7/8.07

www.dotnet-magazin.de

Deutschland 8,50 € Österreich 9,60 € Schweiz 16,50 sFr

S&S

dot.net magazin

dot.net magazin

.NET, Visual Studio & More

mit CD →

LINQ

Vier Buchstaben, die die (Datenbank-)Welt verändern werden

LINQ to SQL

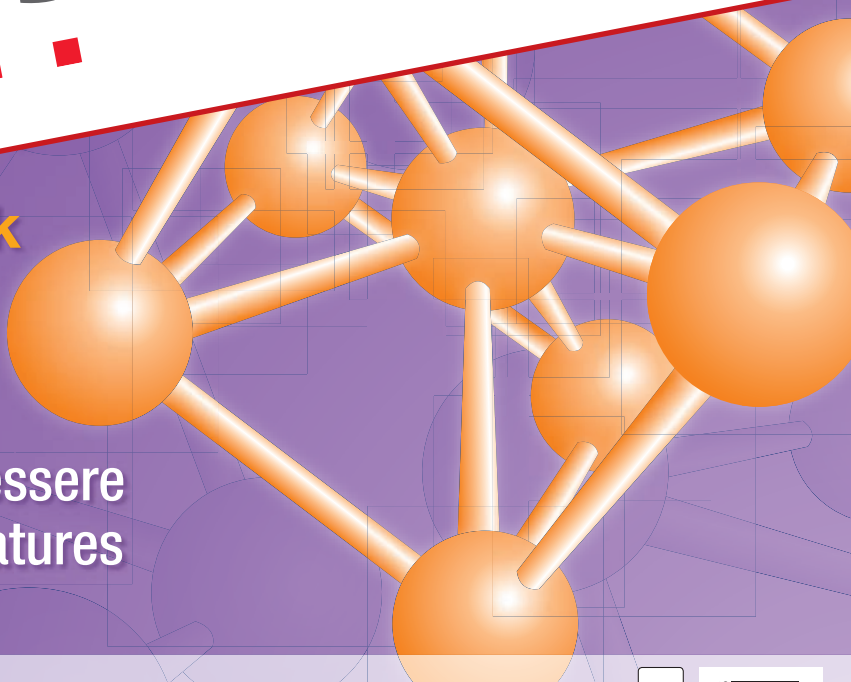
Endlich da: ORM-Funkti…
von Micro…

…icro Framework

Kleiner geht es nicht mehr

Enterprise Library 3.0

Neue Applikationsblöcke, bessere VS-Integration, .NET 3.0-Features



CD-Inhalt: Testversionen, .NET Tools & More

- Advantage Database Server 8.1
- NetAdvantage for .NET 2007 Vol.1
- MQ Batch Toolkit v1.2.0
- {smartassembly} 2.1.2627
- Spices.NET 5.2
- TurboDB Managed
- AxCMS.net
- DataBlock 1.3.1
- db4o 6.1
- NAnt 0.85
- .NET 2.0 Provider for Firebird
- Pidgin 2.0.0
- Py2Exe 0.6.6

u.v.a.m.

Datenträger enthält nur Lehr- oder Infoprogramme

4 195809 808509 08

Gewünschte Abhängigkeit

Die Dependency Properties bei der WPF

von Thomas Huber

Die in .NET 3.0 neu eingeführten Dependency Properties sind in der Windows Presentation Foundation (WPF) die Grundlage für Merkmale wie Animationen, Data Binding oder Styles. Reguläre Properties lassen sich mit diesen von der WPF zur Verfügung gestellten „Services“ nicht verwenden und erreichen diese Funktionalität nur durch zusätzlichen Code. Obwohl die Implementierung von Dependency Properties meist nur beim Erstellen eigener WPF-Controls benötigt wird, ist ein Verständnis ihrer Funktionsweise für einen professionellen Einsatz der WPF ein absolutes Muss.

In .NET 3.0 werden Eigenschaften wie gewohnt über die klassischen CLR-Wrapper (Common Language Runtime) implementiert. Jedoch steckt dahinter in vielen Fällen nicht, wie meist üblich, nur ein privates Feld, sondern der Zugriff auf eine so genannte *Dependency Property*. Dependency Properties werden in der *Property Engine* der WPF registriert. Sie besitzen einen integrierten Benachrichtigungsmechanismus und ermöglichen damit der WPF das Überwachen ihrer Werte. Darüber hinaus berechnet die WPF bzw. die Programmlogik der WPF, die als Property Engine bezeichnet wird, den Wert einer Dependency Property „abhängig“ – daher der Name „Dependency Property“ – von anderen Zuständen in der Anwendung und im System. So beeinflussen Animationen, Data Bindings, Styles oder hierarchisch im Element Tree höher liegende Elemente den Wert einer Dependency Property. Gibt es mehrere Quellen, die den Wert einer Dependency Property ändern wollen, wird dies durch die WPF priorisiert. Beispielsweise erhält eine Animation beim Setzen des Wertes einer Dependency Property Vorrang vor einem Data Binding. Da all die in Tabelle 1 dargestellten Services mit klassischen Properties nicht nutzbar sind – für eine Animation wäre z.B. eine zusätzliche Implementierung mit einem Timer notwendig – stellt sich an dieser Stelle zu Recht die Frage, ob jetzt in Zukunft jede Eigenschaft als Dependency Property implementiert wird. Die Antwort lautet Nein. Eigenschaften werden üblicherweise nur dann als Dependency Property implementiert, wenn sie für die Verwendung der Services in Tabelle 1 vorgesehen sind.

Im weiteren Verlauf wird gezeigt, wie Dependency Properties funktionieren und wie sie implementiert werden. Dabei wird speziell auf die pro Typ definierten Meta-

kurz & bündig

Inhalt

Ein Überblick über die Implementierung der in .NET 3.0 neu eingeführten *Dependency Properties* und wie sie mithilfe von Metadaten und Callbacks mit weiterer Programmlogik verbunden werden – darüber hinaus werden die *Attached Properties* vorgestellt

Zusammenfassung

Dependency Properties bilden in der WPF u.a. die Grundlage für Animationen, Data Bindings oder Styles. Sie wurden mit .NET 3.0 eingeführt und erweitern die klassischen Properties. Für einen Einsatz der WPF ist für .NET-Entwickler ein Grundverständnis der Dependency Properties notwendig

Quellcode (C#)

Diverse Projektmappen

Quellcode auf CD

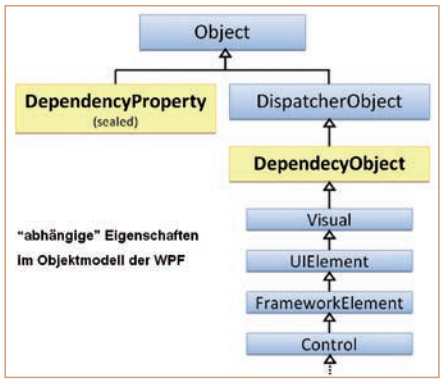


Abb. 1: WPF-Objektmodell mit DependencyObject und DependencyProperty

daten und Callbacks eingegangen. Auch ein Blick auf die so genannten *Attached Properties* – also Eigenschaften, die in einer Klasse erstellt und auf Objekten anderer Klassen gesetzt werden – kommt in diesem Artikel nicht zu kurz.

DependencyObject und DependencyProperty

Mit den Klassen *DependencyObject* und *DependencyProperty* stehen im Namespace *System.Windows* die beiden Keyplayer für die Funktionalität der Dependency Properties bereit. Das Zusammenspiel von *DependencyObject* und *DependencyProperty* wird im Folgenden an einem kleinen Beispiel aufgezeigt. Dazu versucht die Klasse *SimpleLabel* die Dependency Property *FontSize* der Klasse *Control* (Namespace: *System.Windows.Controls*) nachzubauen. Klassisch wird eine Eigenschaft wie *FontSize* vom Typ *double* mithilfe eines privaten Feldes und eines CLR-Wrappers implementiert (siehe Listing 1). Eine Dependency Property – wie die *FontSize* der Klasse *Control* – besteht jedoch aus einem öffentlich, statischen Feld vom Typ *DependencyProperty* und einem CLR-Wrapper (siehe Listing 2).

Über die Überladung der statischen Methode *Register* der Klasse *DependencyProperty* wird in Listing 2 das statische Feld *FontSizeProperty* vom Typ *DependencyProperty* initialisiert. Dabei werden der *Register*-Methode der Eigenschaftsname, der Typ des Rückgabewerts als auch der Typ, der die Dependency Property besitzt, übergeben. Das statische Feld zeigt jetzt auf ein in der WPF registriertes *DependencyProperty*-Objekt. Dieses Objekt stellt den Schlüssel für den eigentlichen Eigenschaftswert dar. Dies erklärt, dass das *FontSizeProperty*-Feld statisch vorliegt. Zum Zugriff auf den Wert werden die aus *DependencyObject* – wovon *FrameworkElement* und damit auch *SimpleLabel* abgeleitet sind (Abbildung 1) – geerbten, öffentlichen Instanz-Methoden *GetValue* und *SetValue* verwendet. Ein CLR-Wrapper kapselt in den *get*- und *set*-Accessoren den Aufruf dieser Methoden.

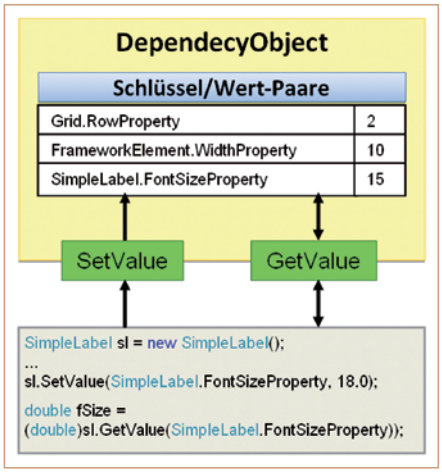


Abb. 2: Ein DependencyObject-Objekt, wie das SimpleLabel, verwaltet die gesetzten Dependency Properties vereinfacht gesehen in Form von Schlüssel/Wert-Paaren

DependencyObject definiert die Logik für das Speichern und Verwalten von Dependency Properties. Intern wird dazu, vereinfacht betrachtet, eine Collection von Schlüssel/Wert-Paaren verwendet, wobei die Schlüssel konkrete Objekte vom Typ *DependencyProperty* sind. Damit also ein Objekt Werte einer Dependency Property speichern und verwalten kann, muss es zwingend vom Typ *DependencyObject* sein. Daher steht diese Klasse in der Klassenhierarchie der WPF ganz weit oben (Abbildung 1). Auf ein *DependencyProperty*-Objekt zeigt das in *SimpleLabel* implementierte statische Feld *FontSizeProperty*. Dieses Feld lässt sich somit als „Schlüssel" in *GetValue* und *SetValue* verwenden (Abbildung 2).

Durch den klassischen CLR-Wrapper, der den Aufruf von *GetValue* und *SetValue* kapselt, lässt sich die *FontSize* in *Simple-*

Listing 1

Klassische Implementierung der FontSize-Property

```
public class SimpleLabel:FrameworkElement
{
 private double fontSize = 11;

 public double FontSize
 {
  get { return this.fontSize; }
  set { this.fontSize = value;}
 }
 ...
}
```

Listing 2

Implementierung als Dependency Property

```
Public class SimpleLabel: FrameworkElement
{
 public static readonly DependencyProperty
                                    FontSizeProperty
  = DependencyProperty.Register("FontSize", typeof(
                        double), typeof(SimpleLabel));

 public double FontSize
 {
  get { return (double)GetValue(FontSizeProperty); }
  set { SetValue(FontSizeProperty, value); }
 }
 ...
}
```

Feature	Beschreibung
Animationen	Der Wert jeder Dependency Property kann animiert werden.
Metadaten	Zusammen mit dem Typ einer Dependency Property werden Metadaten definiert. Unter anderem ist mithilfe der Metadaten ein Default-Wert für eine Dependency Property definierbar.
Expressions	Expressions ermöglichen, dass der Wert einer Dependency Property zur Laufzeit berechnet wird.
Data-Binding	Das Data Binding ist in der WPF über Expressions implementiert. Die Ziel-Eigenschaft, die an den Wert der Source-Eigenschaft gebunden wird, muss eine Dependency Property sein.
Styles	Die Werte einer oder mehrerer Dependency Properties auf einem Control können über Styles gesetzt werden
Vererbung	Der Wert einer Dependency Property ist – wenn in den Metadaten entsprechend angegeben – über den Element Tree vererbbar.

Tabelle 1: Auf Dependency Properties aufbauende Services der WPF

Label wie eine gewöhnliche Property verwenden und es fällt nicht auf, dass dahinter eine Dependency Property steckt:

```
SimpleLabel sl = new SimpleLabel();
sl.FontSize = 18.0
```

Aus C#-Sicht ist der CLR-Wrapper optional, da auf einem *DependencyObject* wie *SimpleLabel* auch ein direkter Aufruf von *GetValue* und *SetValue* möglich ist. Wird die *FontSize* des *SimpleLabel* in XAML gesetzt, setzt der XAML-Parser allerdings die Existenz des CLR-Wrappers voraus. Jedoch ruft die WPF zur Laufzeit für in XAML definierte Zugriffe auf Dependency Properties direkt *SetValue* und *GetValue* auf. Daher ist es wichtig, dass die CLR-Wrapper für Dependency Properties außer dem Aufruf von *SetValue* und *GetValue* keinerlei Logik enthalten. Ist zusätzliche Logik notwendig, so werden dafür die später beschriebenen Metadaten und Callbacks verwendet.

Data Binding und Animation

Die in *SimpleLabel* als Dependency Property implementierte *FontSize* hat gegenüber der klassischen Property aus Listing 1 jetzt den Vorteil, dass sie mit den Services der WPF wie Data Binding und Animationen verwendet werden kann. Zum Test wird in der Klasse *SimpleLabel* die aus *UIElement* geerbte Methode *OnRender* überschrieben, um darin ein Rechteck und Text mit der aktuellen *FontSize* zu „zeichnen" (Listing 3).

In einem kleinen WPF-Windows-Projekt wird das *SimpleLabel* einem *Grid* hinzugefügt. Ein *EventTrigger* deklariert für das *MouseDown*-Event eine Animation der *FontSize* vom Wert *20* zum Wert *30* und zurück. Neben dem *SimpleLabel* wird eine *TextBox* hinzugefügt, deren Text an die *FontSize* des *SimpleLabel* „gebunden" wird (Listing 4).

Wird die Anwendung gestartet und auf das *SimpleLabel* geklickt, so startet die Animation und die *TextBox* ändert ihren Wert stetig und gelangt nach zwei Sekunden beim Wert *30* an. Der Text des *SimpleLabel* ändert jedoch seine Grösse als auch die Anzeige der *FontSize* nicht. Er bleibt weiterhin auf dem Wert *20* stehen (Abbildung 3). Wie der Wert der *TextBox* zeigt, erfüllen Animation und Data Binding zwar ihre Aufgabe, das *SimpleLabel* wird allerdings nicht neu gezeichnet. Um bei einer Änderung der *FontSize* eine Neuzeichnung (Aufruf *OnRender*) auszulösen, ist in der WPF kein zusätzlicher Code erforderlich. Stattdessen wird die *FontSizeProperty* bei ihrer Registrierung mit Metadaten ausgestattet, die diese Information enthalten.

Abb. 3: Animation und Data Binding funktionieren, jedoch wird das SimpleLabel bei einer Änderung der FontSize nicht neu gerendert

Definition von Metadaten

Für eine Dependency Property lassen sich per Klasse Metadaten definieren. Bemerkt die „Property Engine" der WPF eine Änderung einer Dependency Property, so werden die für den Typ geltenden Metadaten ausgelesen und bestimmte Aktionen aufgrund dieser Metadaten ausgelöst.

Zur Definition von Metadaten wird der Methode *DependencyProperty.Register* ein *PropertyMetadata*-Objekt übergeben. Wird kein *PropertyMetadata*-Objekt übergeben – wie es beim Registrieren der *FontSizeProperty* in *SimpleLabel* in Listing 2 der Fall ist – so verwendet die WPF Default-Metadaten. Neben der *Register*-Methode definiert die Klasse *DependencyProperty* mit den Instanz-Methoden *MetadataOverride* und *AddOwner* weitere Möglichkeiten zum Setzen der Metadaten, doch dazu

Listing 3

Im Label wird Text mit der FontSize gezeichnet

```
public class SimpleLabel:FrameworkElement
{
 ...
 protected override void OnRender(DrawingContext ctx)
 {
  base.OnRender(ctx);

  ctx.DrawRectangle(Brushes.LightBlue, null,
                                new Rect(this.RenderSize));

  FormattedText txt =
   new FormattedText("SimpleLabel (
                                Click to animate)\nFontSize: "
    + string.Format("{0:0.0}", FontSize)
    , CultureInfo.CurrentCulture
    , FlowDirection.LeftToRight
    , new Typeface("Arial")
    , this.FontSize,
    Brushes.Black);

  ctx.DrawText(txt,new Point(0,0));
 }
}
```

Listing 4

Die FontSize wird über einen EventTrigger animiert

```
<Window x:Class="DependencyPropertyInWPF_
                                SimpleLabel.Window1"
 xmlns="http://schemas.microsoft.com/winfx/2006/
                                xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/
                                2006/xaml"
 xmlns:local="clr-namespace:
            DependencyPropertyInWPF_SimpleLabel"
 Title="DependencyPropertyInWPF_
            SimpleLabel" Width="500" Height="200"
  Background="GhostWhite" WindowStyle="ToolWindow"
 >
<Grid>
 <Grid.RowDefinitions>
  <RowDefinition/>
  <RowDefinition/>
 </Grid.RowDefinitions>
 <local:SimpleLabel x:Name="sl"
        FontSize="20"
        Margin="5">
  <local:SimpleLabel.Triggers>
   <EventTrigger RoutedEvent="FrameworkElement
                                .MouseDown">
    <EventTrigger.Actions>
     <BeginStoryboard>
      <Storyboard>
       <DoubleAnimation Storyboard.TargetName="sl"
            Storyboard.TargetProperty="FontSize"
            From="20"
            To="30"
            Duration="0:0:2"
            AutoReverse="true"
            />
      </Storyboard>
     </BeginStoryboard>
    </EventTrigger.Actions>
   </EventTrigger>
  </local:SimpleLabel.Triggers>
 </local:SimpleLabel>
 <TextBox Text="{Binding ElementName=sl,Path=
                                FontSize}"
    Grid.Row="1"
    Margin="5" Height="50"/>
</Grid>
</Window>
```

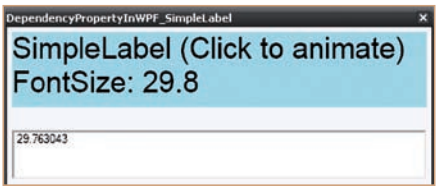


Abb. 4: Animation der FontSize – dank Metadaten wird bei jeder Änderung ein Rendering des SimpleLabel ausgelöst

später mehr. Mit einem *PropertyMetadata*-Objekt lassen sich Callbacks und ein Default-Wert definieren. So lässt sich z.B. für die *FontSize* des *SimpleLabel* ein Default-Wert von 11.0 festlegen:

```
public static readonly DependencyProperty FontSizeProperty
= DependencyProperty.Register("FontSize"
 ,typeof(double)
 ,typeof(SimpleLabel)
 ,new PropertyMetadata(11.0));
```

Wird auf einem *SimpleLabel*-Objekt die *FontSize*-Eigenschaft nicht explizit gesetzt, so gibt ein Aufruf von *GetValue* 11.0 zurück. Wurde die FontSize gesetzt, lässt sich der gesetzte Wert zur Laufzeit durch Aufruf der in *DependencyObject* definierten Methode *ClearValue* zurücksetzen:

```
SimpleLabel sl = new SimpleLabel();
sl.SetValue(SimpleLabel.FontSizeProperty,15);
sl.ClearValue(SimpleLabel.FontSizeProperty);
double fSize = (double)sl.GetValue(SimpleLabel
                                   .FontSizeProperty);
```

Listing 5

FontSize wird mit Metadaten im statischen Konstruktor registriert

```
public class SimpleLabel:FrameworkElement
{
 public static readonly DependencyProperty
                                   FontSizeProperty;

 static SimpleLabel()
 {
  FrameworkPropertyMetadata meta =
                    new FrameworkPropertyMetadata();

  meta.DefaultValue = 11.0;
  meta.AffectsRender = true;

  FontSizeProperty =
   DependencyProperty.Register("FontSize"
             ,typeof(double)
             ,typeof(SimpleLabel)
             ,meta);
 }
 ...
}
```

Die Variable *fSize* in obigem Codeausschnitt enthält jetzt den Default-Wert 11 der *FontSizeProperty*. Die Windows Presentation Foundation stellt für die Definition von Metadaten einen Subtyp von *PropertyMetadata* bereit. *FrameworkPropertyMetadata* bietet neben einem Default-Wert und den noch zu beschreibenden Callbacks diverse Einstellungsmöglichkeiten für Benutzeroberflächen. Unter anderem lässt sich damit für die *FontSizeProperty* des *SimpleLabel* neben dem Default-Wert festlegen, dass eine Änderung der Dependency Property ein Rendering verursacht (*AffectsRender*) Zur Übersicht wird die *FontSizeProperty* nicht mehr bei der Deklaration, sondern im statischen Konstruktor initialisiert – veranschaulicht in Listing 5.

An Stelle der Initialisierung im statischen Konstruktor ist nach wie vor eine Initialisierung direkt bei der Deklaration des *FontSizeProperty*-Feldes möglich. Dazu werden im Konstruktor der Klasse *FrameworkPropertyMetadata* die gewünschten Eistellungen mit der Enumeration *FrameworkPropertyMetadataOptions* angeben (Listing 6). Die Enumeration definiert das *Flags*-Attribut, mehrere Einstellungen

Listing 6

FontSize wird mit Metadaten registriert

```
public class SimpleLabel:FrameworkElement
{
 public static readonly DependencyProperty
                                   FontSizeProperty=
   DependencyProperty.Register("FontSize"
   ,typeof(double)
   ,typeof(SimpleLabel)
   new FrameworkPropertyMetadata(11.0
    ,FrameworkPropertyMetadataOptions.AffectsRender),
 );
 ...
}
```

Listing 7

Eine abgeleitete Klasse überschreibt Metadaten

```
public class SubLabel:SimpleLabel
{
 static SubLabel()
 {
  SimpleLabel.FontSizeProperty.OverrideMetadata(
   typeof(SubElement)
  ,new System.Windows.FrameworkPropertyMetadata(24.0
   ,FrameworkPropertyMetadataOptions.AffectsRender));
 }
 ...
}
```

lassen sich somit mit dem bitweisen *OR* verknüpfen.

Mit der Implementierung der Metadaten macht ein weiterer Test der Anwendung Sinn. Wird zur Laufzeit auf das *SimpleLabel* geklickt, wird die *FontSize* im *SimpleLabel* jetzt auch für das Auge sichtbar animiert. Die WPF liest die Metadaten der *FontSizeProperty* für den Typ *SimpleLabel* aus und ruft aufgrund der darin definierten Informationen (*AffectsRender*) bei jeder Änderung die Methode *OnRender* in der Klasse *SimpleLabel* auf.

Metadaten überschreiben

Metadaten werden von der WPF per Typ verwaltet. In obigem Beispiel wurde der statischen *DependencyProperty.Register*-Methode *SimpleLabel* als Besitzer-*Type* übergeben. Zu *SimpleLabel* gehören damit auch die übergebenen Metadaten. Für einen Subtyp von *SimpleLabel* besteht die Möglichkeit, diese Metadaten zu überschreiben. Dies geschieht im statischen Konstruktor durch Aufruf der in *DependencyProperty* definierten Instanz-Methode *OverrideMetadata*. So definiert die von *SimpleLabel* abgeleitete Klasse *SubLabel* für die *FontSize* einen Default-Wert von 24.0 (Listing 7).

Als Schlüssel für *GetValue* und *SetValue* lassen sich *SimpleLabel.FontSizeProperty* als auch *SubLabel.FontSizeProperty* verwenden, beide zeigen auf das gleiche *DependencyProperty*-Objekt:

```
SubLabel sub = new SubLabel();
Console.WriteLine(sub.GetValue(SimpleLabel
                                   .FontSizeProperty));
 // Output: 24
Console.WriteLine(sub.GetValue(SubLabel
                                   .FontSizeProperty));
 // Output: 24
```

Die Rolle der Callbacks

Für die in *SimpleLabel* implementierte *FontSize* bedarf es noch einer Validierung. Der Wert einer Schriftgröße sollte nicht negativ sein. Klassisch wird diese Logik im *set*-Accessor des CLR-Wrappers implementiert. Für in XAML deklarierte Zugriffe werden jedoch zur Laufzeit direkt die in *DependencyObject* definierten Methoden *GetValue* und *SetValue* an Stelle des CLR-Wrappers aufgerufen. Darüber hinaus steht es dem Entwickler in C# frei, ob er für den Zu-

griff auf eine Dependency Property den CLR-Wrapper oder die Methoden *GetValue* und *SetValue* verwendet. Somit darf der CLR-Wrapper ausser *GetValue* und *SetValue* keinerlei Logik besitzen. Für eine Validierung wird der statischen *Register*-Methode ein *ValidateValueCallback* übergeben. Er kapselt eine Methode mit der entsprechenden Validierungslogik und wird vor jeder Änderung aufgerufen. So wird der Programmfluss bei einer *FontSize* kleiner 0 mit einer *InvalidOperationException* abgebrochen (Listing 8).

Sind neben dem Validieren der *FontSize* weitere Aktionen erforderlich, so nehmen die Metadaten zusätzliche Callbacks entgegen. Einer dieser Callbacks ist der *CoerceValueCallback*. Er wird bei jeder Änderung aufgerufen und kann den Wert einer Dependency Property mit seinem Rückgabewert „erzwingen". Gibt es Zustände des *SimpleLabel*, bei denen eine Schriftgröße zwingend den Wert 10 haben muss, wird dies mit dem *CoerceValueCallback* implementiert. Für alles was über das Validieren und Erzwingen von Werten hinausgeht, steht der *PropertyChangedCallback* bereit. Er ist wie der *CoerceValueCallback* Teil der Metadaten und wird auch bei jeder Änderung der Dependency Property aufgerufen. Die *DependencyPropertyChangedEventArgs* dieses Callbacks gewähren einen Zugriff auf den alten und neuen Wert der Dependency Property:

```
meta.PropertyChangedCallback =
        new PropertyChangedCallback(FontSizeChanged);
...
public static void FontSizeChanged(DependencyObject depOb,
      DependencyPropertyChangedEventArgs e){object o =
                                        e.OldValue; ...}
```

Während der *ValidateValueCallback* direkt mit der *DependencyProperty* zusammenhängt, sind der *CoerceValueCallback* und der *PropertyChangedCallback* Teil der Metadaten und können sich somit von Typ zu Typ unterscheiden.

Existierende DependencyProperties

Ein *DependencyProperty*-Objekt stellt den „Schlüssel" zum eigentlichen Wert dar. Vor der Implementierung einer Dependency Property ist es eine gute Idee zu prüfen, ob für den gewünschten Zweck nicht schon ein *DependencyProperty*-Objekt in einer anderen Klasse existiert, da es im Idealfall für eine spezielle Dependency Property lediglich einen „Schlüssel" gibt. Denn nur wenn alle *DependencyObject*-Objekte denselben „Schlüssel" verwenden, ist der Zugriff auf den Wert einer Dependency Property konsistent und der Wert kann zum Beispiel über den Element Tree vererbt werden. Daher sollte auch die Klasse *SimpeLabel* für die *FontSize* dasselbe *DependencyProperty*-Objekt wie die Klasse *Control* verwenden. Dies wurde bei der Klasse *SimpleLabel* vernachlässigt und wird an dieser Stelle nachgeholt, da die Klasse *Control* im statischen Feld *FontSizeProperty* bereits eine Dependency Property für den gewünschten Zweck definiert. Anstatt in *SimpleLabel* mit der statischen Methode *Register* eine neue Dependency Property in der WPF zu registrieren, wird die – ebenfalls in der *DependencyProperty*-Klasse befindliche – Instanz-Methode *AddOwner* aufgerufen. Sie fügt einer bereits registrierten *DependencyProperty* einen weiteren „Besitzer" hinzu. Die Initialisierung der *FontSizeProperty* in *SimpleLabel* sieht somit wie folgt aus:

```
public static readonly DependencyProperty
                                    FontSizeProperty =
  Control.FontSizeProperty.AddOwner(typeof(SimpleLabel));
```

Nachdem die *FontSizeProperty* in *SimpleLabel* nun lediglich einen weiteren „Besitzer" zur in *Control* definierten *FontSizeProperty* hinzufügt, zeigen zur Laufzeit die statischen *FontSizeProperty*-Felder beider Klassen auf dasselbe *DependencyProperty*-Objekt bzw. denselben „Schlüssel". Folglich lassen sich die Felder beider Klassen zum Zugriff auf die Dependency Property verwenden:

```
SimpleLabel sl = new SimpleLabel();
sl.SetValue(Control.FontSizeProperty, 40.0);
```

Listing 8

Eigenschaftswerte überprüfen per Callback

```
public class SimpleLabel:FrameworkElement
{
 ...
 static SimpleLabel()
 {
 ...
  FontSizeProperty =
   DependencyProperty.Register("FontSize"
              ,typeof(double)
              ,typeof(SimpleLabel)
              ,meta
              ,new ValidateValueCallback(
                                  ValidateFontSize));
 }
 private static bool ValidateFontSize(object value)
 {
  return (double)value >= 0;
 }
 ...
}
```

Listing 9

FontSize des Windows wird an Wert der ComboBox gebunden

```
<Window x:Class="DependencyPropertyInWPF_
                                  AddOwner.Window1"
  ...
  FontSize="{Binding ElementName=cboFontsizes,
                          Path=SelectionBoxItem}">
<Grid>
 <Grid.RowDefinitions>
  <RowDefinition/>
  <RowDefinition/>
 </Grid.RowDefinitions>
 <Grid.ColumnDefinitions>
  <ColumnDefinition/>
  <ColumnDefinition Width="Auto"/>
 </Grid.ColumnDefinitions>
 <local:SimpleLabel x:Name="sl"
          Margin="5"/>
 <TextBox Text="{Binding ElementName=sl,
                                  Path=FontSize }"
     Grid.Row="1"
     Margin="5"/>
 <StackPanel x:Name="stack1"
      Margin="5"
      Grid.Column="1"
      Grid.RowSpan="2"
      Width="250">
  <ComboBox x:Name="cboFontsizes"/>
  <TextBlock TextWrapping="Wrap" >^<LineBreak/>
  Combobox setzt FontSize auf dem Window.
  Die FontSize wird im Element Tree von
  hierarchisch tieferliegenden
  Elementen "vererbt"
  (Hier sind diese Elemente u.a.:
  TextBlock, TextBox, SimpleLabel
  als auch die ComboBox selbst)
  </TextBlock>
 </StackPanel>
</Grid>
</Window>
```

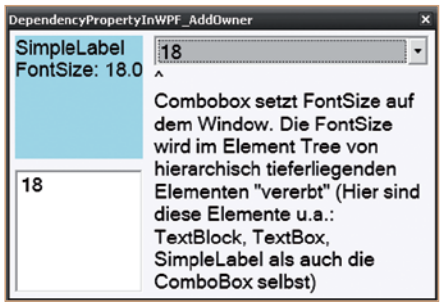


Abb. 5: Schriftgröße wird anhand des Element Tree vererbt

```
double fSize = (double)sl.GetValue(SimpleElement
                                   .FontSizeProperty);
// fSize = 40.0;
```

Eine Überladung der *AddOwner*-Methode nimmt als zweites Argument typspezifische Metadaten entgegen. In obigem Code werden keine Metadaten übergeben. Damit werden die von *Control* definierten Metadaten auch für *SimpleLabel* übernommen. Diese Metadaten der *Control .FontSizeProperty* haben die Option *FrameworkPropertyMetadataOptions.Inherits* gesetzt. Der Wert der *FontSizeProperty* wird infolgedessen über die Hierarchie des Element Trees vererbt. Da *SimpleLabel* für die *FontSizeProperty* jetzt dasselbe *DependencyProperty*-Objekt und nicht einen anderen „Schlüssel" verwendet, nimmt die *FontSizeProperty* des *SimpleElement* auch in diesem Vererbungsprozess teil. Zum Test wird dem Fenster, welche das *SimpleLabel* und die *TextBox* enthält, eine *ComboBox* hinzugefügt, die im *Loaded*-Ereignis des *Window*-Objekts mit Werten von 10 bis 20 gefüllt wird. Die *FontSize* des *Window*-Objekts wird in XAML an den aktuellen Wert der *ComboBox* „gebunden" – zu sehen in Listing 9.

Wird in der *ComboBox* eine *FontSize* ausgewählt, so „erben" die Elemente den Wert des *Window*-Objekts über den Element Tree – vorausgesetzt, ihre *FontSize* wurde noch nicht gesetzt. Auch das *SimpleLabel* erbt den *FontSize*-Wert des *Window*-Objekts, da es dasselbe *DependencyProperty*-Objekt verwendet – siehe Abbildung 5.

Die Rolle der Attached Properties

Eine interessante Möglichkeit der Dependency Properties ist, dass sie sich auf beliebigen *DependencyObject*-Objekten setzen lassen. Dabei müssen die *DependencyObject*-Objekte nicht vom Typ der Klasse sein, die die *DependencyProperty* besitzt. Diese Möglichkeit wird insbesondere beim Layout verwendet. Ein *Grid* benötigt auf den Kind-Elementen unter anderem eine *Row*-Eigenschaft, um im Layout-Prozess die Elemente korrekt anzuordnen. Anstatt diese und zig andere Eigenschaften in einer Basis-Klasse wie *Control* zu definieren und damit ein nicht erweiterbares System zu erstellen, wurden solche Eigenschaften in der WPF als Dependency Properties implementiert. So lässt sich auf dem *SimpleLabel* die *Grid.RowProperty* setzen. Und das obwohl die Klasse *SimpleLabel* nichts über die in der Klasse *Grid* definierte Dependency Property *Grid.RowProperty* weiß:

```
SimpleLabel sl = new SimpleLabel();
sl.SetValue(Grid.RowProperty, 1);
```

Wird wie im obigen Codeausschnitt eine in Klasse A deklarierte und initialisierte *DependencyProperty* auf einem Objekt der Klasse B gesetzt, so wird auch von einer „Attached Property" gesprochen. Auf Objekten einer von *DependencyObject* abgeleiteten Klasse wie zum Beispiel *Control* lassen sich durch diese Möglichkeit zur Laufzeit weitere, in anderen Klassen implementierte, Dependency Properties hinzufügen – daher „attached" – ohne eine weitere Sub-Klasse von *Control* zu erstellen. Wie der obere Codeausschnitt zeigt, lässt sich die *Grid.RowProperty* auch dann auf dem *SimpleElement* oder einem anderen, beliebigen *DependencyObject* setzen, wenn sich dieses nicht in einem *Grid* befindet. Wird das *SimpleLabel* später zu einem *Grid* hinzugefügt, verwendet das *Grid* den auf *SimpleLabel* gesetzten Wert der *Grid.RowProperty*. Wird es nicht zu einem *Grid* hinzugefügt, so ist der Wert der Dependency Property zwar trotzdem auf dem *SimpleLabel* gesetzt, wird aber von keinem anderen Objekt verwendet. Folglich passiert nichts weiter.

Ist eine Dependency Property speziell für die Verwendung als Attached Property vorgesehen, wie das bei der *Grid.RowProperty* der Fall ist, wird die *DependencyProperty* an Stelle von *Register* mit der Methode *RegisterAttached* initialisiert und registriert (Listing 10). Die Methoden unterscheiden sich nur darin, dass bei *Register* die Metadaten per *Type* und bei *RegisterAttached* die Metadaten für alle *DependencyObject*-Objekte erstellt werden [2].

Während C# zwischen „gewöhnlichen" Dependency Properties und Attached Properties keinen Unterschied macht – es wird bei beiden an *SetValue* und *GetValue* ein *DependencyProperty*-Objekt als „Schlüssel" übergeben – definiert XAML für den Zugriff auf Attached Properties eine spezielle Syntax. Mit der Attached-Property-Syntax wird die *Grid.RowProperty* auf dem *SimpleLabel* in XAML wie folgt gesetzt:

```
<SimpleLabel Grid.Row="1" x:Name="sl"/>
```

Der XAML-Parser setzt allerdings für die Attached-Property-Syntax die Existenz von zwei statischen Methoden in der Klasse *Grid* voraus. Diese sind laut WPF-Konvention für jede Attached Property anstatt

Listing 10

Die Dependency Property wird mit RegisterAttached initialisiert

```
public class Grid
{
 public static readonly RowProperty;

 static Grid()
 {
  RowProperty = DependencyProperty.RegisterAttached
  ("Row"
   ,typeof(int)
   ,typeof(Grid)
   ,new FrameworkPropertyMetadata(0,...)
  );
 }
}
```

Listing 11

Statistische Set- und Get-Methoden der Klasse Grid

```
public class Grid
{
 public static readonly RowProperty;

 ...
 public static int GetRow(UIElement element)
 {
  return (int)element.GetValue(RowProperty);
 }

 public static void SetRow(UIElement element, int value)
 {
  element.SetValue(RowProperty, value);
 }
}
```

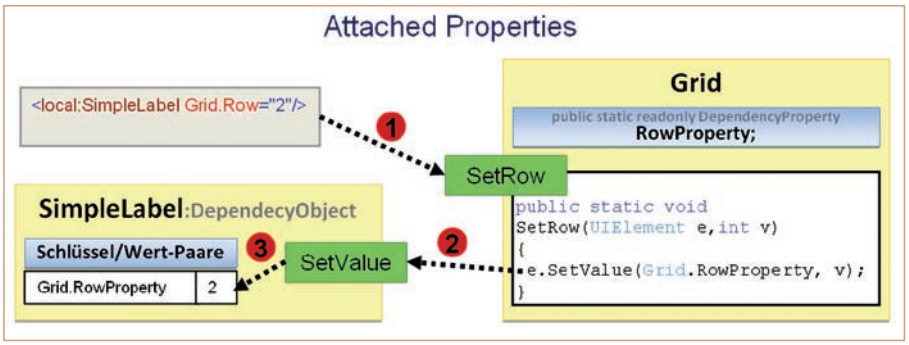


Abb. 6: Eine Klasse, die Attached Properties definiert, kann aber muss nicht zwingend vom Typ DependencyObject sein

des CLR-Wrappers zu erstellen. Sie müssen folgendem Muster entsprechen (*[Propertyname]* ist durch den Eigenschaftsnamen zu ersetzen):

```
public static Set[Propertyname](DependencyObject o,
                                object value){
 o.SetValue(MyDependencyProperty,value); }

public static object Get[Propertyname](
                                DependencyObject o){
 return o.GetValue(MyDependencyProperty); }
```

Wie zu sehen ist, rufen die Methoden intern lediglich *GetValue* und *SetValue* der übergebenen *DependencyObject*-Instanz auf. Die WPF-Konvention für diese beiden Methoden erlaubt es, für die Parameter und Rückgabewerte der Methoden Sub-Typen von *DependencyObject* bzw. *object* zu verwenden. Davon macht auch die Klasse *Grid* für den Zugriff auf die *RowProperty* Gebrauch – dargestellt in Listing 11.

Da die Attached-Property-Syntax in XAML auf die statischen *Set*- und *Get*-Methoden aufbaut, entspricht der obere XAML-Ausschnitt folgendem Code in C#:

```
SimpleLabel sl = new SimpleLabel();
Grid.SetRow(sl,1);
```

Hervorzuheben ist die Tatsache, dass Attached Properties keinen klassischen CLR-Wrapper haben. Aber ebenso wie beim CLR-Wrapper sollten auch die beiden statischen *Get*- und *Set*-Methoden außer dem Aufruf von *GetValue* und *SetValue* auf dem übergebenen *DependencyObject*-Objekt keine weitere Logik enthalten. Da bei einer „Attached Property" der Wert nicht auf Objekte der Klasse gesetzt wird, die das *DependencyProperty*-Objekt definiert, muss diese Klasse nicht zwingend vom Typ *DependencyObject* sein. Im zuvor gezeigten Beispiel ist es das *Grid*, das die *RowProperty* nicht auf sich selbst, sondern auf dem *SimpleLabel* setzt (Abbildung 6).

Dependency Properties als Fundament

Dependency Properties öffnen nicht nur das Tor zu den Services in der Windows Presentation Foundation, sie bilden auch ihr Fundament. Microsoft ist es gelungen, das Konzept der Dependency Properties mit einem konsistenten und geradlinigen Objektmodell umzusetzen. Es dürfte aber auch jedem Entwickler klar sein, dass die Property Engine der Windows Presentation Foundation im Hintergrund viele Aufgaben wahrnimmt und somit vermutlich einen der komplexesten Teile des .NET Framework 3.0 darstellt. Durch die interne Komplexität ist es jedoch nach außen umso einfacher, wie die ohne große Probleme zu implementierende Vererbung oder Animation der *SimpleLabel.FontSizeProperty* gezeigt hat. Während früher für eine Animation zusätzlicher Code mithilfe eines Timers und einem dazugehörigen Event Handler implementiert wurde, steht die Animation in der WPF als Service zur Verfügung und kann für Dependency Properties genutzt werden. Neben all den Möglichkeiten wie Styles, Data Binding usw. sind die Attached Properties natürlich, insbesondere auch aufgrund der speziellen Syntax für XAML, ein willkommenes Feature. Ursprünglich lediglich für Layout-Zwecke angedacht, fanden sich während der Entwicklung der Windows Presentation Foundation viele weitere Stellen für den Einsatz von Attached Properties (z.B. die *TextElement.FontSizeProperty*). Trotz der internen Berechnungen und zusätzlichen Aufwände, die die WPF rund um Dependency Properties hat, besitzen sie gegenüber klassischer Properties lediglich beim Setzen des Wertes einen kleinen Performance-Nachteil [1]. Für Interessierte, die weitere „Experimente" durchführen möchten, sind die Klassen *DependencyPropertyDescriptor* und *DependencyPropertyHelper* sehr interessant. Darüber hinaus lassen sich mit Lutz Roeders .NET Reflector [3] auch viele Erkenntnisse über bereits in der WPF implementierte Dependency Properties gewinnen.

Thomas Huber arbeitet bei der Trivadis AG im Bereich Application Development als Trainer und Entwickler für Microsoft Technologien. Für Fragen, Anregungen, Kritik und Wünsche rund um das Thema .NET und WPF ist er unter thomas.huber@trivadis.com zu erreichen.

Links & Literatur

[1] Optimizing WPF Application Performance: msdn2.microsoft.com/en-us/library/aa970683.aspx

[2] Nick Kramers Blog: blogs.msdn.com/nickkramer/archive/2005/08/25/456024.aspx

[3] Lutz Roeder's Reflector: www.aisto.com/roeder/dotnet/

[3] MSDN Properties: msdn2.microsoft.com/en-us/library/ms753192.aspx