Der Generationenkonflikt fällt aus

Interoperabilität zwischen WPF und Windows Forms

von Thomas Huber und Christoph Pletz

Mit der Einführung von .NET 3.0 bietet Microsoft neuerdings zwei Programmiermodelle für die Entwicklung von Windows-Anwendungen unter .NET an. Während die bereits mit .NET 1.0 eingeführten Windows Forms weiterhin unterstützt werden, ist die Windows Presentation Foundation (WPF) die neue strategische Plattform für die Windows-User-Interface-Programmierung. Obwohl Windows Forms und WPF eine weitestgehend identische Zielsetzung haben – die Aufbereitung und Darstellung des Inhalts von Fenstern – besitzen sie technisch kaum Gemeinsamkeiten.

Vor der Auseinandersetzung mit der Interoperabilität zwischen Windows Forms und WPF sollte jeder Entwickler ein grundlegendes Verständnis beider Programmiermodelle besitzen. Damit ist nicht nur das Schreiben von Code gemeint, sondern auch das Verständnis dafür, wie die beiden Modelle ihre Aufgabe – die Aufbereitung und Darstellung des Inhalts von Fenstern – realisieren. Dies geschieht bei der WPF anders als bei Windows Forms. Daher werden zunächst beide Programmierschnittstellen etwas genauer beleuchtet, bevor auf die Interoperabilität zwischen Windows Forms und WPF mit 3D-Effekten, Animationen, Property Mapping und gemeinsamen Datenquellen eingegangen wird.

kurz & bündig

Inhalt

Wann ist es sinnvoll, WinForms und WPF zu kombinieren und wie werden die Klassen aus dem Namespace *System.Windows .Forms. Integration* für verschiedene Interoperabilitäts-Szenarien eingesetzt

Zusammenfassung

Es gibt die Möglichkeit, WinForms und WPF zu kombinieren. Dadurch lassen sich z.B. WPF-Elemente mit 3D-Effekten in WinForms einbinden. WinForms-Controls sind auch in der WPF einsetzbar

Quellcode

C#

Quellcode auf CD

Zwei Generationen unter der Lupe

Windows Forms ist eine objektorientierte Programmierschnittstelle, welche die Konzepte der klassischen Windows-Programmierung durch .NET-Klassen kapselt. In ihren Grundzügen ist die Programmierung von Windows bis zur Einführung von WPF seit Windows 1.0 weitestgehend unverändert geblieben. Ein Steuerelement innerhalb eines Fensters – und damit auch jedes Windows-Forms-Control – ist aus Sicht von Windows wiederum ein Fenster. Jedes Fenster wird über einen Window-Handle (*HWND*-Datentyp in C/C++, bzw. *System.IntPtr* in .NET) referenziert und ist für das Zeichnen seines Bereichs des Bildschirms verantwortlich. Das Zeichnen auf Bereiche, die Teil eines anderen Window-Handle sind, ist nicht möglich. Erkennt Windows, dass ein Fenster oder ein Teil davon neu gezeichnet werden muss, etwa weil die Anwendung neu gestartet wurde

© Software & Support Verlag GmbH

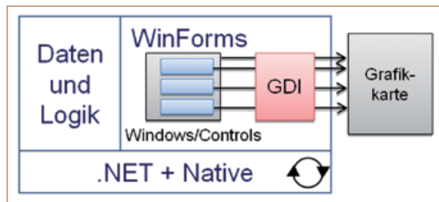


Abb. 1: Windows Forms verwendet das Graphics Device Interface zur Bildschirmausgabe

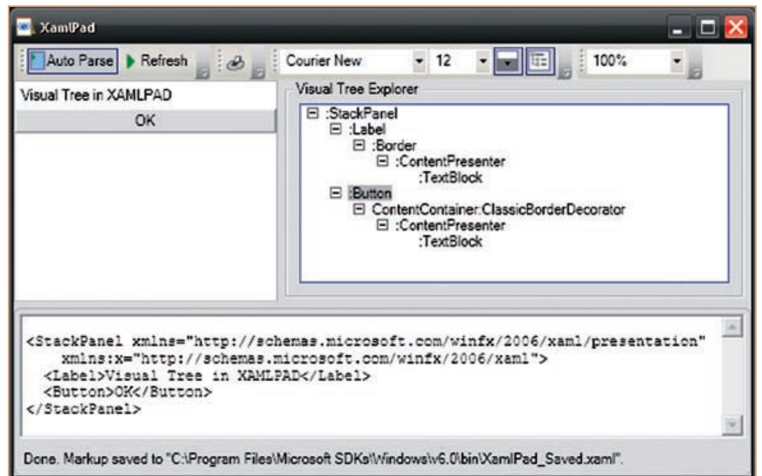
oder durch ein anderes Fenster verdeckt war, sendet Windows eine *WM_PAINT*-Nachricht an das zu zeichnende Fenster. Für ein Windows-Forms-Formular resultiert daraus der Aufruf der *OnPaint*-Methode. Für das Zeichnen selbst werden die Zeichenoperationen des GDI+ (*Graphics Device Interface*) verwendet, welches in .NET durch die Klasse *System.Drawing.Graphics* gekapselt ist (Abbildung 1).

MILCORE statt GDI+

Im Gegensatz zu Windows Forms war eines der Design-Ziele bei der WPF, nicht GDI+, sondern DirectX für die Darstellung einzusetzen. Dadurch steht die Leistung moderner Grafikkarten nicht nur in Spielen, sondern auch in „gewöhnlichen" Windows-Anwendungen zur Verfügung. DirectX wird durch eine neue Programmierschnittstelle namens *MILCORE* gekapselt – *MIL* steht dabei für „Media Integration Layer". MILCORE ist zurzeit noch nicht öffentlich, sondern wird nur Microsoft intern verwendet. Microsoft denkt jedoch darüber nach, dies zu einem späteren Zeitpunkt zu ändern. MILCORE ist nicht in .NET, sondern in nativen Code geschrieben und wird auch vom *Windows Desktop Manager* (WDM) in Windows Vista für die Darstellung der Betriebssystem-Oberfläche verwendet. Fähigkeiten von Windows Vista wie *Flip 3D* basieren auf der Funktionalität von MILCORE. Den darzustellenden Bildschirminhalt verwaltet MILCORE in Form einer Baumstruktur, dem sog. *Composition Tree*. Dieser Baum besteht aus Knoten, welche Metadaten und Zeichnungsinformationen enthalten. Bei Änderungen am Composition Tree generiert MILCORE DirectX-Befehle, die die Änderungen visuell umsetzen.

WPF ist ein auf MILCORE aufbauendes .NET-Programmiermodell. Das Prinzip der Entwicklung von WPF-Anwendungen und -Controls besteht darin, eine Hierarchie von visuellen Elementen zu erzeugen. Der WPF-Entwickler verwendet in der Regel eine Kombination aus XAML (*Extensible Application Markup Language*) – einer XML-basierten Beschreibungssprache für WPF-Anwendungen – und Programmcode gemeinsam mit Data Binding und Templates. Daraus wird eine Hierarchie aus Controls wie *Window*, *Button* etc. erzeugt und modifiziert. Die vom Entwickler verwendeten Controls setzen sich wiederum aus einfacheren visuellen Elementen wie *Rectangle*, *TextBlock* und *Border* zusammen. Die Hierarchie der visuellen Elemente wird als *Visual Tree* bezeichnet und kann im XamlPad mithilfe des Visual Tree Explorers betrachtet werden (Abbildung 2). Der Visual Tree bildet das Gegenstück zum auf MILCORE-Seite bestehenden *Compostion Tree* (Abbildung 3). Der Visual Tree enthält alle visuellen Elemente einer WPF-Anwendung. Die Typen aller Elemente im Visual Tree sind direkt oder indirekt von der abstrakten Klasse *System.Windows.Media.Visual* abgeleitet. Diese Klasse beinhaltet versteckte Logik, um mit dem *Composition Tree* auf der *native*-Seite von MILCORE zu kommunizieren. Über einen zweiseitigen Kommunikationskanal sind der Visual Tree und der Composition Tree miteinander verbunden. Über diesen werden Änderungen auf die jeweilig andere Seite übertragen. Die beiden Trees sind aber nicht 100 Prozent identisch, z.B. können einem *Visual* im Visual Tree mehrere Knoten im Composition Tree entsprechen. Objekte mit hohem Speicherplatzbedarf, wie etwa Bitmaps, werden gemeinschaftlich verwendet.

Abb. 2: In XamlPad lässt sich der Visual Tree im Visual Tree Explorer betrachten



Durch die Darstellung mit DirectX besitzen Elemente der WPF nicht nur ihren eigenen Pixelausschnitt wie die Window Handles in Windows Forms, sondern können auch auf Pixel anderer *Visual*-Objekte zugreifen. Dadurch öffnen sich in der UI-Programmierung völlig neue Möglichkeiten wie z.B. die Darstellung von halbtransparenten Elementen – ein Element kann einfach über die Anzeige/Pixel eines vorherigen Elements zeichnen. Für Interoperabilität zwischen Windows Forms und WPF ist festzuhalten, dass verschiedene Technologien, wie GDI+ und DirectX, nicht gemischt werden können. Ein Pixel gehört immer genau zu einer Technologie. Es ist somit nicht möglich, ein WPF-*UIElement* halbtransparent über ein Windows-Forms-*Control* zu platzieren.

Warum Windows Forms und WPF kombinieren?

Aufgrund der technischen Unterschiede zwischen Windows Forms und WPF gibt es keine automatisierte Methode, etwa durch einen Wizard, ein bestehendes Windows-Forms-Projekt in ein WPF-Projekt

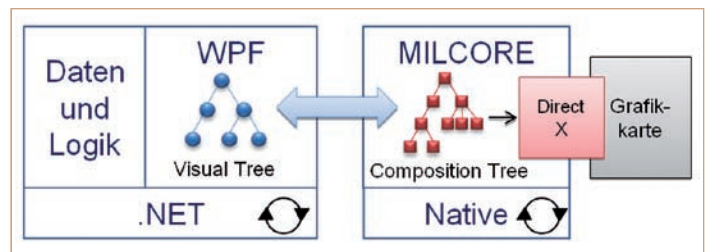


Abb. 3: Die WPF kommuniziert zur Bildschirmausgabe mit der „nativen" Komponente MILCORE

© Software & Support Verlag GmbH

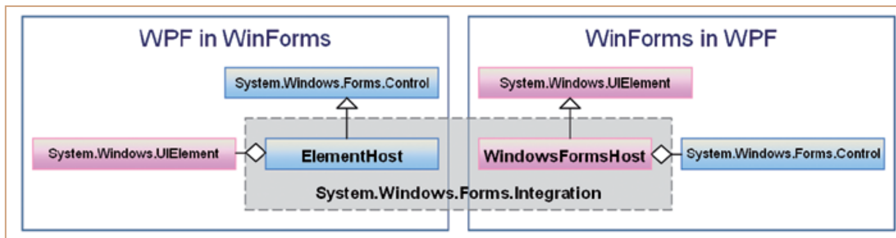


Abb. 4: Die Klassen WindowsFormsHost und ElementHost bilden die zentralen Komponenten für die Interoperabilität zwischen Windows Forms und WPF

zu migrieren, um die Entwicklung auf der neuen Plattform fortzusetzen. Aus diesem Grund werden die meisten Projekte, die mit Windows Forms als Grundlage begonnen wurden, auch basierend auf dieser Technologie weiterentwickelt. Dank der Interoperabilität zwischen Windows Forms und WPF können aber auch in solchen Projekten die neuen Features der WPF verwendet werden. Unter anderem sind dies die Audio/Video-Unterstützung, 3D-Grafik, Animationen und die verbesserte Integration von Dokumenten.

Entwickler, die heute mit einem WPF-Projekt beginnen, stehen vor dem Problem, dass die angebotene Auswahl an Controls – sowohl von Microsoft als auch von Drittherstellern – noch nicht mit denjenigen für Windows Forms vergleichbar ist. Beispielsweise enthält der WPF selbst lediglich ein rudimentäres *DataGrid Control*. Auch hier kommt Interop zwischen den beiden Technologien zu Hilfe und

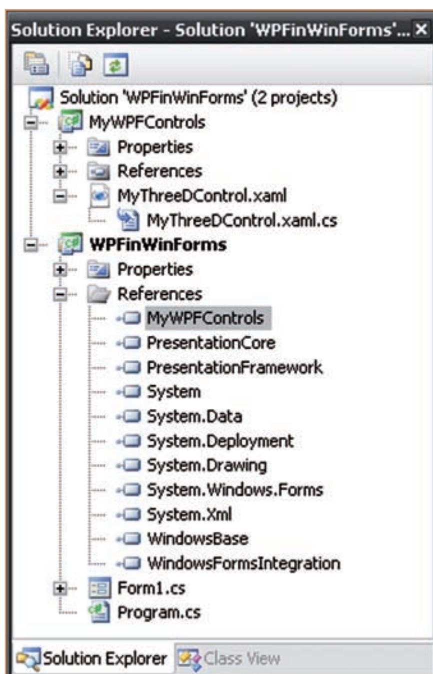


Abb. 5: Die Custom Control Library (WPF) MyWPFControls wird im Windows-Forms-Projekt referenziert

ermöglicht die Verwendung altbekannter Windows-Forms-Controls in WPF-Anwendungen. Wurde viel Aufwand in die Entwicklung eigener Windows-Forms-Controls gesteckt, so stellt der Interop-Ansatz an Stelle einer Neuentwicklung in der WPF eventuell eine sinnvolle Alternative dar.

Der Schlüssel zur Verbindung beider Welten

Die notwendigen Komponenten für Interop zwischen WPF und Windows Forms liegen in der mit dem .NET Framework 3.0 installierten Assembly *WindowsForms-Integration.dll*. Die Klassen *Windows-FormsHost* und *ElementHost* aus dem Namespace *System.Windows.Forms.Integration* bilden die zentralen Bausteine. Sie kapseln jeweils über eine *Child*-Eigenschaft das *UIElement/Control* aus der anderen Technologie (Abbildung 4). Sie enthalten die Funktionalität, um verschiedene Interop-Szenarien abzubilden. Neben der Nutzung eines Windows-Forms-Controls in einer WPF-Anwendung lassen sich aus einer WPF-Anwendung auch modale und nicht modale Windows-Forms-Formulare öffnen. Umgekehrt stehen dem Entwickler zur Nutzung der WPF-Features in bestehenden Windows-Forms-Anwendungen die gleichen Szenarien zur Verfügung (Tabelle 1).

WPF in Windows Forms

Eine Windows-Forms-Anwendung kann visuell nicht mit einer WPF-Anwendung mithalten. Darüber hinaus ist die Bedienung einer gut strukturierten WPF-Anwendung im Allgemeinen angenehmer und weicher. Erscheinen 3D-Effekte oder Animationen im ersten Moment noch als Spielerei, so kann dies in Zukunft – sinnvoll und an der richtigen Stelle eingesetzt – ein entscheidendes Kriterium für eine Applikation sein. Dank Interop können auch bereits existierende Windows-Forms-Anwendungen von WPF-Features profitieren. Zum Einsatz von WPF in einem Windows-Forms-Projekt werden die folgenden vier Assemblies referenziert:

- *WindowsFormsIntegration.dll*
- *PresentationFramework.dll*
- *PresentationCore.dll*
- *WindowsBase.dll*

Die erste Assembly enthält die Komponenten für Interop, die drei letzteren die Hauptkomponenten der WPF. Optional werden zu den Referenzen des Windows-Forms-Projekts zusätzliche Assemblies mit weiteren WPF-Elementen hinzugefügt. In folgendem Beispiel ist das die *My-WPFControls*-Assembly, die ein WPF-*UserControl* vom Typ *MyThreeDControl* enthält (Abbildung 5).

Im Code Editor beginnt die eigentliche Arbeit mit dem Erstellen eines Objekts des gewünschten WPF-Elements – dem *MyThreeDControl* – und eines Objekts der Klasse *ElementHost*. *ElementHost* ist von *Windows.Forms.Control* abgeleitet und bringt somit alle Voraussetzungen mit, um in einer Windows-Forms-Anwendung eingesetzt zu werden (Abbildung 4). Dem erstellten *ElementHost*-Objekt wird über die *Child*-Eigenschaft ein Objekt vom Typ *System.Windows.UIElement* zugewiesen – in diesem Fall das *MyThreeDControl*. Der *ElementHost* selbst wird der *Controls*-Collection des Formulars hinzugefügt und dann mit der *SetBounds*-Methode an der gewünschten Stelle positioniert. Gut platziert ist der Sourcecode im *Load*-Ereignis des Formulars (Listing 1). Der Windows-Forms-Designer von Visual Studio 2005 ist zur Einbindung eines *ElementHost* (noch)

Windows Forms	WPF
Hosten von WPF-UIElementen (*ElementHost*)	Hosten von Windows-Forms-Controls (*WindowsFormsHost*)
Popup modale/nicht modale WPF-Fenster	Popup modale/nicht modale Windows-Forms-Fenster
Gemeinsame Datenquellen	

Tabelle 1: Interop-Szenarien zwischen Windows Forms und WPF

© Software & Support Verlag GmbH

Abb. 6: Eine Windows-Forms-Anwendung mit einem WPF-User-Control mit 3D-Effekten

nicht geeignet. Doch der Nachfolger von Visual Studio (Codename „Orcas") wird mit Sicherheit mehr Möglichkeiten bieten.

Das in Listing 1 zur *Child*-Eigenschaft des *ElementHost* hinzugefügte *MyThreeDControl* beinhaltet ein dreidimensionales Element – siehe Abbildung 6. Über die Eigenschaft *AxisAngleRotation3D* gibt das *MyThreeDControl* ein gleichnamiges Objekt aus dem Namespace *System.Windows.Media.Media3D* zurück. Die *AxisAngleRotation3D* definiert wiederum eine *Angle*-Eigenschaft – sie ist vom Typ *double* und spiegelt die Rotation des 3D-Objekts in Grad wider. Mit den bereits im Windows-Forms-Projekt referenzierten Assemblies lassen sich natürlich die WPF-Features ebenfalls wie Animationen nutzen. Mit einer auf die *Angle*-Eigenschaft der *AxisAngleRotation3D* angewandten *DoubleAnimation* wird das 3D-Objekt im Event Handler einer *Windows.Forms.ComboBox* animiert (Listing 2).

Neben dem „Hosten" von WPF-Elementen in Windows Forms lassen sich auch modale und nicht-modale WPF-Dialoge aus Windows Forms öffnen (Tabelle 1). Dazu wird die *Show*- oder *ShowDialog*-Methode eines erstellten WPF-*Window*-Objekts aufgerufen. Dieser Aufruf unterscheidet sich nicht von dem in einer reinen WPF-Applikation. Zur korrekten Funktion eines nicht-modalen WPF-Dialogs ist vor dem Aufruf der *Show*-Methode aus Windows Forms die statische Methode *EnableModelessKeyboardInterop(Window window)* mit dem *Window*-Objekt als Übergabe-Parameter aufzurufen:

```
System.Windows.Window window =
                    new System.Windows.Window();
window.Content = new System.Windows.Controls
                                    .TextBox();
ElementHost.EnableModelessKeyboardInterop(window);
window.Show();
```

Intern wird dadurch ein Nachrichtenfilter in der Windows-Forms-Applikation erstellt, der alle Tastatur-Nachrichten an das WPF-Fenster weiterleitet, sobald dieses das aktive Fenster darstellt. Ohne diesen Filter empfängt das nicht-modale WPF-Fenster die Tastatur-Nachrichten nicht korrekt. Für modale WPF-Fenster ist der Aufruf von *EnableModelessKeyboardInterop* nicht erforderlich, was bereits aus dem Namen der Methode mit „Modeless" hervorgeht.

Windows Forms in der WPF

Wie bereits in der Einleitung erwähnt wurde, gibt es heute für die WPF noch nicht die Vielfalt an Controls, wie sie für Windows Forms besteht. Für bereits existierende Windows-Forms-Controls stellt die Interoperabilität von Windows Forms in WPF eine Alternative zur Neuentwicklung dar. Ist die Entscheidung für eine Integration von Windows Forms gefallen, erwartet den Entwickler ein nahezu identisches Vorgehen zu dem im vorherigen Abschnitt dargestellten Fall, der Implementierung von WPF in Windows Forms. Allerdings besteht jetzt die Wahl zwischen einer Implementierung in XAML oder in C#, wobei

Listing 1

```
...
private void Form1_Load(object sender, EventArgs e)
{
 myWPFElement = new MyWPFControls.MyThreeDControl();

 ElementHost host = new ElementHost();
 host.SetBounds(12, 12, 215, 174);
 host.Child = myWPFElement;

 this.Controls.Add(host);
 ...
}
...
```

Listing 2

```
private void cboGrad_SelectedIndexChanged(
                          object sender, EventArgs e)
{
  double from = myWPFElement.AxisAngleRotation3D
                                        .Angle;
  double to = (int)this.cboGrad.SelectedItem;
  DoubleAnimation ani = new DoubleAnimation(
              from, to, new System.Windows.Duration(
                      TimeSpan.FromSeconds(2)));

myWPFElement.AxisAngleRotation3D.BeginAnimation(
          AxisAngleRotation3D.AngleProperty, ani);
}
```


AUF DEN PUNKT GEBRACHT

Jetzt online abonnieren
dotnetmagazin.de
S&S

© Software & Support Verlag GmbH

XAML als Markup-Sprache C# natürlich nicht vollständig ersetzt. Wurde zuvor der *ElementHost* eingesetzt, kommt jetzt sein Pendant zum Zuge – die Klasse *WindowsFormsHost*. Im WPF-Projekt werden die folgenden Assemblies referenziert:

- *WindowsFormsIntegration.dll*
- *System.Windows.Forms.dll*

Für das Setzen bestimmter Eigenschaften in C# wie zum Beispiel die *Location*-Eigenschaft der *Windows.Forms.Control*-Klasse muss zum erfolgreichen Kompilieren zusätzlich die *System.Drawing.dll* referenziert werden. Mit *System.Windows.UIElement* als (indirekte) Basisklasse ist *WindowsFormsHost* „WPF-kompatibel" (Abbildung 4). Über die *Child*-Eigenschaft nimmt der *WindowsFormsHost* ein Windows-Forms-*Control* entgegen. Zum Verwenden von mehreren Windows-Forms-Controls in einem *WindowsFormsHost* wird ein Layout-Container wie zum Beispiel ein Windows-Forms-Panel genutzt:

```
WindowsFormsHost host = new WindowsFormsHost();
System.Windows.Forms.Panel pnl =
                    new System.Windows.Forms.Panel();
pnl.Controls.Add(txtUserName);
pnl.Controls.Add(txtPassword);
host.Child = pnl;
```

Der Umgang mit dem *WindowsFormsHost* ist weitestgehend analog zu dem mit dem *ElementHost*. Um Compiler-Schwierigkeiten aus dem Weg zu gehen, wird für Windows-Forms-Controls der voll qualifizierte Typ-Bezeichner oder ein Namespace-Alias verwendet. Die in Windows enthaltenen visuellen Stile werden von den Windows-Forms-Controls durch einen Aufruf der *EnableVisualStyles*-Methode der Windows-Forms-*Application*-Klasse unterstützt. Der Aufruf muss vor dem Erstellen der Windows-Forms-Controls erfolgen.

```
System.Windows.Forms.Application.EnableVisualStyles();
```

Im Gegensatz zur C#-Initialisierung eines Windows-Forms-*Control* in der Windows Presentation Foundation bietet XAML eine interessante Alternative. Um in XAML Windows-Forms-Controls zu verwenden, bedarf es eines Namespace-Mapping zwischen CLR-Namespace und XML-Namespace (*xmlns*).

```
xmlns:wf=»clr-namespace:System.Windows.
Forms;assembly=System.Windows.Forms»
```

Die Windows-Forms-Komponenten müssen in XAML mit dem frei wählbaren Prefix – in diesem Fall „wf" – verwendet werden. Dadurch erkennt der XAML-Prozessor, dass eine solche Komponente aus Windows Forms stammt. Im Folgenden wird eine *DataGridView* in XAML erstellt. Dazu

Listing 3

```
<Window x:Class="WinFormsInWPF_XamlAndCSharp
                                          .Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/
                                    xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/
                                            2006/xaml"
  xmlns:wf="clr-namespace:System.Windows
                  .Forms;assembly=System.Windows.Forms"
  Title="Windows Forms in WPF - XamlAndCSharp"
  Height="300"
  Width="300"
  Loaded="OnLoad">
 <Grid x:Name="myWPFGrid">
  <Grid.RowDefinitions>
   <RowDefinition/>
   <RowDefinition/>
  </Grid.RowDefinitions>
  <WindowsFormsHost Grid.Row="1">
   <wf:DataGridView x:Name="dgvCustomers">
    <wf:DataGridView.Dock>
     <wf:DockStyle>Fill</wf:DockStyle>
    </wf:DataGridView.Dock>
    <wf:DataGridView.SelectionMode>
     <wf:DataGridViewSelectionMode>
      FullRowSelect
     </wf:DataGridViewSelectionMode>
    </wf:DataGridView.SelectionMode>
   </wf:DataGridView>
  </WindowsFormsHost>
 </Grid>
</Window></Window>
```

Listing 4

```
...
using wf = System.Windows.Forms;

namespace WinFormsInWPF_XamlAndCSharp
{
 public partial class Window1 : System.Windows.Window
 {
  ...
  private void OnLoad(object sender, RoutedEventArgs e)
  {
   WindowsFormsHost host = new WindowsFormsHost();
   wf.DataGridView dgvCustomersClone =
                                new wf.DataGridView();
   dgvCustomersClone.Dock = wf.DockStyle.Fill;
   dgvCustomersClone.SelectionMode =
         wf.DataGridViewSelectionMode.FullRowSelect;

   host.Child = dgvCustomersClone;

   myWPFGrid.Children.Add(host);
   ...
  }
 }
}
```

Rund um den WindowsFormsHost

HwndHost als Basisklasse von WindowsFormsHost

WindowsFormsHost ist von der Klasse *System.Windows.Interop.HwndHost* abgeleitet. *HwndHost* stellt die Grundfunktionalität zur Nutzung von „Window-Handle"-basierten Technologien in WPF-Applikationen bereit. Für die Verwendung von Win32-Controls in WPF muss eine eigene Klasse von *HwndHost* abgeleitet und die abstrakten Methoden *BuildingWindowCore* und *DestroyWindowCore* überschrieben werden. Für Interop „Windows Forms in WPF" befindet sich mit der Klasse *WindowsFormsHost* bereits ein vollständig implementierter Subtyp im .NET Framework.

Einschränkungen des WindowsFormsHost

Ein WPF-*Window* wird bei der Erstellung in einen Top-Level „Window-Handle" gesetzt, der von allen darunter liegenden Elementen gemeinsam genutzt wird. Das heißt pro WPF-*Window* gibt es einen *Window*-Handle, und nicht wie in Windows Forms einen *Window*-Handle pro Control. Wird ein *WindowsFormsHost* einem WPF-*Window* hinzugefügt, so wird für den *WindowsFormsHost* ein zusätzlicher *Window*-Handle erstellt. Die WPF nutzt Win32, um diesen *Window*-Handle relativ zum WPF-*Window* zu positionieren. Aufgrund des separaten *Window*-Handle für den *WindowsFormsHost* resultieren einige Einschränkungen:

- Der *WindowsFormsHost* kann weder transformiert, skaliert noch rotiert werden
- Der *WindowsFormsHost* erscheint in der Z-Order ganz oben, über allen anderen WPF-Elementen im gleichen WPF-*Window*
- Transparenzeffekte werden für den *WindowsFormsHost* nicht unterstützt
- Ist die Maus über dem *WindowsFormsHost*, so erhält die WPF-Anwendung keine gerouteten *MouseEvents* und die WPF-Eigenschaft *IsMouseOver* ist *false*

Einen genauen Überblick gibt es in der MSDN-Dokumentation [1].

© Software & Support Verlag GmbH

wird ein <*WindowsFormsHost*>-Element eingefügt, das ein <*wf:DataGridView*>-Element enthält. Damit auch für C# der Zugriff auf die *DataGridView* möglich ist, wird auf dem <*wf:DataGridView*>-Element der Name *dgvCustomers* vergeben. Eigenschaften wie *Dock* und *SelectionMode* sind problemlos in XAML initialisierbar (Listing 3). Der Nachteil einer Implementierung der Windows-Forms-Komponente in XAML ist der fehlende IntelliSense-Support. Daher werden (noch) nicht eingefleischte XAML-Fans nach wie vor eine C#-Initialisierung bevorzugen (Listing 4).

Zum Öffnen von nicht-modalen Fenstern beinhaltet auch der *WindowsFormsHost* eine statische Methode zur Einrichtung eines Nachrichtenfilters. Der einmalige Aufruf der parameterlosen Methode *EnableWindowsFormsInterop* genügt, um einen Nachrichtenfilter in der WPF-Anwendung einzurichten, der entsprechende Nachrichten an das WinForms-Fenster weiterleitet.

PropertyMapping – WPF/Windows Forms

In einer WPF-Anwendung werden Eigenschaften, die auf höheren Leveln des Visual Tree gesetzt werden, an darunterliegende Elemente propagiert. Wird zum Beispiel in einem WPF-*Window* die *Background*-Eigenschaft gesetzt, wird diese Eigenschaft von der WPF an alle im Visual Tree darunterliegenden Elemente weitergeleitet. Auf diese Weise und natürlich auch auf direktem Wege werden auch Eigenschaften des *WindowsFormsHost* gesetzt. Die Eigenschaften des im *WindowsFormsHost* gekapselten Windows-Forms-Control werden jedoch nicht durch die WPF gesetzt. Der Grund dafür liegt in den verschiedenen Eigenschaftsmodellen von Windows Forms und WPF. Beispielsweise besitzen WPF-Elemente eine *Background*-Eigenschaft, während ein Windows-Forms-Control für die gleiche Funktionalität eine *BackColor*-Eigenschaft bereitstellt. Damit sich aber, wie bei dem vorherigem Beispiel aufgezeigt, ein Wechsel der Hintergrundfarbe auch auf das im *WindowsFormsHost* gekapselte Windows-Forms-Control auswirkt, muss der *WindowsFormsHost* selbst tätig werden. Dazu nutzt er ein Objekt der im Namespace *System.Windows.Forms.Integration* enthaltenen Klasse *PropertyMap*. Die *PropertyMap* definiert die Eigenschafts-Zuordnungen (*PropertyMappings*) zwischen WPF und Windows Forms. Sie besteht aus *Keys* und *Values*. Keys sind *Strings*, die den WPF-Eigenschaften entsprechen. Values sind *Delegates* vom Typ *PropertyTranslator*. Wird eine Eigenschaft auf dem *WindowsFormsHost* gesetzt, so wird der entsprechende *Delegate* in der *PropertyMap* gesucht und aufgerufen. Der *Delegate* zeigt typischerweise auf eine Methode, die die entsprechende Eigenschaft des gekapselten Controls setzt (Abbildung 7).

Die *PropertyMap* des *WindowsFormsHost* beinhaltet bereits vordefinierte Zuordnungen, wie beispielsweise *IsEnabled* zu *Enabled*, für die keine weitere Implementierung notwendig sind. Es lassen sich jedoch weitere Zuordnungen hinzufügen und bestehende entfernen, ersetzen oder erweitern. Über die *PropertyMap*-Eigenschaft des *WindowsFormsHost*-Objekts wird auf das *PropertyMap*-Objekt zugegriffen. So lässt sich beispielsweise die *FlowDirection*-Eigenschaft der WPF der *TextAlign*-Eigenschaft einer Windows-Forms-TextBox zuordnen. Wird die *FlowDirection*-Eigenschaft auf dem *WindowsFormsHost* geändert, so wird die *TextAlign*-Eigenschaft der *TextBox* gesetzt – zu sehen in Listing 5. Bestehende Eigenschaftszuordnungen lassen sich durch zusätzliche Handler-Methoden erweitern:

```
WindowsFormsHost host = new WindowsFormsHost();
...
host.PropertyMap["IsEnabled"] +=
              new PropertyTranslator(OnEnabledChange);
```

Analog zum *WindowsFormsHost* führt auch der *ElementHost* die Zuordnung von Eigenschaften durch. Auch der *ElementHost* hat in seiner *PropertyMap* bereits vordefinierte Zuordnungen der gängigsten Eigenschaften. Eine Übersicht der vordefinierten Zuordnungen im *WindowsFormsHost* und *ElementHost* befindet sich unter [1].

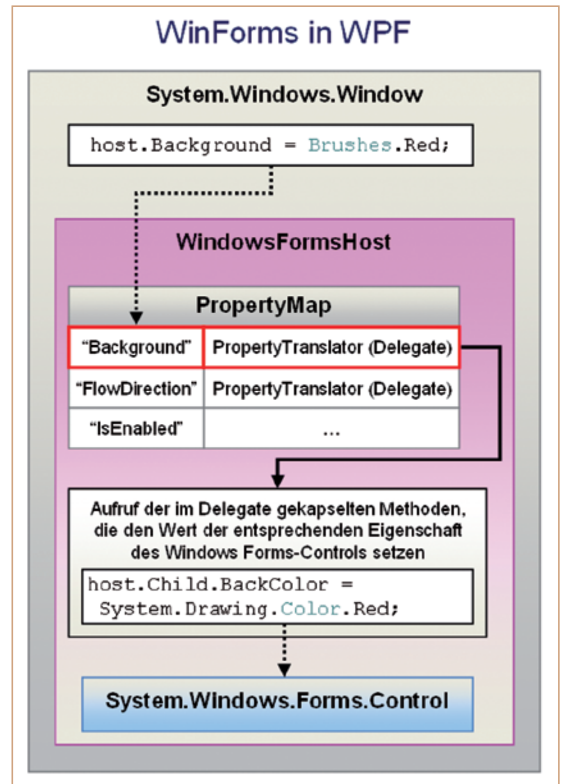


Abb. 7: Zuordnung von WPF- zu Windows-Forms-Eigenschaften („PropertyMapping“) im WindowsFormsHost

Gemeinsame Datenquellen

Die Nutzung einer gemeinsamen Datenquelle ist in vielen Hybrid-Applikation unverzichtbar. Nachfolgend wird eine WPF-Applikation erstellt, die verschiedene WPF-Elemente und zwei Windows-Forms-*DataGridViews* mit einer gemeinsamen Datenquelle einsetzt (Abbildung 8). Die Datenquelle besteht aus zwei in einer Master-Detail-Beziehung stehenden Tabellen – *Customers* und *Orders* – die aus einem XML-File in das *CustomerDataSet* geladen werden. In XAML werden für eine Master-Detail-Ansicht zwei *WindowsFormsHost* mit je einer *DataGridView* nach dem Schema in Listing 3 erstellt – *dgvCustomers* und *dgvOrders*. Die gemeinsame Datenquelle wird durch eine Windows-Forms-*BindingSource* definiert, die das *CustomerDataSet* als *DataSource* und die *Customers*-Tabelle als *DataMember* hat. Beide *DataGridView*-Objekte zeigen durch folgenden Code, der in einer Windows-Forms-Anwendung gleich aussehen würde, bereits die gewünschte Funktionalität.

```
bindingSource.DataSource = dataSet;
bindingSource.DataMember =
```

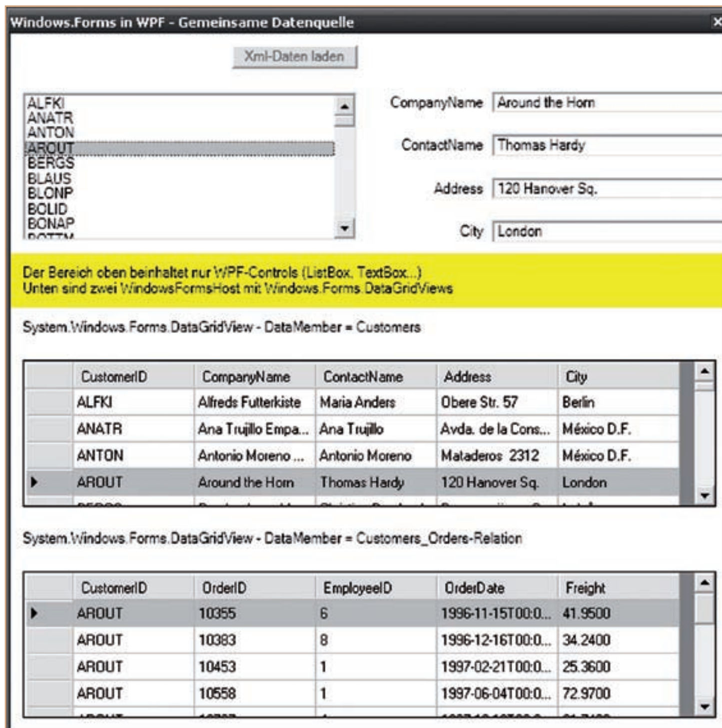


Abb. 8: WPF-Anwendung mit zwei DataGridViews von Windows Forms

```
                        dataSet.Customers.TableName;
dgvCustomers.DataSource = bindingSource;
dgvOrders.DataSource = bindingSource;
dgvOrders.DataMember = "Customers_Orders";
```

Um die WPF-Elemente vom Typ *ListBox* und *Textbox* ebenfalls mit der Datenquelle zu verbinden, werden diese mittels Data Binding an die gewünschte Tabellenspalte „gebunden" (Listing 6). Wird der *DataContext*-Eigenschaft des WPF-*Window* und der *ItemsSource*-Eigenschaft der *ListBox* die *BindingSource* zugewiesen, zeigen auch die WPF-Elemente die Daten korrekt an.

Zur Synchronisation von WPF und Windows Forms ist noch ein wenig Nachhilfe notwendig. Wird eine *CustomerID* in der WPF-*ListBox* ausgewählt, synchronisieren sich zwar alle WPF-*Textbox*-Objekte korrekt, aber die *DataGridView*-Objekte zeigen keine Reaktion. Wird hingegen in der *DataGridView* ein anderer

Listing 5

```
...
using System.Windows.Forms.Integration;
using wf = System.Windows.Forms;

namespace WinFormsInWPF_PropertyMap
{
 public partial class Window1 : System.Windows.Window
 {
  public Window1()
  {
   InitializeComponent();
  }
  private void OnLoad(object sender, RoutedEventArgs e)
  {
   WindowsFormsHost host = new WindowsFormsHost();
   wf.TextBox txt = new wf.TextBox();
   txt.Text = "Eine simple Windows.Forms.TextBox";
   host.Child = txt;
   myWPFGrid.Children.Add(host);

   // Ersetzen des vordefinierten FlowDirection Mappings
   host.PropertyMap.Remove("FlowDirection");
   host.PropertyMap.Add("FlowDirection",
     new PropertyTranslator(OnFlowDirectionChange));
  }

  private void OnFlowDirectionChange(
                 object h, string propertyName, object value)
  {
   System.Windows.FlowDirection f = (
                     System.Windows.FlowDirection)value;

   WindowsFormsHost host = h as WindowsFormsHost;
   wf.TextBox txt = host.Child as wf.TextBox;

   txt.TextAlign = System.Windows.FlowDirection
                              .LeftToRight == f ?
     wf.HorizontalAlignment.Left :
     wf.HorizontalAlignment.Right;
  }
  ...
}
```

Listing 6

```
<Window x:Class="WinFormsInWPF_Data.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"
  Title="Windows.Forms in WPF - Gemeinsame Datenquelle"
  Height="600"
  Width="600"
  WindowStyle="ToolWindow"
  Loaded="OnLoad">
 <Grid>
  <Grid.Resources>
   ...

   <DataTemplate x:Key="ListItemsTemplate">
    <StackPanel Orientation="Horizontal">
     <TextBlock Text="{Binding Path=CustomerID}"/>
    </StackPanel>
   </DataTemplate>
  </Grid.Resources>

  ...

  <ListBox Margin="10" Grid.Row="1" IsSynchronizedWithCurrentItem=
       "True" ItemTemplate="{StaticResource ListItemsTemplate}" x:Name="listBox1"/>

  <Grid Grid.Row="1" Grid.Column="1">
   <TextBox Grid.Column="1" Text="{Binding Path=CompanyName}"/>
   <TextBlock Grid.Row="1"Text=»ContactName» Style=»{StaticResource txtStyle1}"/>
   <TextBox Grid.Row="1" Grid.Column="1" Text="{Binding Path=ContactName}"/>
   <TextBlock Grid.Row="2"Text="Address" Style="{StaticResource txtStyle1}"/>
   <TextBox Grid.Row="2" Grid.Column="1" Text="{Binding Path=Address}"/>
   <TextBlock Grid.Row="3"Text="City" Style="{StaticResource txtStyle1}"/>
   <TextBox Grid.Row="3" Grid.Column="1" Text="{Binding Path=City}"/>
  </Grid>
  ...
 </Grid>
</Window>
```


© Software & Support Verlag GmbH

„Customer" ausgewählt, zeigt die zweite *DataGridView* die richtigen „Orders" an, aber die WPF-Elemente bleiben auf dem bereits zuvor angezeigten „Customer" stehen. Dies hat folgenden Grund: Das Data Binding der WPF nutzt ein aus der *BindingSource* erstelltes Objekt vom Typ *ICollectionView* als Zeiger-Manager für den aktuellen Record einer Collection. Die beiden *DataGridView*-Objekte hingegen verwenden die *Position*-Eigenschaft der *BindingSource* als Zeiger. Der Zeiger des *BindingSource*-Objekts und der des Objekts vom Typ *ICollectionView* werden nicht miteinander synchronisiert. Zwei passende Event Handler lösen dieses Problem und sorgen für die entsprechende Synchronisation (Listing 7).

Wie geht's weiter?

Mit den Komponenten aus dem Namespace *System.Windows.Forms.Integration* lassen sich beide Generationen auf einfache Art und Weise vereinen. Auch nach dem Erscheinen der WPF wird Windows Forms bei der Entwicklung von Windows-Applikationen weiterhin eine wichtige Rolle spielen. Im Gegensatz zur WPF stehen Windows Forms heute (noch) die leistungsfähigeren Controls und Dialogfenster zur Verfügung. Dagegen bietet die WPF unter anderem für grafische Herausforderungen eindeutig den besseren Ansatz. Das dosierte Mischen beider Welten wird in der Praxis aufgrund bestehender Anwendungen nicht unüblich sein. Der Entwickler kann dabei mit *WindowsFormsHost* und *ElementHost* auf zwei Klassen zugreifen, die im Hintergrund ganz heimlich das ganze Management mit *MessageLoops*, *KeyEvents* etc. steuern. Was die aktuellen Designer in Visual Studio 2005 betrifft, so ist bezüglich Interop noch kein wirkliches Rapid Application Development möglich. Der Windows-Forms-Designer als auch die aktuelle Version des WPF-Designers sind noch nicht in der Lage, die in den Hosts enthaltenen Controls korrekt darzustellen. Damit ist wohl klar, was bei allen Interop-Begeisterten mit auf der Wunschliste der Designer-Features von „Visual Studio 2008" (Codename „Orcas") steht.

Listing 7

```
...
using wf = System.Windows.Forms;

namespace WinFormsInWPF_Data
{
 public partial class Window1 : System.Windows.Window
 {
  private CustomerDataSet dataSet;
  private wf.BindingSource bindingSource;

  public Window1()
  {
   InitializeComponent();
  }

  private void OnLoad(object sender, RoutedEventArgs e)
  {
   dataSet = new CustomerDataSet();

   bindingSource = new wf.BindingSource();

   bindingSource.DataSource = dataSet;
   bindingSource.DataMember =
                          dataSet.Customers.TableName;

   this.DataContext = bindingSource;

   this.listBox1.ItemsSource = bindingSource;

   dgvCustomers.DataSource = bindingSource;

   dgvOrders.DataSource = bindingSource;
   dgvOrders.DataMember = "Customers_Orders";

   // EventHandler um Windows.Forms mit WPF synchron
   //zu halten
   ICollectionView cv = CollectionViewSource
                          .GetDefaultView(bindingSource);
   cv.CurrentChanged += new EventHandler(
                                    cv_CurrentChanged);

   // EventHandler um WPF mit Windows.Forms synchron
   // zu halten.
   bindingSource.PositionChanged +=
               new EventHandler(bs_PositionChanged);
  }

  private void cv_CurrentChanged(object sender,
                                          EventArgs e)
  {
   ICollectionView cv = sender as ICollectionView;
   bindingSource.Position = cv.CurrentPosition;
  }

  private void bs_PositionChanged(object sender,
                                          EventArgs e)
  {
   ICollectionView cv = CollectionViewSource
                          .GetDefaultView(bindingSource);
   if (cv.CurrentPosition != bindingSource.Position)
    cv.MoveCurrentToPosition(bindingSource.Position);
  }
  ...
 }
}
```

Links & Literatur

[1] Migration und Interoperabilität mit WPF und Windows-Applikationen: msdn2.microsoft.com/en-us/library/ms753178.aspx

[2] Windows Forms / WPF Interoperabilität FAQ: www.windowsforms.net/Samples/Go%20To%20Market/InterOP/Crossbow%20FAQ.doc

[3] MSDN TV: Mike Henderlight zum Thema Windows Forms und WPF Interoperabilität: msdn.microsoft.com/msdntv/episode.aspx?xml=episodes/en/20060216crossbowmh/manifest.xml

[4] Dirk Frischalowski: Offen für alles – Das Zusammenspiel der WPF mit Windows Forms, ActiveX und MFC, in: *dot.net magazin* 1/2.07

Thomas Huber studierte Wirtschaftsinformatik und arbeitet als Consultant bei der Trivadis AG. Neben seiner Trainer-Tätigkeit für Microsoft-Technologien arbeitet er als Entwickler in den Bereichen .NET (MCPD), Java und Oracle. Mit dem .NET Framework 3.0 hat er sich seit dem Erscheinen der ersten Previews auseinandergesetzt. Sie erreichen ihn unter thomas.huber@trivadis.com.

Christoph Pletz arbeitet als Senior Consultant für Microsoft-Technologien bei der Trivadis AG. Seine Tätigkeit erstreckt sich von Software-Entwicklung über die Erstellung von Kursen und Seminaren bis zur Architektur-Beratung. Er ist Sprecher bei Entwickler-Konferenzen und war auch bereits auf der BASTA! zu sehen. Er ist unter christoph.pletz@trivadis.com erreichbar.

trivadis
makes IT easier.

Trivadis AG
Elisabethenanlage 9
CH-4051 Basel
Tel.: +41-61-279 97 55
Fax: +41-61-279 97 56
E-Mail: info-basel@trivadis.com
www.trivadis.com