TextBlock** durch ein **AccessText**-Element:

```
<Button Name="count" Width="100" IsDefault="True">
    <StackPanel>
        <Image Source="avares://ButtonBitmap/Assets/count.png"
               RenderOptions.BitmapInterpolationMode="HighQuality" />
        <AccessText HorizontalAlignment="Center">_Reset</AccessText>
    </StackPanel>
</Button>
```

Mit Bildern, Beschriftungen und Zugriffstasten sieht das Zählprogramm so aus:

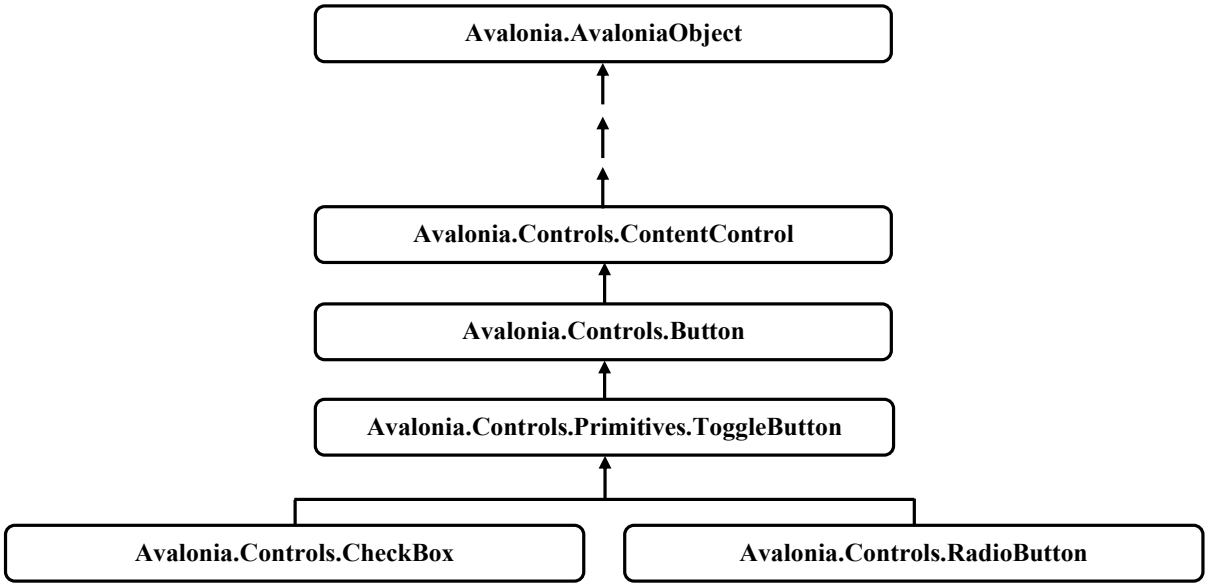


11.3.2 Kontrollkästchen und Optionsfelder

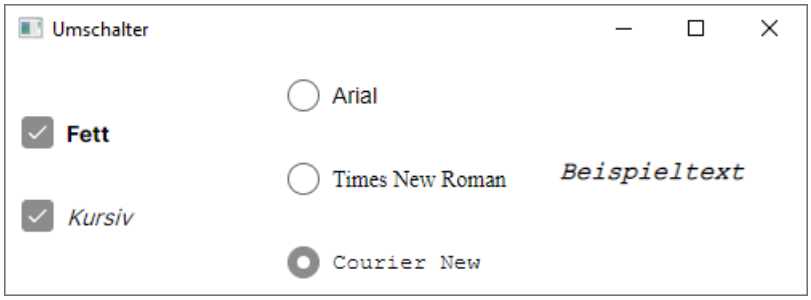
Dieser Abschnitt behandelt zwei Umschalter mit den möglichen Werten *An* und *Aus*:

- Für **Kontrollkästchen** steht die Klasse **CheckBox** bereit.
- Für ein **Optionsfeld** aus Schaltern, von denen zu einem Zeitpunkt genau einer eingerastet sein kann, verwendet man Objekte der Klasse **RadioButton**.

Die Klassen **CheckBox** und **RadioButton** haben **ToggleButton** als gemeinsame Basisklasse und stammen indirekt von der im Abschnitt 11.3.1 vorgestellten Klasse **Button** ab:



Im folgenden Programm kann man für den Text eines **Label**-Steuerelements über zwei Kontrollkästchen die Schriftauszeichnung (normal, **fett**, *kursiv*, ***fett-kursiv***) und über ein Optionsfeld die Schriftartenfamilie wählen:[1]



Beim Fensterdesign per XAML-Code kommt ein dreispaltiger **Grid**-Layoutcontainer zum Einsatz. In den beiden ersten Spalten arbeitet jeweils ein vertikal orientierter **StackPanel**-Layoutcontainer mit den **CheckBox**- bzw. **RadioButton**-Objekten. In der dritten Spalte befindet sich das **Label**-Objekt mit dem Beispieltext:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="Umschalter.MainWindow"
        Title="Umschalter" Height="150" Width="500">
    <Grid ColumnDefinitions="*,*,*">
        <StackPanel  VerticalAlignment="Center" Button.Click="CheckBox_Click">
            <CheckBox Name="chkBold" Content="Fett" Margin="10"
                      FontWeight="Bold" />
            <CheckBox Name="chkItalic" Content="Kursiv " Margin="10"
                      FontStyle="Italic"/>
        </StackPanel>
        <StackPanel  Grid.Column="1" VerticalAlignment="Center"
                     Button.Click="RadioButton_Click">
            <RadioButton Name="rbArial" Content="Arial" Margin="10"
                         FontFamily="Arial" IsChecked="True"/>
            <RadioButton Name="rbTimesNewRoman" Content="Times New Roman" Margin="10"
                         FontFamily="Times New Roman"/>
            <RadioButton Name="rbCourierNew" Content="Courier New" Margin="10"
                         FontFamily="Courier New"/>
        </StackPanel>
```

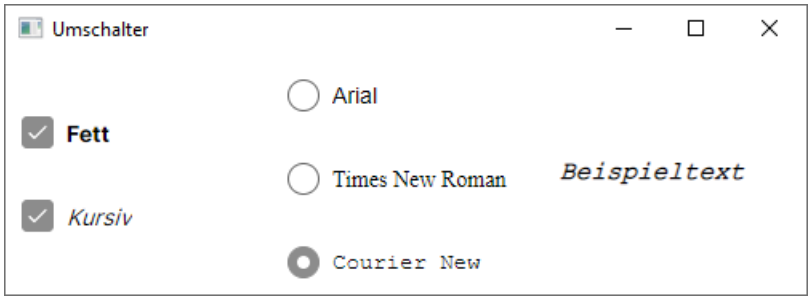
[1] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
...\BspUeb\Steuerelemente\Umschalter

```
        <Label  Name="lblText" Grid.Column="2" Content="Beispieltext"
                Margin="10" VerticalAlignment="Center"
                FontFamily="Arial" FontSize="16" />
    </Grid>
</Window>
```

Der Text im **Content**-Attribut der **CheckBox** zum Aktivieren der kursiven Auszeichnung endet mit einem Leerzeichen,

```
Content="Kursiv "
```

weil der letzte Buchstabe anderenfalls unvollständig erscheint:



Damit beim Programmstart die Option Arial markiert ist, erhält das zugehörige **RadioButton**-Objekt für die Eigenschaft **IsChecked** den Wert **true**. Das **Label**-Objekt verwendet beim Start eine Schrift aus der Familie Arial ohne Auszeichnung in der Größe 16.

Alle unmittelbar zu einem Container gehörenden **RadioButton**-Objekte bilden eine *Gruppe*, wobei nur *ein* Mitglied eingerastet sein kann (Wert **true** bei der Eigenschaft **IsChecked**). Alternativ kann man die Gruppenzugehörigkeit explizit durch das XAML-Attribut **GroupName** festlegen. Alle **RadioButton**-Elemente mit demselben **GroupName**-Wert bilden eine Gruppe, z. B.:

```
<RadioButton Name="rbOne" GroupName="g1" Content="One" ... />
```

Für die beiden Kontrollkästchen (mit den Feldnamen `chkBold` und `chkItalic`) ist im Programm dieselbe **Click**-Behandlungsmethode `CheckBox_Click()` zuständig, die für das separate Ein- bzw. Ausschalten der Schriftattribute **fett** und *kursiv* sorgt:[1]

```
private void CheckBox_Click(object sender, RoutedEventArgs e) {
    lblText.FontWeight = chkBold.IsChecked.GetValueOrDefault() ?
        FontWeights.Heavy : FontWeights.Normal;
    lblText.FontStyle = chkItalic.IsChecked.GetValueOrDefault() ?
        FontStyles.Italic : FontStyles.Normal;
}
```

Weil **IsChecked** den Typ **Nullable<bool>** (alias **bool?**) besitzt, ist in den logischen Ausdrücken der Konditionaloperatoren ein **GetValueOrDefault()** - Aufruf erforderlich, um den benötigten **bool**-Wert zu produzieren, z. B.:

```
chkBold.IsChecked.GetValueOrDefault()
```

---

[1] Das Programm verwendet zur Bezeichnung der Instanzvariablen die sogenannte *Ungarische Notation*, wobei ein Präfix den Datentyp andeutet (z. B. `lblText`). Diese in früheren Zeiten der Windows-Programmierung verbreitete Konvention gilt mittlerweile als veraltet. Microsoft empfiehlt auf der Webseite zu Namenskonventionen
https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/general-naming-conventions
explizit:
DO NOT use Hungarian notation.
Wir verwenden sie trotzdem, weil der zentrale Kritikpunkt nicht allzu zwingend erscheint: Bei einem Wechsel des Datentyps müsse der Name geändert werden. Der Wechsel des Datentyps ist bei einem Steuerelement unwahrscheinlich.

Die Methode liefert genau dann die Rückgabe **true**, wenn **IsChecked** den definierten Wert **true** besitzt.

Beim aufperlenden **Click**-Routingereignis empfiehlt es sich, die für beide **CheckBox**-Objekte zuständige **Click**-Behandlungsmethode dem gemeinsamen **StackPanel**-Container zuzuweisen (vgl. Abschnitt 7.6):

```
<StackPanel  VerticalAlignment="Center" Button.Click="CheckBox_Click">
    <CheckBox Name="chkBold" Content="Fett" Margin="10" FontWeight="Bold" />
    <CheckBox Name="chkItalic" Content="Kursiv" Margin="10" FontStyle="Italic"/>
</StackPanel>
```

Das Beispielprogramm verwendet auch für die **RadioButton**-Objekte eine gemeinsame **Click**-Behandlungsmethode:

```
private void RadioButton_Click(object sender, RoutedEventArgs e) {
    if (e.Source is RadioButton rbSource)
        lblText.FontFamily = rbSource.FontFamily;
}
```

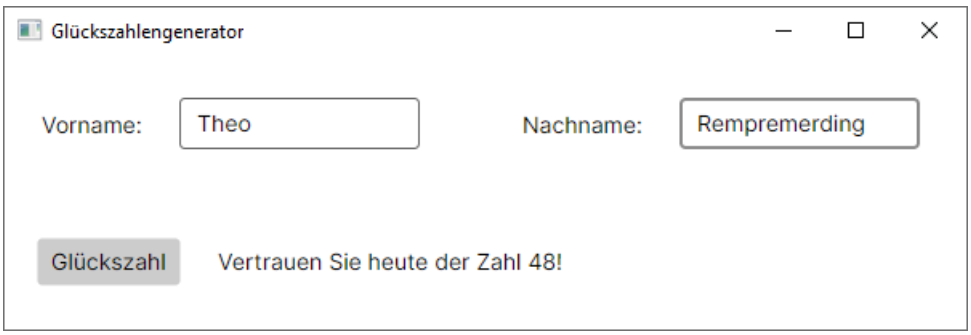
Das **Label** übernimmt vom angeklickten **RadioButton** die Schriftartfamilie (ein Objekt der Klasse **FontFamily** im Namensraum **Avalonia.Media**).

Wir weisen die **Click**-Behandlungsmethode der **RadioButton**-Objekte dem gemeinsamen **StackPanel**-Container zu:

```
<StackPanel Grid.Column="1" VerticalAlignment="Center"
            Button.Click="RadioButton_Click">
    <RadioButton Name="rbArial" Content="Arial" Margin="10"
                 FontFamily="Arial" IsChecked="True"/>
    <RadioButton Name="rbTimesNewRoman" Content="Times New Roman" Margin="10"
                 FontFamily="Times New Roman"/>
    <RadioButton Name="rbCourierNew" Content="Courier New" Margin="10"
                 FontFamily="Courier New"/>
</StackPanel>
```

11.3.3 Texteingabefelder

Kurze Texteingaben erfasst man mit **TextBox**-Steuerelementen. Das folgende Programm zur Berechnung der persönlichen Glückszahl in Abhängigkeit vom Vor- und Nachnamen verwendet zwei Objekte aus dieser Klasse:[1]



Das GUI-Design und die Ereignismethodenzuweisung erfolgen per XAML:

[1] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
...\BspUeb\Steuerelemente\Texterfassung\TextBox

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="TextBox.MainWindow"
        Title="Glückszahlengenerator" Height="170" Width="600"
        Loaded="Window_Loaded">

    <Grid RowDefinitions="*,*" ColumnDefinitions="*,*"
          TextBox.TextChanged="TextBox_TextChanged">
        <StackPanel Orientation="Horizontal" >
            <Label Content ="Vorname:" Margin="20" VerticalAlignment="Center"/>
            <TextBox Name="vorname" Width="150" VerticalAlignment="Center"/>
        </StackPanel>
        <StackPanel Orientation="Horizontal" Grid.Column="1">
            <Label Content ="Nachname:" Margin="20" VerticalAlignment="Center"/>
            <TextBox Name="nachname" Width="150"  VerticalAlignment="Center"/>
        </StackPanel>
        <StackPanel Orientation="Horizontal" Grid.ColumnSpan="2" Grid.Row="1">
            <Button Content="Glückszahl" Margin="20"
                    Click="Button_Click" Focusable="False" IsDefault="True"
                    VerticalAlignment="Center"/>
            <Label Name="info" Width="400" VerticalAlignment="Center" />
        </StackPanel>
    </Grid>
</Window>
```

Über die **TextBox**-Eigenschaft **Text** kann man auf den Inhalt eines Texteingabefeldes zugreifen, z. B. in der folgenden Behandlungsmethode zum **Click**-Ereignis des Befehlsschalters:

```
private void Button_Click(object sender, RoutedEventArgs e) {
    string vn = vorname.Text?.ToUpper() ?? "";
    string nn = nachname.Text?.ToUpper() ?? "";
    int seed = 0;
    if (vn.Length > 0 && nn.Length > 0) {
        foreach (char c in vn)
            seed += c;
        foreach (char c in nn)
            seed += c;
        var zzg = new Random(DateTime.Today.Day + seed);
        info.Content = "Vertrauen Sie heute der Zahl " +
            (zzg.Next(100) + 1).ToString() + "!";
        valToRemove = true;
    }
}
```

Eine beim Ereignis **TextChanged** registrierte Methode macht es möglich, auf eine Veränderung der **Text**-Eigenschaft zu reagieren. Im Beispielprogramm verschwindet die Glückszahlenanzeige, sobald sich einer der zugrundeliegenden Texte ändert:

```
private void TextBox_TextChanged(object sender, TextChangedEventArgs e) {
    if (valToRemove) {
        info.Content = strInst;
        valToRemove = false;
    }
}
```

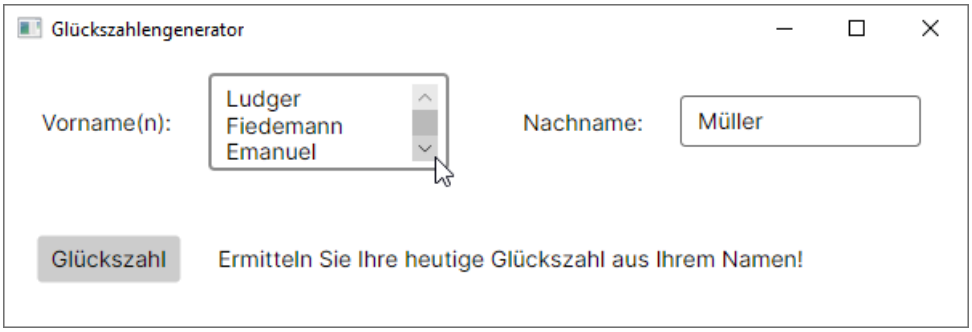
Beim aufperlenden **TextChanged**-Routingereignis empfiehlt es sich, die für beide **TextBox**-Objekte zuständige Behandlungsmethode dem gemeinsamen **Grid**-Container zuzuweisen (siehe Abschnitt 7.6):

```
<Grid RowDefinitions="*,*" ColumnDefinitions="*,*"
      TextBox.TextChanged="TextBox_TextChanged">
  . . .
</Grid>
```

TextBox-Steuerelemente bieten einigen Bedienungskomfort, z. B.:

- Textmarkierung per Maus oder Tastatur
- Kommunikation mit der Zwischenablage über die Tastenkombinationen **Strg+C**, **Strg+X** und **Strg+V**
- Mehrstufige Rücknahme der letzten Änderungen über **Strg+Z**
- Mehrstufige Wiederherstellung der zuletzt verworfenen Änderungen über **Strg+Y**
- Kontextmenü mit den Items **Ausschneiden**, **Kopieren** und **Einfügen**

Eine *mehrzeilige* **TextBox**



lässt sich so realisieren:

- Man aktiviert den automatischen Zeilenumbruch über den Wert **Wrap** für das Attribut **TextWrapping**.
- Der Wert **true** für das Attribut **AcceptsReturn** ermöglicht den manuellen Zeilenumbruch per **Enter**-Taste.
- Man sorgt über das **Height**-Attribut für ausreichend Platz.
- Mit dem Wert **Auto** oder **Visible** für die angefügte Eigenschaft **ScrollViewer.VerticalScrollBarVisibility** erhält man einen vertikalen Rollbalken.

Der Glückszahlengenerator erhält durch den folgenden XAML-Code eine mehrzeilige **TextBox** für multiple Vornamen:

```
<TextBox Name="vorname" Width="150" Height="61" VerticalAlignment="Center"
         TextWrapping="Wrap" AcceptsReturn="True"
         ScrollViewer.VerticalScrollBarVisibility="Auto" />
```

Im Beispiel hat die Ernennung des **Button**-Objekts zum Standardschalter (über den Wert **true** der Eigenschaft **IsDefault**, siehe Abschnitt 11.3.1.2) keinen Effekt, wenn das **Enter**-berechtigte mehrzeilige Texteingabefeld den Tastaturfokus besitzt (siehe Abschnitt 11.3.4). Der Standardschalter ist per **Enter** auslösbar, wenn das einzeilige (rechte) Textfeld den Tastaturfokus besitzt.

Trotz **TextWrapping** eignet sich die Klasse **TextBox** nur für kurze Texteingaben. Mit dem Steuerelement **RichTextBox** erhält man einen kompletten Texteditor.

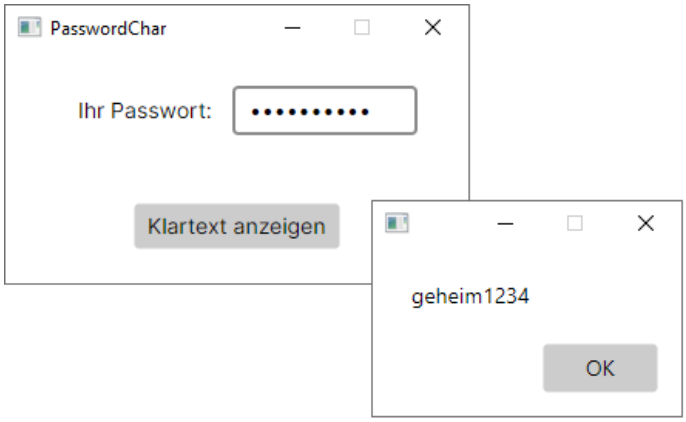
Eine zur Erfassung von **Passwörtern** geeignete **TextBox** erhält man durch die Zuweisung eines Maskierungszeichens an die Eigenschaft **PasswordChar**, z. B.:

```
<TextBox Name="pw" PasswordChar="•" Width="120" Height="30"/>
```

Daraufhin zeigt die **TextBox** ein geändertes Verhalten:

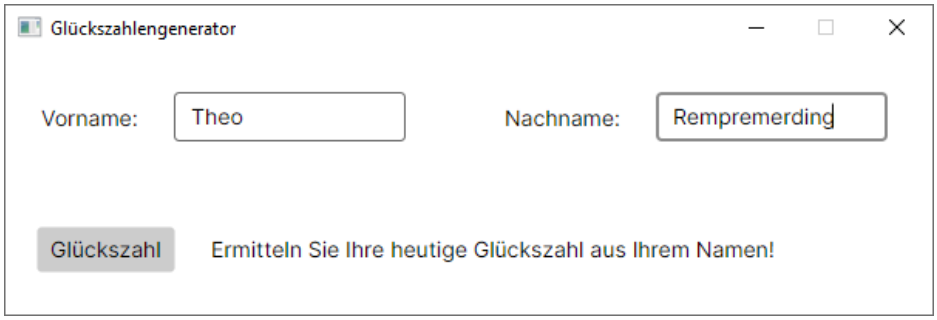
- Statt der eingegebenen Zeichen ist das Maskierungszeichen zu sehen.
- Ein Passwort lässt sich nicht in die Zwischenablage kopieren.

Ein erfasstes Passwort befindet sich als Klartext im Speicher:[1]



11.3.4 Tastaturfokus

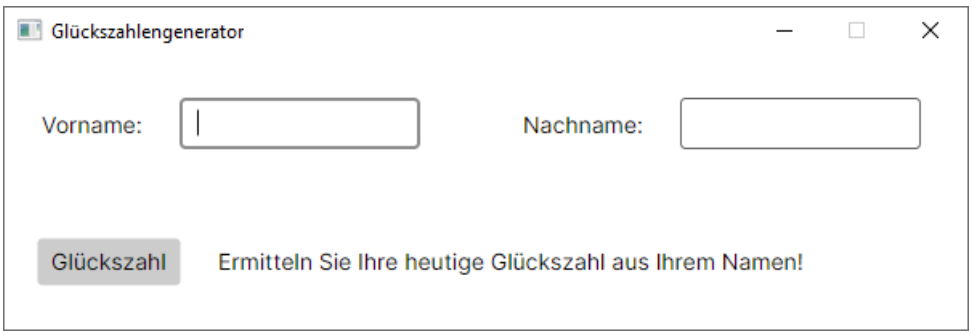
Im **TextBox**-Demonstrationsprogramm (siehe Abschnitt 11.3.3) erhält der Schalter zur Anforderung einer persönlichen Glückszahl über seine Eigenschaft **IsDefault** die Rolle des Standardschalters (siehe Abschnitt 11.3.1.2). Folglich ist er auch *dann* per **Enter**-Taste auslösbar, wenn eines der Texteingabefelder den Tastaturfokus besitzt (erkennbar am verstärkten Rahmen und an der Texteinfügemarke):



In diesem Programm bietet es sich sogar an, die Schaltfläche über den Wert **false** für die Eigenschaft **Focusable** aus der Liste der fokussierbaren Steuerelemente zu entfernen, damit die Tabulatortaste nur noch zwischen den beiden Texteingabefeldern wechselt:

```
<Button Content="Glückszahl" . . . IsDefault="True" Focusable="False" />
```

Im Glückszahlengenerator sollte das Texteingabefeld für den Vornamen schon beim Programmstart den Tastaturfokus haben:



---

[1] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
...\BspUeb\Steuerelemente\Texterfassung\PasswordChar

Eine Lösungsmöglichkeit besteht darin, das zu privilegierende Steuerelement über die Methode **Focus()** der Klasse **UIElement** aufzufordern, den Fokus zu übernehmen, z. B.:

```
vorname.Focus();
```

Damit das **TextBox**-Steuerelement `vorname` schon beim Laden des Fensters den Tastaturfokus beansprucht, steckt man den Methodenaufruf in eine Behandlungsmethode zum **Loaded**-Ereignis der Fensterklasse, z. B. so:

- Man weist die noch nicht definierte Behandlungsmethode `Window_Loaded()` im XAML-Code dem Fenster zu:

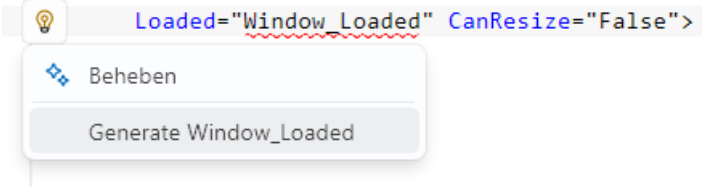
```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="TextBox.MainWindow"
        Title="Glückszahlengenerator" Height="170" Width="600"
        Loaded="Window_Loaded" CanResize="False">

    . . .

</Window>
```

  und provoziert eine Fehlermeldung.

- Über die Glühbirnen-Schnellaktion zum „Fehler“

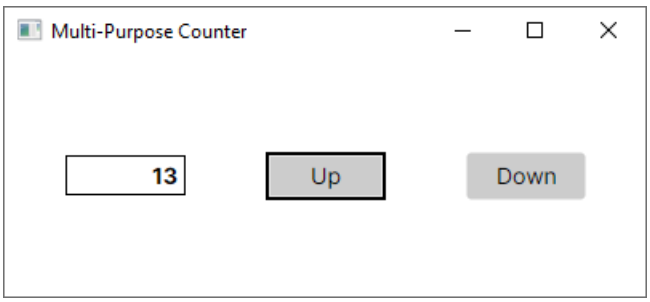


  erhält man einen Behandlungsmethodenrohling in der Code Behind - Datei.

- Dort ist nur noch der Methodenaufruf zu ergänzen:

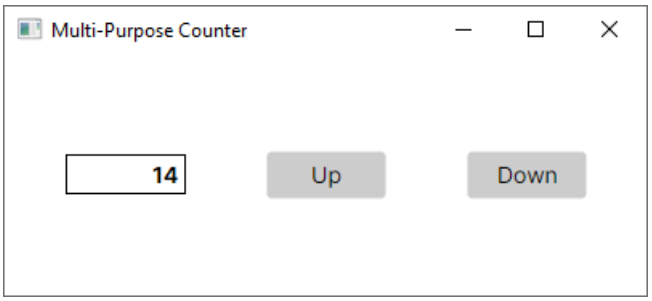
```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    vorname.Focus();
}
```

Wenn ein **Button**-Steuerelement den Tastaturfokus erhält, dann ist sein **Click**-Ereignis per **Enter**- und per Leertaste auslösbar. Ein *per Tabulatortaste* erworbener Tastaturfokus ist bei einem Schalter per Voreinstellung optisch an einem kräftigen schwarzen Rahmen zu erkennen. Das ist in der folgenden Variante des Beispiels aus dem Abschnitts 11.3.1.1 zu sehen:[1]



Wenn ein Schalter *per Mausklick* den Tastaturfokus erhält, dann erscheint aber *kein* schwarzer Rahmen:

[1] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
...\BspUeb\Steuerelemente\Button\UpDown

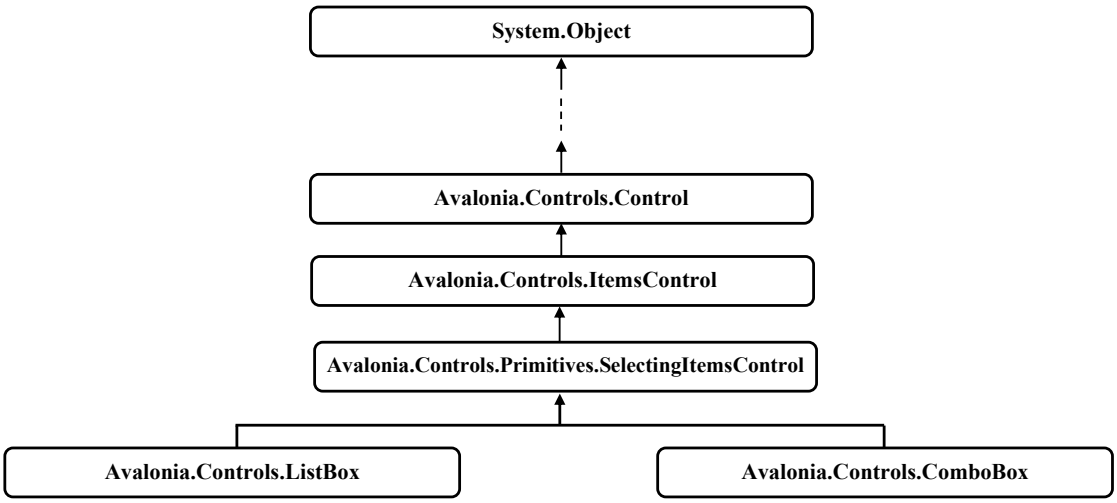


Wir beurteilen dieses (auch bei WPF zu beobachtende) Verhalten kritisch:

- Beim Schalter mit Tastaturfokus lässt sich das **Click**-Ereignis per **Enter**- und per Leertaste auslösen. Nach dem Fokuserwerb *per Tabulatortaste* zeigt der kräftige schwarze Rahmen diese Auslöseoptionen zurecht an.
- Hat ein Schalter den Tastaturfokus *per Mausklick erworben*, fehlt aber der Hinweis auf die **Click**-Auslösung per Tastatur.
- Anschließend hat ein nicht markierter Schalter den Tastaturfokus, was zu Bedienungsfehlern führen kann. Im Up-Down – Beispiel ist für den Benutzer nicht erkennbar, ob ein Druck auf die **Enter**- oder Leertaste den Zählerstand erhöht oder reduziert. Während im Zählprogramm lediglich eine falsche Summe resultiert, ist z. B. bei Schaltern zur Steuerung der Vertikalbewegung eines Luftfahrzeugs ein Irrtum folgenreich.
- Weil viele Benutzer zur Bedienung eines Programm alternierend sowohl die Maus als auch die Tastatur verwenden, tritt die riskante Konstellation oft und unerwartet auf.

11.3.5 Listen- und Kombinationsfelder

Dieser Abschnitt behandelt die Steuerelementklassen **ListBox** und **ComboBox**, die gemeinsame Vorfahren besitzen:

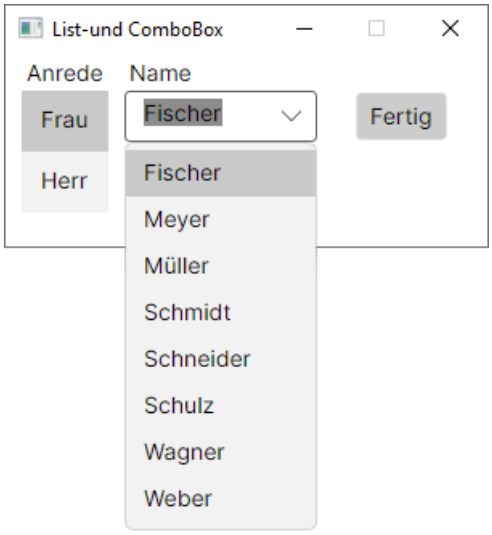


Ein **ListBox**-Steuerelement präsentiert eine Liste von Elementen, von denen der Benutzer (z. B. durch Mausklicks) eines oder mehrere wählen kann.

Das **ComboBox**-Steuerelement erfragt einen Text durch eine Kombination aus einem einzeiligen Texteingabefeld und einer Drop-Down - Liste, und der Benutzer hat *zwei* Möglichkeiten zur Beantwortung, sofern die Eigenschaft **IsEditable** den Wert **true** besitzt (siehe unten):

- Text eingeben
- Drop-Down - Liste öffnen und ein Item wählen

Das folgende Programm erlaubt per **ListBox** die Wahl einer Anrede und erleichtert per **ComboBox** die Eingabe des Namens, indem es eine Liste mit den häufigsten Nachnamen bereithält:[1]



11.3.5.1 ListBox mit Items befüllen oder an eine externe Kollektion binden

Ein **ListBox**-Steuerelement lässt sich per XAML mit Items befüllen, z. B.:

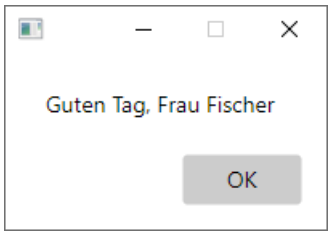
```
<ListBox Name="listBox" Margin="10,0,0,0" SelectedIndex="0">
    <ListBoxItem Content="Frau" />
    <ListBoxItem Content="Herr" />
</ListBox>
```

Diese landen in einem Objekt der Klasse **ItemCollection**, das über die Eigenschaft **Items** ansprechbar ist und Elemente vom Typ **ListBoxItem** verwaltet. Weil **Items** die Inhaltseigenschaft von **ListBox** ist, kann man die Items direkt angeben (siehe Abschnitt 5.2.3.3).

Über die Methoden **Add()**, **Remove()** usw. lässt sich die **ItemCollection** zur Laufzeit per Programm verändern, z. B.:

```
listBox.Items.Add("Hrl.");
```

Ein Klick auf den Schalter `Fertig` im **ComboBox/ListBox** – Demonstrationsprogramm produziert eine **MessageBox** mit Gruß, z. B.:



Wie man das gewählte **ListBox**-Item extrahiert, ist im Abschnitt 11.3.5.3 zu sehen.

Meist befinden sich die in einem **ListBox**-Steuerelement anzuzeigenden Elemente in einer externen Kollektion, und dann ist eine Datenbindung zwischen dieser Kollektion als Quelle und dem **ListBox**-Objekt als Ziel die beste Lösung (siehe Kapitel 9). Dazu weist man der **ListBox**-Eigenschaft **ItemsSource** die Kollektion zu.

---

[1] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
...\BspUeb\Steuerelemente\Kollektions-Präsentation\List-und ComboBox

Besonders geeignet sind Kollektionen vom Typ **ObservableCollection<T>**, weil diese Klasse das Interface **INotifyCollectionChanged** implementiert (siehe Abschnitt 9.6.2). Diese Kollektionen melden Änderungen der Zusammenstellung per Ereignis an ein verbundenes **ListBox**-Steuerelement, das daraufhin seine Anzeige aktualisiert.

Für eine simple Demonstration recyclen und modifizieren wir das im Kapitel 9 vorgestellte Personenverwaltungsprogramm:[1]

- Eine **ListBox** präsentiert Objekte der im Abschnitt 9.6.1 vorgestellten Klasse `Person`.
- Die Klasse `Daten` enthält eine Eigenschaft namens `Personen` vom Typ **ObservableCollection**<`Person`>:

```
public class Daten {
  public ObservableCollection<Person> Personen { get; } = [];

  public Daten() {
    Personen.Add(new Person() {FirstName = "Susanne",LastName = "Brgl",Age = 45});
    Personen.Add(new Person() {FirstName = "Randolf",LastName = "Schmitt",Age=34});
  }
}
```

- Das Anwendungsobjekt erstellt in der Methode **OnFrameworkInitializationCompleted()** ein Objekt der Klasse `Daten` und legt es als **DataContext** für das Anwendungsfenster fest:

```
public override void OnFrameworkInitializationCompleted() {
    if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
        desktop.MainWindow = new MainWindow {
            DataContext = new Daten()
        };
    base.OnFrameworkInitializationCompleted();
}
```

In der XAML-Deklaration des Fensters binden wir die **ItemsSource**-Eigenschaft des **ListBox**—Steuerelements an die beobachtbare Kollektion:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="using:DataTemplateDemo"
        x:Class="DataTemplateDemo.MainWindow"
        x:DataType="local:Daten"
        Title="DataTemplate" Width="300" SizeToContent="Height">

    <StackPanel Margin="10" Spacing="10" >
        <ListBox ItemsSource="{Binding Personen}">
          <ListBox.ItemTemplate>
            <DataTemplate>
              <Grid ColumnDefinitions="*,*,*" ColumnSpacing="10">
                <TextBlock Grid.Column="0" Text="{Binding FirstName}" />
                <TextBlock Grid.Column="1" Text="{Binding LastName}"
                           FontWeight="Bold"/>
                <TextBlock Grid.Column="2" Text="{Binding Age}" />
              </Grid>
            </DataTemplate>
          </ListBox.ItemTemplate>
        </ListBox>
        <Button Content="Person hinzufügen" HorizontalAlignment="Center"
                Click="BtnAddPerson_Click" />
    </StackPanel>
</Window>
```

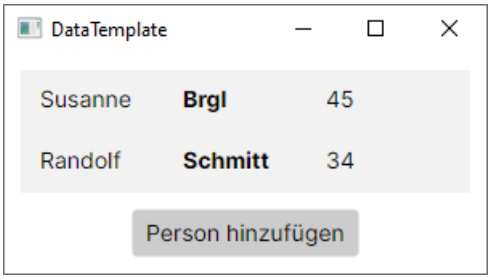
---

[1] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
...\BspUeb\Steuerelemente\Kollektions-Präsentation\DataTemplate

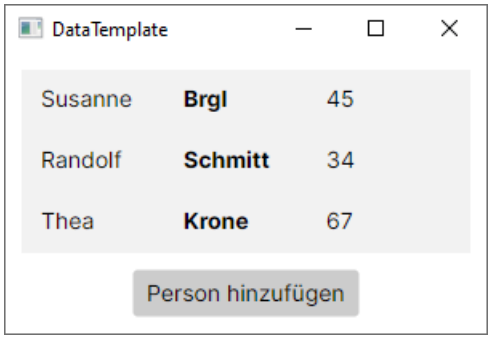
Um die Anzeige der `Person`-Eigenschaften gestalten zu können, weisen wir der **ListBox**-Eigenschaft **ItemTemplate** ein Objekt der Klasse **DataTemplate** (Namensraum **Avalonia.Markup.Xaml.Templates**) zu.[1] Es verwendet ein **Grid**-Layout mit drei Spalten, die jeweils einen **TextBlock** zur Darstellung einer `Person`-Eigenschaft enthalten. Weil die **ItemsSource**-Eigenschaft der **ListBox** an die beobachtbare `Personen`-Kollektion gebunden ist, erhält ein **ListBoxItem** automatisch ein Kollektionselement als **DataContext**. In den **Binding**-Attributen zur Versorgung der **Text**-Eigenschaften der **TextBlock**-Elemente ist als Pfad lediglich die jeweilige `Person`-Eigenschaft anzugeben. Von den Möglichkeiten zur individuell formatierten Darstellung der `Person`-Eigenschaften machen wir nur sparsam Gebrauch.

Zur Demonstration der automatischen Aktualisierung der **ListBox** bei einer Änderung der beobachteten Kollektion enthält das Programm einen Schalter, der einen weiteren Fall in die Kollektion einfügt:

Vor dem Klick auf **Person hinzufügen**



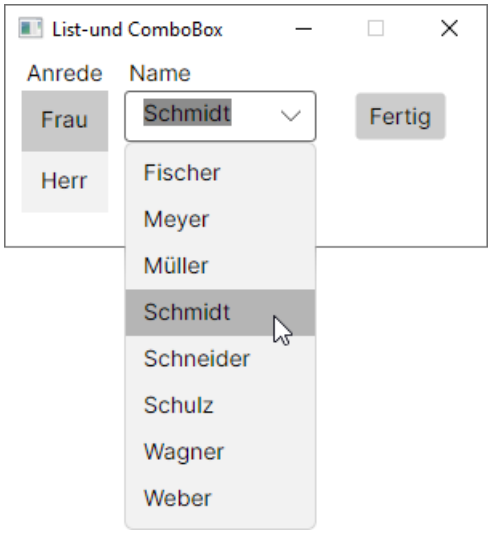
Nach dem Klick auf **Person hinzufügen**



Sobald eine **ListBox** per **ItemsSource**-Eigenschaft mit einer externen Kollektion verbunden ist, lässt sich die interne, per **Items**-Eigenschaft ansprechbare **ItemCollection** nicht mehr verwenden.

11.3.5.2 ComboBox mit Items befüllen oder an eine externe Kollektion binden

Zur Erläuterung des **ComboBox**-Steuerelements kehren wir zum Beispielprogramm des Abschnitts im Abschnitt 11.3.5 mit Wahlmöglichkeiten für Anrede und Namen zurück:



Im XAML-Code zeigen sich wenig Unterschiede zwischen der **ListBox** mit den Anreden und der **ComboBox** mit den häufigsten Nachnamen:

---

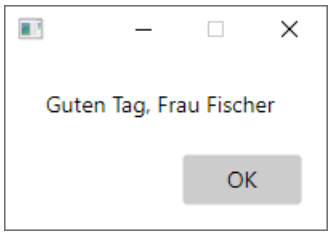
[1] https://docs.avaloniaui.net/docs/data-templates/introduction-to-data-templates

```
<ComboBox Name="comboBox" Margin="10,0,0,0" Width="120"  IsEditable="True">
    <ComboBoxItem Content="Fischer"/>
    <ComboBoxItem Content="Meyer"/>
    <ComboBoxItem Content="Müller"/>
    <ComboBoxItem Content="Schmidt"/>
    <ComboBoxItem Content="Schneider"/>
    <ComboBoxItem Content="Schulz"/>
    <ComboBoxItem Content="Wagner"/>
    <ComboBoxItem Content="Weber"/>
</ComboBox>
```

In seiner internen, über die Eigenschaft **Items** ansprechbaren **ItemCollection** verwaltet eine **ComboBox** Elemente der von **ListBoxItem** abstammenden Klasse **ComboBoxItem**.

Ob ein **ComboBox**-Objekt ein Texteingabefeld anbietet, hängt von der Eigenschaft **IsEditable** ab, die im Beispiel den Wert **true** besitzt.

Ein Klick auf den Schalter `Fertig` im Demonstrationsprogramm produziert eine **MessageBox** mit Gruß, z. B.:



Wie man das gewählte Item bzw. den eingetragenen Text extrahiert, ist im Abschnitt 11.3.5.3 zu sehen.

Wie eine **ListBox** kann man auch eine **ComboBox** an eine externe Kollektion binden, statt die interne **ItemCollection** zu verwenden. Diese Möglichkeit haben wir im Kapitel 9 über die Datenbindung genutzt und dabei die **ItemsSource**-Eigenschaft per XAML versorgt:[1]

```
<ComboBox Name="comboBox" ItemsSource="{Binding Personen}" Grid.Column="1"
          DisplayMemberBinding="{Binding FirstName}"
          PlaceholderText="Person wählen ..." Margin="0,0,0,10"/>
```

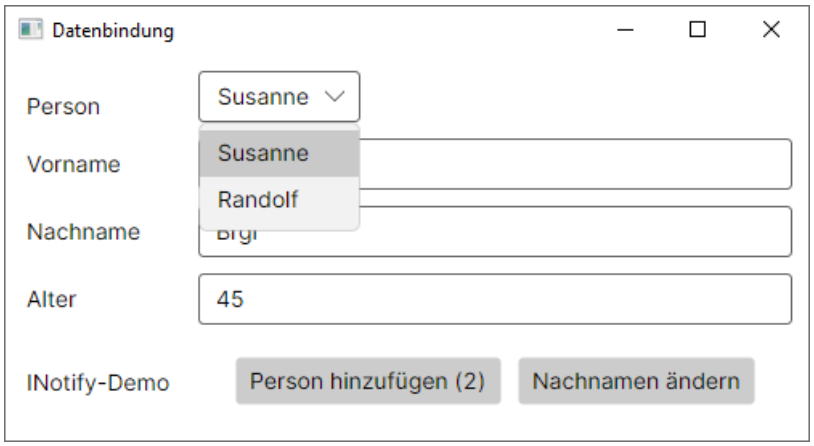
Das **Binding**-Objekt benötigt eine versorgte **DataContext**-Eigenschaft mit einer Kollektion als Referenzziel. Die automatische, in der Container-Verschachtelung nach außen voranschreitende Suche endet beim Fensterobjekt, das in der vom Anwendungsobjekt ausgeführten Initialisierungsmethode **OnFrameworkInitializationCompleted()** ein Objekt der Klasse `Daten` (mit der Eigenschaft `Personen` vom Typ **ObservableCollection**<`Person`>) als **DataContext** erhält:

```
public override void OnFrameworkInitializationCompleted() {
    if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
        desktop.MainWindow = new MainWindow {
            DataContext = new Daten()
        };
    base.OnFrameworkInitializationCompleted();
}
```

Die **ComboBox** zeigt stets die aktuelle Besetzung der beobachtbaren Kollektion:

---

[1] Ein Visual Studio - Projekt mit dem Programm ist hier zu finden:
…\BspUeb\Datenbindung\Personenverwaltung



Das folgende Attribut im XAML-Element **ComboBox**

```
DisplayMemberBinding="{Binding FirstName}"
```

sorgt dafür, dass von den Kollektionselementen vom Typ `Person` die Eigenschaft `FirstName` zu sehen ist.

Sobald eine **ComboBox** per **ItemsSource**-Eigenschaft mit einer externen Kollektion verbunden ist, lässt sich die interne, per **Items**-Eigenschaft ansprechbare **ItemCollection** nicht mehr verwenden.

11.3.5.3 Gewähltes Item ermitteln

Wie sich in der **Click**-Behandlungsmethode zur Schaltfläche des Beispielprogramms mit Wahlmöglichkeiten für Anrede und Namen zeigt, ist bei einem **ListBox**-Objekt die aktuelle Wahl über die Eigenschaft **SelectedItem** ansprechbar:

```
private async void Button_Click(object sender, RoutedEventArgs e) {
    string? anrede = "";
    if (listBox.SelectedItem is ListBoxItem lbItem)
        anrede = lbItem.Content?.ToString();

    string? name = comboBox.Text;
    if (string.IsNullOrWhiteSpace(name))
        return;

    var box = MessageBoxManager.GetMessageBoxStandard("",
                                    $"Guten Tag, {anrede} {name}", ButtonEnum.Ok);
    await box.ShowWindowDialogAsync(this);
}
```

Weil diese Eigenschaft den Datentyp **Object?** besitzt, ist die Textextraktion über die Eigenschaft **Content** erst nach einer Typprüfung und -umwandlung möglich.

Weil Avalonia 12 kein Analogon zur WPF-Klasse **MessageBox** bietet, verwenden wir erneut das NuGet-Paket **MessageBox.Avalonia** (siehe Abschnitt 4.9).

Von einem **ComboBox**-Objekt erfährt man die aktuelle Wahl bzw. Eintragung des Benutzers über die Eigenschaft **Text**. Weil sich das Ergebnis einer Auswahl per Drop-Down – Liste in der Eigenschaft **Text** befindet, ist der Zugriff auf die Eigenschaft **SelectedItem** nur dann erforderlich, wenn man auf einfache Weise einen individuellen (nicht in der Liste befindlichen) Namen erkennen möchte, z. B.:

```
string? name = comboBox.Text;
string? selName = "";
if (comboBox.SelectedItem is ComboBoxItem cbItem)
    selName = cbItem.Content?.ToString();
if (name != selName)
    name += " (individuell)";
```

Bei einem **ListBox**-Objekt kann man über die Eigenschaft **SelectedIndex** dafür sorgen, dass zur Bedienungserleichterung initial ein Listenelement markiert ist, z. B.:

```
<ListBox Name="listBox" Margin="10,0,0,0" SelectedIndex="0">
    <ListBoxItem Content="Frau" />
    <ListBoxItem Content="Herr" />
</ListBox>
```

Bei einem **ComboBox** wählt man mit derselben Eigenschaft ein Listenelement, das initial im Texteingabefeld erscheinen soll.

Die Klassen **ListBox** und **ComboBox** bieten zahlreiche Ereignisse an (z. B. **SelectionChanged**), die im Beispielprogramm nicht interessieren.

11.4 Übungsaufgaben zum Kapitel 11

1) Für das im Abschnitt 11.2.3.1 behandelte Routingereignis **Click** kann man nicht nur der Quelle (dem **Button**-Element) die Behandlungsmethode `Button_Click()` zuweisen, sondern auch dem **Panel**-Container:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="Einstieg.MainWindow"
        Title="Steuerelemente - Einstieg" Height="160" Width="320"
        Background="Beige">
    <Panel Button.Click="Button_Click">
        <Button Content="Farbe wechseln" HorizontalAlignment="Center"
                VerticalAlignment="Center" Click="Button_Click" />
    </Panel>
</Window>
```

Nun klappt die Ereignisbehandlung aber scheinbar nicht mehr: Ein Klick auf den Schalter bleibt ohne sichtbare Folgen.

Wenn man dieselbe Behandlungsmethode zusätzlich auch noch dem **Window**-Element zuweist,

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="Einstieg.MainWindow"
        Title="Steuerelemente - Einstieg" Height="160" Width="320"
        Background="Beige" Button.Click="Button_Click" >
    <Panel Button.Click="Button_Click">
        <Button Content="Farbe wechseln" HorizontalAlignment="Center"
                VerticalAlignment="Center" Click="Button_Click" />
    </Panel>
</Window>
```

dann funktioniert der Farbwechsel wieder. Wie ist das zu erklären?

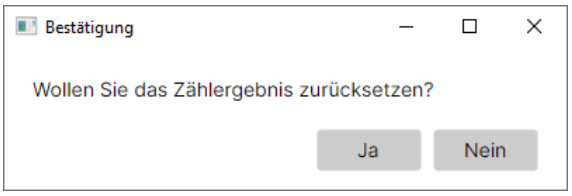
2) Zur Realisation der in Avalonia 12 fehlenden Message-Box ist man nicht auf ein NuGet-Paket angewiesen, sondern kann auch eine geeignete Klasse mit dieser Funktionalität definieren. Im Ordner

...\BspUeb\Steuerelemente\Eigene MessageBox\Klasse ConfirmationWnd

finden Sie zwei Dateien zur Realisation der von **Window** abgeleiteten Klasse `ConfirmationWnd`:[1]

- **ConfirmationWnd.axaml**
- **ConfirmationWnd.axaml.cs**

Ein `ConfirmationWnd`-Fenster besitzt einen **TextBlock** mit einem per Konstruktor festgelegten Text und zwei Schalter, z. B.:



Die **Click**-Behandlungsmethoden der Schalter schließen das Fenster durch einen **Close()** – Aufruf mit Übergabe eines vom gedrückten Schalter abhängigen **bool**-Werts:

```
private void YesBnt_Click(object? sender, RoutedEventArgs e) {
    Close(true);
}

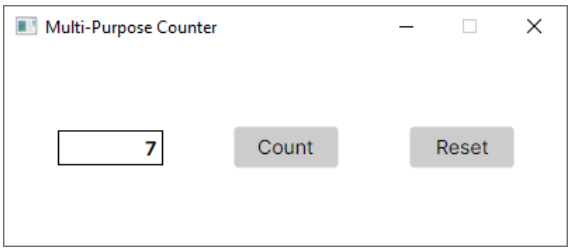
private void NoBnt_Click(object? sender, RoutedEventArgs e) {
    Close(false);
}
```

Als **Window**-Ableitung beherrscht die Klasse `ConfirmationWnd` die Methode **ShowDialog<TResult>()**,

public Task<TResult> ShowDialog<TResult>(Window *owner*)

die ein **Task<TResult>** - Objekt mit der **Close()** – Rückgabe liefert. Man sollte die Methode **ShowDialog<TResult>()** per **await**-Operator aufrufen, was in einer Methode mit dem Modifikator **async** passieren muss (siehe unten). Über die **async**/**await** – Technik der asynchronen C# - Programmierung informiert z. B. das Kapitel 17 in Baltes-Götz & Frischknecht 2026.

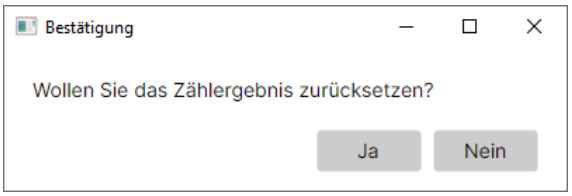
Mit Hilfe der Klasse `ConfirmationWnd` lässt sich z. B. im Zählprogramm des Abschnitts 11.3.1



nach der (eventuell unabsichtlichen) Betätigung des **Reset**-Schalters eine Sicherheitsfrage vornehmen:

---

[1] Die Klasse `MsgBox` verwendet Ideen der folgenden Webseite:
https://docs.avaloniaui.net/troubleshooting/controls/messagebox



Um das Zählprogramm auf dem Stand im Ordner

...\BspUeb\Steuerelemente\Button\Text

entsprechend zu erweitert, müssen Sie ...

- die oben angegebenen Dateien (**ConfirmationWnd.axaml**, **ConfirmationWnd.axaml.cs**) in den Projektordner kopieren
- und in die Ereignisbehandlungsmethode zu den **Button**-Objekten des Zählprogramms die Nachfrage einbauen.

Als Parameter erwartet die vom `ConfirmationWnd`-Objekt ausgeführte Methode **ShowDialog<bool>()** eine Referenz auf das besitzende **Window**-Objekt, also auf das Fensterobjekt des Programms. Diese Referenz liefert die **TopLevel**-Methode **GetTopLevel()** zu einem Parameter vom Typ **Visual**:

```
private async void Button_Click(object? sender, Avalonia.Interactivity.RoutedEventArgs e) {
  if (e.Source == count)
    anzahl++;
  else {
    if (GetTopLevel(this) is Window window) {
      var msgBox = new ConfirmationBox.ConfirmationWnd(
                                          "Wollen Sie das Zählergebnis zurücksetzen?");
      var result = await msgBox.ShowDialog<bool>(window);
      if (!result)
        return;
    }
    anzahl = 0;
  }
  label.Content = anzahl.ToString();
}
```

Das Fehlen einer **MessageBox** begründen die Avalonia-Architekten mit der Plattformunabhängigkeit.[1] Unter Linux und Windows funktionieren allerdings sowohl die in dieser Übungsaufgabe vorgeschlagene Lösung als auch die Klassen aus dem NuGet-Paket **MessageBox.Avalonia** (siehe Abschnitt 4.9).

[1] https://docs.avaloniaui.net/troubleshooting/controls/messagebox

12 MVVM

In großen Projekten ist die Separierung von ...

- **GUI-Layout und GUI-Logik** (z. B. Stile und Animationen),
- **Präsentationslogik** (z. B. Verwaltung der präsentierten Daten und des GUI-Zustands)
- und **Anwendungslogik** (z. B. Algorithmen, Geschäftsregeln, Datenpersistenz)

von hoher Relevanz (engl.: *separation of concerns*):[1]

- Eine komplexe Anwendung gewinnt an **Modularität**, was den Aufwand zur Wartung und Weiterentwicklung reduziert.
- Die lose Kopplung der Einzelbestandteile erlaubt eine **einfache Testung**. Weil z. B. die Präsentationslogik eines Fensters in einer sogenannten *Ansichtsmodellklasse* ohne Kenntnis von Steuerelementen codiert ist (siehe Abschnitt 12.1.2), kann man sie unabhängig von der Bedienoberfläche testen.
- Aufgrund der losen Kopplung von Komponenten, die möglichst wenig voneinander wissen (z. B. keine gegenseitigen Referenzen besitzen), steigt die Aussicht auf eine **Wiederverwendung** der Bestandteile.
- Die **Kooperation** von Software-Entwicklern und Layout-Gestaltern profitiert.

Die Entwicklung von Microsofts GUI-Lösung ***WPF*** (***W**indows **P**resentation **F**oundation*) hat sich eng an dem von John Gossman (2005) vorgeschlagenen Anwendungsdesign nach dem ***MVVM***-Muster (***M**odel-**V**iew-**V**iew**M**odel*) orientiert, wobei die GUI- und Datenbindungsdeklaration per ***XAML*** (*e**X**tensible **A**pplication **M**arkup **L**anguage*) von zentraler Bedeutung ist (siehe z. B. Gehlot & Karamchandani 2014). Avalonia UI ist wie WPF stark auf das MVVM-Anwendungsdesign ausgerichtet.[2] Im bisherigen Verlauf des Manuskripts haben wir zur Erleichterung des Einstiegs das MVVM-Muster ausgeblendet. Das ist nicht länger sinnvoll, weil z. B. Ansichtsmodellklassen bei der im Menüsystem von Avalonia verwendete *Kommandobindung* eine wesentliche Rolle spielen (siehe Abschnitte 12.3 und 13.1).

In der ***BCL*** (***B**ase **C**lass **L**ibrary*) von .NET sind essenzielle Schnittstellen für die MVVM-Architektur (z. B. **INotifyPropertyChanged**, **INotifyDataErrorInfo**, **ICommand**) und einige unterstützende Klassen (z. B. **ObservableCollection<T>**) enthalten. Es fehlen allerdings *Implementationen* der MVVM-Schnittstellen, sodass zur Umsetzung der MVVM-Architektur das Verfassen von Boilerplate-Code in erheblichem Umfang erforderlich ist. Um den Codierungsaufwand in Grenzen zu halten, empfiehlt sich die Verwendung einer Bibliothek mit Implementationen der MVVM-Schnittstellen. Im Abschnitt 12.4 stellen wir das als Nuget-Paket verfügbare *MVVM Community Toolkit* von Microsoft vor.[3] Es enthält u. a. ...

- die Klasse **ObservableObject**, die die Schnittstelle **INotifyPropertyChanged** implementiert. Von **ObservableObject** abgeleitete Modell- oder Ansichtsmodellklassen können die Ansicht über geänderte Eigenschaften informieren.
- die von **ObservableObject** abstammende Klasse **ObservableValidator**, die die Schnittstelle **INotifyDataErrorInfo** implementiert. Von **ObservableValidator** abgeleitete Modell- oder Ansichtsmodellklassen können eine Validierung von Benutzereingaben vornehmen.

---

1 https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm

2 https://docs.avaloniaui.net/docs/fundamentals/the-mvvm-pattern

3 https://learn.microsoft.com/en-us/dotnet/communitytoolkit/mvvm/

- die Klassen **RelayCommand** und **RelayCommand<T>**, die die Schnittstelle **ICommand** implementieren. Objekte mit diesen Typen taugen als handlungskompetente Referenzziele von **ICommand**-Eigenschaften in Ansichtsmodellklassen, an die man Steuerelemente (z.B. Menü-Items und Befehlsschalter) zum Auslösen von Kommandos binden kann.

Auch bei Verwendung einer Bibliothek wächst mit der MVVM-Architektur die Komplexität einer Anwendung. So ist z. B. die Umsetzung von Benutzerwünschen durch Kommandos aufwändiger als das Erstellen traditioneller Ereignisbehandlungsmethoden (siehe Abschnitt 12.3). Bei einer kleinen Anwendung lohnt sich die strikte Beachtung der MVVM-Architektur nicht.

12.1 Verteilung von Zuständigkeiten auf lose gekoppelte Komponenten

In einer Anwendung mit MVVM-Architektur kooperieren drei lose gekoppelte Komponenten:

- Die Bedienoberfläche (kurz: **Ansicht**, engl. **View**) ist per XAML deklariert und besteht im Allgemeinen aus mehreren Fenstern.
- Zu jedem Fenster gehört ein **Ansichtsmodell** (engl. **View Model**, oft zusammengeschrieben als **ViewModel**). Dabei handelt es sich um eine Klasse, die im Fenster zu präsentierende Daten bereitstellt und den Zustand des Fensters verwaltet, z. B. die sichtbaren Bedienelemente festlegt.
- Die Klassen im **Modell** sind für die Anwendungslogik und die Datenpersistenz zuständig.

Mit den Aufgaben und Kommunikationsverfahren der Komponenten beschäftigen wir uns anschließend in Detail:



12.1.1 Modell

Die Modellklassen ...

- sorgen für die Verfügbarkeit/Persistenz der Daten, indem sie auf Datenbanken, Web-Dienste oder lokale Dateien zugreifen.
- sind für die Anwendungslogik zuständig. In einer Speditionsverwaltung berechnet eine Modellklasse z. B. die Route zu den Lade/Entladepunkten und stellt die Einhaltung von Regeln sicher (z. B. Verbot negativer Lagerbestände).
- sind *nicht* über die Ansichtsmodelle zu den Fenstern informiert (besitzen keine Referenzen auf deren Klassen).
- sind *nicht* über die View-Komponente (Bedienoberfläche) informiert (besitzen keine Referenzen auf deren Klassen).

Typische Beispiel für Modellklassen sind:

- ***POCO***-Klassen (***P**lain **O**ld **C**LR **O**bjects*)
  Diese Klassen sind nicht an eine bestimmte Umgebung (z. B. Avalonia, WPF) gebunden und besitzen Handlungskompetenzen zur Lösung der Aufgaben aus dem Anwendungsbereich.
- ***DTO***-Klassen (***D**ata **T**ransfer **O**bjects*)
  Sie dienen zum Datenaustausch und besitzen kaum Handlungskompetenzen.

- Entitätsklassen
  Diese Klassen können z. B. zu einer vorhandenen Datenbank von einer ***ORM***-Software (***O**bject **R**elational **M**apping*) wie *Entity Framework Core* erstellt werden.

Im Unterschied zu den kurzen Demonstrationsprogrammen im Manuskript enthalten reale Anwendungen oft eine zwei- oder dreistelligen Zahl von Modellklassen, sodass eine Untergliederung mehr Übersicht schafft:

- Domänenmodell
  Diese Klassen modellieren die bei einer objektorientierten Analyse festgestellten Akteure im Anwendungsbereich mit ihren Merkmalen, Handlungskompetenzen und Beziehungen.
- Datenzugriffsschicht (engl.: *data access layer*)
  Dazu gehören Modellklassen im Direktkontakt mit Datenquellen bzw. -senken.

Die Modellkomponente ist in der MVVM-Architektur genauso konzipiert wie in der älteren ***MVC***-Architektur (***M**odel **V**iew **C**ontroller*).

12.1.2 Ansichtsmodell (ViewModel)

Je komplexer das Modell ausfällt, desto eher empfiehlt sich die Entwicklung einer separierenden und vermittelnden Schicht von Ansichtsmodellklassen zwischen dem Modell und der Bedienoberfläche (McCarter 2010). Eine Ansichtsmodellklasse ...

- ist einer Ansicht zugeordnet, um sie mit Daten zu versorgen und durch Präsentationslogik zu unterstützen.
- ist zuständig für das Verhalten und den Zustand der Ansicht (angezeigte Daten, Nutzbarkeit von Steuerelementen), aber nicht für das Layout. Gossman (2005) bezeichnet eine Ansichtsmodellklasse als *Abstraktion* der zugeordneten Ansicht. Weil die Ansichtsmodellklasse von der Ansicht entkoppelt ist, kann man ...
  - die Ansicht austauschen,
  - die Ansichtsmodellklasse einfach testen (ohne Beteiligung von Bedienelementen und Bedienern). Ein Modultest (engl.: *unit test*) ist lediglich ein zur Ansicht alternativer Nutzer der Ansichtsmodellklasse.
- enthält ...
  - Eigenschaften mit Daten zur Nutzung durch daran gebundene Steuerelemente (siehe Kapitel 9),
  - Eigenschaften mit dem Zustand der Ansicht,
  - Eigenschaften vom Typ **ICommand** zum Starten von Kommandos durch daran gebundene Steuerelemente (siehe Abschnitt 12.3).
- kennt die Ansicht *nicht* (besitzt z. B. keine Referenzen auf GUI-Objekte). Zur Kommunikation mit der Ansicht dient die Datenbindungs- und Benachrichtigungstechnik von Avalonia (vgl. Kapitel 9). Ein Ansichtsmodellobjekt dient der Ansicht als **DataContext**. Eine Ansichtsmodellklasse benötigt dank Datenbindung keinen Code zur Aktualisierung der Ansicht.
- kennt die Modellklassen, um über Methodenaufrufe Daten abzurufen oder zu aktualisieren. Zum Bezug von Mitteilungen über die Aktualisierung von Modelldaten dienen Ereignisse der Modellklassen (z. B. **PropertyChanged**, **CollectionChanged**) (siehe Abschnitt 9.6). In einem typischen Fall meldet eine Ansichtsmodellklasse eine Behandlungsmethode beim **PropertyChanged**-Ereignis einer Modellklasse an und löst in dieser Methode das eigene **PropertyChanged**-Ereignis aus. Daraufhin aktualisiert die Avalonia-Datenbindungstechnik automatisch die verbundenen Steuerelemente.

- vermittelt zwischen Ansicht und Modell, die keinen direkten Kontakt miteinander haben, indem es ...
  - Daten zwischen Modell und Ansicht überträgt,
  - Kommandos der Ansicht in Methodenaufrufe an das Modell übersetzt.
- transformiert Daten nach den Bedürfnissen der Ansicht (z. B. Wandlung eines Datumsobjekts mit der internationalen Datums- und Zeitangabe in eine Zeichenfolge mit dem lokalen Format).
- validiert Benutzereingaben (siehe Abschnitt 9.7). Das geschieht manchmal (z. B. in dem seit Kapitel 9 entwickelten Personenverwaltungsprogramm) durch Delegation an eine die Schnittstelle **INotifyDataErrorInfo** implementierende Modellklasse (siehe Abschnitt 12.1.4). Wenn eine Ansichtsmodellklasse als **DataContext** z. B. eine Kollektion mit Objekten einer Modellklasse veröffentlicht, dann werden durch verbundene Steuerelemente erfasste Daten bei entsprechendem **Binding**-Pfad durch die Modellklasse validiert.

  Avalonia nimmt die Validierung von Benutzereingaben *nach* dem Speichern vor, weist also irreguläre Daten nicht ab. Das spricht für die strikte Umsetzung der MVVM-Konzeption:
  - Die Ansicht kann nur Eigenschaften des Ansichtsmodells ändern.
  - Das Ansichtsmodell erledigt die Validierung und leitet nur korrekte Daten an das Modell weiter.

  Wir werden das Personenverwaltungsprogramm im weiteren Verlauf das aktuellen Kapitels auf eine MVVM-konforme Arbeitsweise umstellen (siehe Abschnitt 12.4.4).
- ist vergleichbar mit ...
  - dem von Martin Fowler 2004 beschriebenen *Presentation Model*[1]
  - einer Controller-Klasse in einer MVC-Anwendungsarchitektur (*Model View Controller*).

Gossman (2005) bezeichnet eine Ansichtsmodellklasse auch als *Model of a View*. Als Anwendungsbeispiel beschreibt er ein GUI-Programm zur Verwaltung eines MSBuild-Projekts, das auch bei einem Avalonia-Programm beteiligt ist und eine Projektdatei (mit der Namenserweiterung **csproj**) zum Speichern von Einstellungen verwendet (vgl. Abschnitt 2.4.8):

- Das MSBuild-Projekt mit seinen Daten und den MSBuild-Funktionen bildet das Modell. Es lässt sich nicht nur über das GUI-Programm verwalten, sondern auch per Konsole (Eingabeaufforderung).
- Die Bedienoberfläche des grafischen Verwaltungsprogramms präsentiert eine Baumansicht mit Kontextmenüs sowie ein Textfenster mit Rollbalken.
- Das Ansichtsmodell erstellt aus MSBuild-Konzepten für die Ansicht konsumierbare Datenbindungsquellen, speichert die aktuelle Auswahl der Baumansicht und enthält Kommandos zur Verwaltung des MSBuild-Projekts.

Erstellt man im Visual Studio ein neues Projekt mit der Vorlage **Avalonia .NET MVVP App**, dann erhält die Ansichtsmodellklasse zum Hauptfenster `MainWindow` den Namen `MainViewModel`.[2]

Das Öffnen eines zusätzlichen Fenster auf Initiative einer Ansichtsmodellklasse verstößt gegen die strikte MVVM-Trennungsregel: Weil eine Ansichtsmodellklasse keine Ansicht (Fensterklasse) kennen darf (weder die eigene, noch eine andere), kommt das Erstellen eines Objekts einer

---

[1] https://martinfowler.com/eaaDev/PresentationModel.html

[2] In älteren Avalonia-Dokumentationen ist die Bezeichnung `MainWindowViewModel` zu finden, orientiert an der Konvention, den Namen einer Ansichtsmodellklasse mit dem Namen der unterstützten Ansichtsklasse beginnen zu lassen und *`ViewModel`* anzuhängen, sodass für die Ansichtsmodellklasse zur Ansichtsklasse `MainWindow` der Name `MainWindowViewModel` resultiert.

Fensterklasse per Konstruktor nicht in Frage. Das MVVM Community Toolkit enthält für diese und andere Kommunikationsbedürfnisse eine Nachrichtentechnik (siehe Abschnitt 12.4.5).

Zwischen Ansicht und Modell das Ansichtsmodell als vermittelnde bzw. separierende Schicht einzuschieben, hat z. B. die eben beschriebenen Vorteile bei der Validierung von Benutzereingaben. Weil Eigenschaften von Modellobjekten zusätzlich als Eigenschaften von Ansichtsmodellobjekten vorhanden sind, entsteht aber auch Aufwand. Bei einem kleinen Projekt mit einem simplen Modell lohnt sich die MVVM-Struktur nicht (siehe Abschnitt 12.1.4).

12.1.3 Ansicht (View)

Eine Ansicht (z. B. das Anwendungsfenster) ...

- ist in der Regel per XAML deklariert und sollte in der begleitenden Code Behind – Datei nur wenig C# - Code enthalten.
- wird durch ein zugeordnetes Ansichtsmodellobjekt mit Daten und Präsentationslogik versorgt. Die Ansicht kennt das unterstützende Ansichtsmodellobjekt, das in der Regel als **DataContext** auf **Window**-Ebene fungiert.
- enthält Steuerelemente (z. B. vom Typ **TextBox**), die per Datenbindungs- und Benachrichtigungstechnik an Eigenschaften des Ansichtsmodells gebunden sind. Das erlaubt die automatisierte Anzeige aktueller Daten und (bei bidirektionaler Bindung) auch die Änderung von Daten.
- nimmt Anwenderwünsche entgegen, indem Bedienelemente eine Nutzung der vom Ansichtsmodell beherrschten Kommandos ermöglichen.

In einer MVVM-Anwendung enthält die Code-Behind – Datei einer Fensterklasse (z. B. **MainWindow.axaml.cs**) in der Regel nur noch einen Konstruktor, der sich auf einen Aufruf der Methode **InitializeComponent()** beschränkt (vgl. Abschnitt 6.2), z. B.:

```
using Avalonia.Controls;

namespace Einstieg.Views;

public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
    }
}
```

Weil die Begriffe *Präsentationslogik* und *GUI-Logik* leicht zu verwechseln sind, folgt ein Klärungsversuch:

- Die GUI-Logik ...
  - gehört zur Ansicht (realisiert durch XAML und Code Behind),
  - beschäftigt sich damit, *wie* eine Darstellung erfolgen soll (z. B. durch responsives oder animiertes Layout),
  - ist stark von einem GUI-Framework (z. B. Avalonia, WPF, MAUI) abgängig,
  - erlaubt keine Modultests (engl.: *unit tests*).
- Die Präsentationslogik ...
  - gehört zum Ansichts*modell* (realisiert durch C#),
  - beschäftigt sich damit, *was* dargestellt werden soll, und welche Funktionalitäten dem Benutzer verfügbar sein sollen, wobei aber keine Steuerelemente direkt beeinflusst werden,
  - ist möglichst wenig von einem GUI-Framework abhängig,
  - erlaubt Modultests ohne Beteiligung einer grafischen Bedienoberfläche.

12.1.4 Variable Aufgabenverteilung

In der Praxis erfolgt die Verteilung von Aufgaben auf die MVVM-Klassen nicht dogmatisch. So können z. B. die Schnittstellen **INotifyPropertyChanged** und **INotifyDataErrorInfo** in Ansichtsmodellklassen und/oder in Modellklassen implementiert sein. Mit dieser Liberalität lässt sich eine nicht immer erwünschte Konsequenz des zwischen Modell und Ansicht eingeschobenen Ansichtsmodells abmildern: Wenn ein Ansichtsmodell viele Modellklassenobjekte an die Ansicht weitergibt, dann resultieren im Ansichtsmodell entsprechend viele Stellvertreterobjekte (McCarter 2010).

Eine Möglichkeit zur Vermeidung von Stellvertreterobjekten besteht darin, im Ansichtsmodell eine Kollektion mit Modellobjekten zu definieren und der Ansicht anzubieten, z. B.:

```
public class MainViewModel {
  public ObservableCollection<Person> Personen { get; } = [];

  public MainViewModel () {
    Personen.Add(new Person() {FirstName = "Susanne",LastName = "Brgl",Age = 45});
    Personen.Add(new Person() {FirstName = "Randolf",LastName = "Schmitt",Age = 34});
  }
}
```

Sind Steuerelemente der Ansicht bidirektional an Eigenschaften von Kollektionselementen gebunden, dann kann die Ansicht entgegen einer strikten MVVM-Auslegung Modellobjekte verändern, z. B.:

```
<StackPanel Margin="10" >
    <Grid ColumnDefinitions="110,*">
        <Label Grid.Column="0" Content="Person" VerticalContentAlignment="Center" />
        <ComboBox Name="comboBox" ItemsSource="{Binding Personen}" Grid.Column="1"
                  DisplayMemberBinding="{Binding FirstName}"
                  PlaceholderText="Person wählen ..." Margin="0,0,0,10"/>
    </Grid>
    <Grid ColumnDefinitions="110,*" RowDefinitions="Auto,Auto,Auto"
          VerticalAlignment="Center" RowSpacing="10" Margin="0,0,0,20"
          DataContext="{Binding #comboBox.SelectedItem}" x:DataType="local:Person">
        <Label Grid.Column="0" Grid.Row="0" Content="Vorname"
                    VerticalContentAlignment="Center" />
        <TextBox Grid.Column="1" Grid.Row="0" Text="{Binding FirstName}" />
    . . .
    </Grid>
 . . .
</StackPanel>
```

Das Ansichtsmodellobjekt vermittelt als **DataContext** der Ansicht einen Zugriff auf Modellobjekte, ist aber bei der Validierung von Schreibzugriffen der Ansicht auf Modellobjekten außen vor.

Indem Modellklassen eine erforderliche Validierung vornehmen, schützen sie die Einhaltung ihrer Regeln (z. B. Verbot negativer Kontostände oder Altersangaben), ohne auf die Sorgfalt eines Ansichtsmodells angewiesen zu sein. Bei der Validierung durch Modellklassen sind allerdings zwei Risiken zu beachten:

- Eine Validierung enthält oft auch die Formulierung von Fehlermeldungen, und solche Texte sind eher der Ansicht oder dem Ansichtsmodell zuzuordnen. Eine Lösungsmöglichkeit sind die vom C# - Compiler bekannten Fehlercodes (z. B. CS0103).
- Durch die Validierung kann eine Modellklasse zu groß werden. In einem solchen Fall bietet es sich an, einer Hilfsklasse die Validierung zu überlassen.

Wenn keine Validierung erforderlich ist, oder Modellklassen die Validierung vornehmen, dann ist in einer MVVM-Anwendung die Änderung von Modellobjekten durch Benutzereingaben ohne Beteiligung der Ansichtsmodellklasse akzeptabel.

Wenn ein Ansichtsmodell der Ansicht Stellvertreterobjekte (z. B. Kopien von Modellobjekten) anbietet, dann sind Zusatzleistungen möglich (z. B. die Rücknahme von Änderungen). Der Aufwand ist am ehesten in einer komplexen Anwendung gerechtfertigt, wobei ein Ansichtsmodell z. B. statt einer Liste mit Modellobjekten

```
public ObservableCollection<Person> Personen { get; } = [];
```

eine Liste mit Ansichtsmodellobjekten für die Ansicht bereitstellt:

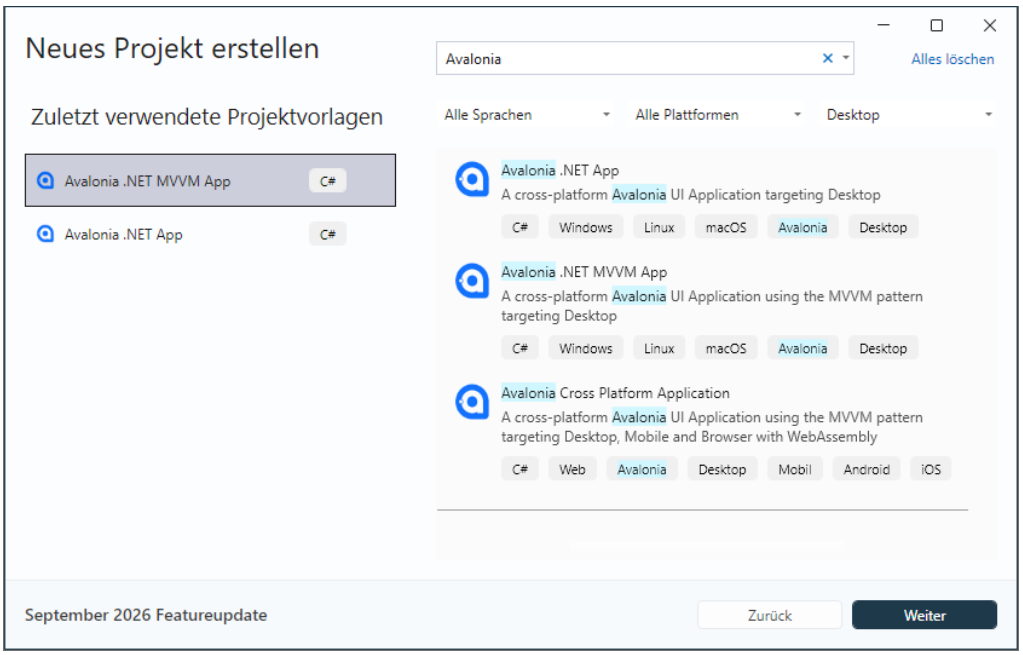
```
public ObservableCollection<ViewModelPerson> Personen { get; } = [];
```

Für dieses Design spricht auch die Avalonia-Validierungstechnik, die in der Regel erst *nach* dem Speichern eine Validierung vornimmt, sofern keine Typunverträglichkeit das Speichern verhindert (z. B. bei der Altersangabe *dreizehn*). Im Abschnitt 12.4.2 ist im Zusammenhang mit dem MVVM Community Toolkit eine *abweisende* Validierung zu sehen (ohne Speicherung ungeprüfter Werte), die aber ohne die Avalonia-Technik zur Fehlermeldung an Benutzer auskommen muss.

Wir präsentieren im Abschnitt 12.4.4 einen Vorschlag, der die Avalonia-Validierungstechnik nutzt und Fehler von den Modellobjekten fernhält. Dabei erfolgt die Validierung im Ansichtsmodell, das nur korrekte Daten an das Modell weiterleitet. Diese Konstruktion belegt den Nutzen des zwischen Ansicht und Modell vermittelnden Ansichtsmodells.

12.2 Beispiel

Zur Demonstration der MVVM-Architektur erstellen wir ein neues Projekt unter Verwendung der Vorlage **Avalonia .NET MVVM App**:[1]

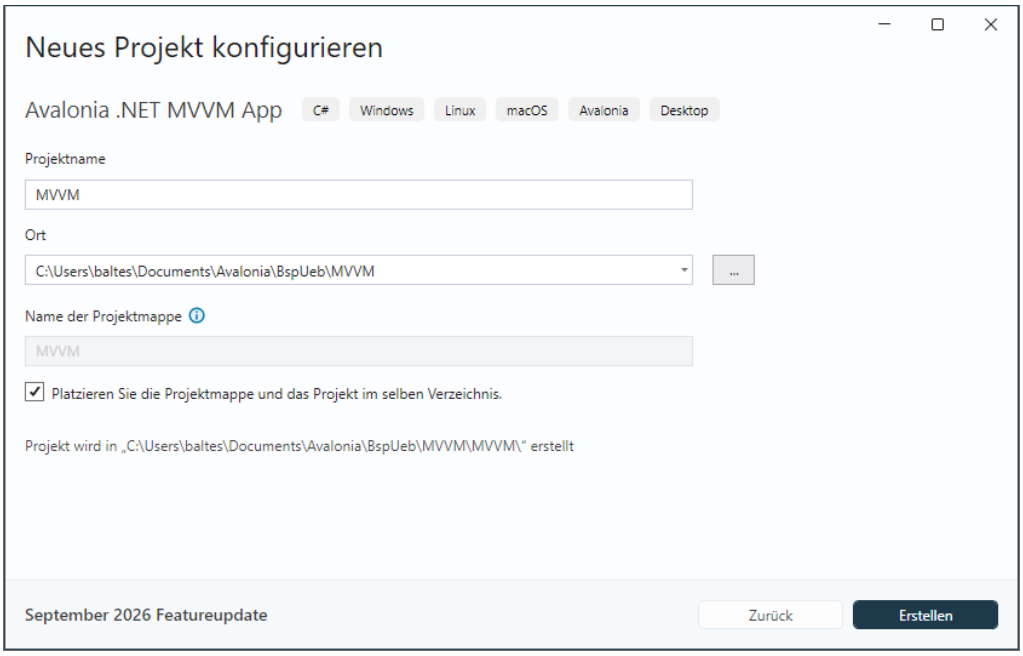


Im ersten Schritt entsteht eine MVVM-Variante des Personenverwaltungsprogramms auf dem Stand von Abschnitt 9.7 (über die Datenvalidierung):

---

[1] Ein Visual Studio - Projekt mit dem Einstieg in eine Serie von MVVM-Varianten der Personenverwaltung ist hier zu finden:

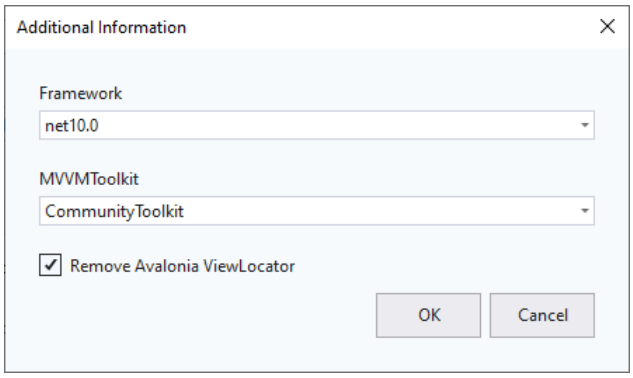
...\BspUeb\MVVM\Einstieg



Es soll eine Serie von Projekten mit sukzessive weiterentwickelten Varianten des Personenverwaltungsprogramms entstehen. Um dabei den Aufwand zu begrenzen, verwenden wir in allen Projekten den Namensraum `MVVM`. Weil das Visual Studio den Projektnamen als Bezeichnung für den Wurzelnamensraum übernimmt, hat das Einstiegsprojekt den Namen `MVVM` erhalten. Einen Projektnamen kann man leicht ändern:

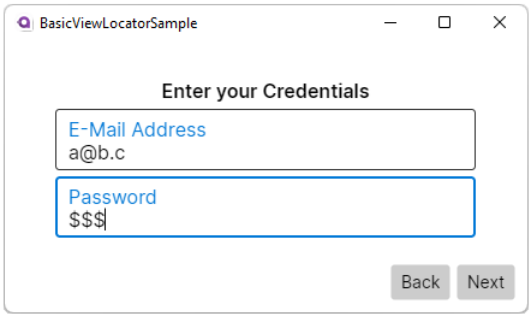
- **csproj**-Datei umbenennen
- **slnx**-Datei umbenennen und darin den Projektnamen ändern.

Obwohl wir das MVVM-Muster zunächst ohne die „Magie“ eines Toolkits kennenlernen wollen, ist eine Entscheidung zu treffen:



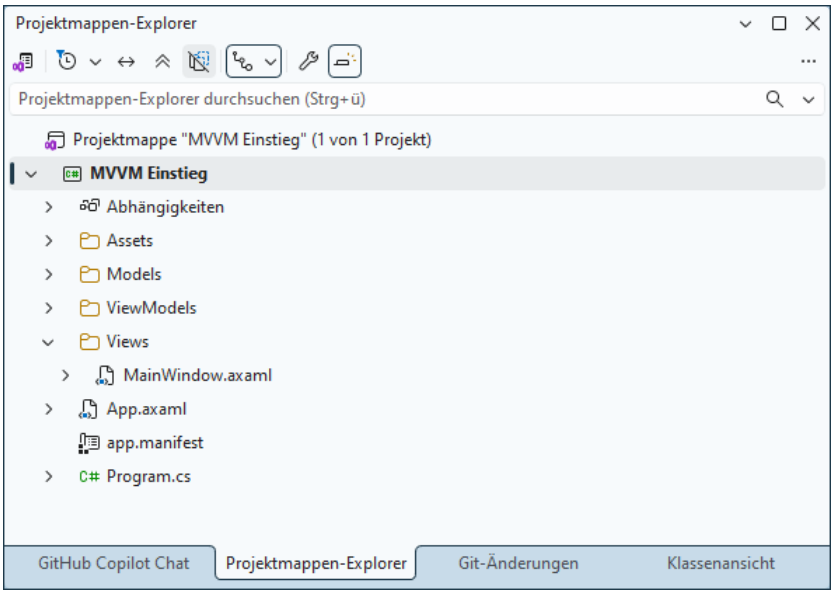
Wir wählen das **CommunityToolkit** und verzichten auf den **ViewLocator**. Der ist nützlich in einer Anwendung mit der Möglichkeit, zwischen mehreren Seiten zu navigieren. Die Benutzer können dort per Navigationskommando die Seite bzw. die **ViewModel**-Klasse wechseln, und der **ViewLocator** lädt das passende **View**-Objekt bzw. die **UserControl** - XAML-Deklaration, z. B.:[1]

[1] https://avaloniaui.github.io/Avalonia.Samples/src/Avalonia.Samples/Routing/BasicViewLocatorSample/



Unsere Desktop-Anwendung hat nur *ein* traditionelles Fenster und *eine* **ViewModel**-Klasse ohne Möglichkeit zum Seitenwechsel.

Zur Unterstützung der MVVM-Struktur hat das neue Projekt die Unterordner **Models**, **ViewModels** und **Views** erhalten:



Wir übernehmen aus dem Ordner

...\BspUeb\Validierung

die Datei **Person.cs** in den Projektunterordner **Models** und ändern die **namespace**-Direktive der Datei in:

```
namespace MVVM.Models;
```

Die Klasse `Daten` aus dem Quellprojekt gelangt mit dem neuen Namen `PersonServices` in der **Models**-Ordner des MVVM-Projekts. Wir betrachten sie als Bestandteil des Modells:

```
using System.Collections.ObjectModel;

namespace MVVM.Models;

public class PersonServices {
  static readonly ObservableCollection<Person> people = [];

  static PersonServices() {
    people.Add(new Person() {FirstName = "Susanne", LastName = "Brgl", Age = 45});
    people.Add(new Person() {FirstName = "Randolf", LastName = "Schmitt", Age = 34});
  }

  public static ObservableCollection<Person> GetPeople() {
      return people;
  }
}
```

Die Ansicht erlangt im MVVM-Muster nur durch die Vermittlung des Ansichtsmodellobjekts eine Referenz auf die im Modell befindliche Sammlung von `Person`-Instanzen. Wir nehmen an der vom Visual Studio aufgrund der Projektvorlage **Avalonia .NET MVVM App** erstellten Klasse `MainViewModel` einige Änderungen vor:

```
using System.Collections.ObjectModel;
using MVVM.Models;

namespace MVVM.ViewModels;

public class MainViewModel {
    public ObservableCollection<Person> Personen { get; }

    public MainViewModel() {
        Personen = PersonServices.GetPeople();
    }
}
```

Weil wir das MVVM Community Toolkit aktuell noch nicht nutzen, entfernen wir ...

- den Modifikator **partial** aus der Klassendefinition,
- die von **ComunityToolkit.Mvvm.ComponentModel.ObservableObject** abstammende Klasse **ViewModelBase** aus dem Projekt.

Die Ansichtsmodellklasse bezieht von der Klasse `PersonServices` über die Methode `GetPeople()` die zu verarbeitenden Daten und stellt sie der Ansicht über die Eigenschaft `Personen` zur Verfügung. Weil sich die Klasse `MainViewModel` im Namensraum `MVVM.ViewModels` befindet, die Klasse `PersonServices` hingegen im Namensraum `MVVM.Models`, erhält die Quellcodedatei eine **using**-Direktive. Insgesamt ist in der aktuellen Klasse `MainViewModel` noch nicht viel von der im Abschnitt 12.1.2 skizzierten Funktionalität einer Ansichtsmodellklasse vorhanden.

Wir übernehmen aus dem Ordner

...\BspUeb\Validierung

den XAML-Code der Datei **MainWindow.axaml** mit einigen Änderungen in die gleichnamige Datei im **Views**-Ordner des MVVM-Projekts:

- Die in der der Projektvorlage **Avalonia .NET MVVM App** enthaltene XML-Namensraumdeklaration
  ```
  xmlns:vm="using:MVVM.ViewModels"
  ```
  und das nutzende Element
  ```
  <Design.DataContext>
      <vm:MainViewModel />
  </Design.DataContext>
  ```
  behalten wir bei in der Erwartung, in der XAML-Vorschau realistische Daten zu sehen.
- Im **x:Class** – Attribut des **Window**-Elements ist der Namensraum der Hauptfensterklasse zu ändern:
  ```
  x:Class="MVVM.Views.MainWindow"
  ```
- Im **x:DataType** – Attribut des **Window**-Elements ist die Ansichtsmodellklasse anzugeben (vgl. Abschnitt 9.2 zur übersetzten Datenbindung in Avalonia):
  ```
  x:DataType="vm:MainViewModel"
  ```
- Das in der der Projektvorlage **Avalonia .NET MVVM App** enthaltene Anwendungssymbol behalten wir bei:
  ```
  Icon="/Assets/avalonia-logo.ico"
  ```
- Die Titelzeilenbeschriftung ändern wir in:
  ```
  Title="MVVM (Toolkit-Einstieg)"
  ```

- Im **Grid**-Container mit den Details zum ausgewählten Fall der **ComboBox**
  ```
  DataContext="{Binding #comboBox.SelectedItem}"
  x:DataType="local:Models.Person"
  ```
  ist wegen der übersetzten Datenbindung die Modellklasse `Person` anzugeben. Das ist kein Verstoß gegen das MVVM-Muster, denn im Pfad einer Datenbindung muss man auch die Eigenschaften der Modellklassen nennen.
- Wir entfernen aus den **Button**-Deklarationen die **Click**-Behandlungsmethoden. Ereignisbehandlungsmethoden sind in der Code Behind – Datei zu einer MVVM-Ansicht unerwünscht. Wir werden sie durch gebundene Kommandos der Ansichtsmodellklasse ersetzen (siehe Abschnitt 12.3). Die aus der Projektvorlage **Avalonia .NET MVVM App** entstandene Code Behind – Datei **MainWindow.axaml.cs** entspricht dem MVVM-Muster:
  ```
  using Avalonia.Controls;
  namespace MVVM.Views;
  public partial class MainWindow : Window {
      public MainWindow() {
          InitializeComponent();
      }
  }
  ```

Die Methode **OnFrameworkInitializationCompleted()** der Anwendungsklasse `App` (siehe Abschnitt 4.7.1) erstellt neben dem Fensterobjekt auch das Ansichtsklassenobjekt und weist der **DataContext**-Eigenschaft des Fensterobjekts das Ansichtsklassenobjekt zu:

```
public partial class App : Application {
    public override void Initialize() {
        AvaloniaXamlLoader.Load(this);
    }

    public override void OnFrameworkInitializationCompleted() {
        if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop) {
            desktop.MainWindow = new MainWindow {
                DataContext = new MainViewModel(),
            };
        }
        base.OnFrameworkInitializationCompleted();
    }
}
```

Man muss die **DataContext**-Eigenschaft des Fensterobjekts nicht in der **App**-Methode **OnFrameworkInitializationCompleted()** versorgen, sondern kann alternativ dazu ein **Window**-Eigenschaftselement in der XAML-Deklaration verwenden:

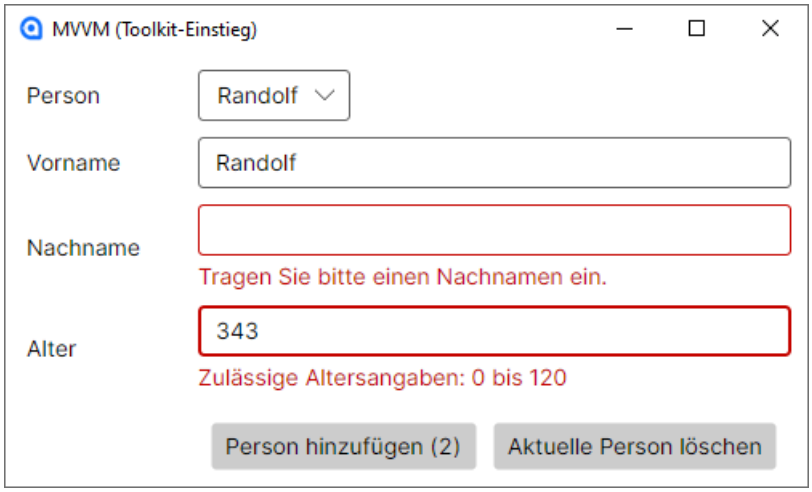
```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:vm="using:MVVM.ViewModels"
        xmlns:local="using:MVVM"
        x:Class="MVVM.Views.MainWindow"
        x:DataType="vm:MainViewModel"
        Icon="/Assets/avalonia-logo.ico"
        Width="500" SizeToContent="Height" MinHeight="240"
        Title="MVVM (Toolkit-Einstieg)">

    <Window.DataContext>
        <vm:ViewModels.MainViewModel/>
    </Window.DataContext>

     . . .

</Window>
```

Das Programm ist einsatzfähig, doch sind die Schalter vorübergehend funktionslos:



12.3 Kommandobindung

Die im Kapitel 9 behandelten Techniken zur Datenbindung, Änderungsbenachrichtigung und Validierung sind für die MVVM-Architektur von zentraler Bedeutung, weil sie die lose Kopplung von Ansicht, Ansichtsmodell und Modell unterstützen. Mit Hilfe der nun zu beschreibenden Kommandobindung lässt sich die erwünschte Trennung der Bedienoberfläche (GUI-Layout und -Logik) von der Präsentations- und Anwendungslogik entscheidend voranbringen. Statt z. B. auf **Click**-Ereignisse durch Behandlungsmethoden der Fensterklasse (im Code Behind) zu reagieren, ...

- definiert man Kommandoklassen, die das Interface **ICommand** implementieren,
- erweitert die Ansichtsmodellklasse um Eigenschaften vom Typ **ICommand**, die auf Objekte implementierender Klassen zeigen,
- und bindet Kommando-fähige Steuerelemente (z. B. Befehlsschalter oder Menü-Items) über ihre **Command**-Eigenschaften an die **ICommand**-Eigenschaften der Ansichtsmodellklasse.

Bei korrekter Implementierung der Schnittstelle **ICommand** erfährt ein Bedienelement von der Ausführbarkeit des verbundenen Kommandos und zeigt zum Vorteil der Benutzer die Verwendbarkeit optisch an.

Aus der MVVM-konformen Separation resultieren u. a. die folgenden Vorteile:

- Während Ereignisbehandlungsmethoden durch die Kopplung an die Bedienoberfläche schwer zu testen sind, bieten die **ICommand**- und die Ansichtsmodellklassen gute Voraussetzungen für das Erstellen von Modultests.
- In **ICommand**-Klassen konzentrierte Logik ist leichter wiederzuverwenden als die in Code Behind – Dateien realisierte.

Die deklarative Bindung von Daten und Kommandos per XAML ist der wesentliche Unterschied zwischen einer MVVM-Ansichtsmodellklasse und einer Controller-Klasse in einer Anwendung mit MVC-Architektur (Model View Controller), die eine Referenz auf die Ansicht benötigt.

Die von WPF angebotenen *vordefinierten* Kommandos, die von der Basisklasse **RoutedCommand** abstammen, fehlen in Avalonia (siehe Abschnitt 12.12.4.6 in Baltes-Götz & Frischknecht 2026).

Wir entwickeln sukzessive die Kommandotechnik von der manuellen Realisation bis zur Nutzung von Bibliotheksklassen und Code-Generatoren:

- Am Anfang steht eine transparente, aber aufwändige Lösung, die für jedes Kommandoobjekt eine eigene Klasse definiert (siehe Abschnitt 12.3.2).
- Es folgt die selbst implementierte Klasse **RelayCommand**, die über Konstruktorparameter das Instanziieren von unterschiedlichen Kommandoobjekten erlaubt (siehe Abschnitt 12.3.3).
- Im Abschnitt 12.4.3 verwenden wir die Klasse **RelayCommand** aus dem MVVM Community Toolkit, statt eine eigene Klasse zu definieren.
- Im Abschnitt 12.4.4.3 verwenden wir die Klasse **RelayCommand** aus dem MVVM Community Toolkit indirekt, indem wir Ansichtsmodellmethoden mit dem *Attribut* **RelayCommand** dekorieren, wobei ein Generator im Toolkit hinter den Kulissen viel C# - Quellcode beisteuert.

Durch die (direkte oder indirekte) Verwendung der Toolkit-Klasse **RelayCommand** spart man Aufwand (Boilerplate Code). Weil dieses Vorgehen viele Details versteckt, ist es durch die Vorerfahrungen mit expliziten Klassen am besten zu verstehen.

Statt in jedem Schritt ein neues Projekt anzulegen, kopieren wir den Projektordner vom letzten Stand und nehmen die folgenden Vorbereitungen für das neue Projekt vor:

- Projektordner umbenennen
- **csproj**-Datei umbenennen
- **slnx**-Datei umbenennen und darin den Projektnamen ändern
- die Ordner **.vs**, **bin** und **obj** löschen

12.3.1 ICommand

Zur Realisation der in einer MVVM-Anwendung eingesetzten Kommandos dienen Klassen, die das Interface **ICommand** im Namensraum **System.Windows.Input** implementieren:

```
public interface ICommand {
   void Execute(Object? parameter);
   bool CanExecute(Object? parameter);
   event EventHandler? CanExecuteChanged;
}
```

Von den Schnittstellen-Mitglieder wird das folgende Verhalten erwartet:

- Die Methode **Execute()** erledigt die zum Kommando gehörenden Arbeiten.
- Über die Methode **CanExecute()** informiert die implementierende Klasse darüber, ob aktuell eine Ausführung der Methode **Execute()** möglich ist. Ein verbundenes Steuerelement signalisiert aufgrund einer **CanExecute()** – Abfrage optisch seine Nutzbarkeit.
- Das **ICommand**-Ereignis **CanExecuteChanged** signalisiert eine mutmaßliche Änderung der **CanExecute()** - Rückgabe. Avalonia reagiert auf dieses Ereignis mit einem **CanExecute()** - Aufruf.

Erfüllt z. B. die Klasse `AddPersonCmd` (siehe Abschnitt 12.3.2.1)

```
public class AddPersonCmd : ICommand {
    public bool CanExecute(object? parameter) { ... }
    public void Execute(object? parameter) { ... }
    public event EventHandler? CanExecuteChanged { ... }
}
```

die Schnittstelle **ICommand**, dann kann man in einer Ansichtsmodellklasse eine **ICommand**-Eigenschaft durch ein `AddPersonCmd`-Objekt initialisieren, z. B.:

```
public ICommand AddPersonCmd { get; } = new AddPersonCmd();
```

Die Methode **Execute()** führt die zum Kommando gehörenden Operationen aus, wenn ein verbundenes Steuerelement das Kommando anstößt (z. B. ein gewähltes **MenuItem** oder ein geklickter **Button**). Das folgende XAML-Element

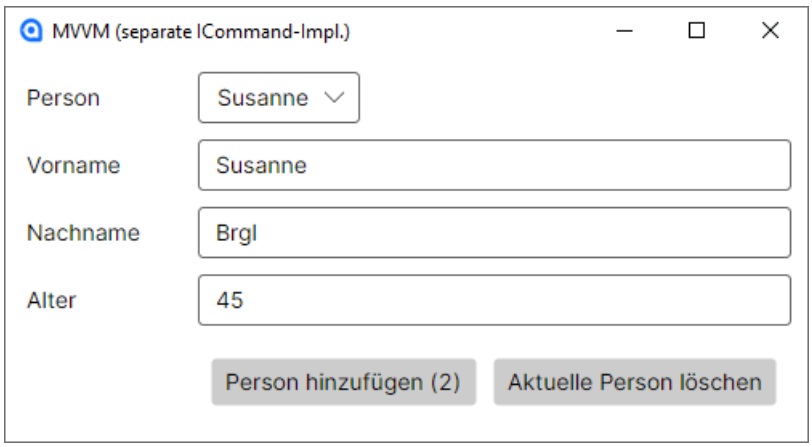
```
<Button Command="{Binding AddPersonCmd}" CommandParameter="{Binding}">
```

bindet einen Schalter über seine **Command**-Eigenschaft an das Kommando `AddPersonCmd`. Beim Aufruf erhält die **ICommand**-Methode **Execute()** ein Objekt mit den erforderlichen Daten (z. B. das die **ICommand**-Eigenschaft anbietende Objekt der Ansichtsmodellklasse). Dieses Objekt legt man durch die **CommandParameter**-Eigenschaft eines verbundenen Steuerelements fest.

Ob ein Steuerelement als verfügbar angezeigt sein soll, ermittelt die Avalonia-Kommandotechnik über die **ICommand**-Methode **CanExecute()**. Diese Methode erhält beim Aufruf dasselbe Parameterobjekt wie die Methode **Execute()**.

12.3.2 Separate ICommand-Implementationen

Wir statten nun im Personenverwaltungsprogramm die Befehlsschalter mit den Funktionen zur Aufnahme bzw. zum Löschen eines Falls aus:



Diese Funktionen realisieren wir *nicht* durch Ereignisbehandlungsmethoden, sondern durch Kommandos. Dazu legen wir im Projekt den Unterordner **Commands** an und erstellen dort die Klassen `AddPersonCmd` und `DeletePersonCmd`, die das Interface **ICommand** implementieren.[1]

12.3.2.1 Kommandoklasse `AddPersonCmd`

Dass die Klasse `AddPersonCmd` erheblich mehr Aufwand verursacht als eine verwandte Ereignisbehandlungsmethode liegt nicht nur an der MVVM-Architektur, sondern auch am angestrebten Bedienungskomfort: Ob ein Kommando ausführbar ist, und ein verbundener Schalter somit eine Funktion besitzt, soll für die Benutzer erkennbar sein.

---

[1] Ein Visual Studio - Projekt mit der aktuellen Variante des Personenverwaltungsprogramms ist hier zu finden:
...\BspUeb\MVVM\Separate ICommand-Implementationen

```
using MVVM.Models;
using MVVM.ViewModels;
using System;
using System.ComponentModel;
using System.Linq;
using System.Windows.Input;

namespace MVVM.Commands;

public class AddPersonCmd : ICommand {
  readonly PropertyChangedEventHandler pceh;

  public AddPersonCmd() {
    pceh = (sender, args) => RaiseCanExecuteChanged();
  }

  public bool CanExecute(object? parameter) {
    if (parameter is not MainViewModel viewModel || viewModel.Personen == null)
      return false;
    Person? lastPerson = viewModel.Personen.LastOrDefault();
    if (lastPerson == null)
      return true;
    if (lastPerson.FirstName == "NN" || lastPerson.LastName == "NN" || lastPerson.Age == 0)
      return false;
    else
      return true;
  }

  public void Execute(object? parameter) {
    if (parameter is not MainViewModel viewModel || viewModel.Personen == null)
      return;
    if (viewModel.Personen.Count > 0)
      viewModel.Personen.Last().PropertyChanged -= pceh;
    Person person = new() { FirstName = "NN", LastName = "NN", Age = 0 };
    person.PropertyChanged += pceh;
    viewModel.Personen.Add(person);
    viewModel.SelectedItem = person;
  }

  public event EventHandler? CanExecuteChanged;

  public void RaiseCanExecuteChanged() => CanExecuteChanged?.Invoke(this, EventArgs.Empty);
}
```

Solange der Benutzer für eine neue `Person`-Instanz noch nicht alle Initialisierungswerte (z. B. den Vornamen *NN* für *Nomen Nominandum*) durch gültige Werte ersetzt hat,

```
Person person = new() { FirstName = "NN", LastName = "NN", Age = 0 };
```

soll das Hinzufügen einer weiteren Person gesperrt sein, weil ansonsten ein irregulärer Fall entstehen würde. Nach jeder Änderung des neuen Falls soll das `AddPersonCmd`-Objekt sein **CanExecuteChanged**-Ereignis auslösen, damit Avalonia per **CanExecute()** – Aufruf die Verfügbarkeit des Kommandos prüft und ggf. den Schalter (de)aktiviert. Die Initiative muss vom `Person`-Objekt ausgehen, das aufgrund der **INotifyPropertyChanged**-Implementation mit einem **PropertyChanged**-Ereignis ausgestattet ist (siehe Abschnitt 9.6.1).

Ein `AddPersonCmd`-Objekt besitzt zur Übergabe an neu erzeugte `Person`-Instanzen ein Delegatenobjekt vom Typ **PropertyChangedEventHandler** basierend auf einer per Lambda-Ausdruck realisierten Methode, die das **CanExecuteChanged**-Ereignis des Kommandoobjekts auslöst:

```
public class AddPersonCmd : ICommand {
  readonly PropertyChangedEventHandler pceh;

  public AddPersonCmd() {
    pceh = (sender, args) => RaiseCanExecuteChanged();
  }
  . . .
  public event EventHandler? CanExecuteChanged;

  public void RaiseCanExecuteChanged() =>
                CanExecuteChanged?.Invoke(this, EventArgs.Empty);
}
```

In seiner **Execute()** – Methode weist ein `AddPersonCmd`-Objekt dem **PropertyChanged**-Ereignis einer neuen Person den **PropertyChangedEventHandler** zu,

```
Person person = new() { FirstName = "NN", LastName = "NN", Age = 0 };
person.PropertyChanged += pceh;
```

sodass jede Änderung einer beobachteten Eigenschaft (`FirstName`, `LastName`, `Age`) eine Neubeurteilung der Kommando-Ausführbarkeit per **CanExecute()** veranlasst.

Die Methode **Execute()** fügt den so vorbereiteten neuen Fall in die `Personen`-Kollektion ein:

```
viewModel.Personen.Add(person);
```

Für diese Aktionen ist eine Referenz auf das Objekt der Ansichtsmodellklasse `MainViewModel` erforderlich, die **Execute()** per Parameter erhält. Weil die Modellklasse `PersonServices` eine Referenz auf die Kollektion `Personen` besitzt, ist die neue Person auch im Modell bekannt.

Außerdem weist **Execute()** die neue Person der Ansichtsmodell-Eigenschaft `SelectedItem` zu (siehe Abschnitt 12.3.2.3):

```
viewModel.SelectedItem = person;
```

Weil wir die **ComboBox**-Eigenschaft **SelectedItem** mit der gleichnamigen Ansichtsmodelleigenschaft verbinden, zeigt die **ComboBox** den neuen Fall an (zum Datenbindungsmodus **TwoWay** siehe Abschnitt 12.3.2.4):

```
<ComboBox Name="comboBox" Grid.Column="1" ItemsSource="{Binding Personen}"
          SelectedItem="{Binding SelectedItem, Mode=TwoWay}"
          DisplayMemberBinding="{Binding FirstName}"
          PlaceholderText="Person wählen ..." Margin="0,0,0,10"/>
```

Beim zuvor aufgenommenen Fall befindet sich eventuell der **PropertyChangedEventHandler**-Handler des `AddPersonCmd`-Objekts in der **PropertyChanged**-Aufrufliste. Daher führt die Methode **Execute()** vor der Aufnahme des neuen Falls eine Bereinigung durch:

```
if (viewModel.Personen.Count > 0)
    viewModel.Personen.Last().PropertyChanged -= pceh;
```

Die Methode **CanExecute()** signalisiert durch die Rückgabe **true** ein ausführbares Kommando, wenn ...

- das per Parameter bekannt gegebene Objekt der Ansichtsmodellklasse samt `Personen`-Kollektion existiert,
- und die `Personen`-Kollektion ...
  - entweder leer ist
  - oder am Ende einen vollständig initialisierten Fall enthält.[1]

12.3.2.2 Kommandoklasse `DeletePersonCmd`

Zum Löschen des aktuell vom Programm angezeigten Falls erstellen wir im Ordner **Commands** die **ICommand**-Implementation `DeletePersonCmd`:

```
using MVVM.Models;
using MVVM.ViewModels;
using System;
using System.Collections.ObjectModel;
using System.Linq;
using System.Windows.Input;

namespace MVVM.Commands;

public class DeletePersonCmd : ICommand {
  public bool CanExecute(object? parameter) {
    if (parameter is not MainViewModel viewModel || viewModel.Personen == null
                                                 || viewModel.SelectedItem == null)
      return false;
    else
      return true;
  }

  public void Execute(object? parameter) {
    if (parameter is not MainViewModel viewModel || viewModel.Personen == null
                                                 || viewModel.SelectedItem == null)
      return;
    ObservableCollection<Person> personen = viewModel.Personen;
    personen.Remove(viewModel.SelectedItem);
    viewModel.SelectedItem = personen.Count > 0 ? viewModel.Personen.LastOrDefault() : null;
  }

  public event EventHandler? CanExecuteChanged;

  public void RaiseCanExecuteChanged() => CanExecuteChanged?.Invoke(this, EventArgs.Empty);
}
```

Die Methode **CanExecute()** prüft anhand der per Parameter erhaltenen Referenz auf das Ansichtsmodellobjekt, ob die (mit der **ComboBox** synchronisierte) Eigenschaft **SelectedItem** von **null** verschieden ist.

Unter derselben Bedingung führt die Methode **Execute()** die folgenden Operationen aus:

- Sie löscht den von **SelectedItem** referenzierten Fall aus der `Personen`-Kollektion.
- Sie aktualisiert die Ansichtsmodelleigenschaft `SelectedItem` auf den letzten verbleibenden Fall oder auf **null**, wenn kein Fall mehr vorhanden ist. Aufgrund der bestehenden Bindung passt sich die **ComboBox** an.

Wie die Klasse `AddPersonCmd` bietet auch die Klasse `DeletePersonCmd` die Methode `RaiseCanExecuteChanged()` an, die das **CanExecuteChanged**-Ereignis auslöst, wenn Interessenten vorhanden sind. Für die Existenz eines Interessenten sorgt Avalonia UI, damit die Ansicht

---

[1] Die Erweiterungsmethode **LastOrDefault()** der statischen Klasse **Enumerable** im Namensraum **System.Linq** unterstützt die Schnittstelle **IEnumerable<TSource>** implementierende Klassen und liefert zu einer Liste das letzte Element oder **null** im Fall einer leeren Liste (zur ***LINQ***-Technik (***L**anguage **I**ntegrated **Q**uery*) siehe z. B. Kapitel 18 in Baltes-Götz & Frischknecht 2026).

dank **CanExecute()** – Aufruf bzgl. der Ausführbarkeit des Kommandos auf dem aktuellen Stand bleibt. Die Ansichtsmodellklasse ruft `RaiseCanExecuteChanged()` bei jedem Schreibzugriff auf die Eigenschaft `SelectedItem` auf (siehe Abschnitt 12.3.2.3).

12.3.2.3 Ansichtsmodellklasse

In der Ansichtsmodellklasse `MainViewModel`

```
using MVVM.Commands;
using MVVM.Models;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace MVVM.ViewModels;

public class MainViewModel : INotifyPropertyChanged {
    public ObservableCollection<Person>? Personen { get; }

    public MainViewModel() {
        Personen = PersonServices.GetPeople();
        SelectedItem = Personen?.Count > 0 ? Personen[0] : null;
    }

    public Person? SelectedItem {
        get; set {
            if (value == field)
                return;
            field = value;
            OnPropertyChanged();
            AddPersonCmd.RaiseCanExecuteChanged();
            DeletePersonCmd.RaiseCanExecuteChanged();
        }
    }

    public event PropertyChangedEventHandler? PropertyChanged;

    protected void OnPropertyChanged([CallerMemberName] string propertyName = "") {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    public AddPersonCmd AddPersonCmd { get; } = new AddPersonCmd();
    public DeletePersonCmd DeletePersonCmd { get; } = new DeletePersonCmd();
}
```

sind zur Realisation der Kommandobindung im Vergleich zum Abschnitt 12.2 einige Änderungen erforderlich:

- Um das im **ComboBox**-Steuerelement ausgewählte Element beeinflussen zu können, erhält die Ansichtsmodellklasse die Eigenschaft `SelectedItem`.
- Das Implementieren der Schnittstelle **INotifyPropertyChanged** macht es möglich, die Bedienoberfläche über eine Änderung der Eigenschaft `SelectedItem` zu informieren (siehe Abschnitt 9.6.1).
- Dazu ist das Ereignis **PropertyChanged** erforderlich, das die Ansichtsmodellklasse in der Methode `OnPropertyChanged()` auslöst.
- Schließlich erhält die Ansichtsmodellklasse die **get-only** – Eigenschaften `AddPersonCmd` und `DeletePersonCmd`, die auf Objekte der gleichnamigen Kommandoklassen zeigen, die beide die Schnittstelle **ICommand** implementieren.

Der **set**-Block der Eigenschaft `SelectedItem` ruft die Methode `OnPropertyChanged()` auf, um die Ansicht (speziell das dortige **ComBox**-Steuerelement) über Änderungen der Eigenschaft (z. B. aufgrund der Ergänzung eines Falls) zu informieren (siehe Abschnitt 9.6.1). Außerdem ruft der **set**-Block die **RaiseCanExecuteChanged()** – Methoden der beiden Kommandoobjekte auf, um **CanExecute()** Anfragen zur Ausführbarkeit zu initiieren. Nach dem Einfügen eines neuen Falls in die `Personen`-Kollektion endet z. B. die **CanExecute()** – Anfrage an das `AddPersonCmd`-Objekt negativ, weil die Eigenschaften des neuen Falls noch nicht initialisiert sind. War die `Personen`-Kollektion vor dem Einfügen leer und der Schalter zum Löschen folglich nicht verfügbar, dann ändert sich die **CanExecute()** – Rückgabe des `DeletePersonCmd`–Objekts durch das Einfügen eines Falls in der positiven Richtung. Nach dem Löschen eines Falls kann sich die **CanExecute()** – Rückgabe des `DeletePersonCmd`–Objekts in der negativen Richtung ändern, wenn keine Fälle mehr vorhanden sind.

12.3.2.4 XAML-Code der Ansicht

Nach der Erstellung der Kommandoklassen `AddPersonCmd` und `DeletePersonCmd` sowie der Erweiterung der Ansichtsmodellklasse `MainViewModel` machen wir uns an die Modifikation der Ansicht in der Datei **MainWindow.xaml**:

- Unter den jetzigen Voraussetzungen einer Ansichtsmodellklasse, die ...
  - das Interface **INotifyPropertyChanged** implementiert
  - und die Eigenschaft `SelectedItem` vom deklarierten Typ `Person` besitzt,

  lassen sich die Eigenschafts-Bindungen im XAML-Code vereinfachen:
  - Wir binden die **ComboBox**-Eigenschaft `SelectedItem` bidirektional an die gleichnamige Ansichtsmodelleigenschaft:

    ```
    <ComboBox Name="comboBox" Grid.Column="1"
              ItemsSource="{Binding Personen}"
              SelectedItem="{Binding SelectedItem, Mode=TwoWay}"
              DisplayMemberBinding="{Binding FirstName}"
              PlaceholderText="Person wählen ..." Margin="0,0,0,10"/>
    ```

    Die Richtung von der Quelle zum Ziel sorgt dafür, dass die **ComboBox** auf eine in **Execute()** – Methoden vorgenommene Änderung der Ansichtsmodelleigenschaft `SelectedItem` reagiert. Aufgrund der Bidirektionalität ändert eine per **ComboBox** vorgenommene Auswahl die `SelectedItem`-Eigenschaft des Ansichtsmodellobjekts.
  - Daher können wir die **TextBox**-Objekte im Details-Bereich des Fensters an die Ansichtsmodelleigenschaft **SelectedItem** binden statt an die **ComboBox**-Eigenschaft, sodass der separate **DataContext** für den **Grid**-Container mit den **TextBox**-Objekten (siehe Abschnitt 12.2) überflüssig wird, z. B.:

    ```
    <TextBox Name="firstName" Grid.Column="1" Grid.Row="0"
             Text="{Binding SelectedItem.FirstName" Margin="0,0,0,10" />
    ```

- In den **Button**-Elementen binden wir die **Command**-Eigenschaften an die zugehörigen, Objekte mit **ICommand**-Implementation referenzierenden Eigenschaften des Ansichtsmodells. Die **Click**-Ereignisse der **Button**-Elemente stoßen daher die zugehörigen Kommandos an, während keine Ereignisbehandlungsmethoden in der Code Behind – Datei vorhanden sind. Um an die **CanExecute()** – und an die **Execute()** – Methoden der Kommandoklassen das Objekt der Ansichtsmodellklasse per Parameter zu übergeben, weisen wir den **CommandParameter**-Attributen der **Button**-Elemente jeweils ein **Binding**-Objekt zu. Es kommt ohne Pfadspezifikation aus, weil das Objekt der Ansichtsmodellklasse als **DataContext** eingestellt ist:

```
<StackPanel Grid.Column="1" HorizontalAlignment="Center"
            Orientation="Horizontal" Margin="0,20,0,0">
    <Button Command="{Binding AddPersonCmd}"
            CommandParameter="{Binding}">
        <Button.Content>
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="Person hinzufügen (" />
                <TextBlock Text="{Binding Personen.Count}" />
                <TextBlock Text=")" />
            </StackPanel>
        </Button.Content>
    </Button>
    <Button Command="{Binding DeletePersonCmd}"
            CommandParameter="{Binding}"
            Content="Aktuelle Person löschen" Margin="10,0,0,0" />
</StackPanel>
```

- Wir entfernen das Label mit dem nicht mehr passenden Inhalt `INotify-Demo`:

```
<Label Grid.Column="0" Content="INotify-Demo"
       VerticalContentAlignment="Center" Margin="0,20,0,0" />
```

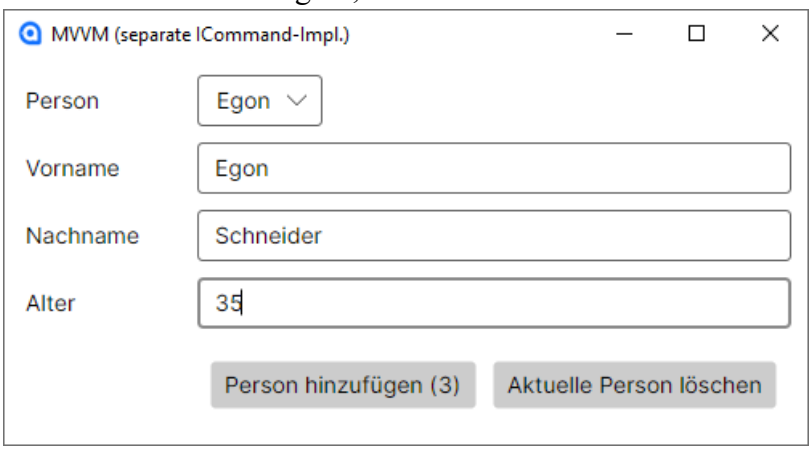
- Schließlich erhalten die **Text**-Bindungen der **TextBox**-Elemente zur Erfassung der `Person`-Eigenschaften als Wert für die Eigenschaft **UpdateSourceTrigger** den Wert **PropertyChanged**, z. B.:

```
<TextBox Name="firstName" Grid.Column="1" Grid.Row="0" Margin="0,0,0,10"
         Text="{Binding SelectedItem.FirstName,
                UpdateSourceTrigger=PropertyChanged}" />
```

Damit führt jede Änderung einer **Text**-Zeichenfolge zur Aktualisierung der Kommandobindungsquelle, statt die am Fokusverlust erkennbare Beendigung des Editierens abzuwarten. Die Aktualisierung der Kommandobindungsquelle hat eine erneute **CanExecute()** – Abfrage zur Folge, sodass der Schalter zum Kommando `AddPersonCmd` frühestmöglich wieder verfügbar ist, z. B.:

```
public bool CanExecute(object? parameter) {
   . . .
  if (lastPerson.FirstName=="NN" || lastPerson.LastName=="NN" || lastPerson.Age==0)
    return false;
  else
    return true;
}
```

Anderenfalls müssten die Benutzer nach dem vollständigen Erfassen einer neuen Person das zuletzt benutzte **TextBox**-Feld verlassen, um den Schalter wieder verfügbar zu machen. Mit dem neuen Trigger-Wert ist der Schalter **Person hinzufügen** nach der vollständigen Bearbeitung des neuen Falls sofort verfügbar, z. B.:



12.3.3 Eigene Kommandoklasse RelayCommand

Im Bemühen um eine Vereinfachung des Personenverwaltungsprogramms ersetzen wir die beiden Kommandoklassen `AddPersonCmd` und `DeletePersonCmd` durch *eine* Klasse namens `RelayCommand`, die das Erstellen von Kommandoobjekten für beliebige parameterfreie Aufgaben erlaubt. Dazu hat die Klasse zwei Mitglieder mit Delegatentypen für die **CanExecute()** – bzw. **Execute()** – Methode eines Kommandos sowie zwei zur Initialisierung der Delegatenvariablen dienende Konstruktorparameter:[1]

```
using System;
using System.Windows.Input;

namespace MVVM.Commands;

public class RelayCommand : ICommand {
    readonly Action _execute;
    readonly Func<bool>? _canExecute;

    public RelayCommand(Action execute) : this(execute, null) { }

    public RelayCommand(Action execute, Func<bool>? canExecute) {
        ArgumentNullException.ThrowIfNull(execute);
        _execute = execute;
        _canExecute = canExecute;
    }

    public bool CanExecute(object? parameter) {
        return _canExecute == null || _canExecute();
    }

    public void Execute(object? parameter) {
        _execute();
    }

    public event EventHandler? CanExecuteChanged;

    public void RaiseCanExecuteChanged() =>
            CanExecuteChanged?.Invoke(this, EventArgs.Empty);
}
```

Es steht auch ein Konstruktor mit *einem* Parameter zur Verfügung, der einen **Action**-Delegaten erwartet und den Doppelparameter-Konstruktor mit dem Wert **null** als **Func<bool>** - Aktualparameter aufruft. In dieser Situation liefert die Methode **CanExecute()** generell die Rückgabe **true**, sodass ein verbundenes Steuerelement generell verfügbar ist.

Im Namen der Klasse `RelayCommand` kommt zum Ausdruck, dass sie zur Weiterleitung der Aufgaben an andere Klassen dient. Eine Klasse mit diesem Namen und Verhalten wird uns auch im MVVM Community Toolkit begegnen (siehe Abschnitt 12.4.3).

Die **ICommand**-Methoden **CanExecute()** und **Execute()** ignorieren ihren **Object**-Parameter und rufen die parameterfreien Delegaten auf.

Wir erstellen in der Ansichtsmodellklasse `MainViewModel` für jedes Kommando ein Paar von Methoden mit passenden Delegatensignaturen:

---

[1] Ein Visual Studio - Projekt mit der aktuellen Variante des Personenverwaltungsprogramms ist hier zu finden: **...\BspUeb\MVVM\RelayCommand**

```
using MVVM.Commands;
using MVVM.Models;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Linq;
using System.Runtime.CompilerServices;

namespace MVVM.ViewModels;

public class MainViewModel : INotifyPropertyChanged {
    public ObservableCollection<Person>? Personen { get; }
    PropertyChangedEventHandler pceh;
    public RelayCommand AddPersonCmd { get; }
    public RelayCommand DeletePersonCmd { get; }

    public MainViewModel() {
        AddPersonCmd = new RelayCommand(AddPerson, CanAddPerson);
        DeletePersonCmd = new RelayCommand(DeletePerson, CanDeletePerson);
        Personen = PersonServices.GetPeople();
        SelectedItem = Personen?.Count > 0 ? Personen[0] : null;
        pceh = (sender, args) => AddPersonCmd.RaiseCanExecuteChanged();
    }

    public Person? SelectedItem {
        get; set {
            if (value == field)
                return;
            field = value;
            OnPropertyChanged();
            DeletePersonCmd.RaiseCanExecuteChanged();
            AddPersonCmd.RaiseCanExecuteChanged();
        }
    }

    public bool CanAddPerson() {
        if (Personen == null)
            return false;
        Person? lastPerson = Personen.LastOrDefault();
        if (lastPerson == null)
            return true;
        if (lastPerson.FirstName == "NN" || lastPerson.LastName == "NN"
                                         || lastPerson.Age == 0)
            return false;
        else
            return true;
    }

    public void AddPerson() {
        if (Personen == null)
            return;
        if (Personen.Count > 0)
            Personen.Last().PropertyChanged -= pceh;
        Person person = new() { FirstName = "NN", LastName = "NN", Age = 0 };
        person.PropertyChanged += pceh;
        Personen.Add(person);
        SelectedItem = person;
    }

    public bool CanDeletePerson() {
        if (SelectedItem == null)
            return false;
        else
            return true;
    }
```

```
        public void DeletePerson() {
            if (Personen == null || SelectedItem == null)
                return;
            Personen.Remove(SelectedItem);
            SelectedItem = Personen.LastOrDefault();
        }

        public event PropertyChangedEventHandler? PropertyChanged;

        protected void OnPropertyChanged([CallerMemberName] string propertyName = "") {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }
    }
```

Beim Erstellen eines Kommandoobjekts übergeben wir dem `RelayCommand`-Konstruktor als Aktualparameter zwei auf Instanzmethoden der Ansichtsmodellklasse (z. B. `AddPerson()`, `CanAddPerson`()) basierende Delegatenobjekte. Die von einem solchen Delegatenobjekt gekapselte Methode wird vom Ansichtsmodellobjekt ausführt, das dabei seine Eigenschaften `Personen` und `SelectedItem` zur Verfügung hat.

Die Implementation des Ereignisses **CanExecuteChanged** erfolgt in der Klasse **RelayCommand** genauso wie in den separaten Kommandoklassen. Nun definiert aber die Ansichtsmodellklasse einen **PropertyChangedEventHandler** und weist ihn in `AddPerson()` dem **PropertyChanged**-Ereignis einer neuen `Person`-Instanz zu.

Dank der Klasse `RelayCommand` kann die Ansichtsmodellklasse *mehrere* Kommandos realisieren, was aber nicht zu einer unübersichtlichen Klasse führen sollte.

12.3.4 Kommandobindung an eine Methode

Avalonia UI ermöglicht auch die direkte Kommandobindung an eine Methode, wobei die Schnittstelle **ICommand** *nicht* im Spiel ist. Man erhält eine vereinfachte Syntax, muss aber auf die Anzeige der Kommandoverfügbarkeit in der Bedienoberfläche verzichten.

In einer vereinfachten Variante des Personenverwaltungsprogramms streichen wir im Vergleich zum Abschnitt 12.3.3 aus der Ansichtsmodellklasse ...

- die **ICommand**- bzw. `RelayCommand`-Eigenschaften `AddPersonCmd` und `DeletePersonCmd`
- sowie die Methoden `CanAddPerson()` und `CanDeletePerson()`.

Es verbleiben die öffentlichen Methoden `AddPerson()` und `DeletePerson()`:[1]

```
public void AddPerson() {
    if (Personen == null)
        return;
    Person person = new() { FirstName = "NN", LastName = "NN", Age = 0 };
    Personen.Add(person);
    SelectedItem = person;
}

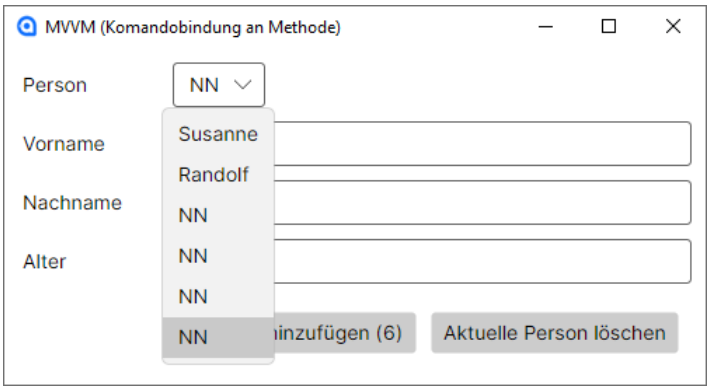
public void DeletePerson() {
    if (Personen == null || SelectedItem == null)
        return;
    Personen.Remove(SelectedItem);
    SelectedItem = Personen.LastOrDefault();
}
```

[1] Ein Visual Studio - Projekt mit der aktuellen Variante des Personenverwaltungsprogramms ist hier zu finden: **...\BspUeb\MVVM\Kommandobindung an eine Methode**

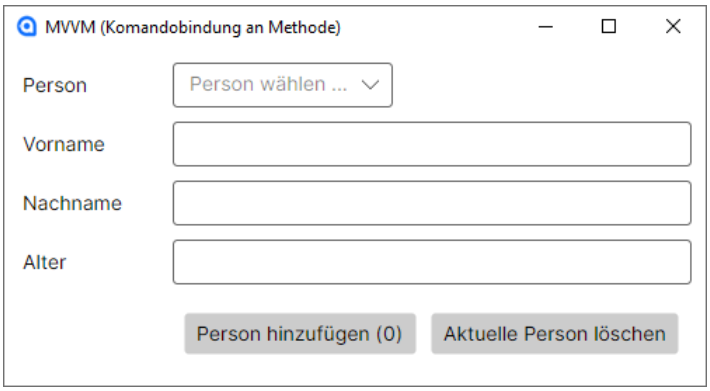
Im XAML-Code des Fensters weisen wir den **Command**-Eigenschaften der Schalter statt einer **ICommand**-Eigenschaft einen Methodennamen zu, z. B.:

```
<Button Command="{Binding AddPerson}">
```

Das Programm sieht auf den ersten Blick unverändert aus, erlaubt aber aufgrund des nunmehr unregulierten **Hinzufügen**-Schalters eine Anhäufung von uninitialisierten Fällen, z. B.:



Außerdem ist der Schalter zum Löschen von Fällen auch bei einer leeren Personenkollektion scheinbar noch verwendbar:



Wenn durch den Verzicht auf die **CanExecute()** – Methode und das **CanExecuteChanged**-Ereignis keine Verschlechterung der Bedienbarkeit eintritt, spricht nichts gegen ein Kommando ohne **ICommand**-Implementation.

12.4 MVVM Community Toolkit

Das MVVM Community Toolkit vereinfacht die Entwicklung einer MVVM-Anwendung drastisch, was wir nun anhand des zuletzt im Abschnitt 12.3.3 überarbeiteten Personenverwaltungsprogramms *mit* Schalterverfügbarkeitsanzeige demonstrieren.[1] Wir erstellen eine Kopie des Projekts im Ordner

...\BspUeb\MVVM\RelayCommand

und ersetzen die manuellen Implementationen der Schnittstellen **INotifyPropertyChanged**, **INotifyDataErrorInfo** und **ICommand** mit Hilfe des Community Toolkits durch weitgehend automatisierte Lösungen.[2] Dabei profitieren wir vom Wissen um die zugrundeliegenden Techniken.

Das NuGet-Paket **CommunityToolkit.Mvvm** müssen wir nicht manuell installieren, weil es durch die Nutzung der Projektvorlage **Avalonia .NET MVVM App** automatisch eingerichtet wurde.

---

[1] https://learn.microsoft.com/en-us/dotnet/communitytoolkit/mvvm/

[2] Ein Visual Studio - Projekt mit der Toolkit-Variante des Personenverwaltungsprogramms ist hier zu finden:
...\BspUeb\MVVM\Toolkit\Einstieg

12.4.1 ObservableObject

Benötigt man für eine Modell- oder Ansichtsmodellklasse nur die Datenbindung und die Änderungsbenachrichtigung, dann bietet sich aus dem Toolkit die Basisklasse **ObservableObject** an, die die Schnittstelle **INotifyPropertyChanged** (siehe Abschnitt 9.6.1) sowie die für *bevorstehende* Werteänderungen zuständige, bislang im Manuskript noch nicht erwähnte Schnittstelle **INotifyPropertyChanging** implementiert.[1]

Für die Datenbindung (GUI-Aktualisierung) ist ausschließlich die Schnittstelle **INotifyPropertyChanged** mit dem Ereignis **PropertyChanged** relevant. Man kann die Schnittstelle **INotifyPropertyChanging** mit dem Ereignis **PropertyChanging** z. B. zur Realisation einer **Rückgängig**-Funktion (engl.: **Undo**) einsetzen. Das Ereignis **PropertyChanging** tritt vor der Werteänderung und dem darauffolgenden Ereignis **PropertyChanged** auf.

Im Demonstrationsprojekt implementiert auf dem kopierten Stand (siehe Abschnitt 12.3.3) die Ansichtsmodellklasse `MainViewModel` das Interface **INotifyPropertyChanged**:

```
public class MainViewModel : INotifyPropertyChanged {
    public ObservableCollection<Person>? Personen { get; }
    PropertyChangedEventHandler pceh;
    public RelayCommand AddPersonCmd { get; }
    public RelayCommand DeletePersonCmd { get; }

public MainViewModel() {
    AddPersonCmd = new RelayCommand(AddPerson, CanAddPerson);
    DeletePersonCmd = new RelayCommand(DeletePerson, CanDeletePerson);
    Personen = PersonServices.GetPeople();
    SelectedItem = Personen?.Count > 0 ? Personen[0] : null;
    pceh = (sender, args) => AddPersonCmd.RaiseCanExecuteChanged();
}

public Person? SelectedItem {
    get; set {
        if (value == field)
            return;
        field = value;
        OnPropertyChanged();
        DeletePersonCmd.RaiseCanExecuteChanged();
        AddPersonCmd.RaiseCanExecuteChanged();
    }

    . . .

    public event PropertyChangedEventHandler? PropertyChanged;

    protected void OnPropertyChanged([CallerMemberName] string propertyName = "") {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

An die `MainViewModel`-Eigenschaften `Personen` und `SelectedItem` ist das **ComboBox**-Steuerelement der Ansicht gebunden:

```
<ComboBox Name="comboBox" Grid.Column="1" ItemsSource="{Binding Personen}"
          SelectedItem="{Binding SelectedItem, Mode=TwoWay}"
          DisplayMemberBinding="{Binding FirstName}"
          PlaceholderText="Person wählen ..." Margin="0,0,0,10"/>
```

Eine Änderungsbenachrichtigung muss die Ansichtsmodellklasse nur für `SelectedItem` vornehmen, weil `Personen` vom Typ **ObservableCollection**<Person> ist, der die Schnittstelle **INotifyCollectionChanged** implementiert.

---

[1] https://learn.microsoft.com/en-us/dotnet/api/microsoft.toolkit.mvvm.componentmodel.observableobject

Durch die Nutzung von **ObservableObject** als Basisklasse für `MainViewModel` kann man sich das Implementieren der Schnittstelle **INotifyPropertyChanged** ersparen. Dabei sind die folgenden Änderungen an der Klassendefinition vorzunehmen:

- Für das Toolkit-Paket ist in der Klassendefinition die folgende **using**-Direktive erforderlich, die das Visual Studio bei Verwendung von Toolkit-Typen automatisch erstellt:
  ```
  using CommunityToolkit.Mvvm.ComponentModel;
  ```
- Die Klasse benötigt den Modifikator **partial**, weil Codegeneratoren im Toolkit zur Klassendefinition beitragen. Aufgrund der Projektvorlage **Avalonia .NET MVVM App** automatisch entstandene Ansichtsmodellklassen erhalten generell den Modifikator **partial**, den wir bisher als überflüssig entfernt haben.
- Statt der implementierten Schnittstelle **INotifyPropertyChanged** ist die Basisklasse **ObservableObject** anzugeben, sodass insgesamt der folgende Definitionskopf entsteht:
  ```
  public partial class MainViewModel : ObservableObject { . . . }
  ```
- Wir löschen das Ereignis `PropertyChanged` und die Methode `OnPropertyChanged()`.
- Die Eigenschaft `SelectedItem` erhält den seit C# 13 auch bei Eigenschaften zulässigen Modifikator **partial**, damit wir das Toolkit durch das Attribut **ObservableProperty** auffordern können, den zur Realisation der Änderungsbenachrichtigung erforderlichen Quellcode zu erstellen:
  ```
  [ObservableProperty]
  public partial Person? SelectedItem { get; set; }
  ```
  Mit der manuellen **set**-Implementation der Eigenschaft `SelectedItem` entfallen auch die die Methodenaufrufe zum Auslösen der `CanExecuteChanged`-Ereignisse der Kommandoobjekte:
  ```
  DeletePersonCmd.RaiseCanExecuteChanged();
  AddPersonCmd.RaiseCanExecuteChanged();
  ```
  In der vom Toolkit-Quellcodegenerator definierten und bei einer `SelectedItem`-Änderung aufgerufenen Methode `OnSelectedItemChanged()` werden wir auf analoge Weise das Auslösen der `CanExecuteChanged`-Ereignisse veranlassen (siehe Abschnitt 12.4.3).

Seit der Version 8.4 des Community Toolkits empfiehlt des Entwicklerteam, partielle Eigenschaften statt der vorher üblichen privaten backing fields zu verwenden, z. B.:[1]

```
[ObservableProperty]
Person? selectedItem;
```

Damit wird es möglich, ...

- zwischen den Zugriffsmethoden **get** und **set** zu differenzieren, um z. B. den **set**-Zugriff durch den Modifikator **private** zu beschränken:
  ```
  [ObservableProperty]
  public partial string? Name { get; private set; }
  ```
- Attribute an die Eigenschaften und an die Zugriffsmethoden zu heften.

Man muss dem Toolkit das Implementieren der partiellen Eigenschaften überlassen und darf *keine Anweisungen* einfügen, z. B.:

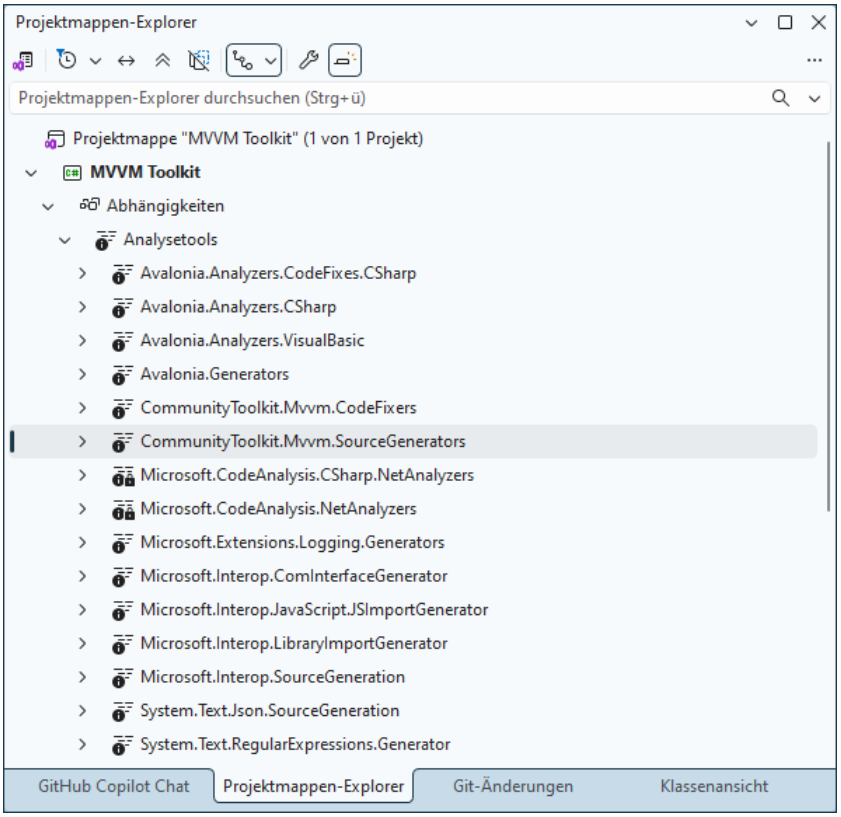
---

[1] https://devblogs.microsoft.com/dotnet/announcing-the-dotnet-community-toolkit-840/

```
[ObservableProperty]
public partial Person? SelectedItem {
    get; set {
          if (value == field)
              return;
          field = value;
           . . .
    }
}
```

Die erste Renovierung der Ansichtsmodellklasse `MainViewModel` mit Hilfe des Toolkits ist nun erfolgreich bewältigt. Wir werden später weitere Änderungen vornehmen und zur Realisation der Kommandobindung die Klasse **RelayCommand** aus dem Toolkit statt unserer Eigenkreation verwenden (siehe Abschnitt 12.4.3).

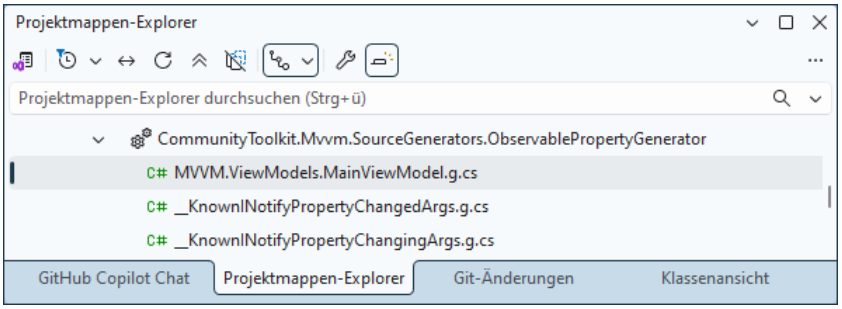
Über die Anwesenheit der Toolkit-Codegeneratoren informiert ein Blick auf den Projektknoten **Abhängigkeiten > Analysetools** im **Projektmappen-Explorer**:



Unter **CommunityToolkit.Mvvm.SourceGenerators** ist der Eintrag

CommunityToolkit.Mvvm.SourceGenerators.ObservablePropertyGenerator

mit (partiellen) Klassendefinitionen zu sehen:

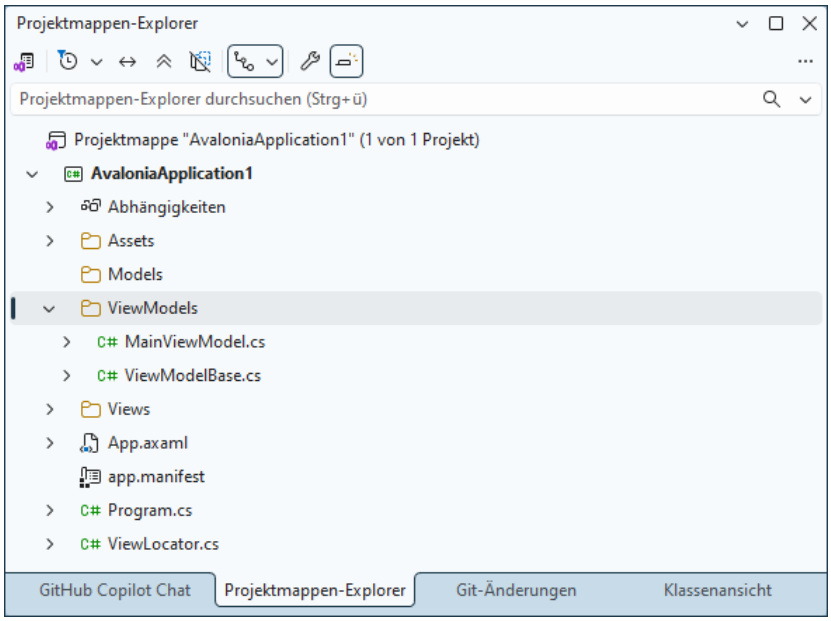


Dieser Quellcode befindet sich im Hauptspeicher, ist also in keiner Festspeicherdatei zu finden (Lischke 2025). Man kann die Quellen aber im Editor öffnen, um z. B. den Beitrag zu unserer Ansichtsmodellklassen-Definition zu inspizieren:

```
partial class MainViewModel {
  [global::System.CodeDom.Compiler.GeneratedCode(
    "CommunityToolkit.Mvvm.SourceGenerators.ObservablePropertyGenerator", "8.4.0.0")]
  [global::System.Diagnostics.CodeAnalysis.ExcludeFromCodeCoverage]
  public partial global::MVVM.Models.Person? SelectedItem {
    get => field;
    set {
        if (!global::System.Collections.Generic.
           EqualityComparer<global::MVVM.Models.Person?>.Default.Equals(field, value)) {
          OnSelectedItemChanging(value);
          OnSelectedItemChanging(default, value);
          OnPropertyChanging(global::CommunityToolkit.Mvvm.ComponentModel.__Internals.
                             __KnownINotifyPropertyChangingArgs.SelectedItem);
          field = value;
          OnSelectedItemChanged(value);
          OnSelectedItemChanged(default, value);
          OnPropertyChanged(global::CommunityToolkit.Mvvm.ComponentModel.__Internals.
                            __KnownINotifyPropertyChangedArgs.SelectedItem);
        }
    }
  }
  . . .
}
```

Die vom Generator definierte und bei einer `SelectedItem`-Änderung aufgerufene Methode `OnSelectedItemChanged()` (mit einem `Person`-Parameter) werden wir im Abschnitt 12.4.3 verwenden.

Ein neu mit der Projektvorlage **Avalonia .NET MVVM App** angelegtes Projekt enthält im Ordner **ViewModels** die Klassen **MainViewModel** und **ViewModelBase**:



Die Klasse **ViewModelBase** stammt von **ObservableObject** ab

```
public abstract class ViewModelBase : ObservableObject {
}
```

und dient als Basisklasse von `MainViewModel`:

```
public partial class MainViewModel : ViewModelBase {
    [ObservableProperty]
    public partial string Greeting { get; set; } = "Welcome to Avalonia!";
}
```

Weil in unserem Beispiel `MainViewModel` die *einzige* Ansichtsmodellklasse ist, haben wir der Klarheit halber **ViewModelBase** gelöscht und `MainViewModel` direkt von **ObservableObject** abgeleitet.

12.4.2 ObservableValidator

Ist für eine Modell- oder Ansichtsmodellklasse neben der Datenbindung mit Änderungsbenachrichtigung auch eine Validierung erforderlich, dann bietet sich die Basisklasse **ObservableValidator** aus dem Toolkit an. Sie stammt von **ObservableObject** ab und implementiert die Schnittstelle **INotifyDataErrorInfo** (siehe Abschnitt 9.7).[1]

In der Modellklasse `Person` haben wir sowohl **INotifyPropertyChanged** als auch **INotifyDataErrorInfo** implementiert und dazu fast 90 Zeilen aufgewendet. Mit Hilfe der Toolkit-Basisklasse **ObservableValidator** erreichen wir dieselbe Funktionalität mit ca. 20 Zeilen:

```
using CommunityToolkit.Mvvm.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace MVVM.Models;

public partial class Person : ObservableValidator {
    [ObservableProperty]
    [NotifyDataErrorInfo]
    [Required(ErrorMessage = "Tragen Sie bitte einen Vornamen ein.")]
    [RegularExpression(@"^[\p{L}\s'-]+$",
                       ErrorMessage = "Ungültige Zeichen im Vornamen.")]
    public partial string? FirstName { get; set; }

    [ObservableProperty]
    [NotifyDataErrorInfo]
    [Required(ErrorMessage = "Tragen Sie bitte einen Nachnamen ein.")]
    [RegularExpression(@"^[\p{L}\s'-]+$",
                       ErrorMessage = "Ungültige Zeichen im Nachnamen.")]
    public partial string? LastName { get; set; }

    [ObservableProperty]
    [NotifyDataErrorInfo]
    [Range(0, 120, ErrorMessage = "Zulässige Altersangaben: {1} bis {2}")]
    public partial int Age { get; set; }
}
```

Die Klasse `Person` und die beobachtbaren Eigenschaften (`FirstName`, `LastName`, `Age`) sind als **partial** zu deklarieren, weil Quellcodegeneratoren im Toolkit ca. 200 Quellcodezeilen beisteuern.

Der von uns zu verantwortende Quellcode in der Datei **Person.cs** enthält nur noch Deklarationen zu den drei in die Datenbindung und Validierung einbezogenen partiellen Eigenschaften. Wie schon im Abschnitt 12.4.1 berichtet, empfiehlt das Entwicklerteam des Community Toolkits seit der Version 8.4, partielle Eigenschaften statt der vorher üblichen privaten backing fields zu verwenden.

Für die drei Eigenschaften `FirstName`, `LastName` und `Age` veranlassen Attribute eine Verarbeitung durch das Community Toolkit:

- Dass beobachtbare, zur Datenbindung geeignete Eigenschaften entstehen sollen, legt das Attribut **ObservableProperty** fest.
- Mit dem Attribut **NotifyDataErrorInfo** ordnen wir die Validierung an.

---

[1] https://learn.microsoft.com/en-us/dotnet/api/microsoft.toolkit.mvvm.componentmodel.observablevalidator

- Es folgen Attribute zur Deklaration der Validierungskriterien nebst Fehlermeldungstexten:
  - Mit **Required** verbieten wir leere oder nur Leerzeichen enthaltende Eingaben zu den Namenseigenschaften.
  - Das **RegularExpression**-Attribut erlaubt in den Namen beliebige Unicode-Buchstaben (`\p{L}`), das Leerzeichen (`\s`) sowie das Apostroph und den Bindestrich.[1]
  - Mit **Range** legen wir den zulässigen Wertebereich für das Alter fest.

Es ist nicht einfach, mit Hilfe der Klasse **ObservableValidator** ungültige Werte abzuweisen und damit aus dem (Ansichts)modell fernzuhalten. In unserem Beispiel verbleiben reklamierte Werte in der `Personen`-Kollektion, wenn der Benutzer den Fehler nicht behebt. Das Fehlen einer einfachen Option zum Abweisen ungültiger Werte resultiert bei der Klasse **ObservableValidator** aus der Designentscheidung, die Validierung von Werten normalerweise *nach* dem Speichern vorzunehmen.

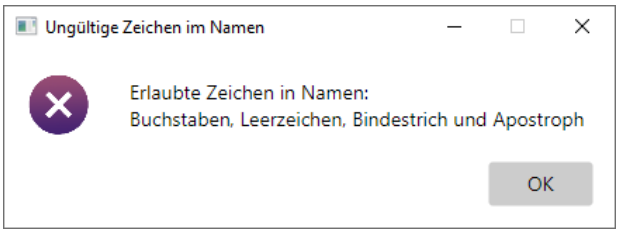
Diese Validierungsstrategie passt zu Ansichtsmodellklassen, die der Ansicht eigene Eigenschaften bzw. Stellvertreter von Modellobjekten anbieten und sicherstellen, dass nur erfolgreich validierte Benutzereingaben ins Modell gelangen. Wir werden im Abschnitt 12.4.4 eine so arbeitende Version des Personenverwaltungsprogramms vorstellen.

Von den **ObservableValidator**-Methoden verweigert nur **TrySetProperty()**, die Annahme ungültiger, an der Validierung gescheiterter Werte. Sie kooperiert aber nicht mit dem im Abschnitt 9.7 beschriebenen Verfahren zur Anzeige von Validierungs-Fehlermeldungen. Die folgende Lösung zur Validierung von Nachnamen verwendet **TrySetProperty()** und

```
string? lastName;
[Required]
[RegularExpression(@"^[\p{L}\s'-]+$")]
public string? LastName {
  get => lastName;
  set {
    if (!TrySetProperty(ref lastName, value, out _))
        ShowErrorMessage();
  }
}

static async void ShowErrorMessage() {
  var box = MessageBoxManager.GetMessageBoxStandard("Ungültige Zeichen im Namen",
    "Erlaubte Zeichen in Namen:\nBuchstaben, Leerzeichen, Bindestrich und Apostroph",
    ButtonEnum.Ok, Icon.Error);
  await box.ShowAsync();
}
```

reagiert auf ungültige Zeichen mit einer Dialogbox:



Weil Avalonia 12 kein Analogon zur WPF-Klasse **MessageBox** bietet, verwenden wir erneut das NuGet-Paket **MessageBox.Avalonia** (siehe Abschnitt 4.9).

[1] Diese Lösung hat ChatGPT 5.5 beigesteuert.

12.4.3 Toolkit-Kommandoklasse RelayCommand

Zur Bindung von GUI-initiierten Kommandos an Methoden der Ansichtsmodellklasse verwenden wir auch nach dem Umstieg auf das Toolkit im Wesentlichen die im Abschnitt 12.3.3 beschriebene Technik:

- Eine Klasse namens `RelayCommand` implementiert die Schnittstelle **ICommand** und enthält Konstruktoren mit Parametern für Delegatenobjekte vom Typ **Action** bzw. **Func<bool>** zur Verwendung in den **ICommand**-Methoden **Execute()** bzw. **CanExecute()**.
- Die Ansichtsmodellklasse ...
  - definiert für die Kommandos zur Aufnahme einer neuen Person in das **ObservableCollection<Person>** - Objekt bzw. zum Löschen einer Person aus dieser Kollektion jeweils eine **Action**- und eine **Func<bool>** - Methode,
  - deklariert die Eigenschaften `AddPersonCmd` und `DeletePersonCmd` vom Typ der aus **ICommand** abgeleiteten Schnittstelle **IRelayCommand** (siehe unten),

    ```
    public IRelayCommand AddPersonCmd { get; }
    public IRelayCommand DeletePersonCmd { get; }
    ```

  - weist den Kommandoeigenschaften im Konstruktor ein auf eigenen Methoden basierendes `RelayCommand`-Objekt zu:

    ```
    public MainViewModel() {
      AddPersonCmd = new RelayCommand(AddPerson, CanAddPerson);
      DeletePersonCmd = new RelayCommand(DeletePerson, CanDeletePerson);
       . . .
    }
    ```

Wir verzichten aber auf eine *eigene* `RelayCommand`-Klasse und verwenden die gleichnamige Klasse aus dem MVVM Community Toolkit.[1] Dazu ist im Quellcode der Ansichtsmodellklasse `MainViewModel` die **using**-Direktive

```
using MVVM.Commands;
```

durch

```
using CommunityToolkit.Mvvm.Input;
```

zu ersetzen, was mit Hilfe der Entwicklungsumgebung bequem per Schnellaktion geschehen kann.

Die Toolkit-Klasse **RelayCommand** implementiert die von **ICommand** abgeleitete Schnittstelle **IRelayCommand**, die wir als Datentyp für die Eigenschaften `AddPersonCmd` und `DeletePersonCmd` verwenden:

```
public partial class MainViewModel : ObservableObject {
  public ObservableCollection<Person>? Personen { get; }
  PropertyChangedEventHandler pceh;
  public IRelayCommand AddPersonCmd { get; }
  public IRelayCommand DeletePersonCmd { get; }

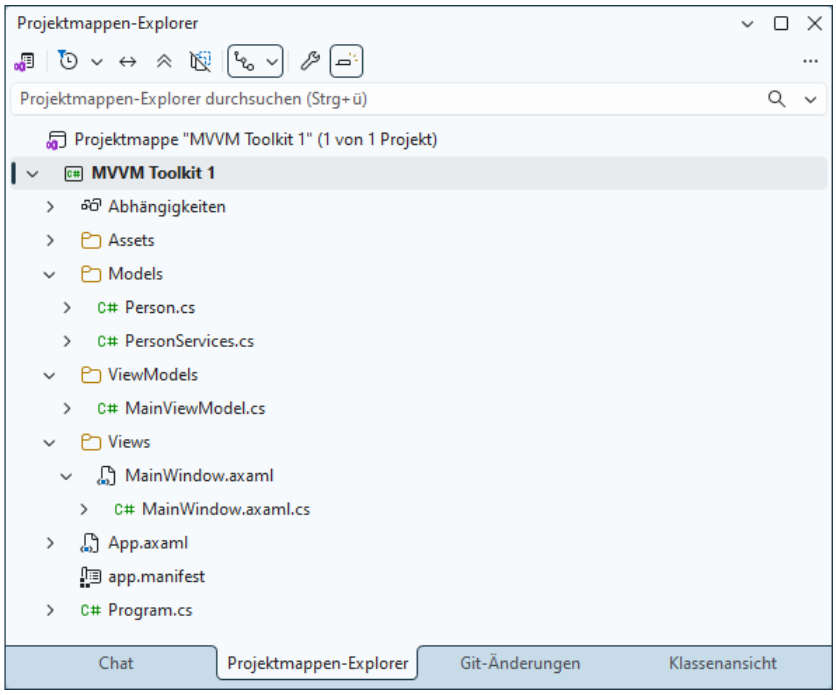
  [ObservableProperty]
  Person? selectedItem;

  public MainViewModel() {
    AddPersonCmd = new RelayCommand(AddPerson, CanAddPerson);
    DeletePersonCmd = new RelayCommand(DeletePerson, CanDeletePerson);
    Personen = PersonServices.GetPeople();
    SelectedItem = Personen?.Count > 0 ? Personen[0] : null;
    pceh = (sender, args) => AddPersonCmd.NotifyCanExecuteChanged();
  }
 . . .
}
```

[1] https://learn.microsoft.com/en-us/dotnet/communitytoolkit/mvvm/relaycommand

Von ihrer Basisschnittstelle **ICommand** unterscheidet sich die Schnittstelle **IRelayCommand** durch die zusätzliche Methode **NotifyCanExecuteChanged()**, die gleich im Einsatz zu sehen ist.

Den Projektordner `Commands` löschen wir komplett, sodass im **Projektmappen-Explorer** nur noch die Ordner **Models**, **ViewModels** und **Views** vorhanden sind:



Statt der Methode `RaiseCanExecuteChanged()` aus unserer eigenen `RelayCommand`-Klasse rufen wir im **PropertyChangedEventHandler**, durch den neue `Person`-Instanzen bei einer Wertänderung die Neubewertung der Ausführbarkeit des `AddPersonCmd`-Kommandos veranlassen, die **IRelayCommand**-Methode **NotifyCanExecuteChanged()** auf:

```
pceh = (sender, args) => AddPersonCmd.NotifyCanExecuteChanged();
```

Weil wir die Eigenschaft `SelectedItem` als **partial** definiert und mit dem Attribut **ObservableProperty** dekoriert haben, müssen wir dem Toolkit das Implementieren des **set**-Blocks überlassen und können hier keine Methodenaufrufe zur Neubewertung von Kommandoverfügbarkeiten unterbringen. Wir benötigen daher eine alternative Technik, um für das folgende Verhalten des Programms zu sorgen:

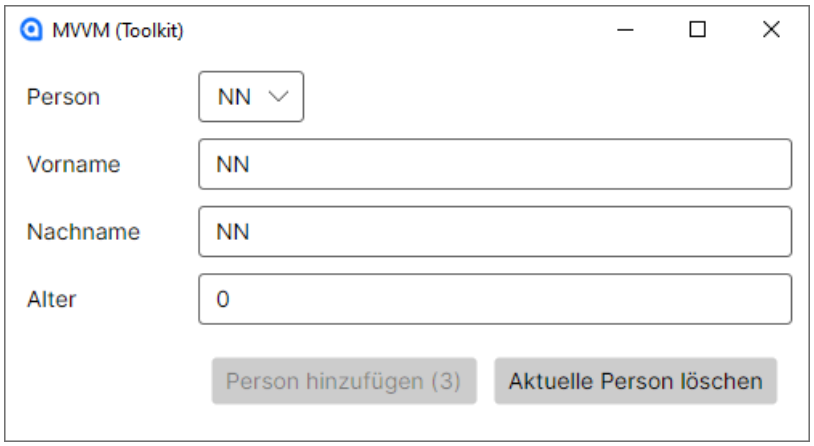
- Nach der Aufnahme einer neuen Person ist das Aufnahmekommando gesperrt bis zur kompletten Initialisierung der Person. In dieser Phase soll der zugehörige Schalter als funktionslos dargestellt sein.
- Nach dem Löschen der letzten Person verliert das zugehörige Kommando die Einsatzfähigkeit, was der Schalter anzeigen soll.

Wir erweitern die `MainViewModel`-Definition um eine partielle Definition der vom Toolkit-Codegerator erstellten und im **set**-Zugriff der Eigenschaft `SelectedItem` aufgerufenen Methode `OnSelectedItemChanged()` (vgl. Abschnitt 12.4.1):

```
partial void OnSelectedItemChanged(Person? value) {
    AddPersonCmd.NotifyCanExecuteChanged();
    DeletePersonCmd.NotifyCanExecuteChanged();
}
```

Der **partial**-Modifikator signalisiert er dem Compiler, dass der Quellcode auf zwei oder mehr Dateien verteilt ist.

Nun verhält sich das mit Hilfe des Toolkits vereinfachte Programm korrekt:



12.4.4 Verbesserte Bedienung und Validierung

Das Personenverwaltungsbeispiel hat seit Kapitel 9 zur Demonstration etlicher Funktionen gedient:

- Datenbindung
- Änderungsbenachrichtigung
- Validierung
- Kommandobindung
- MVVM (mit den Klassen **ObservableObject**, **ObservableValidator**, **RelayCommnand**)

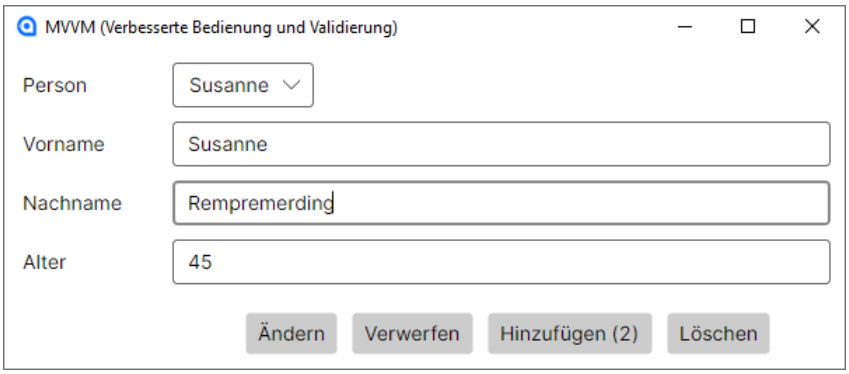
Mittlerweile gibt es einige Kritikpunkte:

- In den Textfeldern vorgenommene Änderungen wirken sich ohne weitere Bestätigung durch den Benutzer auf das Modell aus.
- Insbesondere landen auch irreguläre und von der Validierung beanstandete Änderungen im Modell, weil die Validierung erst nach dem Speichern erfolgt.

Stattdessen sollten ...

- Benutzer die an einem vorhandenen Eintrag vorgenommenen Änderungen bis zu einer expliziten Bestätigung verwerfen können,
- nur korrekte Eintragungen in Modellobjekte gelangen.

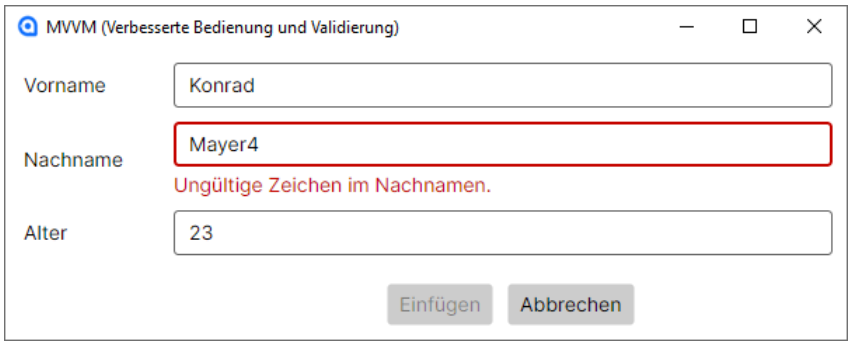
Das leistet die folgende Variante des Programms:



Hier können die Benutzer an einem vorhandenen Fall vorgenommenen Modifikationen ...

- per Klick auf **Ändern** explizit in das Modellobjekt übernehmen
- oder mit einem Klick auf **Verwerfen** rückgängig machen.

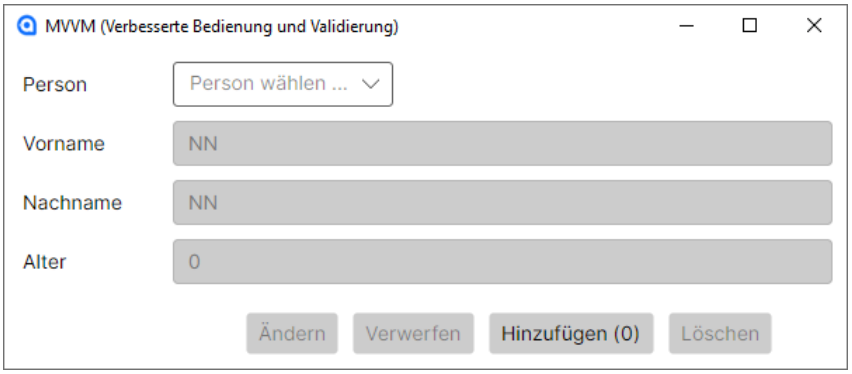
Die Ansicht der neuen Programmversion greift auf Eigenschaften des Ansichtsmodells statt auf Eigenschaften von Modellobjekten zu, sodass sich z. B. ein neuer Fall mit fehlerhaften Daten nicht in die Sammlung mit den Modellobjekten einfügen lässt:



Je nach Bearbeitungsmodus (Editieren versus Anlegen eines Falls) sind unterschiedliche Kommandos verfügbar und Befehlsschalter vorhanden. Die Ansichtsmodellklasse steuert in Abhängigkeit vom Bearbeitungsmodus und vom aktuellen Zustand (z. B. Vorliegen von fehlerhaften Eingaben) die nutzbaren Kommandos. Die Ansicht ...

- zeigt die Ausführbarkeit der Kommandos durch die Schalteroptik an,
- blendet nicht zum Bearbeitungsmodus passende Schalter aus
- und deaktiviert funktionslose Texteingabefelder.

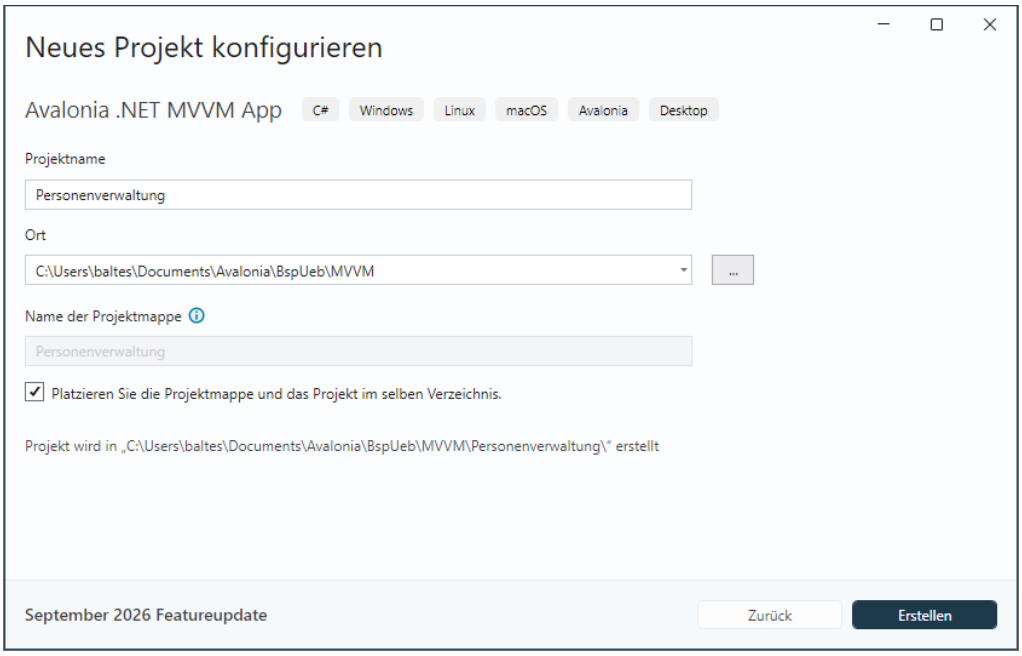
Hat der Benutzer z. B. sämtliche Fälle gelöscht, dann sind alle Bedienelemente gesperrt mit Ausnahme des Schalters zum Hinzufügen von Fällen:



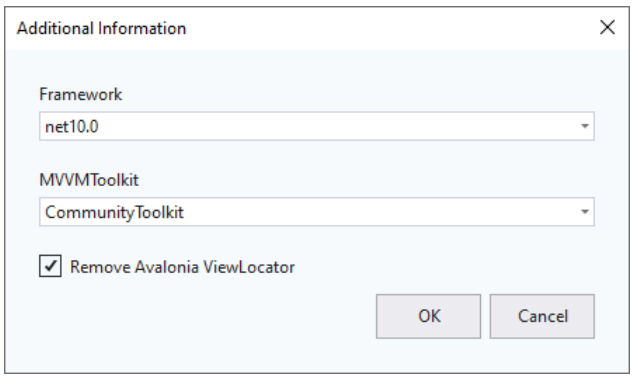
12.4.4.1 Neustart

Nachdem wir das Personenverwaltungsprogramm mehrfach kopiert und modifiziert haben, ist ein Neustart zur Verbesserung der Übersicht sinnvoll.[1] Wir starten ein neues Projekt mit der Vorlage **Avalonia .NET MVVM App**

[1] Ein Visual Studio - Projekt mit der Toolkit-Variante des Personenverwaltungsprogramms ist hier zu finden: **...\BspUeb\MVVM\Toolkit\Verbesserte Bedienung und Validierung**



für .NET 10.0 unter Verwendung des **Community-Toolkit**s ohne Bedarf für den **ViewLocator**:



Im Modell verwenden wie diesmal eine reine Datenklasse:

```
using System;

namespace Personenverwaltung.Models;

public class Person(string firstName, string lastName, int age) {
    public Guid Id { get; } = Guid.NewGuid();
    public string FirstName { get; set; } = firstName;
    public string LastName { get; set; } = lastName;
    public int Age { get; set; } = age;
}
```

Weil z. B. eine Person namens Peter Müller mit 37 Jahren in der Sammlung öfter auftauchen könnte, erweitern wir die Klasse `Person` um eine eindeutige Identifikation per **Guid**-Instanz.[1]

Die zur Simulation eines Datenbestands dienende Klasse `PersonServices` erfährt keine Änderung außer der Nutzung des `Person`-Konstruktors:

[1] https://learn.microsoft.com/en-us/dotnet/api/system.guid

```
using System.Collections.ObjectModel;

namespace Personenverwaltung.Models;

public class PersonServices {
    static readonly ObservableCollection<Person> people = [];

    static PersonServices() {
        people.Add(new Person("Susanne", "Brgl", 45));
        people.Add(new Person("Randolf", "Schmitt",34));
    }

    public static ObservableCollection<Person> GetPeople() {
        return people;
    }
}
```

12.4.4.2 Validierung im ViewModel

Die aus der Vorlage stammende Ansichtsmodelklasse `MainViewModel` benötigt zur Realisation der zu Beginn des Abschnitts 12.4.4 formulierten Ziele einigen Aufwand. Um die nicht-abweisende Validierung ohne das Risiko fehlerhafter Modelldaten nutzen zu können, greifen die Benutzer nicht mehr direkt auf Modellobjekte zu, sondern auf Eigenschaften des Ansichtsmodellobjekts. Im Konstruktor erhalten diese Eigenschaften die Werte des aktuellen Falls, auf den die Ansichtsmodelleigenschaft `SelectedItem` zeigt:

```
public partial class MainViewModel : ObservableValidator {
    public ObservableCollection<Person>? Personen { get; }

    [ObservableProperty]
    public partial Person? SelectedItem { get; set; }

    public MainViewModel() {
        Personen = PersonServices.GetPeople();
        SelectedItem = Personen?.Count > 0 ? Personen[0] : null;
        FirstName = SelectedItem?.FirstName ?? "NN";
        LastName = SelectedItem?.LastName ?? "NN";
        Age = SelectedItem?.Age ?? 0;
        SwitchAddPerson(false);
    }

    . . .

}
```

Außerdem ruft der Konstruktor die Methode `SwitchAddPerson()` zum Wechseln zwischen den Bearbeitungsmodi (Editieren versus Ergänzen einer Person) auf (siehe Abschnitt 12.4.4.4).

Man könnte ein Mitgliedsobjekt der Klasse `Person` in das Ansichtsmodell aufnehmen. Weil nur *ein* Stellvertreter für Modellobjekte erforderlich ist, verwenden wir der Einfachheit halber das Ansichtsmodellobjekt in dieser Funktion.

Die Klasse `MainViewModel` stammt von `ObservableValidator` ab, und anstelle von Modellobjekteigenschaften sind nun Ansichtsmodelleigenschaften beobacht- und validierbar, z. B.:

```
[ObservableProperty]
[NotifyDataErrorInfo]
[NotifyCanExecuteChangedFor(nameof(ChangePersonCommand))]
[NotifyCanExecuteChangedFor(nameof(ResetPersonCommand))]
[NotifyCanExecuteChangedFor(nameof(SavePersonCommand))]
[Required(ErrorMessage = "Tragen Sie bitte einen Nachnamen ein.")]
[RegularExpression(@"^[\p{L}\s'-]+$",
                   ErrorMessage = "Ungültige Zeichen im Nachnamen.")]
public partial string? FirstName { get; set; }
```

Das Attribut **NotifyCanExecuteChangedFor** veranlasst analog zur **RelayCommand**-Methode **NotifyCanExecuteChanged()** für das im Parameter angegebene Kommando eine Prüfung der Ausführbarkeit. Im Beispiel veranlasst jede Änderung der Eigenschaft `FirstName` diese Prüfung für die folgenden Kommandos:

- `ChangePersonCommand` zur Übernahme neuer Werte für den aktuellen Fall in das Modell
- `ResetPersonCommand` zum Verwerfen neuer Werte für den aktuellen Fall
- `SavePersonCommand` zur Übertragung eines neuen Falls in das Modell

Mit der Erstellung dieser Kommandoobjekte aus Ansichtsmodellmethoden (z. B. `ChangePerson()`) aufgrund des Methodenattributs **RelayCommand** beschäftigt sich der nächste Abschnitt.

12.4.4.3 Toolkit-Attribut RelayCommand

Für die aktuelle Programmversion nutzen wir noch stärker als bisher die Attribute bzw. Code-Generatoren im Community Toolkit. Das Ansichtsmodell besitzt sechs **RelayCommand**-Mitgliedsobjekte, doch ist vom Konstruktur dieser Klasse nichts zu sehen. Wir heften **RelayCommand**-*Attribute* an Methoden, die als **Action**-Delegat für ein Kommando geeignet sind (vgl. Abschnitte 12.3.3 und 12.4.3). Im folgenden Beispiel ist die Methode `ChangePerson()` zu sehen, aus der das `ChangePersonCommand` entsteht:

```
[RelayCommand(CanExecute = nameof(CanChangePerson))]
public void ChangePerson() {
    if (SelectedItem == null)
        return;
    SelectedItem.FirstName = FirstName;
    SelectedItem.LastName = LastName;
    SelectedItem.Age = Age;
    ChangePersonCommand.NotifyCanExecuteChanged();
}

public bool CanChangePerson() {
    if (SelectedItem == null)
        return false;
    if (FirstName == SelectedItem.FirstName && LastName == SelectedItem.LastName &&
        Age == SelectedItem.Age)
        return false;
    if (HasErrors || FirstName == "NN" || LastName == "NN" || Age == 0)
        return false;
    return true;
}
```

Die Methode überträgt die Ansichtsmodell-Eigenschaften `FirstName`, `LastName` und `Age` auf das von `SelectedItem` referenzierte `Person`-Objekt. Anschließend veranlasst ein **NotifyCanExecuteChanged()** – Aufruf die Prüfung der Ausführbarkeit und infolgedessen die vorübergehende Deaktivierung des Kommandos. Es wird nach einer zulässigen Änderung einer beobachteten Eigenschaft wieder verfügbar.

Der **CanExecute**-Parameter des **RelayCommand**-Attributs zu `ChangePerson()` erhält die als **Func<bool>** - Delegat geeignete Methode `CanChangePerson()` zur Überprüfung der Ausführbarkeit des Kommandos. Das Kommando ist *nicht* verfügbar, wenn ...

- im Vergleich zu den Eigenschaften der aktuellen Person keine Änderung stattgefunden hat,
- ein Validierungsfehler vorliegt,
- oder ein Feld den Initialwert für neue Personen besitzt.

Das auf der Methode `ResetPerson()` basierende Kommando setzt die geänderten Ansichtsmodelleigenschaften `FirstName`, `LastName` und `Age` auf die Daten das aktuellen Falls zurück:

```
[RelayCommand(CanExecute = nameof(CanResetPerson))]
public void ResetPerson() {
    if (SelectedItem == null)
        return;
    FirstName = SelectedItem.FirstName;
    LastName = SelectedItem.LastName;
    Age = SelectedItem.Age;
    ChangePersonCommand.NotifyCanExecuteChanged();
}
```

Es ist ausführbar, wenn die Methode `CanResetPerson()` Änderungen feststellt:

```
public bool CanResetPerson() {
    if (SelectedItem == null)
        return false;
    if (FirstName == SelectedItem.FirstName && LastName == SelectedItem.LastName &&
        Age == SelectedItem.Age)
        return false;
    return true;
}
```

Das auf der Methode `DeletePerson()` basierende Kommando entfernt den von `SelectedItem` referenzierten Fall aus der `Personen`-Kollektion:

```
[RelayCommand(CanExecute = nameof(CanDeletePerson))]
public void DeletePerson() {
    if (Personen == null || SelectedItem == null)
        return;
    Personen.Remove(SelectedItem);
    SelectedItem = Personen.LastOrDefault();
}
```

Es ist verfügbar, wenn die Eigenschaft `SelectedItem` von **null** verschieden ist:

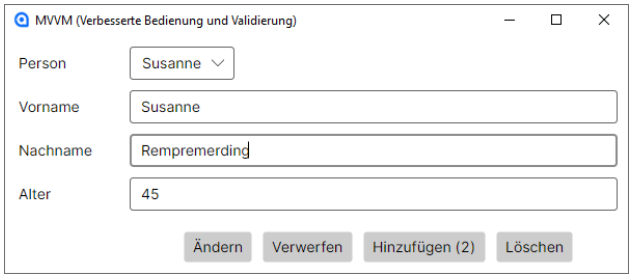
```
public bool CanDeletePerson() {
    if (SelectedItem == null)
        return false;
    else
        return true;
}
```

Im Abschnitt 12.4.4.4 stellen wir die Kommandos zur Aufnahme eines neuen Falls vor.

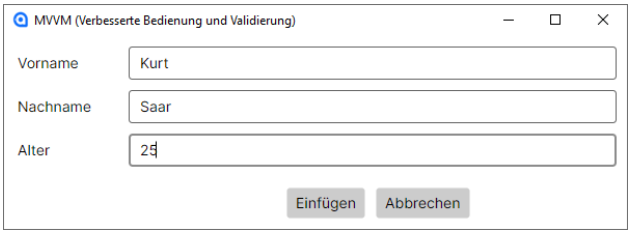
12.4.4.4 Bearbeitungsmodi

Im Programm soll nicht nur die Ausführbarkeit von Kommandos anhand der Schalterdarstellung erkennbar sein, sondern es sollen zwei Bearbeitungsmodi mit unterschiedlichen Bedienelementausstattungen verfügbar sein:

Editieren eines Falls



Einfügen eines Falls



Im Editiermodus ...

- kann man per **ComboBox** zwischen den im Modell (in der `Personen`-Kollektion) vorhandenen Fällen wählen,
- sind die Texteingabefelder nutzbar, wenn die Ansichtsmodelleigenschaft `SelectedItem` auf eine `Person`-Instanz zeigt,
- sind Schalter vorhanden zum ...
  - Ändern der Werte des angezeigten Falls,
  - Verwerfen von vorgenommenen Änderungen,
  - Löschen der angezeigten Person,
  - Einfügen einer neuen Person.

Nach einen Klick auf den Schalter **Hinzufügen** erlaubt das Programm das Einfügen eines neuen Falls in das Modell (in die `Personen`-Kollektion) durch eine angepasste Bedienoberfläche:

- Die **ComboBox** verschwindet
- ebenso wie die Schalter zum ...
  - Ändern der Werte des angezeigten Falls,
  - Verwerfen von vorgenommenen Änderungen,
  - Löschen des angezeigten Falls.
- Die Texteingabefelder sind verfügbar,
- und es erscheinen Schalter zum ...
  - Einfügen eines neuen Falls in das Modell,
  - Abbrechen des Einfügens.

Über beobachtbare **bool**-Eigenschaften schreibt das Ansichtsmodell der Ansicht den Bearbeitungsmodus und die Verfügbarkeit der Texteingabefelder vor:

```
[ObservableProperty]
public partial bool EditPersonMode { get; set; }
[ObservableProperty]
public partial bool FieldsEnabled { get; set; }
```

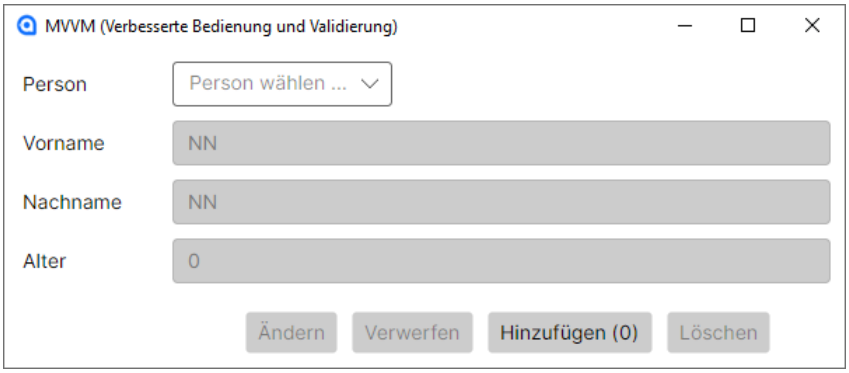
Die Ansicht bindet die **IsVisible**-Attribute von Bedienelementen an die `EditPersonMode`-Eigenschaft des Ansichtsmodells, sodass z. B. der Schalter zum Ändern eines vorhandenen Falls nur im Editiermodus

```
<Button IsVisible="{Binding EditPersonMode}" Command="{Binding ChangePersonCommand}"
        Content="Ändern" Margin="10,0,0,0" />
```

und der Schalter zum Einfügen eines neuen Falls nur im Ergänzungsmodus verfügbar ist, den die Negation der Eigenschaft `EditPersonMode` signalisiert:

```
<Button IsVisible="{Binding !EditPersonMode}" Command="{Binding SavePersonCommand}"
        Content="Einfügen" Margin="10,0,0,0" />
```

Wenn die Eigenschaft `SelectedItem` (z. B. aufgrund einer leeren `Personen`-Kollektion) den Wert **null** besitzt, dann erhält die Eigenschaft `FieldsEnabled` den Wert **false**. In diesem Fall sperrt die Ansicht die Texteingabefelder:



Das Ansichtsmodell benutzt die private Methode `SwitchAddPerson()`, um den Einfügemodus ein- bzw. auszuschalten:

```
void SwitchAddPerson(bool on) {
    if (on) {
        EditPersonMode = false;
        FieldsEnabled = true;
    } else {
        EditPersonMode = true;
        FieldsEnabled = SelectedItem != null;
    }
}
```

Das stets ausführbare Kommando für den Wechsel in den Einfügemodus basiert auf der Methode `AddPerson()`:

```
[RelayCommand]
public void AddPerson() {
    if (Personen == null)
        return;
    FirstName = "NN";
    LastName = "NN";
    Age = 0;
    SwitchAddPerson(true);
    SavePersonCommand.NotifyCanExecuteChanged();
}
```

Es schreibt Initialwerte für den neuen Fall in die `Person`-Eigenschaften des Ansichtsmodells und aktiviert mit dem folgenden `SwitchAddPerson()` - Aufruft den Einfügemodus:

```
SwitchAddPerson(true);
```

Man könnte noch eine Abfrage einbauen, um den Verlust von ungesicherten Änderungen beim Wechsel in den Einfügemodus zu verhindern.

Im Einfügemodus erstellt das auf der Methode `SavePerson()` basierende Kommando

```
[RelayCommand(CanExecute = nameof(CanSavePerson))]
public void SavePerson() {
    if (Personen == null)
        return;
    Person person = new(FirstName, LastName,Age);
    Personen.Add(person);
    SelectedItem = person;
    SwitchAddPerson(false);
}
```

eine neue `Person`-Instanz aufgrund der Einträge in den Texteingabefeldern, sichert den neuen Fall ins Modell (in die `Personen`-Kollektion), weist ihn der Eigenschaft `SelectedItem` zu und kehrt mit dem folgenden `SwitchAddPerson()` - Aufruf zum Editiermodus zurück:

```
SwitchAddPerson(false);
```

Das Kommando ist verfügbar, wenn `CanSavePerson()` eine fehlerfreie Fallinitialisierung meldet:

```
public bool CanSavePerson() {
    if (Personen == null || HasErrors || FirstName == "NN" || LastName == "NN"
                         || Age == 0)
        return false;
    else
        return true;
}
```

Auf der Methode `CancelAdd()` basiert das Kommando, um die Aufnahme eines neuen Falls abzubrechen:

```
[RelayCommand]
public void CancelAdd() {
    FirstName = SelectedItem?.FirstName ?? "NN";
    LastName = SelectedItem?.LastName ?? "NN";
    Age = SelectedItem?.Age ?? 0;
    SwitchAddPerson(false);
}
```

Die Methode schreibt die Daten des aktuellen Falls in die `Person`-Eigenschaften des Ansichtsmodells und endet mit der Rückkehr zum Editiermodus.

12.4.4.5 Umsetzung der Bearbeitungsmodi durch die Ansicht

Wie bisher besitzt die per XAML deklarierte Ansicht eine **ComboBox**

```
<StackPanel Margin="10" >
  <Grid ColumnDefinitions="110,*">
    <Label IsVisible="{Binding EditPersonMode}" Grid.Column="0" Content="Person"
           Margin="0,0,0,10" VerticalContentAlignment="Center" />
    <ComboBox IsVisible="{Binding EditPersonMode}" Grid.Column="1"
              ItemsSource="{Binding Personen}"
              SelectedItem="{Binding SelectedItem, Mode=TwoWay}"
              DisplayMemberBinding="{Binding FirstName}"
              PlaceholderText="Person wählen ..." Margin="0,0,0,10"/>
  </Grid>
    . . .
</StackPanel>
```

mit ...

- der `Personen`-Kollektion als **ItemsSource**
- und dem Attribut **SelectedItem**, das bidirektional an die gleichnamige Ansichtsmodelleigenschaft gefunden ist:

  ```
  SelectedItem="{Binding SelectedItem, Mode=TwoWay}"
  ```

Die **TextBox**-Steuerelemente sind an die Eigenschaften `FirstName`, `LastName` und `Age` des Ansichtsmodells gebunden:

```
<StackPanel Margin="10" >
  <Grid ColumnDefinitions="110,*">
     . . .
  </Grid>

  <Grid IsEnabled="{Binding FieldsEnabled}" ColumnDefinitions="110,*"
        RowDefinitions="Auto,Auto,Auto" >
   <Label Grid.Column="0" Grid.Row="0" Content="Vorname" Margin="0,0,0,10"
          VerticalContentAlignment="Center" />
    <TextBox Name="firstName" Grid.Column="1" Grid.Row="0"
             Text="{Binding FirstName, UpdateSourceTrigger=PropertyChanged}"
             Margin="0,0,0,10" />
    <Label Grid.Column="0" Grid.Row="1" Content="Nachname" Margin="0,0,0,10"
           VerticalContentAlignment="Center" />
    <TextBox Grid.Column="1" Grid.Row="1"
             Text="{Binding LastName, UpdateSourceTrigger=PropertyChanged}"
             Margin="0,0,0,10" />
    <Label Grid.Column="0" Grid.Row="2" Content="Alter" Margin="0,0,0,0"
           VerticalContentAlignment="Center" />
    <TextBox Grid.Column="1" Grid.Row="2" Margin="0,0,0,0"
             Text="{Binding Age, UpdateSourceTrigger=PropertyChanged}" />
  </Grid>

  <Grid ColumnDefinitions="110,*">
   . . .
  </Grid>
</StackPanel>
```

Die Eintragungen der Benutzer wirken sich also *nicht* direkt auf Modellobjekte aus, z. B.:

```
Text="{Binding FirstName, UpdateSourceTrigger=PropertyChanged}"
```

Bei einer Änderung der Ansichtsmodelleigenschaft `SelectedItem` sorgt die vom Toolkit automatisch aufgerufene Methode `OnSelectedItemChanged()` für eine Aktualisierung der Eigenschaften `FirstName`, `LastName` und `Age`:

```
partial void OnSelectedItemChanged(Person? value) {
    FirstName = value?.FirstName ?? "NN";
    LastName = value?.LastName ?? "NN";
    Age = value?.Age ?? 0;
    if (value == null)
        FieldsEnabled = false;
    else
        FieldsEnabled = true;
    ChangePersonCommand.NotifyCanExecuteChanged();
    ResetPersonCommand.NotifyCanExecuteChanged();
    DeletePersonCommand.NotifyCanExecuteChanged();
}
```

Wie im Segment mit der **ComboBox** und im Segment mit den Texteingabefeldern beachtet die Ansicht auch im Segment mit den **Button**-Elementen

```
<StackPanel Margin="10" >
  <Grid ColumnDefinitions="110,*">
     . . .
  </Grid>
  <Grid IsEnabled="{Binding FieldsEnabled}" ColumnDefinitions="110,*"
        RowDefinitions="Auto,Auto,Auto" >
    . . .
  </Grid>

  <Grid ColumnDefinitions="110,*">
    <StackPanel Grid.Column="1" HorizontalAlignment="Center" Orientation="Horizontal"
                Margin="0,20,0,0">
      <Button IsVisible="{Binding EditPersonMode}"
              Command="{Binding ChangePersonCommand}" Content="Ändern"
              Margin="10,0,0,0" />
      <Button IsVisible="{Binding EditPersonMode}"
              Command="{Binding ResetPersonCommand}" Content="Verwerfen"
              Margin="10,0,0,0" />
      <Button IsVisible="{Binding EditPersonMode}"
              Command="{Binding AddPersonCommand}" Margin="10,0,0,0">
          <Button.Content>
            <StackPanel Orientation="Horizontal">
              <TextBlock Text="Hinzufügen (" />
              <TextBlock Text="{Binding Personen.Count}" />
              <TextBlock Text=")" />
            </StackPanel>
          </Button.Content>
      </Button>
      <Button IsVisible="{Binding EditPersonMode}"
              Command="{Binding DeletePersonCommand}"
              Content="Löschen" Margin="10,0,0,0" />
      <Button IsVisible="{Binding !EditPersonMode}"
              Command="{Binding SavePersonCommand}"
              Content="Einfügen" Margin="10,0,0,0" />
      <Button IsVisible="{Binding !EditPersonMode }"
              Command="{Binding CancelAddCommand}"
              Content="Abbrechen" Margin="10,0,0,0" />
    </StackPanel>
  </Grid>
</StackPanel>
```

über Bindungen für die Steuerelementeigenschaften **IsEnabled** bzw. **IsVisible** die Anweisungen des Ansichtsmodells zum Bearbeitungsmodus:

- Durch die Bindung an die **IsEnabled**-Eigenschaft des **Grid**-Containers mit den Texteingabefeldern wirkt sich die Ansichtsmodelleigenschaft `FieldsEnabled` auf alle **TextBox**-Elemente aus:

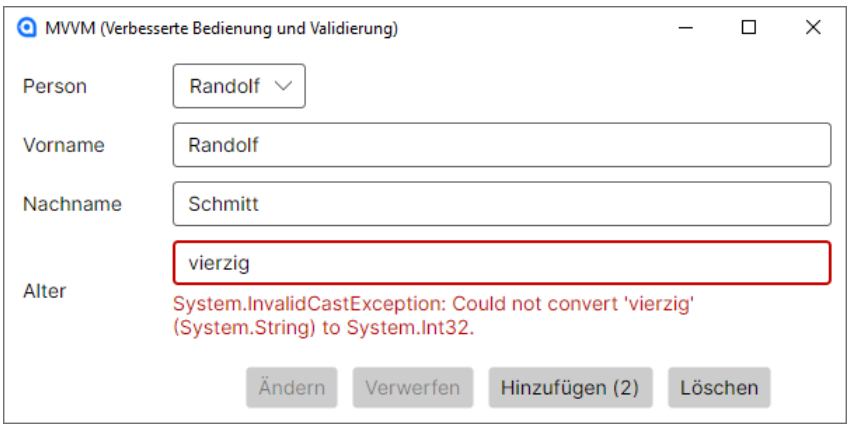
```
<Grid IsEnabled="{Binding FieldsEnabled}" ColumnDefinitions="110,*"
      RowDefinitions="Auto,Auto,Auto" >
```

- Die **IsVisible**-Attribute der **ComboBox** und der **TextBox**-Elemente sind an die Ansichtsmodelleigenschaft `EditPersonMode` gebunden, z. B.:

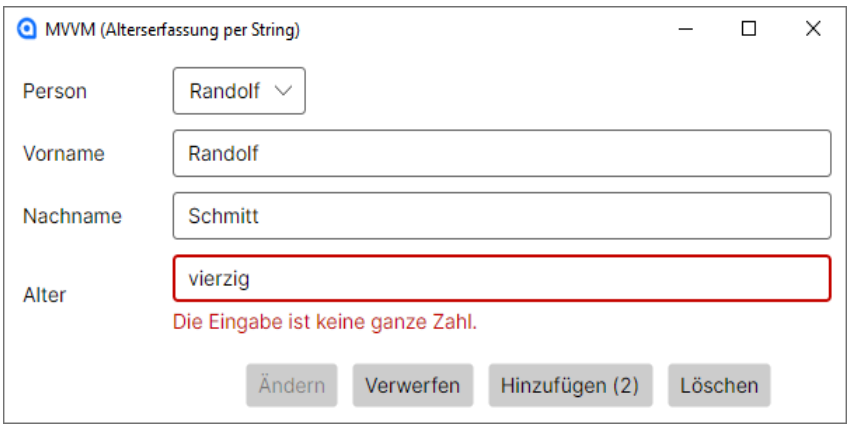
```
<ComboBox IsVisible="{Binding EditPersonMode}" Grid.Column="1"
          ItemsSource="{Binding Personen}"  . . . />
 . . .
<Button IsVisible="{Binding EditPersonMode}"
        Command="{Binding ChangePersonCommand}" . . . />
```

12.4.4.6 Alterserfassung per String

Trägt ein Benutzer in das Feld zur Alterserfassung eine nicht als **int**-Wert interpretierbare Zeichenfolge ein, dann kommt es zu einem Ausnahmefehler und zu einer wenig hilfreichen Fehlerinformation in der Ansicht, z. B.:



Um eine gut verständliche Meldung zu erreichen,



ist einiger Aufwand erforderlich.

Wir präsentieren den Benutzern statt der **int**-Eigenschaft `Age` die **String**-Eigenschaft `Sage` und vermeiden somit einen Ausnahmefehler. In der Ansichtsmodellklasse `MainViewModel` ist neben der öffentlichen **String**-Eigenschaft `Sage` zum Datenaustausch mit dem Modell die private **int**-Eigenschaft `Age` vorhanden:

- Beim Lesen eines Falls entsteht `Sage` aus `Age` per **ToString()**.
- Bei einer erfolgreichen Konvertierung und Validierung entsteht `Age` aus `Sage` per **TryParse()**.

Die Lesekonvertierung findet im Konstruktor

```
Age = SelectedItem?.Age ?? 0;
Sage = Age.ToString();
```

sowie in der Methode **OnSelectedItemChanged()** statt:

```
Age = value?.Age ?? 0;
Sage = Age.ToString();
```

Die umgekehrte Konvertierung findet in der Methode `ValidateAge()` statt, die der interessanteste Bestandteile der verbesserten Ansichtsmodellklasse `MainViewModel` ist:[1]

[1] Ein Visual Studio - Projekt mit der Toolkit-Variante des Personenverwaltungsprogramms ist hier zu finden:
...\BspUeb\MVVM\Toolkit\Alterserfassung per String

```
public partial class MainViewModel : ObservableValidator {
    public ObservableCollection<Person>? Personen { get; }

    [ObservableProperty]
    public partial Person? SelectedItem { get; set; }

    int Age { get; set; }

    public MainViewModel() {
        Personen = PersonServices.GetPeople();
        SelectedItem = Personen?.Count > 0 ? Personen[0] : null;
        FirstName = SelectedItem?.FirstName ?? "NN";
        LastName = SelectedItem?.LastName ?? "NN";
        Age = SelectedItem?.Age ?? 0;
        Sage = Age.ToString();
        SwitchAddPerson(false);
    }

. . .

    [ObservableProperty]
    [NotifyDataErrorInfo]
    [NotifyCanExecuteChangedFor(nameof(ChangePersonCommand))]
    [NotifyCanExecuteChangedFor(nameof(ResetPersonCommand))]
    [NotifyCanExecuteChangedFor(nameof(SavePersonCommand))]
    [Required(ErrorMessage = "Tragen Sie bitte das Alter ein.")]
    [CustomValidation(typeof(MainViewModel), nameof(ValidateAge))]
    public partial string? Sage { get; set; }

    public static ValidationResult? ValidateAge(string? sage,
                                                ValidationContext context) {
        if (!int.TryParse(sage, out int age))
            return new ValidationResult("Die Eingabe ist keine ganze Zahl.");
        if (age < 0 || age > 120)
            return new ValidationResult("Zulässige Altersangaben: 0 bis 120");
        var viewModel = (MainViewModel)context.ObjectInstance;
        viewModel.Age = age;
        return ValidationResult.Success;
    }

. . .

    [RelayCommand(CanExecute = nameof(CanResetPerson))]
    public void ResetPerson() {
        if (SelectedItem == null)
            return;
        FirstName = SelectedItem.FirstName;
        LastName = SelectedItem.LastName;
        Sage = SelectedItem.Age.ToString();
        ChangePersonCommand.NotifyCanExecuteChanged();
    }
    public bool CanResetPerson() {
        if (SelectedItem == null)
            return false;
        if (FirstName == SelectedItem.FirstName &&
            LastName == SelectedItem.LastName && Sage == SelectedItem.Age.ToString())
            return false;
        return true;
    }
```

```
        [RelayCommand]
        public void AddPerson() {
            if (Personen == null)
                return;
            FirstName = "NN";
            LastName = "NN";
            Sage = "0";
            Age = 0;
            SwitchAddPerson(true);
            SavePersonCommand.NotifyCanExecuteChanged();
        }

        [RelayCommand]
        public void CancelAdd() {
            FirstName = SelectedItem?.FirstName ?? "NN";
            LastName = SelectedItem?.LastName ?? "NN";
            Sage = SelectedItem?.Age.ToString() ?? "0";
            SwitchAddPerson(false);
        }

    . . .

        partial void OnSelectedItemChanged(Person? value) {
            FirstName = value?.FirstName ?? "NN";
            LastName = value?.LastName ?? "NN";
            Age = value?.Age ?? 0;
            Sage = Age.ToString();
            if (value == null)
                FieldsEnabled = false;
            else
                FieldsEnabled = true;
            ChangePersonCommand.NotifyCanExecuteChanged();
            ResetPersonCommand.NotifyCanExecuteChanged();
            DeletePersonCommand.NotifyCanExecuteChanged();
        }

    . . .

    }
```

Wir ersetzen das bisherige **Range**-Validierungsattribut für die Eigenschaft `Age`

```
[Range(0, 120, ErrorMessage = "Zulässige Altersangaben: {1} bis {2}")]
public partial int Age { get; set; }
```

durch die Validierungsattribute **Required** und **CustomValidation** für die Eigenschaft `Sage`:

```
[Required(ErrorMessage = "Tragen Sie bitte das Alter ein.")]
[CustomValidation(typeof(MainViewModel), nameof(ValidateAge))]
public partial string? Sage { get; set; }
```

Als Parameter des Attributs **CustomValidation** sind anzugeben:

- Name der validierenden Klasse
  Im Beispiel erledigt das Ansichtsmodell diese Aufgabe.
- Name einer *öffentlichen* und *statischen* Validierungsmethode

Die Validierungsmethode erhält ebenfalls zwei Parameter:

- einen **String**-Parameter mit der zu beurteilenden Zeichenfolge
- einen Parameter vom Typ **ValidationContext** (Namensraum **System.ComponentModel.DataAnnotations**) mit der Eigenschaft **ObjectInstance**, die auf das zu überprüfende Objekt zeigt, also im Beispiel auf das Ansichtsklassenobjekt

Das Ergebnis einer gelungenen Konvertierung und Validierung weisen wir der privaten **int**-Eigenschaft `Age` zu:

```
viewModel.Age = age;
```

Diese Eigenschaft verwenden wir in Schreibzugriffen auf `Person`-Instanzen im Modell, z. B. in der Methode `ChangePerson()`:

```
[RelayCommand(CanExecute = nameof(CanChangePerson))]
public void ChangePerson() {
    if (SelectedItem == null)
        return;
    SelectedItem.FirstName = FirstName;
    SelectedItem.LastName = LastName;
    SelectedItem.Age = Age;
    ChangePersonCommand.NotifyCanExecuteChanged();
}
```

Vom Wechsel auf die Erfassung des Alters durch eine **String**-Eigenschaft sind noch weitere Methoden betroffen:

- `ResetPerson()` und `CanResetPerson()`
  Weil eine nicht-konvertierbare Altersangebe keinen Ausnahmefehler mehr verursacht, kann man sie nun ebenso verwerfen wie andere Eingaben.
- `AddPerson()`
- `CancelAdd()`

Schließlich ist in der XAML-Deklaration die Datenbindung der **TextBox** zur Alterserfassung zu ändern:

```
<TextBox Grid.Column="1" Grid.Row="2" Margin="0,0,0,0"
         Text="{Binding Sage, UpdateSourceTrigger=PropertyChanged}" />
```

12.4.5 ObservableRecipient

Das Erstellen eines zusätzlichen Fensters auf Initiative einer Ansichtsmodellklasse verstößt gegen die strikte MVVM-Trennungsregel: Weil eine Ansichtsmodellklasse keine Ansicht (Fensterklasse) kennen darf (weder die eigene noch eine andere), kommt das Erstellen eines Fensterklassenobjekts nicht in Frage. Das MVVM Community Toolkit bietet für diese und andere Kommunikationsbedürfnisse eine Nachrichtentechnik.[1] Dazu verwendet man für Ansichtsmodellklassen die Basisklasse **ObservableRecipient**, die von **ObservableObject** abstammt und die Schnittstelle **IMessenger** implementiert.

[1] https://learn.microsoft.com/en-us/dotnet/communitytoolkit/mvvm/observablerecipient

13 Weitere Steuerelemente

13.1 Menüoptionen

Nachdem wir die Kommandobindung studiert haben (siehe Abschnitte 12.3 und 12.4), ist die Verwendung der Menüoptionen in Avalonia (Haupt-, Kontext- und Flyout-Menü) leicht zu bewerkstelligen.

13.1.1 Hauptmenü

Für ein Fenster mit Hauptmenü ist der **DockPanel**-Layoutcontainer sehr gut geeignet (siehe Abschnitt 10.3). Er bietet die angefügte Eigenschaft **Dock**, und mit deren Wert **Top** lässt sich das zur Realisation des Hauptmenüs konzipierte Steuerelement **Menu** am oberen Fensterrand fixieren, z. B.:

```
<DockPanel>
    <Menu DockPanel.Dock="Top">
        <MenuItem Header="_Datei">
            <MenuItem Header="_Sichern" Command="{Binding SaveToFileCommand}" />
            <MenuItem Header="_Laden" Command="{Binding LoadFromFileCommand}" />
            <Separator/>
            <MenuItem Header="_Beenden" Command="{Binding ShutdownAppCommand}"/>
        </MenuItem>
            <MenuItem Header="_Hilfe">
            <MenuItem Header="_Version" Command="{Binding VersionCommand}" />
            <MenuItem Header="_Info" Command="{Binding InfoCommand}"/>
        </MenuItem>
    </Menu>

    <StackPanel Margin="10">
     . . .
    </StackPanel>
</DockPanel>
```

Ein **Menu**-Element schafft den Raum für **MenuItem**-Elemente, die per **Header**-Eigenschaft eine Beschriftung erhalten. Auf oberster Ebene in ein Menu eingefügte **MenuItem**-Elemente ...

- erscheinen nebeneinander
- und dienen in der Regel nicht zum Auslösen von Kommandos.

Sie enthalten untergeordnete **MenuItem**-Objekte, die ...

- senkrecht angeordnet unter dem Hauptmenü-Element erscheinen
- und in der Regel eine **Command**-Eigenschaft mit Bindung an ein Kommando der assoziierten Ansichtsmodellklasse besitzen.

Per Unterstrich vereinbart man im **Header** eine **Alt**-Zugriffstaste, um zu einem Menü-Item das Untermenü zu öffnen oder das verbundene Kommando anzustoßen.

Wie man die Items in einem Untermenü per **Separator** gruppiert, zeigt das `Datei` - **MenuItem** des Beispiels, das die seit Kapitel 9 sukzessive entwickelte Personenverwaltung erweitert:[1]

[1] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Menüoptionen\Hauptmenü



Durch die Kommandos zum Sichern und Laden der `Personen`-Liste erhält das Programm eine Dateibasierte Persistenzlösung, die auf neuen statischen Methoden der Klasse `PersonServices` beruht:[1]

```
public class PersonServices {
    static readonly ObservableCollection<Person> people = [];

    public static ObservableCollection<Person> GetPeople() {
        people.Add(new Person("Susanne", "Brgl", 45));
        people.Add(new Person("Randolf", "Schmitt", 34));
        return people;
    }

    static public readonly string JsonFileName = Path.Combine(
                        Environment.GetFolderPath(Environment.SpecialFolder.Personal),
                        "Personenverwaltung", "Personen.json");

    public static async Task SaveToFileAsync(IEnumerable<Person> itemsToSave) {
        Directory.CreateDirectory(Path.GetDirectoryName(JsonFileName)!);
        using var fs = File.Create(JsonFileName);
        await JsonSerializer.SerializeAsync(fs, itemsToSave);
    }

    public static async Task<IEnumerable<Person>?> LoadFromFileAsync() {
        try {
            using var fs = File.OpenRead(JsonFileName);
            return await JsonSerializer.DeserializeAsync<IEnumerable<Person>>(fs);
        } catch (Exception e) when
                (e is FileNotFoundException || e is DirectoryNotFoundException) {
            return null;
        }
    }
}
```

Zur Aufbewahrung der Kollektionselemente dient eine Datei im ***JSON***-Form (***JavaScript Object Notation***) mit dem Namen **Personen.json**.[2] Zur Ablage der Datei legen wir den Ordner **Personenverwaltung** unterhalb des Standardordners **Environment.SpecialFolder.Personal** an, den die für uns relevanten Desktop-Betriebssysteme so zuordnen:

- Linux: **$HOME/Documents**
- macOS: **$HOME/Documents**
- Windows: **C:\Users\<benutzername>\Documents**

---

[1] In die Methoden zum Sichern und Laden sind Ideen von der folgenden Webseite eingegangen: https://github.com/AvaloniaUI/Avalonia.Samples/blob/main/src/Avalonia.Samples/CompleteApps/SimpleToDoList/README.adoc

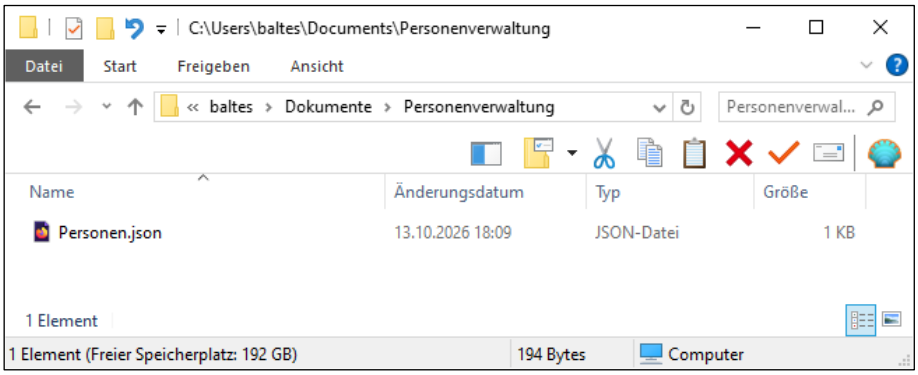
[2] https://de.wikipedia.org/wiki/JSON

Zum Speichern bzw. Laden kommen die Techniken zum (De)serialisieren von JSON-Dateien zum Einsatz (siehe z. B. Abschnitt 16.5 in Baltes-Götz & Frischknecht 2026).

Zwischen den statischen Methoden in der Klasse `PersonServices` und den **Command**-Eigenschaften der Menü-Items in der XAML-Datei vermitteln **RelayCommand**-Definitionen in der Ansichtsmodellklasse `MainViewModel`, die wir über mit Hilfe von MVVM-Toolkit - Attributen vornehmen (siehe Abschnitt 12.4.4.3), z. B.:

```
bool sichernAktiv = false;
[RelayCommand(CanExecute = nameof(CanSaveToFile))]
public async void SaveToFile() {
    if (Personen == null)
        return;
    sichernAktiv = true;
    SaveToFileCommand.NotifyCanExecuteChanged();
    await PersonServices.SaveToFileAsync(Personen);
    sichernAktiv = false;
    SaveToFileCommand.NotifyCanExecuteChanged();
}
public bool CanSaveToFile() {
    if (Personen == null || Personen.Count == 0 || sichernAktiv)
        return false;
    else
        return true;
}
```

Die beim Speichern entstehende Datei



enthält die `Personen`-Kollektionselemente im JSON-Format, z. B.:

```
[
  {"Id":        "15ff8518-ef1a-4e45-82b2-c5395259258f",
   "FirstName": "Susanne",
   "LastName":  "Brgl",
   "Age":       45},
  {"Id":        "9fd3334c-bf82-49ea-849f-5d06bafa0c35",
   "FirstName": "Randolf",
   "LastName":  "Schmitt",
   "Age":       34},
  {"Id":        "61b7aaaf-33b0-49cb-a643-2c49f672a112",
   "FirstName": "Karl",
   "LastName":  "Wallner",
   "Age":       89}
]
```

Zur Bequemlichkeit der Benutzer lädt der Ansichtsmodellklassen-Konstruktor die Personen automatisch, wenn er die erwartete Datei findet. Anderenfalls fordert er Übungsfälle bei der Klasse `PersonServices` an:

```
public MainViewModel() {
    if (File.Exists(PersonServices.JsonFileName))
        LoadFromFile();
    else {
        Personen = PersonServices.GetPeople();
        InitViewModel();
    }
}

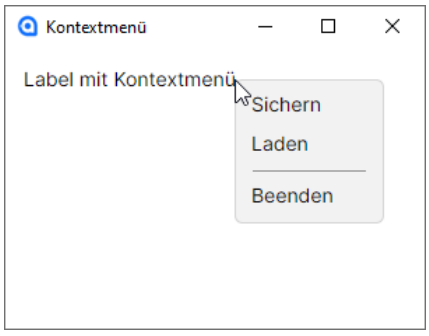
void InitViewModel() {
    SelectedItem = Personen?.Count > 0 ? Personen[0] : null;
    FirstName = SelectedItem?.FirstName ?? "NN";
    LastName = SelectedItem?.LastName ?? "NN";
    Age = SelectedItem?.Age ?? 0;
    Sage = Age.ToString();
    SwitchAddPerson(false);
}
```

13.1.2 Kontextmenü

Zu beliebigen Steuerelementen kann man ein per Rechtsklick zu öffnendes Kontextmenü einrichten.[1] Der folgende XAML-Code vereinbart ein Kontextmenü zu einem **Label**-Objekt über seine (von **Control** geerbte) **ContextMenu**-Eigenschaft:[2]

```
<Label Content="Label mit Kontextmenü " VerticalAlignment="Top" Margin="10">
    <Label.ContextMenu>
    <ContextMenu>
        <MenuItem Header="_Sichern" Command="{Binding SaveToFileCommand}" />
        <MenuItem Header="_Laden" Command="{Binding LoadFromFileCommand}" />
        <Separator/>
        <MenuItem Header="_Beenden" Command="{Binding ShutdownAppCommand}"/>
    </ContextMenu>
    </Label.ContextMenu>
</Label>
```

Das Menü erscheint nach einem Rechtsklick auf das **Label**:



Auf eine sinnvolle Implementation der Kommandos im Ansichtsmodell verzichten wir der Einfachheit halber:

```
using CommunityToolkit.Mvvm.Input;
namespace Kontext.ViewModels;
public partial class MainViewModel : ViewModelBase {
    [RelayCommand] public void SaveToFile() { }
    [RelayCommand] public void LoadFromFile() { }
    [RelayCommand] public void ShutdownApp() { }
}
```

---

[1] https://docs.avaloniaui.net/controls/menus/contextmenu

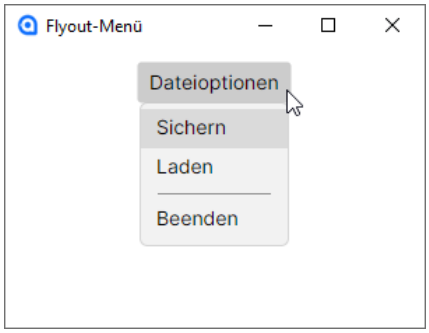
[2] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Menüoptionen\Kontextmenü

13.1.3 Flyout-Menü

Als Alternative zum Kontextmenü kann man zu manchen Steuerelementen ein sogenanntes *Flyout*-Menü einrichten.[1] Der folgende XAML-Code vereinbart ein solches Menü zu einem **Button**-Objekt über seine **Flyout**-Eigenschaft:[2]

```
<Button Content="Dateioptionen" VerticalAlignment="Top" Margin="10"
        HorizontalAlignment="Center">
    <Button.Flyout>
      <MenuFlyout>
              <MenuItem Header="_Sichern" Command="{Binding SaveToFileCommand}" />
              <MenuItem Header="_Laden" Command="{Binding LoadFromFileCommand}" />
              <Separator/>
              <MenuItem Header="_Beenden" Command="{Binding ShutdownAppCommand}"/>
      </MenuFlyout>
    </Button.Flyout>
</Button>
```

Das Menü erscheint nach einem Klick auf den Schalter:



Wie beim Kontextmenü verzichten wir der Einfachheit halber auf eine sinnvolle Implementation der Kommandos im Ansichtsmodell:

```
using CommunityToolkit.Mvvm.Input;
namespace Flyout.ViewModels;
public partial class MainViewModel : ViewModelBase {
    [RelayCommand] public void SaveToFile() { }
    [RelayCommand] public void LoadFromFile() { }
    [RelayCommand] public void ShutdownApp() { }
}
```

13.2 ToolTip

Über angefügte Eigenschaften der Klasse ein **ToolTip** im Namensraum **Avalonia.Controls** realisiert man eine temporär sichtbare Information zu einem Steuerelement:

- Die Anzeige erscheint nach einer kurzen Verzögerung in der Nähe des Steuerelements, wenn sich die Maus über diesem Element befindet.
- Die Anzeige endet, wenn der Mauszeiger den Bereich des zu erläuternden Steuerelements verlässt.

Man nimmt für das zu erläuternde Element ein **ToolTip.Tip** - Attribut in den XAML-Code auf, und die Ansicht erledigt die Aufgabe ohne Beteiligung des Ansichtsmodells. Das folgende Beispiel informiert die Benutzer über die Bedeutung der eingeklammerten Zahl am Ende der Beschriftung des **Hinzufügen**-Schalters im Personenverwaltungsprogramm:

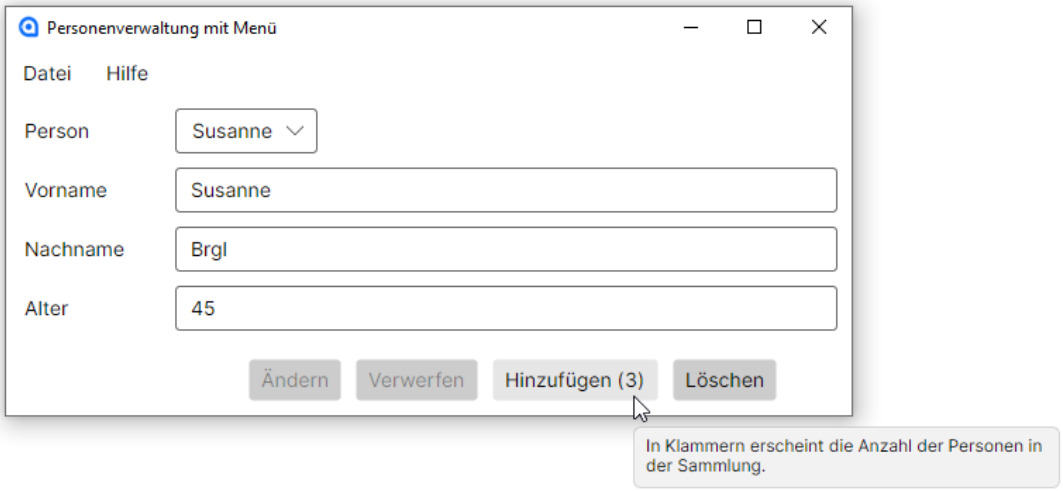
---

[1] https://docs.avaloniaui.net/controls/menus/menuflyout

[2] Ein Visual Studio - Projekt mit dem Beispiel ist hier zu finden:
...\BspUeb\Menüoptionen\Flyout

```
<Button IsVisible="{Binding EditPersonMode}" Command="{Binding AddPersonCommand}"
        Margin="10,0,0,0"
        ToolTip.Tip="In Klammern erscheint die Anzahl der Personen in der Sammlung.">
```

Wer sich für die Zahl interessiert, erhält schnell eine klare Auskunft:



Neben **ToolTip.Tip** zur Vereinbarung des Texts existieren weitere angefügte **ToolTip**-Eigenschaften, z. B.:[1]

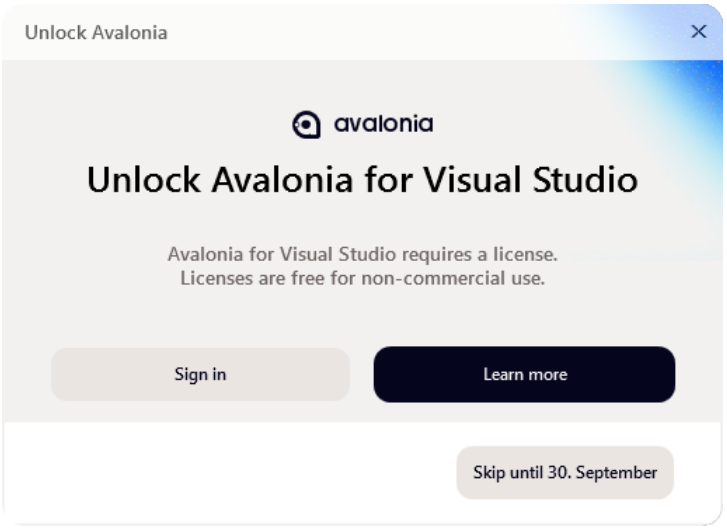
Angefügte Eigenschaft	**Bedeutung**
ToolTip.ShowDelay	Zeit bis zum Auftauchen der Erläuterung in Millisekunden (Voreinstellung: 400 Millisekunden), z. B.: `ToolTip.ShowDelay="1000"`
ToolTip.Placement	Statt am voreingestellten Einsatzort kann man den Text z. B. an der Mausposition auftauchen lassen: `ToolTip.Placement="Pointer"`

[1] https://docs.avaloniaui.net/controls/feedback/tooltip

Anhang

A. Erwerb einer Community-Lizenz für Avalonia UI

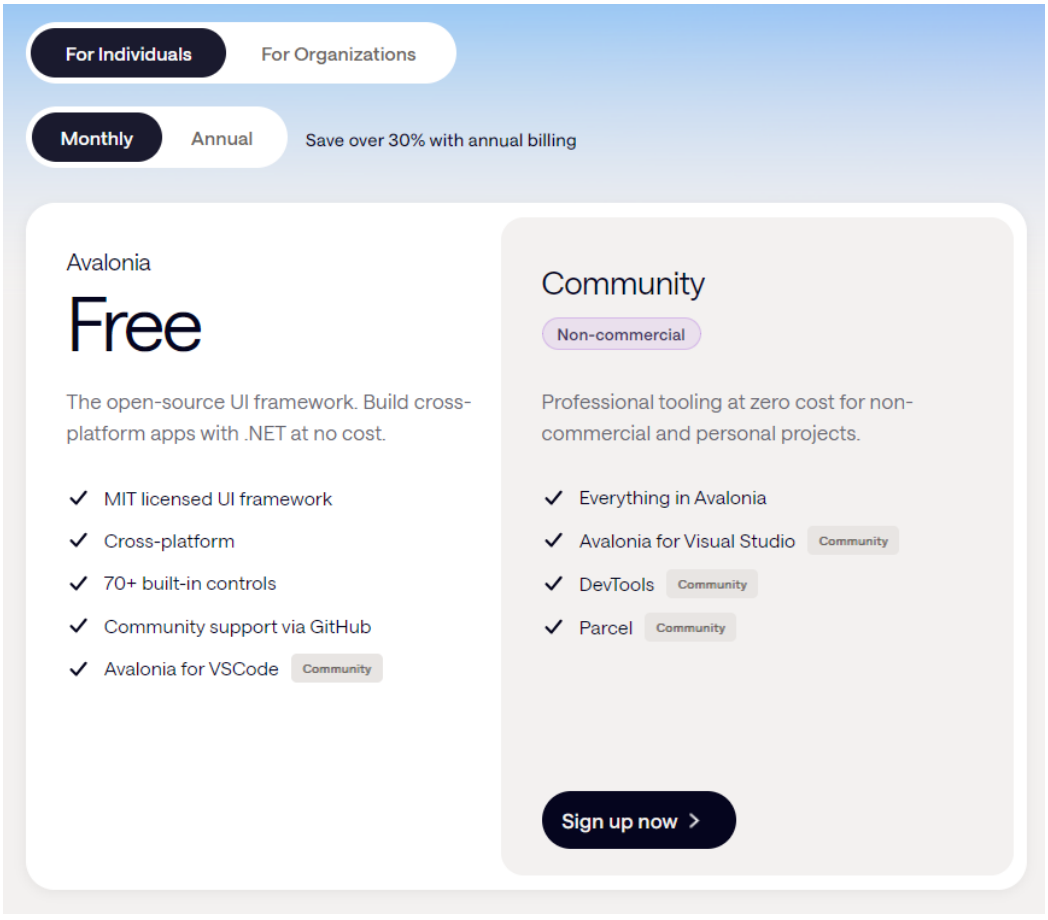
Ist die Visual Studio – Erweiterung zu Avalonia UI installiert, dann erscheint bei der Bearbeitung des ersten – Projekts nach dem Wechsel zum Registerblatt mit der Datei **MainWindow.axaml** (XAML-Deklaration der Bedienoberfläche) ein Hinweis auf die Lizenzlage:



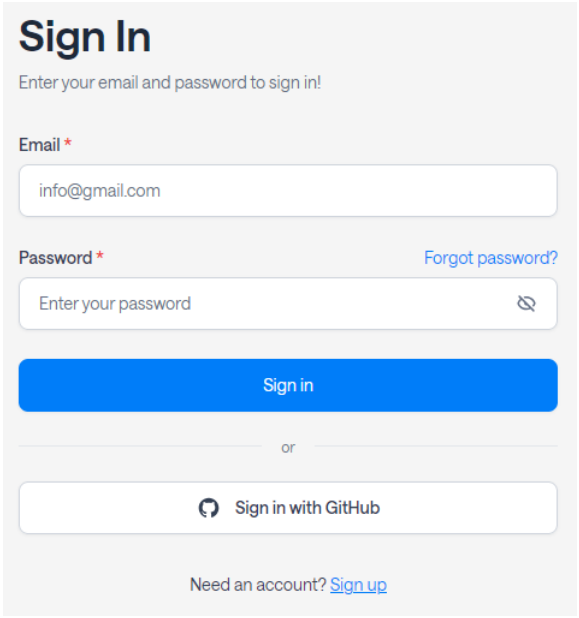
Nach einem Klick auf **Learn more** erscheint eine Webseite, wo man über Klicks auf

- **For Individuals** und
- **Sign up now**

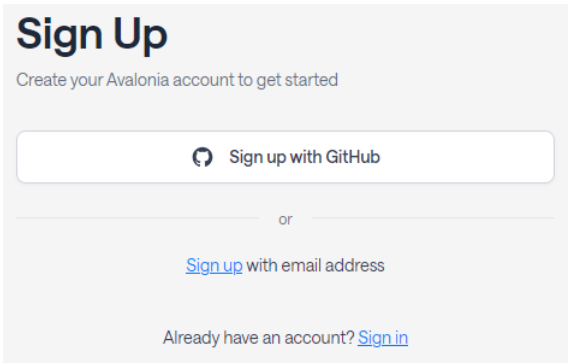
den Lizenzerwerb starten kann:



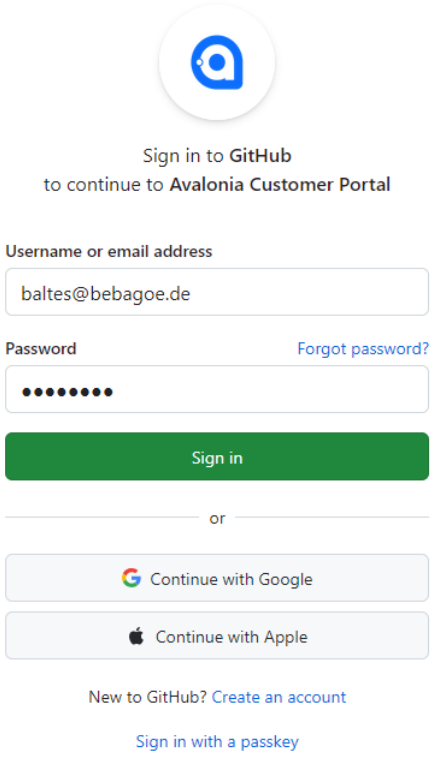
Auf der nächsten Webseite klickt man zum Erwerb eines Kontos auf **Sign up**:



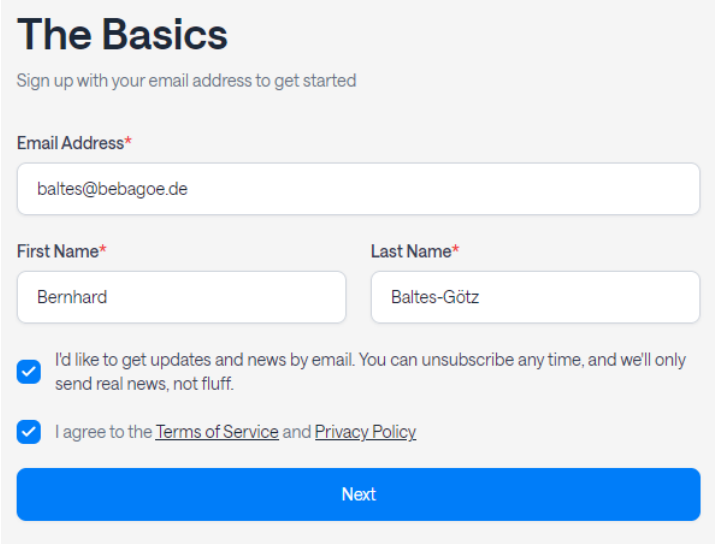
Wer ein GitHub-Konto besitzt,



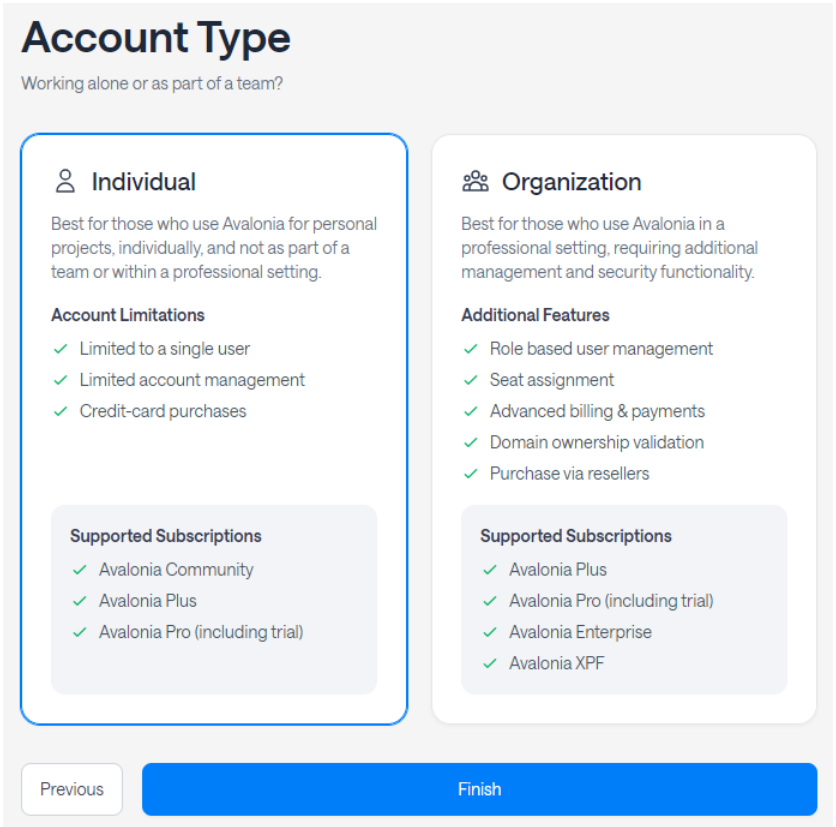
kann es zum Erwerb eines Avalonia-Kontos benutzen, z. B.:



Alternativ ist der Kontoerwerb per Mail-Adresse möglich, z. B.:

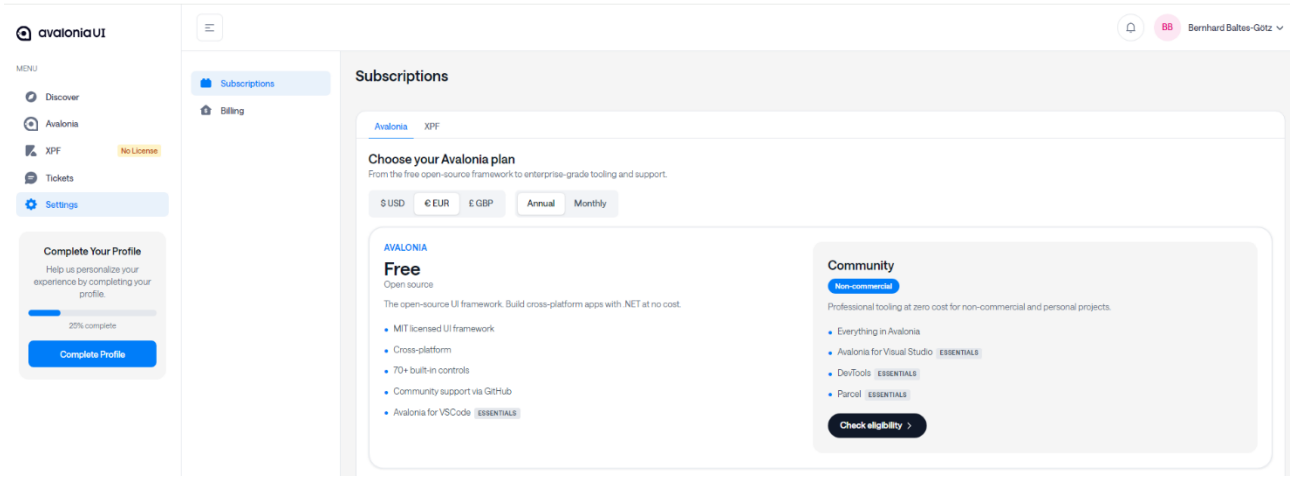


Wir beschreiben den Erwerb eines individuellen Kontos:

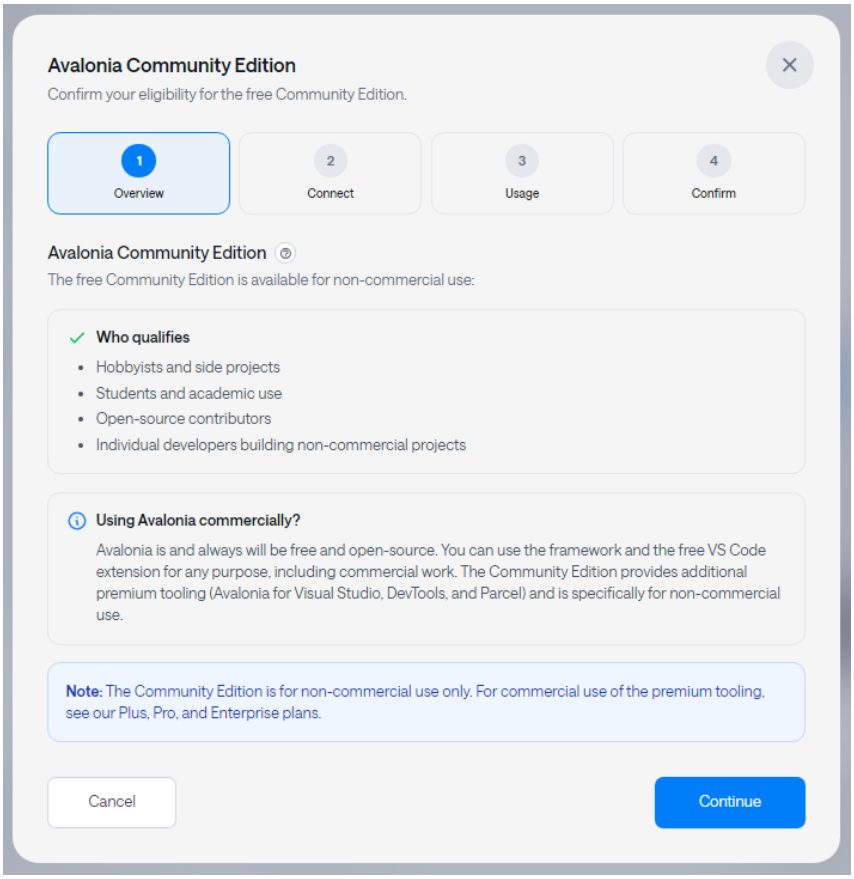


Nach der Bestätigung der Mail-Adresse ist noch ein Passwort zu vereinbaren.

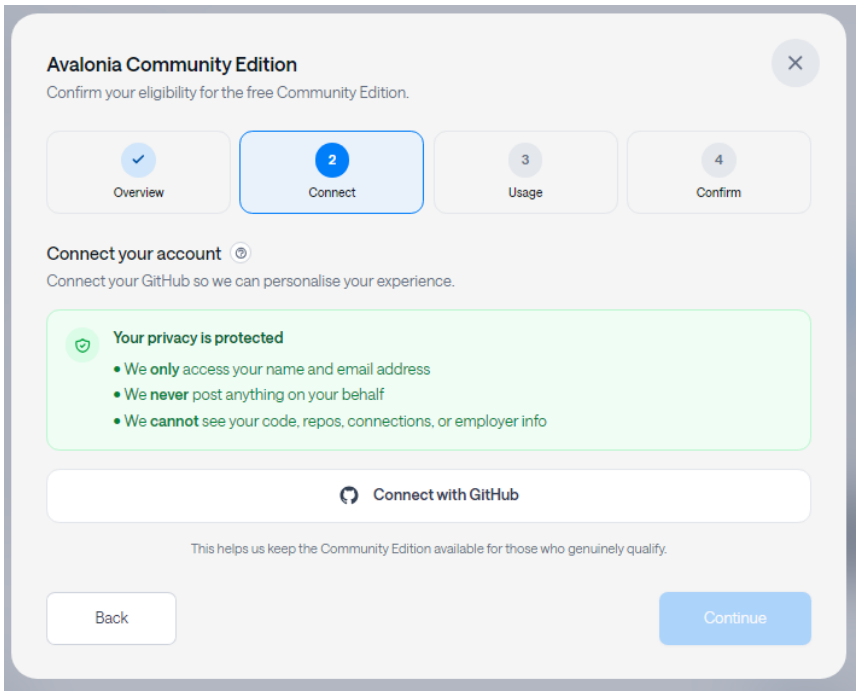
Man gelangt auf das Avalonia-Kundenportal und kann dort nach einem Klick auf den Schalter **Settings** den Erwerb der Community-Lizenz per Klick auf **Check eligibility** einleiten:



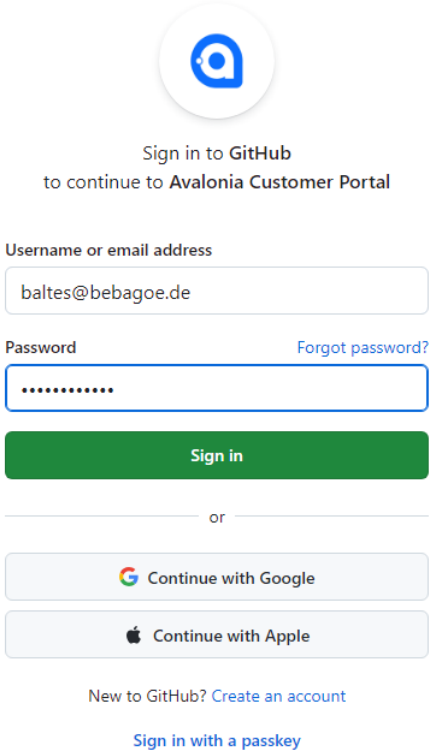
Es folgen Erläuterungen zur Community-Lizenz:



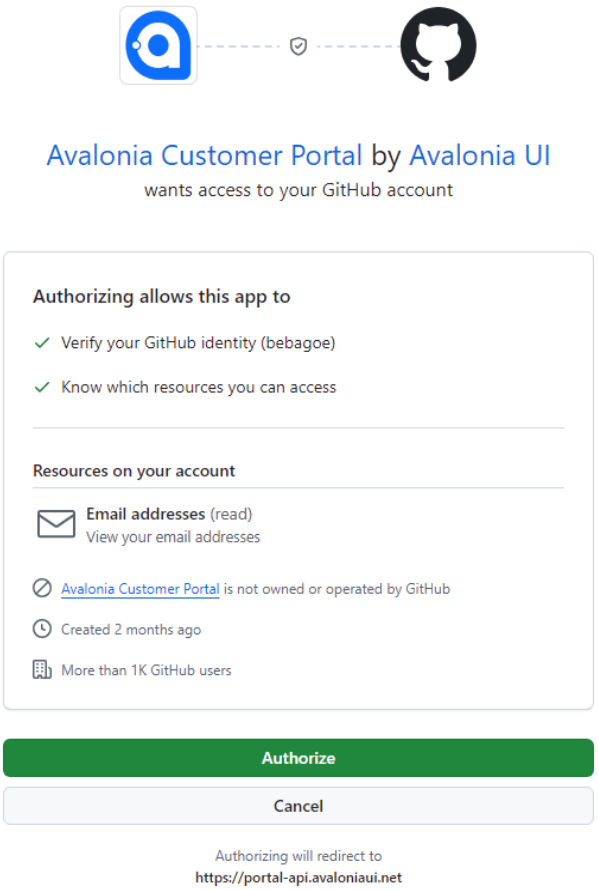
Im 2. Schritt ist eine Anmeldung bei GitHub erforderlich:



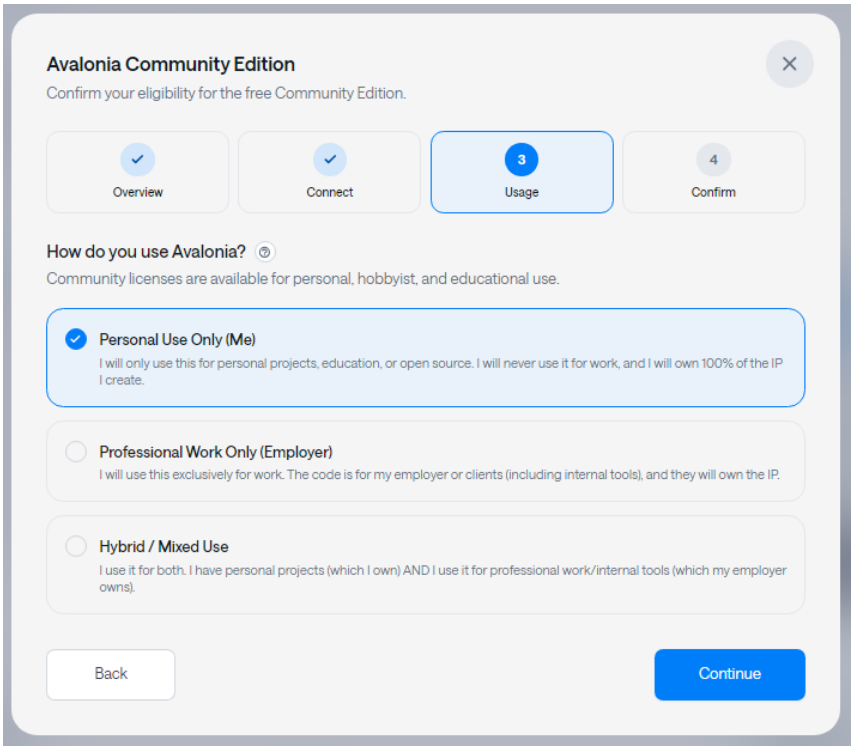
Wer kein GitHub-Konto hat, muss per Klick auf **Create an account** eines anlegen, z. B.:



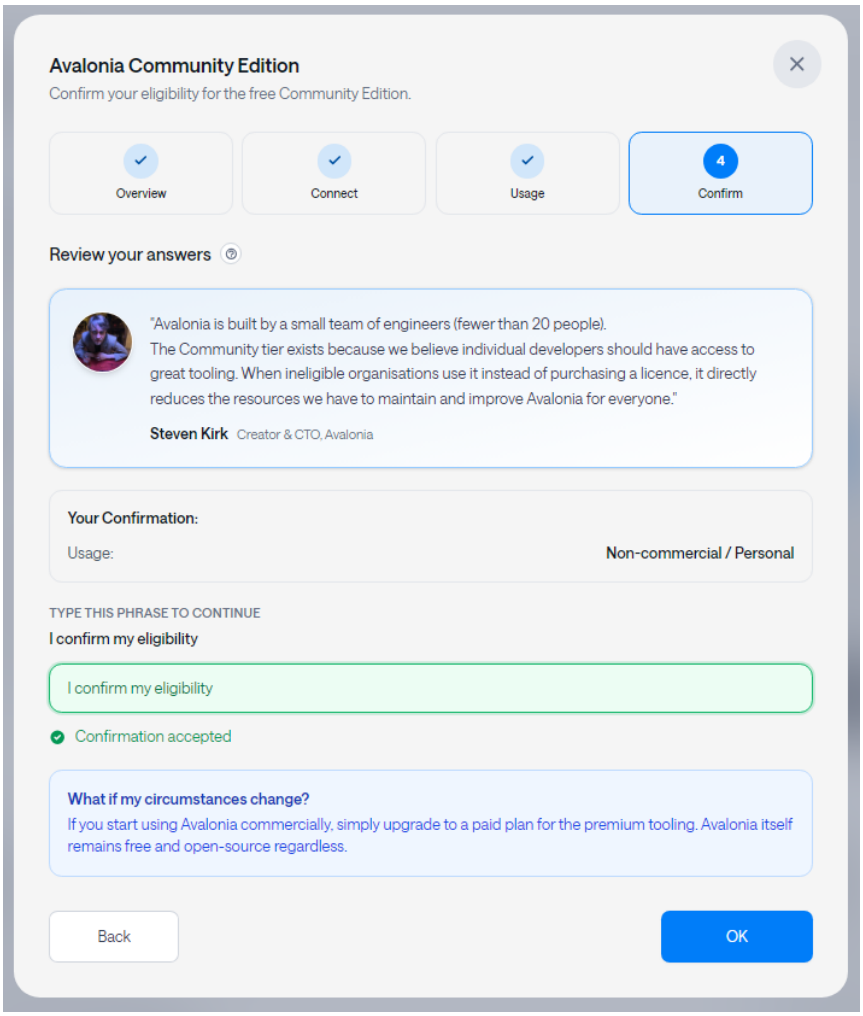
Ein GitHub-Konto benötigt man in der Regel auch zur Nutzung des KI-Assistenten Copilot im Visual Studio, sodass der Aufwand früher oder später entsteht. Man muss dem Avalonia-Kundenportal den Zugriff auf das GitHub-Konto per Klick auf **Authorize** erlauben:



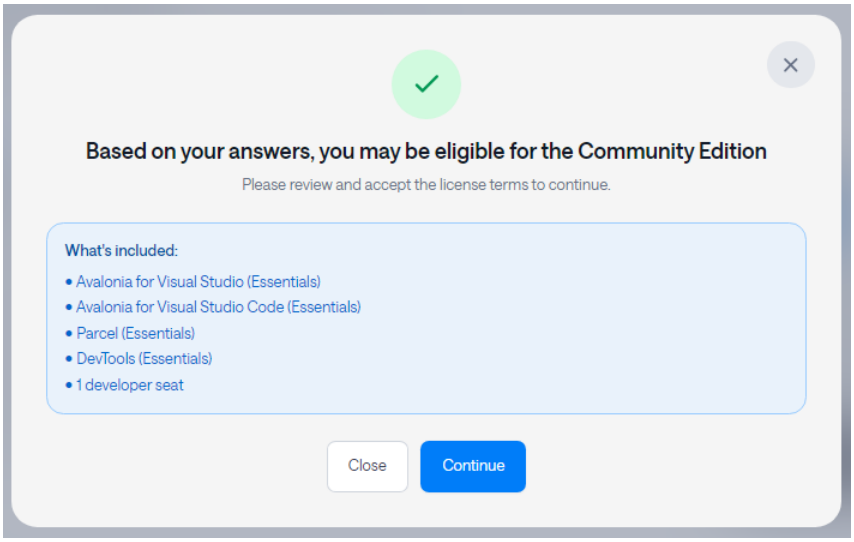
Im 3. Schritt auf dem Weg zur Avalonia-Community-Lizenz bestätigt man die persönliche Nutzungsart:



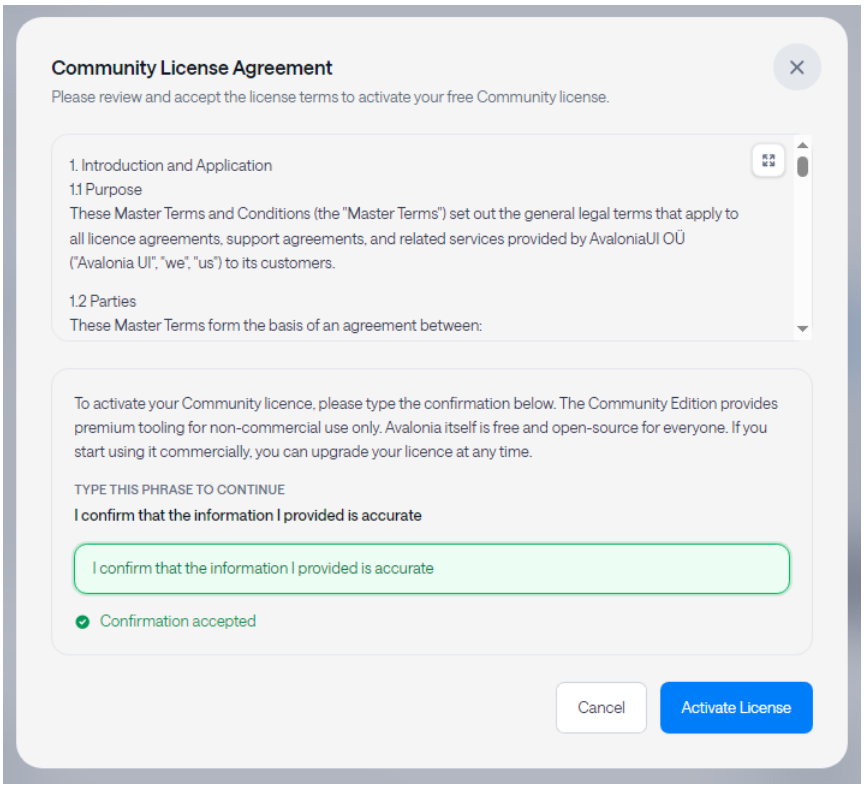
Die Firma hinter Avalonia muss verständlicherweise Verstöße gegen die Lizenzbedingungen erschweren:



Die Berechtigung zum Erwerb einer Community-Lizenz ist nachgewiesen:

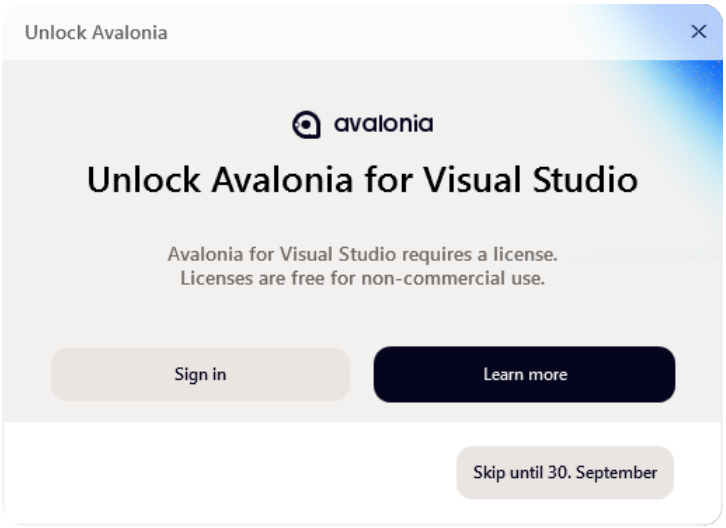


Jetzt sind noch die Lizenzbedingungen (19 DIN-A4 - Seiten) zu akzeptieren:

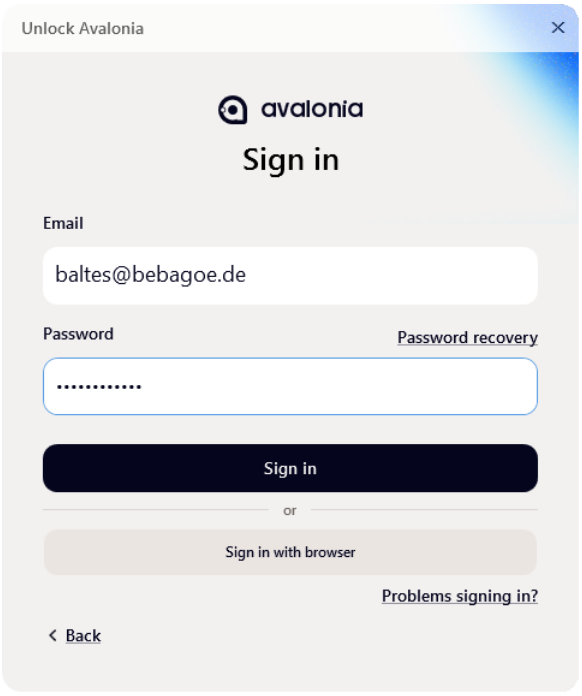


Die aktivierte Lizenz ist ein Jahr lang gültig.

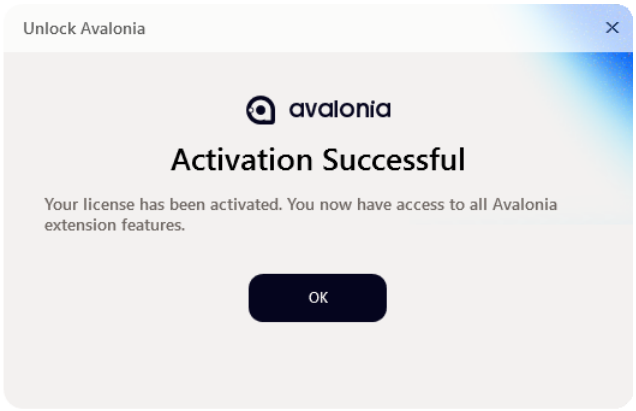
Mit der erworbenen Lizenz lässt sich die Avalonia-Erweiterung für das Visual Studio über zwei Klicks auf **Sign in**



entsperren:

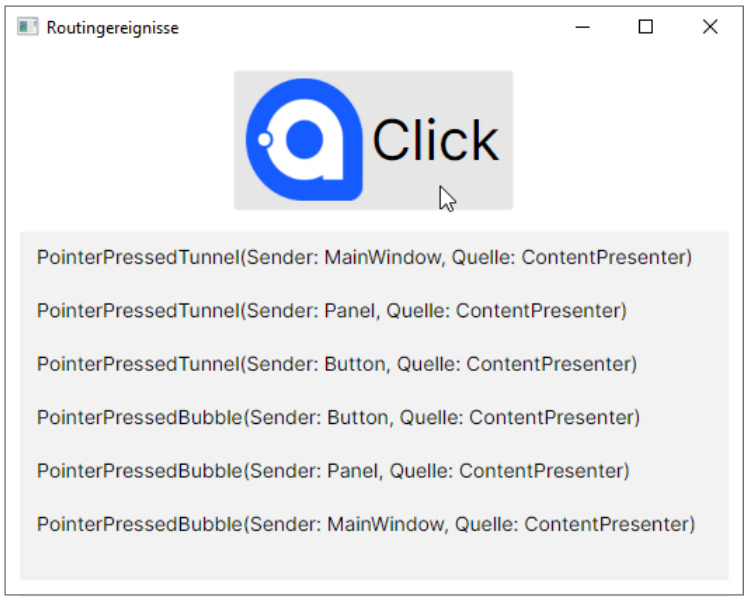


Einer Nutzung der Avalonia-Erweiterung zum Visual Studio steht nichts mehr im Weg:

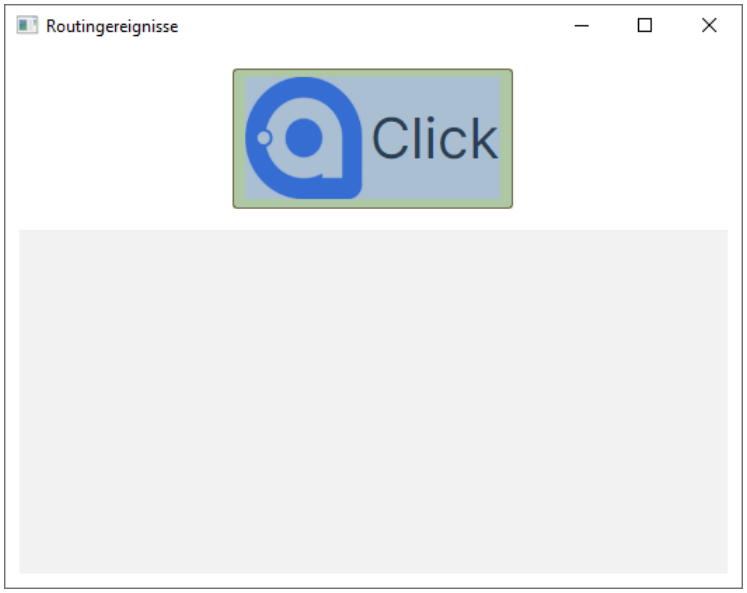


B. Visueller Baum und DevTools

Zu den Elementen eines Fensters existiert nicht nur der **logische Baum** mit den im XAML-Code deklarierten Objekten, sondern auch der feiner aufgelöste **visuelle Baum** mit den darzustellenden Elementen.[1] In der Regel müssen wir nur den logischen Baum beachten, doch für die Mausklick-Trefferdiagnose ist der visuelle Baum relevant. Das hat im Abschnitt 7.3 zu einem verblüffenden Ergebnis nach einem schlecht gesetzten Mausklick im Programm zur Beobachtung von Routingereignissen geführt. Hier taucht das Objekt **ContentPresenter** als Quelle auf:

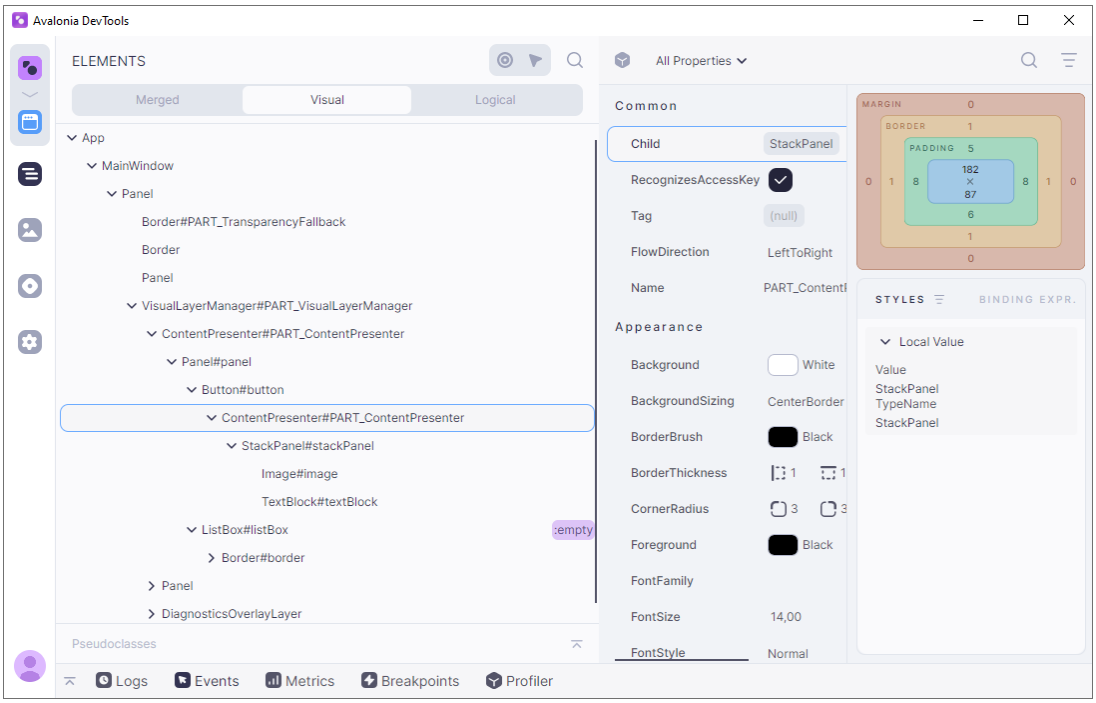


Mit Hilfe der **Avalonia DevTools** kann man den *visuellen* Baum zu einem Fenster



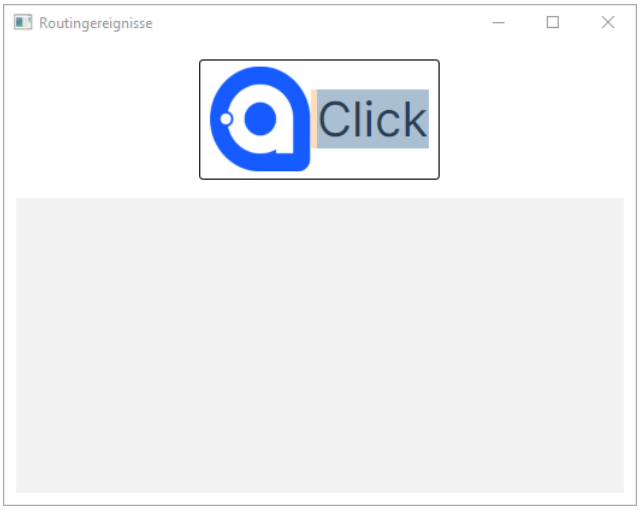
inspizieren und den **ContentPresenter** identifizieren:

[1] https://docs.avaloniaui.net/docs/fundamentals/visual-and-logical-trees



Ein zu tief angesetzter Klick trifft nicht den **TextBlock**, sondern den in Z-Richtung (orthogonal zur Display-Ebene) darunter befindlichen **ContentPresenter**, der als Träger für den **StackPanel**-Container mit dem **Image** und dem **TextBlock** dient.

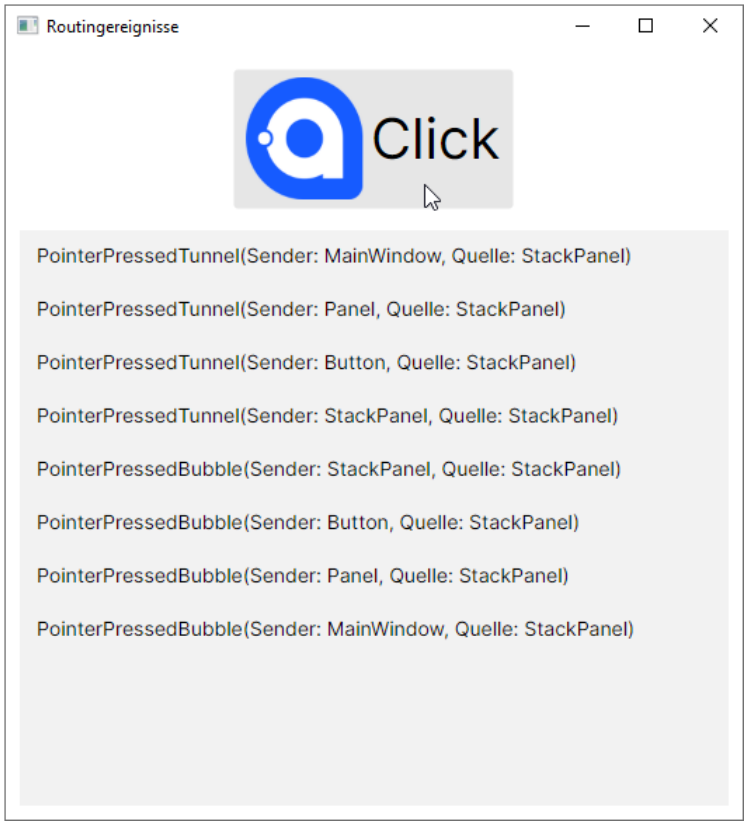
Markiert man in den **DevTools** den **TextBlock**, dann ist seine exakte Trefferzone erkennbar:



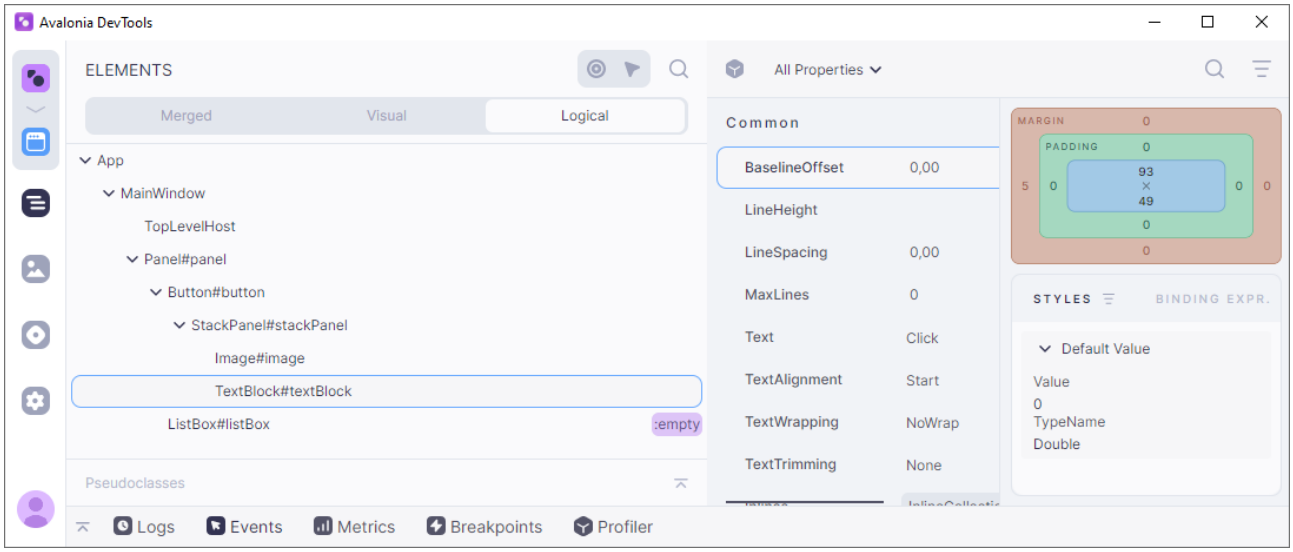
Nun stellt sich die Frage, warum ein (in vertikaler Richtung) zu tief angesetzter Klick nicht den **StackPanel**-Container trifft. Es liegt daran, dass ein Layoutcontainer ohne definierten Hintergrund keine Treffer auffangen kann. Eine beliebige **Background**-Definition macht einen Layoutcontainer zur potenziellen Trefferfläche, z. B. auch ein transparenter Hintergrund:

```
<StackPanel Name="stackPanel" Orientation="Horizontal" Background="Transparent"
            PointerPressed="Handle_PointerPressedBubble">
    . . . .
</StackPanel>
```

Aufgrund dieser Deklaration landet im Programm zur Beobachtung von Routingereignissen ein zu tief angesetzter Klick auf dem **StackPanel**-Container:



Der *logische* Baum enthält die Elemente aus der XAML-Datei:

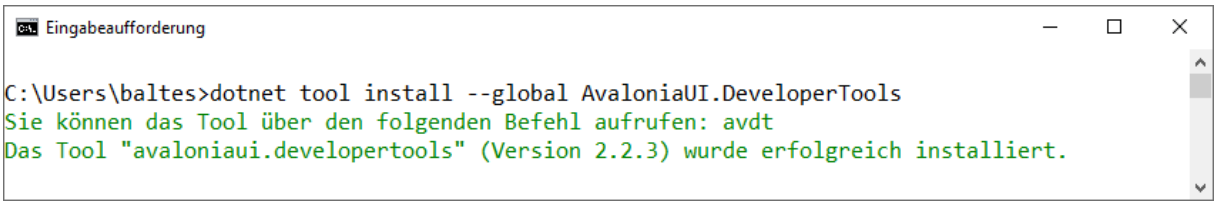


Es bleibt noch zu erklären, wie man die für Einzelpersonen im Rahmen der Community-Lizenz kostenlos verfügbaren **Avalonia DevTools** verwendet.[1] Dazu ist zunächst eine Installation über das Kommando

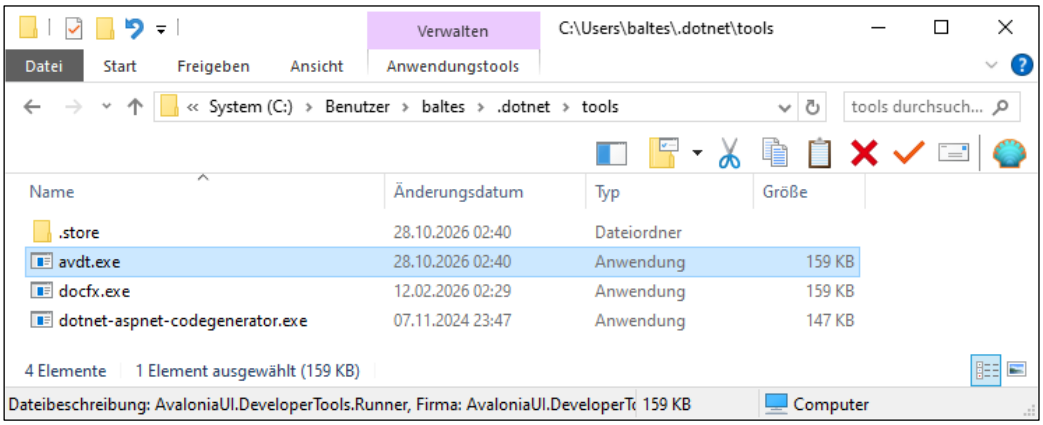
```
>dotnet tool install --global AvaloniaUI.DeveloperTools
```

vorzunehmen. Nach der erfolgreichen Installation

[1] https://avaloniaui.net/pricing?audience=individual



findet man die **DevTools** unter Windows im **.dotnet** - Ordner des Benutzers, z. B.:



Unter Linux und macOS befinden sich die Tools im folgenden Ordner:

~/.dotnet/tools

Zur Verwendung der **DevTools** im Visual Studio startet man das zu untersuchende Programm im Debugger (z. B. mit **F5**) und drückt dann die Taste **F12**.

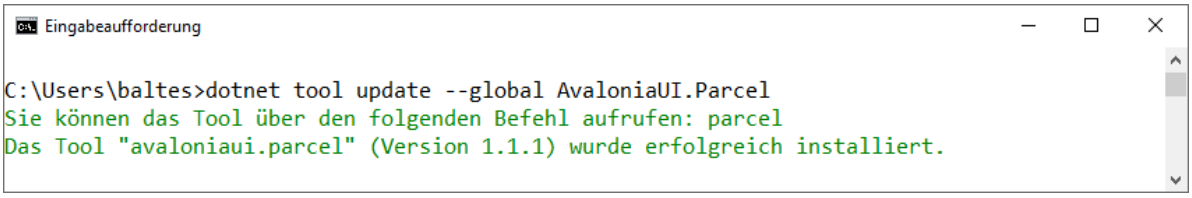
C. Parcel-Werkzeug zum Erstellen von Installationspaketen

Wer ein professionelles, also insbesondere benutzerfreundliches Programm erstellt hat, sollte auch ein benutzerfreundliches Veröffentlichungsverfahren wählen. Dazu gehört die Erstellung von Installationspaketen für die unterstützten Zielbetriebssysteme, um eine gute Systemintegration zu erreichen (z. B. Desktop-Verknüpfung, Standardmethoden zur (De)installation). Dazu bietet die Firma AvaloniaUI OÜ das Programm **Parcel** an. Es ist für Einzelpersonen im Rahmen der Community-Lizenz kostenlos verfügbar,[1] erstellt mit dieser Lizenz aber nur Installationspakete für das Betriebssystem, auf dem Parcel läuft. Da wir das nur für Windows verfügbare Visual Studio als Entwicklungsumgebung verwenden und die Community-Lizenz von Avalonia UI nutzen, können wir nur die Kreation eines Installationsprogramms für Windows vorführen. Mit der für 15 € pro Monat erhältlichen Plus-Lizenz von Avalonia UI lassen sich auf einem Entwicklungsrechner Installationspakete für Linux, macOS und Windows erstellen.

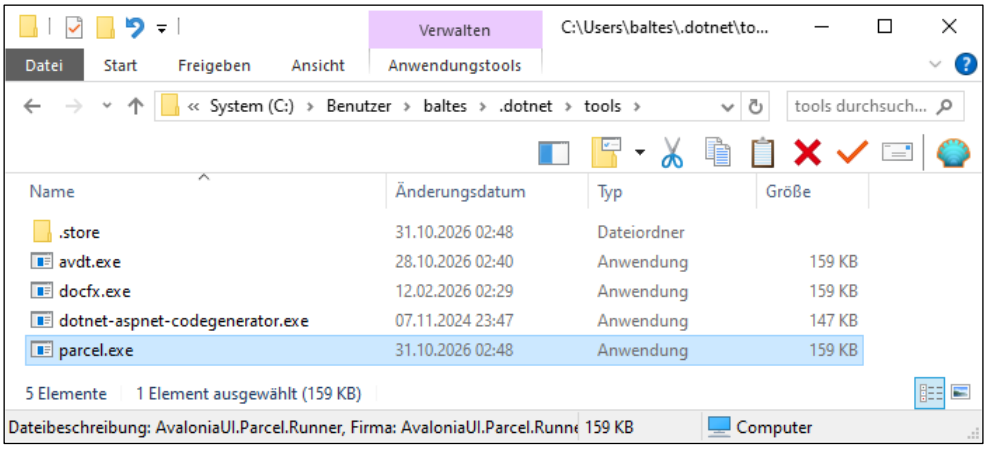
Zur Installation von Parcel dient das folgende Kommando:

```
>dotnet tool update --global AvaloniaUI.Parcel
```

Nach der erfolgreichen Installation



findet man Parcel unter Windows im **.dotnet** - Ordner des Benutzers, z. B.:



Unter Linux und macOS befinden sich die .NET - Tools im folgenden Ordner:

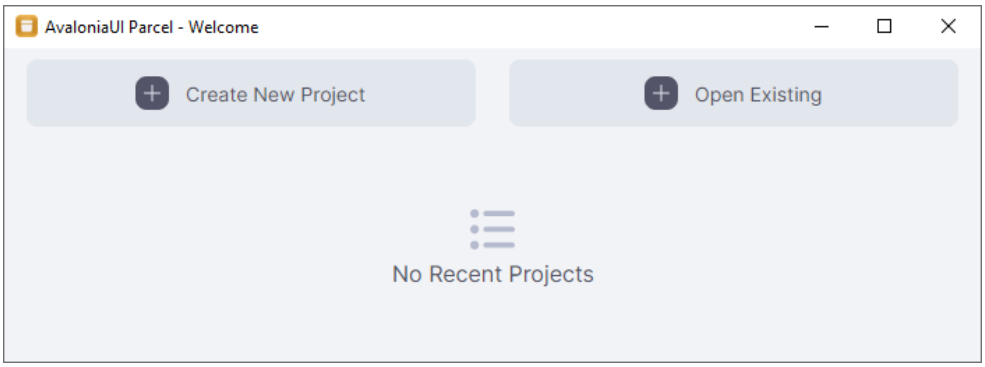
~/.dotnet/tools

Man startet Parcel z. B. per Konsolenfenster,

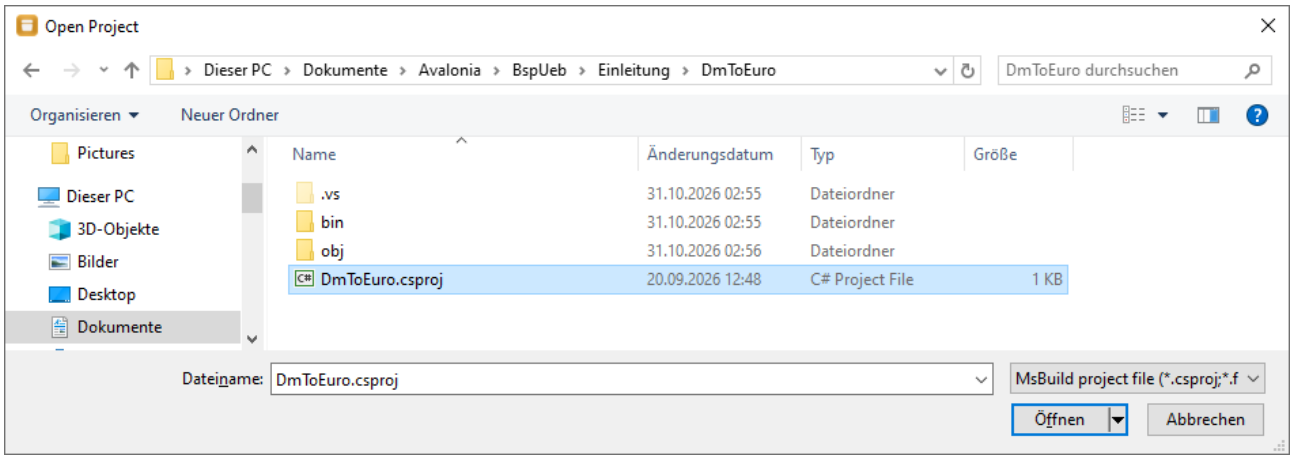


klickt dann auf **Create New Project**

[1] https://avaloniaui.net/pricing?audience=individual

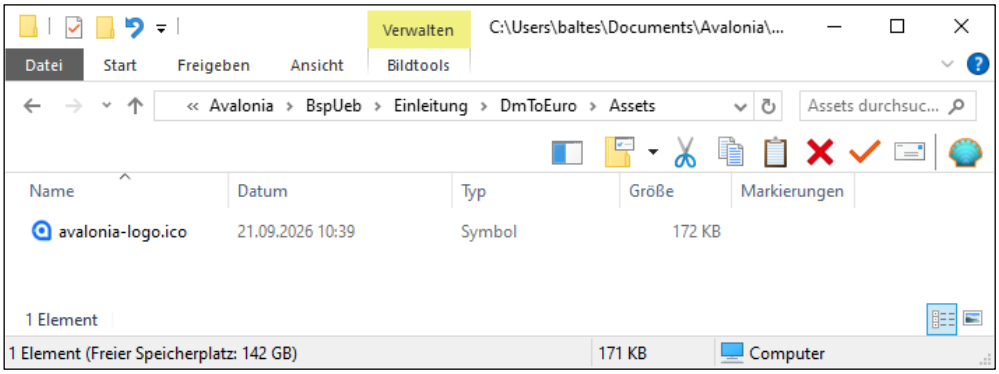


und öffnet die Projektdatei des zu installierenden Programms, z. B. die Projektdatei zu dem im Abschnitt 2.4 erstellten Währungskonverter:



Um das Anwendungsfenster und die ausführbare Windows-Datei mit einem Symbol zu verzieren, haben wir ausgehend vom Zustand im Abschnitt 2.4 ...

- den Projektordner **Assets** mit einer **ico**-Datei angelegt,



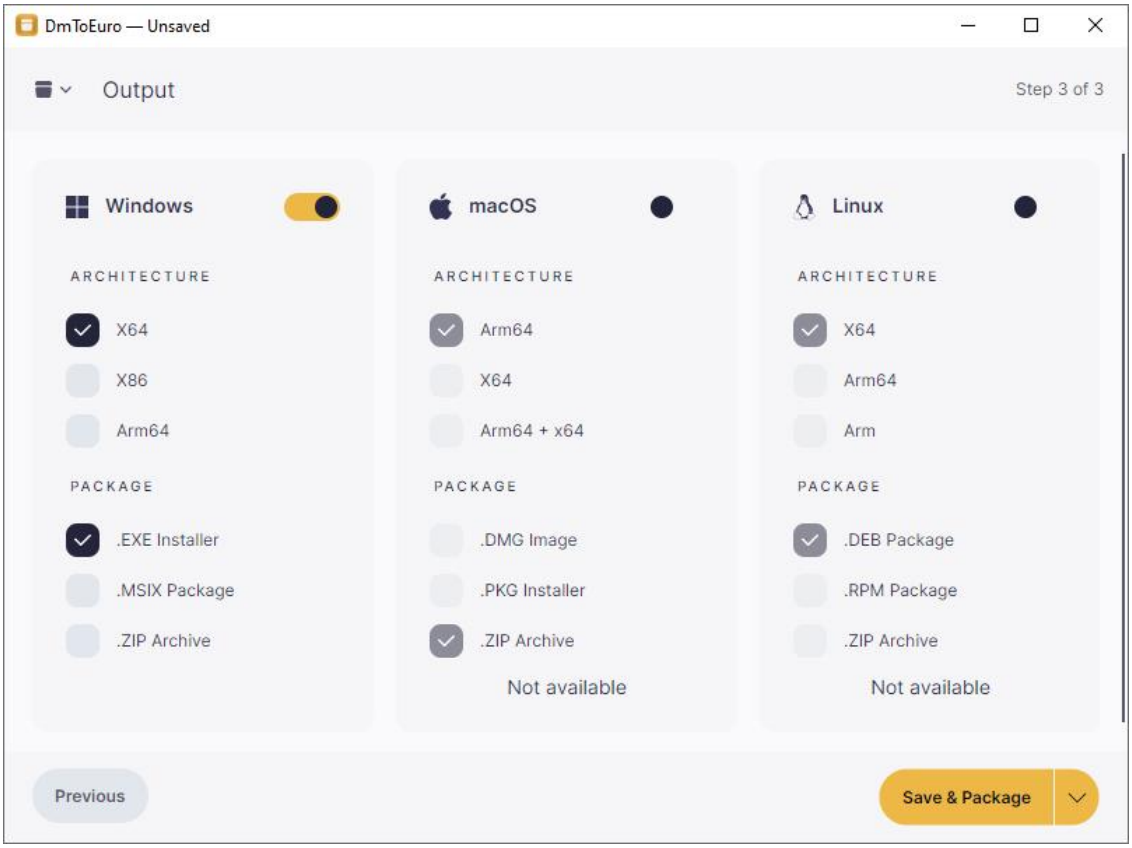
- in der Projektdatei den Ordner **Assets** als Quelle von Ressourcen bekannt gemacht und ein Anwendungssymbol vereinbart,

```
<ItemGroup>
  <AvaloniaResource Include="Assets\**" />
</ItemGroup>
<PropertyGroup>
  <ApplicationIcon>Assets\avalonia-logo.ico</ApplicationIcon>
</PropertyGroup>
```

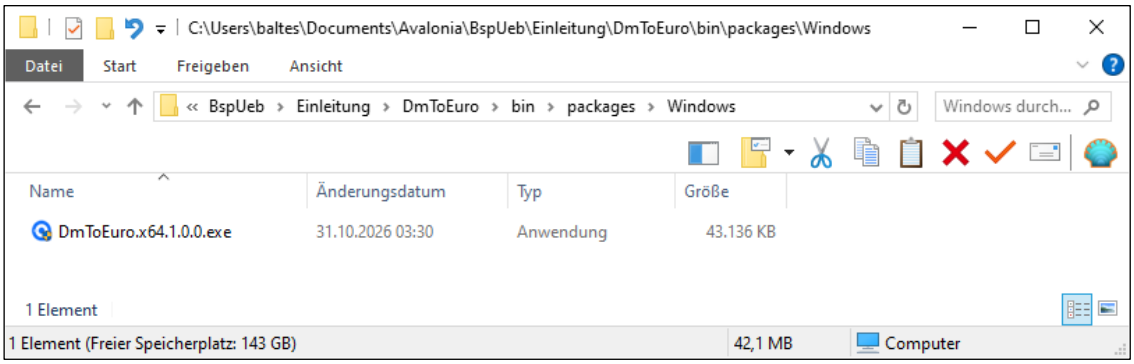
- im XAML-Code des Fensters ein Titelzeilensymbol vereinbart:[1]

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="DmToEuro.MainWindow"
        Icon="/Assets/avalonia-logo.ico"
        Title="DM → Euro" Width="500" Height="180" Loaded="Window_Loaded">
    . . .
</Window>
```

Unter Windows kann man mit Parcel u. a. ein EXE-Installationsprogramm produzieren:

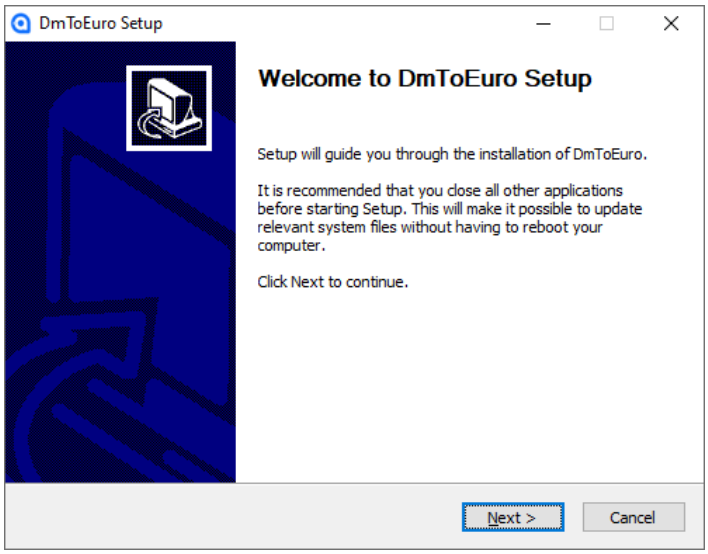


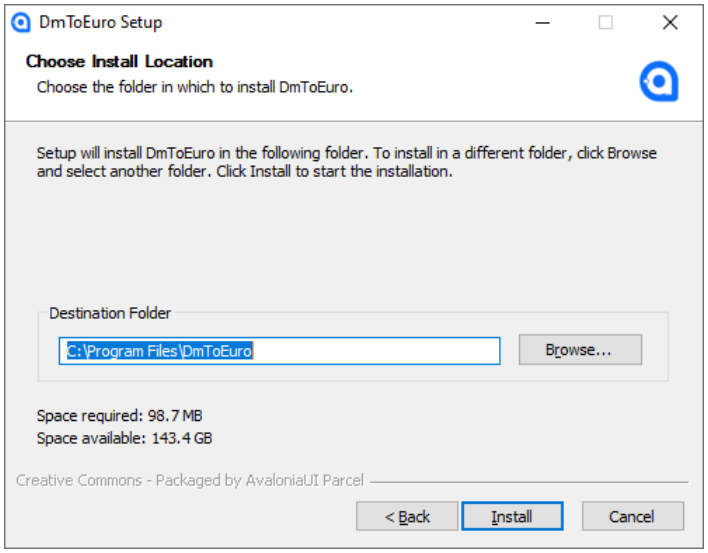
Nach dem Starten des erzeugten Installationsprogramms



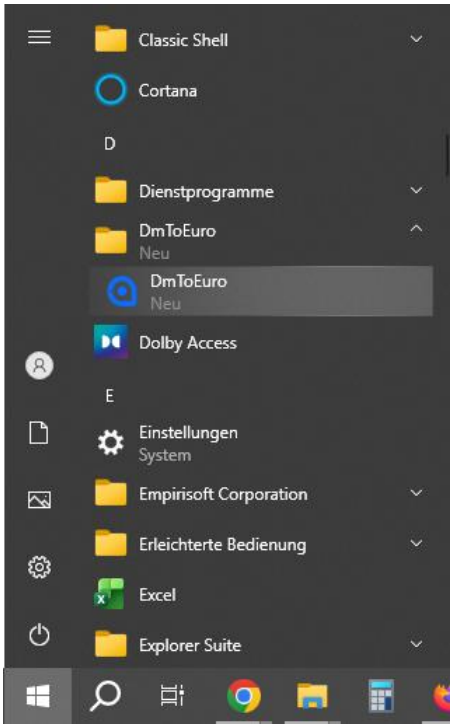
erscheinen die Plattform-üblichen Installationsdialoge, z. B.:

[1] Ein Visual Studio - Projekt mit dem bebilderten Währungskonverter ist hier zu finden:
...\BspUeb\Parcel

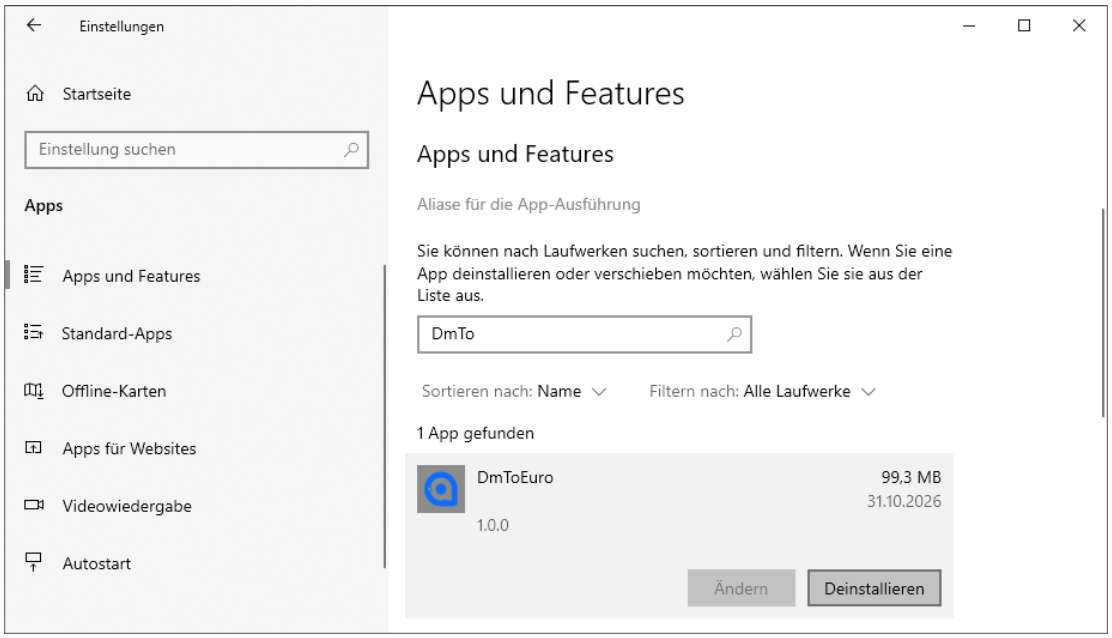




Man startet das installierte Programm per Startmenü,



und auch die Deinstallation erfolgt auf vertraute Weise:


Einstellungen
Startseite
Einstellung suchen
Apps
Apps und Features
Standard-Apps
Offline-Karten
Apps für Websites
Videowiedergabe
Autostart
Apps und Features
Apps und Features
Aliase für die App-Ausführung
Sie können nach Laufwerken suchen, sortieren und filtern. Wenn Sie eine App deinstallieren oder verschieben möchten, wählen Sie sie aus der Liste aus.
DmTo
Sortieren nach: Name
Filtern nach: Alle Laufwerke
1 App gefunden
DmToEuro
99,3 MB
31.10.2026
1.0.0
Ändern
Deinstallieren

D. Lösungsvorschläge zu den Übungsaufgaben

Kapitel 2 (Erstellungswerkzeuge unter Windows)

Aufgabe 2

1. **Richtig**
2. **Richtig**
3. **Falsch**
4. **Richtig**

Kapitel 11 (Steuerelemente)

Aufgabe 1

Ist die Behandlungsmethode dem **Button**- und dem **Panel**-Objekt zugewiesen, dann hebt ein zweiter Aufruf den Effekt des ersten auf. Im Endausbau der unsinniger Konstruktion bewirkt der dritte Aufruf schließlich den sichtbaren Farbwechsel.

Aufgabe 2

Ein Visual Studio – Projekt mit einem Lösungsvorschlag ist hier zu finden:

...\BspUeb\Steuerelemente\Eigene MessageBox\Reset-Nachfrage

Literatur

Baltes-Götz, B. & Frischknecht, P.(2026). *Einführung in das Programmieren mit C# 14*. Online-Dokument: https://bebagoe.de/csharp14/

Bandiera, R. (2025). *Avalonia UI. Cross-platform desktop applications in C#*. Morrisville, NC: Lulu.com

Del Sole, A. (2025). *Avalonia UI Succinctly*. Morrisville, NC: Syncfusion.

Johnson, R. (2025). *Avalonia Development Essentials*. NobleTrex Press.

Lischke, O. (2025). *Weniger Code, klarere Struktur*. Online-Dokument: https://entwickler.de/ux/weniger-code-klarere-struktur

Troelsen, A. & Japikse, P. (2022). *Pro C# 10 with .NET 6* (11th ed.). New York, NY: Apress.

Stichwortregister

A

Abhängigkeitseigenschaften 109
AccessText 167, 169
Activated
  Window-Eigenschaft 62
  Window-Ereignis 63
AddClassHandler<Target> 102
AddHandler() 93, 104
AddOwner() 113
AllowDrop 106
Alt-Tastenbefehl 167, 235
Alt-Zugriffstaste 167
Angefügte Eigenschaften 115
Angefügte Ereignisse 105
Animatable 54
Ansichtsmodell 189
Anwendungstyp 13
AppBuilder 57
Application 63
  Current 64
  XAML-Element 72
ApplicationLifetime 58, 59, 64
ASP.NET Core Razor Pages 9
Assemblyname 41
attached properties 115
Attribut
  XML 72
Ausgabetyp 39
AvaloniaObject 54, 112, 124
AvaloniaProperty 109
AvaloniaResource 100
AvaloniaXamlLoader 82, 87, 155

B

Backend 58
BAML-Dateien 81
Befehlsschalter 164
Binding 81, 125
BindingMode 119
Blazor WebAssembly 9
Border 106, 109, 111, 113
BorderBrush 30
BorderThickness 30
Bubble-Weiterleitung 92
Button 31, 104, 164

C

Call Back - Routinen 2
CallerMemberName 129
CanExecute() 200, 202
CanExecuteChanged 199, 201
CanResize 164
Canvas 149
CheckAccess() 56
CheckBox 169
Children 135
chmod 52
CLI 12
Click 31
Closing
  Window-Ereignis 63
Code-Behind – Dateien 83
CollectionChanged 130
ColumnSpacing
  Grid 138
  UniformGrid 149
ColumnSpan
  Grid 139
COM 57
ComboBox 122, 177
COMException 57
compiled bindings 121
CompiledBinding 119
Configure<TApp>() 57
Content 26, 158
ContentAttribute 78
ContentControl 55
ContentTemplate 55
Control 55
controls 23
Controls
  Klasse 135
Converter 127
CSS 5
CustomValidation 232

D

DataContext 119, 123, 179, 197
DataTemplate 180
Datenbindung 81, 119, 178
Deactivated
  Window-Ereignis 63
Debug
  Klasse 108
Debug-Konfiguration 36
dependency properties 109
DevTools 250
Direktereignisse 92
Dispatcher 55, 64, 66
DispatcherPriority 56
DisplayMemberBinding 123
Dock 141, 235
DockPanel 141
dotnet.exe 12
Drag & Drop 106
DragDrop 106

E

Eigenschaftswertvererbung 114
ElementName 124, 126
Escape-Schalter 167
Execute()
  Kommando 200
Exit
  Application-Ereignis 60
Exit() 61
Exit-Code 60

F

FluentTheme 64
Flutter 53
Flyout-Menü 239
Focus() 34, 176

Focusable 175
FontFamily 172
FontSize 31

G

Geräteunabhängige Pixel 138
GetTopLevel() 185
GetValue() 112
GetValueOrDefault() 171
Grafikelemente 153
Grid 137
GridSplitter 140
GroupName 171

H

HarfBuzz 58
HasErrors 131
Height 25, 156
Hollywood-Prinzip 2
HorizontalAlignment 26, 157
HorizontalContentAlignment 158
HotKey 166

I

IClassicDesktopStyleApplicationLifetime 59
ICommand 198, 199
IControlTemplate 55
IDataTemplate 55
ILSpy 47
Image 100
IMessenger 233
Impeller 53
Ineractive 54
Inhaltseigenschaft
  XAML 78
InitalizeComponent() 85
Initialize() 63
InitializeComponent() 87
Innenrand 31
INotifyCollectionChanged 130
INotifyDataErrorInfo 131
INotifyPropertyChanged 54, 128
INotifyPropertyChanging 211
InputElement 54, 92
IntelliSense 24
Inter 21
InvokeAsync() 56
IRelayCommand 217
IrfanView 100, 168
IsCancel 167
IsChecked 171
IsDefault 31, 165, 175
IsEditable 181
IsEnabled 229
IsVisible 225, 229
ItemCollection 178
ItemsControl 122
ItemsSource 179
  ListBox 178
ItemTemplate 180
IT-Souveränität 5
IValueConverter 126

K

Kombinationsfeld 177
Kommentar
  X(A)ML 72
Kontextmenü 238
Kontrollkästchen 169

L

Label 26, 30
LastOrDefault() 203
Layoutable 54
Layoutcontainer 135
Leertaste 176
ListBox 177
Listenfeld 177
Loaded-Ereignis 34, 176
Logischer Baum 124, 250

M

MainWindow 60
  IClassicDesktopStyleApplicationLifetime-Eigenschaft 60
MainWindowViewModel 190
Manifest 42
Margin 26, 156
Markup-Erweiterungen 80
MaxHeight 156
MaxWidth 156
Mehrzeilige TextBox 174
MenuItem 235
MessageBox.Avalonia 65, 103, 117, 163
MinHeight 156
MinWidth 156
Modaler Dialog 67
Modultest 189
MVVM 187
MVVM Community Toolkit 210

N

Name
  XAML 29
nameof-Operator 129
NaN 156
NotifyCanExecuteChanged() 218
NuGet-Pakete 42, 67
null - (Un)zulässigkeit von Referenztypen 41

O

Obfuskator 49
ObservableCollection<T> 120, 130, 179
ObservableObject 211
ObservableProperty 212, 215
ObservableRecipient 233
ObservableValidator 215
OnFrameworkInitializationCompleted() 60, 63, 120, 154
Optionsfeld 169
Orientation 145, 146
OutputType 39

P

Padding 29, 31, 158
Panel 135, 136
Parcel 52, 254
partial 84, 85, 218
PasswordChar 174
Passwörter 174
Path

Binding 122
pdb-Datei 37
png-Datei 168
Pointer 95
PointerType 95
Post() 56
Profiling 37
Program Database 37
Project Reunion 8
Projektdatei 38
Projektmappe 19
Projektmappen-Explorer 20
property value inheritance 114
PropertyChanged 128
PropertyChangedEventHandler 201

R

RadioButton 169
RaiseEvent() 105
Range
MVVM Community Toolkit 216
Rectangle 147
RegularExpression
MVVM Community Toolkit 216
RelativePanel 146
RelativeSource 124
RelayCommand
manuell 207
MVVM Community Toolkit 217
RemoveHandler() 104
RequestedThemeVariant
Application-Eigenschaft 64, 72
Required
MVVM Community Toolkit 216
ResourceDictionary 75, 80
Resources
Application-Eigenschaft 64
StyledElement-Eigenschaft 62
Ressource
WPF 126
Return-Code 60
Reunion 8
RichTextBox 174
RoutedEvent 91, 104
RoutedEventArgs 94
Routingereignisse 91
Routingstrategien 92
RowSpacing
Grid 138
UniformGrid 149
RowSpan
Grid 139
Rückgängig-Funktion 211

S

Schaltfläche 164
Schriftauszeichnung 170
SDK 11
SelectedItem
ListBox 182
Selector 142
separation of concerns 187
SetColumn() 116
Setter 142
SetValue() 112
SharpDevelop 47
Show() 61
ShowDialog<TResult>() 184
ShowGridLines 139
Shutdown() 60, 61
ShutdownMode 60, 61
ShutdownRequested
IClassicDesktopStyleApplicationLifetime-Ereignis 60
SimpleTheme 64, 114
Single-Thread - Architektur 55
SizeToContent 133
Skia 53
Skia-Grafik 5
Solution 19
SPA 9
Spacing
StackPanel 144
StackPanel 143
Standardnamespace 41
Standardschalter 165
Startup
ApplicationLifetime-Ereignis 60
STAThreadAttribute 57
StaticResourceExtension 80
Steuerelemente 23, 153
Stile 5
Style 142
StyledElement 54, 62
StyledProperty<T> 109, 111
Styles
Application-Eigenschaft 64
StyledElement-Eigenschaft 62

T

TargetFramework 40
TemplatedControl 55, 114
TextAlignment 29
TextBox 29, 172
TextChanged 173
Texteingabefeld 172
TextWrapping 174
Thickness 156
ToolTip 239
TopLevel 55
Transparenzfarbe 168
TryParse() 33
TrySetProperty() 216
Tunnel-Weiterleitung 92
TypeConverter 76
Typkonverter
XAML 76, 79

U

Übersetzte Datenbindung 121
UIElement 34, 176
Umschalter 169
Undo-Funktion 211
Ungarische Notation 171
UnhandledException 64, 66
UniformGrid 148
unit test 189
UpdateSourceTrigger 206
UsePlatformDetect() 21
UWP 8

V

ValidationContext 232
VerticalAlignment 26, 157

VerticalContentAlignment 31, 158
ViewModel 189
ViewModelBase 214
Visual 54
Visual Studio Community 2026 14
Visuelle Baum 250
Visueller Baum 124
VSIX Installer 15, 47

W

WASM 5
wasm-tools 14
WebAssembly 14
Width 25, 156
Window 61
  XAML-Element 73
WindowBase 55
WinRT 8
WinUI 2 8
WithInterFont() 21
WPF-Ereignissystem 91
WrapPanel 145

X

x
  Class 74
  Data 121
  Key 75
  Name 75
XAML 23, 71
  Attributsyntax 76
  Eigenschaftselementsyntax 77
  Inhaltseigenschaft 78
  Kollektionssyntax 79
  Markup-Erweiterungen 80
  Textinhaltsyntax 79
XAML-Eigenschaftselement 77
XAML-Instanzelemente 75
XAML-Typkonverter 76, 79
XamlX-Compiler 63, 81, 155
XML 71
XML-Deklaration 71
XML-Kommentar 72
XML-Namensräume 74
xmlns 74
XPF 5

Z

Zielframework 40
ZIndex 137, 151
Zugriffstaste 167