Kapitel 7

Windows Presentation Foundation 4.0

In diesem Kapitel:

Dr. Holger Schwichtenberg, Manfred Steyer: Microsoft .NET 4.0 - Update, Microsoft Press 2010 (ISBN 978-3-86645-443-9)

Mit WPF brachte Microsoft frischen Wind in die Welt der GUI-Programmierung. Der nächste Schritt besteht nun darin, dieses umfangreiche Framework durch das eine oder andere zusätzliche Feature abzurunden sowie den Brückenschlag zu neuen Technologien, wie Windows 7, herzustellen. Dieses Kapitel beschäftigt sich mit diesen Aspekten.

Neue Steuerelemente

WPF 4.0 bringt einige neue Steuerelemente: einen Kalender, einen `DatePicker` sowie ein lang erwartetes Grid. Darüber hinaus wird kurz nach dem Erscheinen von .NET 4.0 ein Ribbon für WPF zur Verfügung gestellt werden. Dieser Abschnitt geht auf diese Steuerelemente ein.

Calendar und DatePicker

In WPF 4.0 werden einige seit bereits längerer Zeit über das WPF Toolkit [WPFT] verfügbare Steuerelemente enthalten sein. Darunter die Steuerelemente `Calendar` und `DatePicker`. Beide ermöglichen, dem Namen zur Folge, die Anzeige und Eingabe von Datumswerten. Abbildung 7.1 zeigt das `Calendar`-Steuerelement zur Laufzeit. Der dazugehörige XAML-Code findet sich in Listing 7.1.

Abbildung 7.1 *Calendar*-Steuerelement

```
<Calendar
     DockPanel.Dock="Bottom"
     Height="171"
     Name="calendar1"
     Width="198"
     SelectionMode="SingleRange"
     DisplayDate="12/24/2009">

     <Calendar.BlackoutDates>
         <CalendarDateRange
                 Start="12/24/2009"
                 End="1/6/2010" />
     </Calendar.BlackoutDates>
</Calendar>
```

Listing 7.1 *Calendar* mit *Blackout* Dates

`DisplayDate` legt das anzuzeigende Datum fest; `SelectionMode` definiert die Möglichkeiten zur Auswahl von Datumswerten. Die gewählte Option `SingleRange` bedeutet, dass lediglich eine einzige Zeitspanne ausgewählt werden darf. Alternativ dazu stehen die Optionen `MultipleRange`, `SingleDate` und `None` zur Auswahl.

`MultipleRange` legt fest, dass beliebig viele Zeitspannen ausgewählt werden können; `SingleDate`, dass nur ein einziges Datum ausgewählt werden darf und `None` unterbindet jegliche Auswahl seitens des Benutzers. Es besteht auch die Möglichkeit, Datumsbereiche für eine Auswahl zu sperren. Dazu können, wie ebenfalls in Listing 7.1 gezeigt, so genannte `BlackoutDates` hinterlegt werden. Diese werden zur Laufzeit ausgegraut dargestellt.

Zur Festlegung der Granularität der anzuzeigenden Einträge kann die Eigenschaft `DisplayMode` angepasst werden. Vorbelegt ist diese Eigenschaft mit der Option `Month`, sodass – wie in Abbildung 7.1 gezeigt – die Tage eines Monats angezeigt werden. Wird diese Eigenschaft auf *Year* gesetzt, so werden die Monate eines Jahres angezeigt (Abbildung 7.2); wird *Decade* gewählt, so erscheinen die Jahre einer Dekade (Abbildung 7.3). Nach der Auswahl eines Jahres werden dessen Monate gezeigt; nach der Auswahl eines Monats dessen Tage.

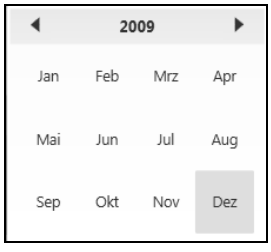

Abbildung 7.2 *Calendar* mit DisplayMode *Year*

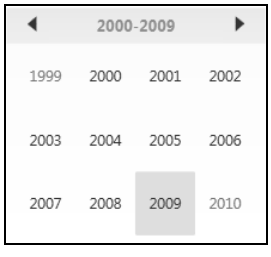

Abbildung 7.3 *Calendar* mit DisplayMode *Decade*

Das Steuerelement `DatePicker` (Abbildung 7.4) ist eine Kombination aus `Calendar` und `DropDown` und erlaubt somit eine kompaktere Darstellung. Von der Handhabung entspricht es dem Steuerelement `Calendar`.

Abbildung 7.4 Geöffneter *DatePicker*

DataGrid

Ein dediziertes Steuerelement zum Darstellen tabellarischer Daten gab es bis dato in WPF nicht. Mit dem `DataGrid` hat sich dies nun geändert.

Einführendes Beispiel

Abbildung 7.5[1] zeigt ein Beispiel des neuen `DataGrid`-Steuerelements zur Laufzeit; Listing 7.2 einen Auszug des dazugehörigen XAML-Codes.

Abbildung 7.5 *DataGrid*

```
<DataGrid
    Name="dataGrid1"
    ItemsSource="{Binding}"
    AutoGenerateColumns="False"
    AlternatingRowBackground="#103D7EC5"
    RowHeaderWidth="20"
    CanUserSortColumns="True">

    <DataGrid.Columns>

      <DataGridTextColumn
        x:Name="colName"
        Header="Name"
        Binding="{Binding Spitzname}"  />
```

[1] Die in diesem Kapitel verwendeten Fotos wurden aus Wikimedia Commons entnommen und stehen unter der GNU-Lizenz für freie Dokumentation. (*http://de.wikipedia.org/w/index.php?title=Datei:Phodopus_roborovskii_front.jpg&filetimestamp=20070726101334*, Urheber: Bullet; *http://de.wikipedia.org/w/index.php?title=Datei:Phodopus_roborovskii_sand.JPG&filetimestamp=20070731191944*, Urheber: Bulli)

```
      <DataGridTextColumn
        x:Name="colGewicht"
        Header="Gewicht"
        Binding="{Binding Gewicht}" />

      <DataGridCheckBoxColumn
        x:Name="colVegetarier"
        Header="Vegatarier?"
        Binding="{Binding Vegetarier}" />

      <DataGridComboBoxColumn
        x:Name="colFarbe"
        Header="Farbe"
        SelectedValueBinding="{Binding FarbeId}"
        SelectedValuePath="FarbeId"
        DisplayMemberPath="Bezeichnung"
        ItemsSource="{Binding Source={StaticResource Farben}}" />

      [...]
</DataGrid>
```

Listing 7.2 Einfaches *DataGrid*

Die Eigenschaft `AutoGenerateColumns` gibt an, ob die einzelnen Spalten für die gebundenen Daten automatisch erstellt werden sollen. Diese Eigenschaft ist standardmäßig auf `true` und erlaubt somit ein rasches Erstellen von Demos und Prototypen. Um die Reihenfolge und Darstellung der einzelnen Spalten beeinflussen zu können, wurde diese Eigenschaft im betrachteten Beispiel auf `false` gesetzt und Angaben zu den einzelnen Spalten weiter unten explizit hinterlegt. Mittels `AlternatingRowBackground` kann die Hintergrundfarbe jeder zweiten Zeile festgelegt werden. Beim `RowHeader`, dessen Breite mit der Eigenschaft `RowHeaderWidth` angegeben werden kann, handelt es sich um die Zelle vor jeder Zeile, welche ein Markieren der gesamten Zeile ermöglicht. Ob das Grid zur Laufzeit durch den Benutzer per Klick auf eine Spaltenüberschrift sortiert werden darf, wird mittels `CanUserSortColumns` angegeben.

Innerhalb von `DataGrid.Columns` werden Informationen zu den anzuzeigenden Spalten angegeben. Dies erfolgt im betrachteten Beispiel durch Verwendung der Elemente `DataGridTextColumn`, `DataGridCheckBoxColumn` und `DataGridComboBoxColumn`. `DataGridTextColumn` zeigt die jeweilige Spalte zum Editieren innerhalb einer *TextBox* an; `DataGridCheckBoxColumn` innerhalb einer `Checkbox` und `DataGridComboBoxColumn` innerhalb einer `ComboBox`. Es handelt sich dabei um Subklassen von `DataGridColumn`. Zur Anzeige von Hyperlinks steht darüber hinaus eine, hier nicht gezeigte, Subklasse `DataGridHyperlinkColumn` zur Verfügung. Wer sich nicht mit den Möglichkeiten dieser vordefinierten Spaltenarten zufriedengeben möchte, kann auf die `DataGridTemplateColumn` zurückgreifen. Diese bietet die Möglichkeit, die innerhalb des Grids anzuzeigenden Steuerelemente selbst festzulegen.

Die Eigenschaft `Header` legt jeweils die Spaltenüberschrift fest. Bei der `DataGridComboBoxColumn` wird mittels `SelectedValueBinding` jene Eigenschaft angegeben, an die der ausgewählte Wert gebunden werden soll. Zum Festlegen der Vorschlagswerte können die Eigenschaften `SelectedValuePath`, `DisplayMemberPath` und `ItemsSource` herangezogen werden. Letztere legt die Datenquelle, von welcher die Vorschlagswerte bezogen werden sollen, fest. Im betrachteten Beispiel handelt es sich hierbei um eine statische `CollectionViewSource`, welche über die Ereignisroutine `Window_Loaded` gesetzt wird (siehe Listing 7.3 und Listing 7.4). Der innerhalb der `ComboBox` anzuzeigende Wert wird mittels `DisplayMemberPath` festgelegt; der für die Datenbindung zu verwendende Wert mittels `SelectedValuePath`.

```
<Window.Resources>
    […]
    <CollectionViewSource x:Key="Farben" />
</Window.Resources>
```

Listing 7.3 *CollectionViewSource* mit Vorschlagswerten für Farben

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    […]
    var farbenSource = (CollectionViewSource)this.FindResource("Farben");
    farbenSource.Source = […];
}
```

Listing 7.4 Festlegen der Vorschlagswerte

Ein Beispiel zur Verwendung der bereits oben erwähnten `DataGridTemplateColumn` findet sich in Listing 7.5. `SortMemberPath` legt dabei fest, nach welchem Attribut eine eventuelle Sortierung der Spalte erfolgen soll. Darüber hinaus wird mittels `DataGridTemplateColumn.CellTemplate` ein Template hinterlegt, welches angibt, wie Zellen dieser Spalte angezeigt werden sollen, wenn sie nicht editiert werden. `DataGridTemplateColumn.CellEditingTemplate` legt analog dazu die Darstellung der jeweiligen Zellen zum Editieren fest.

```
<DataGridTemplateColumn
    Header="Geburtstag"
    x:Name="colGeburtstag"
    SortMemberPath="Geburtstag">

    <DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding Geburtstag, StringFormat=\{0:dd.MM.yyyy\}}"/>
        </DataTemplate>
    </DataGridTemplateColumn.CellTemplate>

    <DataGridTemplateColumn.CellEditingTemplate>
        <DataTemplate>
            <DatePicker
                SelectedDate="{Binding Geburtstag}" />
        </DataTemplate>
    </DataGridTemplateColumn.CellEditingTemplate>

</DataGridTemplateColumn>
```

Listing 7.5 Verwendung von *DataGridTemplateColumn*

Detailbereiche

Das `DataGrid` ermöglicht auch das Anzeigen von Details zur ausgewählten Zeile. In Abbildung 7.5 wird beispielsweise ein Foto für die aktuelle Zeile angezeigt. Somit könnten – ähnlich wie in *WinForms* – verschachtelte Grids realisiert werden. Um die Darstellung dieser Detailinformation zu beeinflussen, wird der Eigenschaft *DataGrid.*`RowDetailsTemplate` ein entsprechendes `DataTemplate` zugewiesen. Listing 7.6 demonstriert dies, indem ein `DataTemplate` mit einem `StackPanel` und einem `Image` hinterlegt wird.

```
<DataGrid.RowDetailsTemplate>
    <DataTemplate>
        <StackPanel Orientation="Horizontal">
            <Image Source="{Binding ImageUrl}"   />
        </StackPanel>
    </DataTemplate>
</DataGrid.RowDetailsTemplate>
```

Listing 7.6 Verwendung des *RowDetailsTemplate*

Validieren von Eingaben

Zum Überprüfen von Eingaben unterstützt das `DataGrid`, analog zu den anderen Steuerelementen, `ValidationRules`. Zum Festlegen der heranzuziehenden `ValidationRules` sind diese der Auflistung `DataGrid.RowValidationRules` hinzuzufügen. Darüber hinaus kann der Eigenschaft `DataGrid.RowValidationErrorTemplate` ein `ControlTemplate` zugewiesen werden. Dieses legt die Darstellung des `RowHeaders` für Zeilen, welche ungültige Werte beinhalten, fest. Ein Beispiel dazu findet sich in Listing 7.7.

```
<DataGrid.RowValidationRules>
     <my:BirthdayValidationRule  />
</DataGrid.RowValidationRules>

<DataGrid.RowValidationErrorTemplate>
     <ControlTemplate>
         <Grid
             VerticalAlignment="Center"
             Margin="1,0,0,0"
             ToolTip="{Binding RelativeSource={RelativeSource FindAncestor, AncestorType={x:Type
DataGridRow}}, Path=(Validation.Errors)[0].ErrorContent}">
             <Label Foreground="Red" FontWeight="Bold">!</Label>
         </Grid>
     </ControlTemplate>
</DataGrid.RowValidationErrorTemplate>
```

Listing 7.7 Validieren von Eingaben in einem *DataGrid*

Zunächst wird festgelegt, dass die benutzerdefinierte `BirthdayValidationRule` zum Validieren herangezogen werden soll. Dabei wird vom XML-Namespace `my`, welcher auf *Window*-Ebene mittels `<Window xmlns:my=»clr-namespace:Sample01« […]>` deklariert wurde und auf den Namespace dieser Klasse (`Sample01`) verweist, Gebrauch gemacht. Anschließend wird ein `ControlTemplate` für `RowValidationErrorTemplate` definiert, welches angibt, dass im Fehlerfall ein *Label* mit einem roten Rufzeichen auszugeben ist. Der `ToolTip` des dieses Label umschließenden Grids wird an die erste für die aktuelle Zeile gefundene Fehlermeldung gebunden. Listing 7.8 enthält den Quellcode der dazugehörigen `ValidationRule`.

```
class BirthdayValidationRule : ValidationRule
{
    public override ValidationResult Validate(object value, System.Globalization.CultureInfo
cultureInfo)
    {
        object oGeburtstag;
        DateTime geburtstag;
```

```
        BindingGroup bg;

        bg = (BindingGroup)value;

        if (bg.Items.Count == 0)
        {
            return ValidationResult.ValidResult;
        }

        object h = bg.Items[0];

        bool ok = bg.TryGetValue(h, "Geburtstag", out oGeburtstag);

        if (!ok)
        {
            return ValidationResult.ValidResult;
        }

        if (oGeburtstag is DateTime)
        {
            geburtstag = (DateTime)oGeburtstag;
            if (geburtstag < DateTime.Now)
            {
                return ValidationResult.ValidResult;
            }
        }

        return new ValidationResult(false, "Geburtstag muss in der Vergangenheit liegen!");

    }
}
```

Listing 7.8 *ValidationRule* für *DataGrid*

Die Methode `Validate` prüft, ob die `BindingGroup` einen Eintrag enthält sowie ob dieser einen Wert für den zu prüfenden Geburtstag aufweist. Wenn nicht, gibt es von der betrachteten `ValidationRule` keine Beanstandung. Ansonsten wird der erhaltene Geburtstag mit `DateTime.Now` verglichen. Ist er »kleiner«, so wird die Methode ohne Fehlermeldung mittels `ValidationResult.ValidResult` beendet. Ansonsten wird am Ende eine Fehlermeldung innerhalb eines neuen `ValidationResults` zurückgegeben.

Gruppieren von Einträgen

Auch das Gruppieren von Datensätzen wird vom WPF DataGrid unterstützt. In Abbildung 7.6 wurden die verwalteten Hamster zum Beispiel nach ihrer Fellfarbe gruppiert.

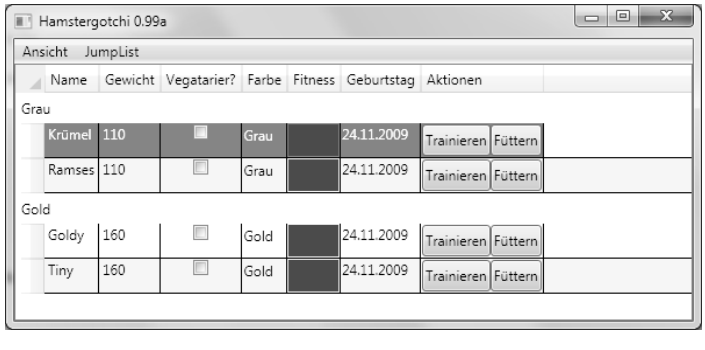

Abbildung 7.6 *DataGrid* mit Gruppierungen

Um in den Genuss dieser Möglichkeit zu kommen, ist zunächst eine `CollectionViewSource`, welche die zu gruppierenden Daten repräsentiert, als `Resource` bereitzustellen. Dieser ist darüber hinaus pro gewünschter Gruppierungsebene eine `PropertyGroupDescription` zu spendieren. Die Eigenschaft `PropertyName` der jeweiligen `PropertyGroupDescription` beinhaltet dabei jeweils den Namen jener Eigenschaft, nach der gruppiert werden soll. Die `CollectionViewSource` kann, wie in Listing 7.4 gezeigt, mit konkreten Werten, z.B. einer Liste mit Entities, versehen werden. Damit das `DataGrid` diese Informationen erhält, muss dessen Binding auf die `CollectionViewSource` verweisen. Listing 7.9 beinhaltet ein Beispiel dafür. Darüber hinaus wird in diesem Beispiel eine Instanz des Converters `farbId2FarbeConverter`, welcher die Id der jeweiligen Farbe in deren Bezeichnung überführt, als `Resource` bereitgestellt.

```
<Window.Resources>
    <my:FarbId2FarbeConverter x:Key="farbId2FarbeConverter" />

    […]

    <CollectionViewSource x:Key="Hamster">
        <CollectionViewSource.GroupDescriptions>
            <PropertyGroupDescription PropertyName="FarbeId" />
        </CollectionViewSource.GroupDescriptions>
    </CollectionViewSource>
</Window.Resources>

[…]

<DataGrid
    Name="dataGrid1"
    ItemsSource="{Binding Source={StaticResource Hamster}}" […]>

[…]
```

Listing 7.9 *DataGrid* mit Datenbindung und Gruppierung

Zur Beeinflussung der Darstellung des Gruppenkopfes besteht unter anderem die Möglichkeit, ein `DataTemplate` für diesen über die Eigenschaft `GroupStyle.HeaderTemplate` des `DataGrid` zu hinterlegen. In Listing 7.10 wird zum Beispiel ein `DataTemplate` mit einem Label, welches an die Eigenschaft `Name` gebunden ist, verwendet. Name bezieht sich hierbei auf den Wert der aktuellen Gruppierung, also auf die jeweilige `FarbId`. Damit anstatt dieser `Id` der Name der Farbe ausgegeben wird, wird im Zuge dieses Bindings auf den in Listing 7.10 bereitgestellten `Converter` referenziert. Die Implementierung dieses Converters findet sich in Listing 7.11.

```
<DataGrid.GroupStyle>
    <GroupStyle>
        <GroupStyle.HeaderTemplate>
            <DataTemplate>
                <Label Content="{Binding Name, Converter={StaticResource farbId2FarbeConverter}}" />
            </DataTemplate>
        </GroupStyle.HeaderTemplate>
    </GroupStyle>
</DataGrid.GroupStyle>
```

Listing 7.10 *Label* mit Verweis auf *Converter*

```
public class FarbId2FarbeConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        int farbeId;
        string strFarbeId;

        strFarbeId = value.ToString();

        try
        {
             farbeId = Int32.Parse(strFarbeId);
        }
        catch
        {
            return value;
        }

        FarbenDAO dao = new FarbenDAO();
        Farbe f = dao.FindById(farbeId);

        return f.Bezeichnung;
    }

    public object ConvertBack(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

Listing 7.11 Converter für die Umwandlung von Farb Ids zu Farben

Es besteht auch die Möglichkeit, die Darstellung einer Gruppe vollständig zu verändern, indem ein neues Template für den `GroupHeaderStyle` definiert wird. Listing 7.12 führt auf diese Weise zum Beispiel einen Expander ein, welcher es erlaubt, die einzelnen Gruppen auf- und zuzuklappen. Abbildung 7.7 zeigt das Resultat dieses Vorhabens: Die erste Gruppe mit der Bezeichnung *Grau* ist zugeklappt; die zweite mit der Bezeichnung *Gold* ist offen.

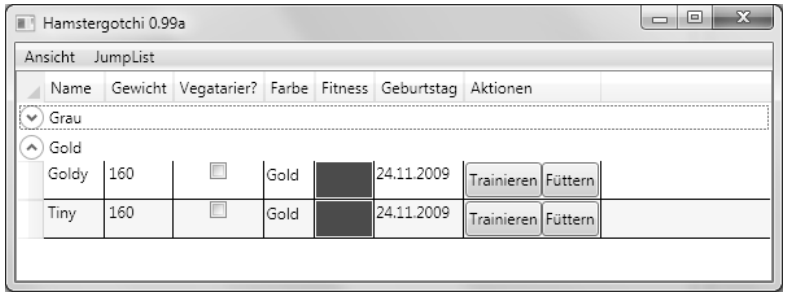

Abbildung 7.7 *DataGrid* mit Gruppierung durch *Expander*

```
<Window.Resources>

    […]

    <Style x:Key="GroupHeaderStyle" TargetType="{x:Type GroupItem}">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type GroupItem}">
                    <Expander IsExpanded="False">
                        <Expander.Header>
                            <TextBlock Text="{Binding Name, Converter={StaticResource farbId2FarbeConverter}}"
                            />
                        </Expander.Header>
                        <ItemsPresenter />
                    </Expander>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>

</Window.Resources>

[…]

<DataGrid […]>
    <DataGrid.GroupStyle>
        <GroupStyle ContainerStyle="{StaticResource GroupHeaderStyle}" />
    </DataGrid.GroupStyle>
[…]
```

Listing 7.12 Gruppierung mit *Expander*

Menüband (Ribbon)

Auch ein `Ribbon`-Steuerelement wird es für WPF 4.0 geben. Allerdings ist dieses nicht im Lieferumfang von Visual Studio 2010/WPF 4.0 enthalten. Stattdessen soll nachgeliefert werden[SCO]. Aus diesem Grund basiert dieser Abschnitt auf der CTP-Version dieses Steuerelements. Diese muss allerdings separat heruntergeladen werden [RIB01].

Einführendes Beispiel

Abbildung 7.8 zeigt ein Beispiel, welches den Einsatz dieses Steuerelements demonstriert; Listing 7.13 den dazugehörigen XAML-Code.

Abbildung 7.8 Einfaches Beispiel für die Verwendung eines *Ribbon*

```
<r:RibbonWindow xmlns="[…]"
    xmlns:x="[…]"
    xmlns:r="clr-namespace:[…]Ribbon[…]"
    xmlns:my="clr-namespace:RibbonDemo"
    x:Class="RibbonDemo.MainWindow"
    x:Name="Window"
    Title="Hamstergotchi 0.99"
    Width="640" Height="480">

    <r:RibbonWindow.Resources>

        <ResourceDictionary>

            […]

            <r:RibbonCommand
                x:Key="FutterCommand"
                CanExecute="OnCanExecute"
                Executed="OnFutterCommand"
                LabelTitle="Füttern"
                LargeImageSource="img/questionIcon.ico"
                SmallImageSource="img/questionIcon.ico"
                />

            <r:RibbonCommand
                x:Key="TrainingCommand"
                CanExecute="OnCanExecute"
                Executed="OnTrainingCommand"
                LabelTitle="Trainieren"
                LargeImageSource="img/questionIcon.ico"
                SmallImageSource="img/questionIcon.ico"
                />

            […]
        </ResourceDictionary>
    </r:RibbonWindow.Resources>

    <StackPanel x:Name="LayoutRoot">

          <r:Ribbon Title="Hamstergotchi 0.99" x:Name="ribbon">

            <r:RibbonTab Label="Hamster" GroupSizeReductionOrder="Füttern,Füttern,Füttern,Training">
                <r:RibbonGroup Name="Training" HasDialogLauncher="True" Command="{StaticResource
                    TrainingCommand}">
```

```
                    <r:RibbonButton Command="{StaticResource TrainingCommand}" />
                    <r:RibbonButton Command="{StaticResource TrainingCommand}" />
                    <r:RibbonButton Command="{StaticResource TrainingCommand}" />
                    <r:RibbonButton Command="{StaticResource TrainingCommand}" />

                </r:RibbonGroup>

                <r:RibbonGroup Name="Füttern">
                    <r:RibbonButton Command="{StaticResource FutterCommand}" />
                    <r:RibbonButton Command="{StaticResource FutterCommand}" />
                    <r:RibbonButton Command="{StaticResource FutterCommand}" />
                    <r:RibbonButton Command="{StaticResource FutterCommand}" />

            <r:RibbonTab Label="Hilfe"/>
        </r:Ribbon>

        <Label Foreground="White" FontSize="18">Hier könnte auch Ihr Hamster abgebildet sein ...</Label>
    </StackPanel>
</r:RibbonWindow>
```

Listing 7.13 Einfaches *Ribbon*

Die Klasse `RibbonWindow` ersetzt hierbei die »herkömmliche« `Window`-Klasse. Die Eigenschaft `Resources` beinhaltet ein `ResourceDictionary`. In diesem finden sich unter anderem `RibbonCommands` für sämtliche Buttons wieder, wobei die einzelnen Buttons auf diese referenzieren. Das Attribut `CanExecute` verweist auf eine Ereignisroutine, welche entscheidet, ob der jeweilige Befehl zurzeit ausgeführt werden kann. Listing 7.14 zeigt eine simple Implementierung einer solchen Routine, welche jederzeit die Ausführung der auf sie verweisenden Befehle erlaubt.

```
private void OnCanExecute(object target, CanExecuteRoutedEventArgs args)
{
    args.CanExecute = true;
}
```

Listing 7.14 Ereignisbehandlungsroutine für *CanExecute*

Das Attribut `Execute` verweist auf jene Ereignisbehandlungsroutine, welche zur Ausführung gebracht werden soll. Der `LabelTitle` spiegelt jene Bezeichnung wieder, welche als Beschriftung für die Buttons, welche auf das Command verweisen, verwendet werden soll und `LargeImageSource` sowie `SmallImageSource` verweisen auf die anzuzeigenden Symbole.

HINWEIS Der Roadmap für das `Ribbon`-Steuerlement [RIB02] zur Folge wird die Abstrahierung durch `RibbonCommands` in der endgültigen Version entfernt werden. Die Eigenschaften, welche in `RibbonCommand` vorherrschen (wie z. B. `LabelTitle`, `SmallImageSource`, `LargeImageSource`) werden in die einzelnen Steuerelemente, wie z. B. `RibbonButton`, verschoben. Darüber hinaus wird die Schnittstelle *ICommand*, welches in WPF für das Trennen von Logik und UI vorgesehen ist, unterstützt werden.

Das eigentliche Ribbon wird durch das Element `Ribbon` definiert. Im betrachteten Beispiel beinhaltet es, wie auch in Abbildung 7.8 zu erkennen ist, drei `RibbonTabs`: *Hamster, Tierarzt* und *Hilfe*. Die Registerkarte *Hamster* besteht aus zwei `RibbonGroups`, welche die Namen *Training* und *Füttern* tragen. Und jede dieser Gruppen beinhaltet insgesamt vier Buttons, welche über das Attribut `Command` auf die beiden weiter oben hinterlegten `RibbonCommands` verweisen.

Ribbons verkleinern

Das Attribut `GroupSizeReductionOrder` im Element `RibbonTab` legt fest, in welcher Reihenfolge die einzelnen Gruppen verkleinert werden sollen, wenn nicht genügend Platz vorhanden ist, weil das Fenster z. B. verkleinert wird. Die im betrachteten Beispiel gewählte Belegung für dieses Attribut bewirkt, dass die Gruppe Füttern zunächst dreimal verkleinert wird, bevor die Gruppe Training Platz abtreten muss. Abbildung 7.9 veranschaulicht diese drei Stadien. Standardmäßig werden die Gruppen gleichmäßig und zyklisch von rechts nach links verkleinert. Dies würde im betrachteten Beispiel einer `GroupSizeReductionOrder` von *Füttern, Training, Füttern, Training, Füttern, Training* entsprechen.

Abbildung 7.9 *Ribbon* in verschiedenen Größen

Standardmäßig verwendet das `Ribbon` den Windows 7-Stil – jenen Stil, den z.B. auch *Paint* oder *WordPad* in Windows 7 verwenden (vgl. Abbildung 7.10). Um den von Office 2007 gewohnten Stil, welcher auch in Abbildung 7.8 und Abbildung 7.9 verwendet wurde, zu erhalten, kann das Office2007-Theme, welches im Lieferumfang des oben genannten Downloads enthalten ist, verwendet werden. Listing 7.15 zeigt, wie dieses eingebunden werden kann.

Abbildung 7.10 *Ribbon* im Windows 7-Stil

```
[…]
<r:RibbonWindow.Resources>

        <ResourceDictionary>

            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary
Source="/RibbonControlsLibrary;component/Themes/Office2007Blue.xaml"/>
            </ResourceDictionary.MergedDictionaries>
[…]
```

Listing 7.15 Verwendung des Themes für den Windows 7-Stil

SplitButton und ToggleButton

Neben herkömmlichen Buttons unterstützen Ribbons noch weitere Steuerelemente. An dieser Stelle sollen zwei davon vorgestellt werden: *SplitButton* und *ToggleButton*. `SplitButton` beinhaltet weitere Buttons, welche in einem Aufklappmenü gezeigt werden, wenn es angeklickt wird. `ToggleButton` ist ein Button, welcher – ähnlich wie eine `Checkbox` – aktiviert oder deaktiviert werden kann. Abbildung 7.11 zeigt diese beiden Buttons zur Laufzeit: `ToggleButton` befindet sich gerade im aktivierten Zustand und `SplitButton` ist gerade aufgeklappt. Es beinhaltet – zu Demonstrationszwecken – einen weiteren `RibbonButton` sowie zwei Menüeinträge. Der dazugehörige XAML-Code findet sich in Listing 7.16.

Abbildung 7.11 Ribbon mit *ToggleButton* und *SplitButton*

```
<r:RibbonToggleButton Command="{StaticResource FutterCommand}" />

<r:RibbonSplitButton Command="{StaticResource FutterCommand}" Name="SomeCommandSplitButton">
    <r:RibbonSplitButton.Items>
        <r:RibbonButton Command="{StaticResource FutterCommand}" />
        <MenuItem Header="A"  Command="{StaticResource FutterCommand}" />
        <MenuItem Header="B"  Command="{StaticResource FutterCommand}" />
    </r:RibbonSplitButton.Items>
</r:RibbonSplitButton>
```

Listing 7.16 *ToggleButton* und *SplitButton*

Symbolleiste

Jemand, der sich Ribbons genauer ansieht, wird feststellen, dass sie auch eine Symbolleiste beinhalten können. Standardmäßig wird diese in der Titelleiste eingeblendet. Abbildung 7.12 zeigt ein Beispiel dafür. Die Symbolleiste beinhaltet zwei Symbole. Der erste Eintrag des gerade aktivierten Menüs offenbart, dass noch ein drittes Symbol eingeblendet werden könnte.

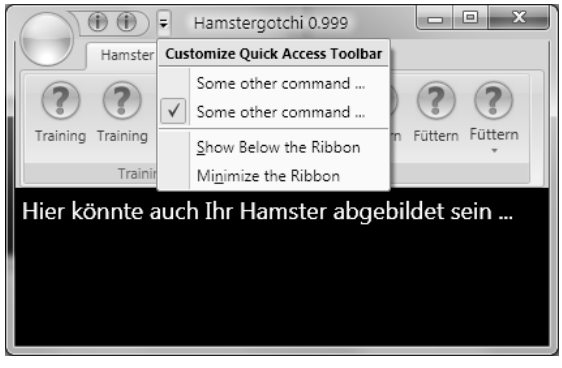

Abbildung 7.12 Ribbons mit Symbolen

Solche Symbole können über die Eigenschaft `Ribbon.QuickAccessToolBar` festgelegt werden. Listing 7.17 demonstriert dies. Über die Eigenschaft `Placement` wird dabei festgelegt, ob das Symbol nur in der Symbolleiste aufscheinen soll (`InToolBar`), ob es sowohl in der Symbolleiste als auch im in Abbildung 7.12 gezeigten Menü zum Anpassen der Symbolleiste erscheinen soll (`InCustomizeMenuAndToolBar`) oder ob es vorerst nur in diesem Menü erscheinen soll, sodass es bei Bedarf aktiviert werden kann (`InCustomizeMenu`).

```
<r:Ribbon.QuickAccessToolBar>
     <r:RibbonQuickAccessToolBar>
         <r:RibbonButton Command="{StaticResource SomeOtherCommand}"
     r:RibbonQuickAccessToolBar.Placement="InCustomizeMenu"/>
         <r:RibbonButton Command="{StaticResource SomeOtherCommand}"
     r:RibbonQuickAccessToolBar.Placement="InToolBar"/>
         <r:RibbonToggleButton Command="{StaticResource SomeOtherCommand}"
     r:RibbonQuickAccessToolBar.Placement="InCustomizeMenuAndToolBar"/>
     </r:RibbonQuickAccessToolBar>
</r:Ribbon.QuickAccessToolBar>
```

Listing 7.17 Verwendung der *QuickAccessToolBar*

Menü

Ribbons beinhalten auch ein Menü, welches per Klick auf das Symbol rechts oben aktiviert wird (siehe Abbildung 7.13). Diese Menüs können auch eine Fußzeile, welche zum Beispiel bestimmte Buttons verfügbar macht, beinhalten. Darüber hinaus finden sich für gewöhnlich im rechten Bereich zuletzt verwendete Dateien wieder.

Abbildung 7.13 Ribbon mit geöffnetem Menü

Der XAML-Code für das in Abbildung 7.13 gezeigte Menü befindet sich in Listing 7.18. Zunächst wird mit der Eigenschaft `RibbonApplicationMenu.RecentItemList` angegeben, dass sich auf der rechten Seite im Menü eine `RibbonHighlightingList` befinden soll. Diese wird an die statische Ressource `letzteHamster`, welche die zuletzt verwendeten Elemente zurückliefert, gebunden. Bei dieser Ressource handelt es sich um eine Instanz einer Subklasse von `ObservableCollection`, welche innerhalb des `ResourceDictionaries` wie folgt definiert wird:

```
<my:LetzteHamster x:Key="letzteHamster" />
```

Die Implementierung dieser Klasse findet sich in Listing 7.19. Instanziiert und zugewiesen wird sie, wie bereits mit Listing 7.4 gezeigt. `MostRecentFileSelected` legt jene Ereignisroutine fest, welche aufgerufen werden soll, wenn ein Eintrag dieser Liste ausgewählt wird; `ItemsSource` beinhaltet das bereits diskutierte Binding und `DisplayMember`, wie gewohnt, den Namen jener Eigenschaft, welche innerhalb der Liste angezeigt werden soll. Eine einfache Implementierung der Ereignisroutine für `MostRecentFileSelected`, welche lediglich den Namen des ausgewählten Hamsters ausgibt, findet sich in Listing 7.20.

Nach der Definition der `RecentItemList` folgt in Listing 7.18 die Definition der Fußzeile innerhalb von `RibbonApplicationMenu.Footer`. Diese beinhaltet ein `DockPanel` mit einem rechts ausgerichteten `RibbonButton`, welches auf den Befehl `SomeOtherCommand` verweist.

Im Anschluss daran wurden die einzelnen Menüeinträge definiert. Es handelt sich dabei um Instanzen von `RibbonApplicationMenuItem`. Der zweite Menüeintrag demonstriert, dass `RibbonApplicationMenuItems` auch ineinander geschachtelt werden können. Darüber hinaus besteht auch die Möglichkeit, »herkömmliche« `MenuItems` innerhalb von `RibbonApplicationMenuItems` anzugeben. Vor dem letzten Menüeintrag wurde darüber hinaus mittels `<Separator />` eine Trennlinie eingefügt (vgl. Abbildung 7.13).

```
<r:Ribbon.ApplicationMenu >
    <r:RibbonApplicationMenu >

        <r:RibbonApplicationMenu.RecentItemList>
```

```
            <r:RibbonHighlightingList
                MostRecentFileSelected="MostRecentFileSelected"
                ItemsSource="{Binding Source={StaticResource letzteHamster}}"
                DisplayMemberPath="Name" />

        </r:RibbonApplicationMenu.RecentItemList>


        <r:RibbonApplicationMenu.Footer>
            <DockPanel LastChildFill="False" >
                    <r:RibbonButton DockPanel.Dock="Right" Margin="2" Command=
                    "{StaticResource SomeOtherCommand}" />
            </DockPanel>
        </r:RibbonApplicationMenu.Footer>

        <!-- Menüeinträge starten hier ... -->

        <r:RibbonApplicationMenuItem Command="{StaticResource SomeOtherCommand}"  />

        <r:RibbonApplicationSplitMenuItem Command="{StaticResource SomeOtherCommand}" >
            <r:RibbonApplicationMenuItem Command="{StaticResource SomeOtherCommand}" />
        </r:RibbonApplicationSplitMenuItem>

        <Separator />

        <r:RibbonApplicationMenuItem Command="{StaticResource SomeOtherCommand}"  />


    </r:RibbonApplicationMenu>
</r:Ribbon.ApplicationMenu>
```

Listing 7.18 Ribbon mit Menü

```
public class LetzteHamster : ObservableCollection<HamsterReferenz>
{
}

public class HamsterReferenz
{
    public HamsterReferenz(string name)
    {
        Name = name;
    }

    public string Name { get; set; }
}
```

Listing 7.19 Liste mit den zuletzt verwendeten Hamstern

```
private void MostRecentFileSelected(object sender, MostRecentFileSelectedEventArgs e)
{
    HamsterReferenz hamster = (HamsterReferenz)e.SelectedItem;
    MessageBox.Show(hamster.Name);
}
```

Listing 7.20 Ereignisbehandlungsroutine für *MostRecentFileSelected*

Windows 7-Integration

Windows 7 brachte einige neue Möglichkeiten für das Gestalten von graphischen Benutzeroberflächen. Die meisten davon können nun auch über WPF verwendet werden. Dieser Abschnitt geht auf diese Möglichkeiten ein.

HINWEIS Die meisten der in diesem Abschnitt vorgestellten Möglichkeiten zur Nutzung von Windows 7-Features setzen die Verwendung von *Aero Glass* voraus.

Vorschaufenster

Windows 7 blendet ein kleines Vorschaufenster ein, wenn die Maus über einen Eintrag in der Taskleiste bewegt wird (Abbildung 7.14).

Abbildung 7.14 Vorschaufenster in Windows 7

Dieses zeigt standardmäßig den Inhalt des jeweiligen Fensters an. Um die Aufmerksamkeit der Anwender auf bestimmte Aspekte zu lenken, bietet WPF 4.0 die Möglichkeit, die Vorschau auf einen bestimmten Teil des Fensters zu beschränken. Dazu bietet die Klasse `Window` eine Eigenschaft `TaskbarItemInfo` an. Über diese kann ein `ThumbnailClipMargin`, welcher angibt, wie viel Platz neben den vier Rändern ausgeblendet werden soll, ausgegeben werden. In Listing 7.21 wird beispielsweise festgelegt, dass 20 Pixel vom linken, 30 Pixel vom oberen, 200 Pixel vom unteren und 300 Pixel vom rechten Rand für die Vorschau weggeschnitten werden sollen. Das Resultat dieser Einstellung findet sich in Abbildung 7.15.

```
<Window.TaskbarItemInfo>
    <TaskbarItemInfo>

        <TaskbarItemInfo.ThumbnailClipMargin>
            <Thickness Left="20" Top="30" Bottom="200" Right="300" />
        </TaskbarItemInfo.ThumbnailClipMargin>

    </TaskbarItemInfo>
</Window.TaskbarItemInfo>
```

Listing 7.21 Beeinflussung des Vorschaufensters

Abbildung 7.15 Vorschaufenster mit Ausschnitt des Fensters

Schaltflächen im Vorschaufenster (Thumbnail Buttons)

Um eine Applikation zu bedienen, ohne sie in den Vordergrund zu holen, bietet Windows 7 die Möglichkeit, Schaltflächen im Vorschaufenster zu platzieren. Abbildung 7.16 zeigt zum Beispiel ein Vorschaufenster, welches vier Schaltflächen zum Navigieren zwischen den vorhandenen Datensätzen anbietet (*Erster*, *Vorheriger*, *Nächster*, *Letzter*).

Abbildung 7.16 Vorschaufenster mit Schaltflächen

Um dieses Vorschaufenster anzupassen, wurde die Klasse `Window` in WPF 4.0 um eine Eigenschaft `TaskbarItemInfo`, welche eine Instanz der gleichnamigen Klasse aufnimmt, erweitert. `TaskbarItemInfo` enthält unter anderem eine Auflistung `ThumbButtonInfos`, welche Instanzen von `ThumbButtonInfo` aufnimmt. Jede `ThumbButtonInfo` beschreibt dabei einen Button. Listing 7.22 demonstriert dies, indem innerhalb dieser Auflistung vier Buttons definiert werden. Die Eigenschaft `DismissWhenClicked` legt fest, ob bei Betätigung des Buttons das Vorschaufenster geschlossen werden soll. Die Eigenschaft `Description` definiert eine Beschreibung des Buttons; `ImageSource` den Pfad zum anzuzeigenden Symbol und `Click` die Ereignisbehandlungsroutine, welche bei einem Klick aufgerufen werden soll.

```
<Window.TaskbarItemInfo>

    <TaskbarItemInfo x:Name="taskBarItemInfo">

        <TaskbarItemInfo.ThumbButtonInfos>
```

```
            <ThumbButtonInfoCollection>

                <ThumbButtonInfo
                    x:Name="tbFirst"
                    DismissWhenClicked="False"
                    Description="Erster"
                    ImageSource="img/first.ico"
                    Click="tbFirst_Click" />

                <ThumbButtonInfo
                    x:Name="tbPrev"
                    DismissWhenClicked="False"
                    Description="Vorheriger"
                    ImageSource="img/prev.ico"
                    Click="tbPrev_Click" />

                <ThumbButtonInfo
                    x:Name="tbNext"
                    DismissWhenClicked="False"
                    Description="Nächster"
                    ImageSource="img/next.ico"
                    Click="tbNext_Click" />

                <ThumbButtonInfo
                    x:Name="tbLast"
                    DismissWhenClicked="False"
                    Description="Letzter"
                    ImageSource="img/last.ico"
                    Click="tbLast_Click" />

            </ThumbButtonInfoCollection>

        </TaskbarItemInfo.ThumbButtonInfos>

    </TaskbarItemInfo>
</Window.TaskbarItemInfo>
```

Listing 7.22 Definition von Schaltflächen für das Vorschaufenster

Da die Aktionen, welche sich hinter diesen Schaltflächen verbergen, häufig auch über andere Wege, wie zum Beispiel Schaltflächen in der Applikation, Menü- und Symbolleisten, aufrufbar sein dürften, besteht alternativ zur Verwendung von Ereignisbehandlungsroutinen die Möglichkeit, über die Eigenschaft `Command` eine Implementierung von `ICommand` zu registrieren.

HINWEIS Die Symbolleiste im Vorschaufenster kann maximal sieben Schaltflächen beinhalten.

Symbol in der Taskleiste verändern

Um den Anwender unaufdringlich über Ereignisse und Fortschritte zu informieren, bietet WPF 4.0 die Möglichkeit, das Symbol in der Taskleiste mit einem anderen, kleineren Symbol zu überblenden, um den Benutzer auf bestimmte Sachverhalte aufmerksam zu machen. Erreicht werden kann dies durch Setzen der Eigenschaft `taskbarItemInfo.Overlay`. Programmatisch könnte dies beispielsweise wie folgt bewerkstelligt werden:

```
this.taskBarItemInfo.Overlay = new BitmapImage(new Uri("img/infoSymbol.ico", UriKind.Relative));
```

Neben Symbolen kann im Taskleisteneintrag auch eine Fortschrittsanzeige, welche durch die Hintergrundfarbe repräsentiert wird, eingeblendet werden. Das folgende Snippet würde zum Beispiel festlegen, dass ein Fortschritt von 50% gemeldet werden soll.

```
this.taskBarItemInfo.ProgressValue = 0.5;
this.taskBarItemInfo.ProgressState = System.Windows.Shell.TaskbarItemProgressState.Normal;
```

Über den `ProgressValue` wird der aktuelle Fortschritt als `double` zwischen 0 und 1 ausgedrückt; über den `ProgressState` der aktuelle Zustand der Abarbeitung. `Normal` bedeutet, dass alles in Ordnung ist und bewirkt eine grünliche Fortschrittsanzeige. Eine Pausierung wird mittels `Paused` angegeben. Dieser Umstand wird von Windows durch eine gelbliche Fortschrittsanzeige ausgedrückt. Im Falle eines Fehlers kann der `ProgressState` auf *Error* gesetzt werden. Dies bewirkt eine rötliche Fortschrittsanzeige. Darüber hinaus kann mit dem Wert `Indeterminate` angegeben werden, dass eine Aufgabe gerade verrichtet wird, ohne dass über den aktuellen Fortschritt informiert wird. In diesem Fall wechselt die Schattierung immer wieder von links nach rechts und vice versa. Zu guter Letzt steht noch die Option `None` zur Verfügung, welche die Fortschrittsanzeige in der Taskbar abschaltet.

Sprunglisten (Jump Lists)

Eine weitere Neuerung, welche ab Windows 7 verfügbar ist, ist die Möglichkeit so genannte Sprunglisten (*Jump Lists)* zu definieren. Dabei handelt es sich um ein Kontextmenü, welches bei Rechtsklick auf einen Eintrag in der Taskleiste erscheint. Die Einträge einer Jump List führen entweder zu einer kürzlich verwendeten Datei oder zum Start einer Applikation. Darüber hinaus können die einzelnen Einträge in verschiedene Kategorien untergliedert werden. Abbildung 7.17 zeigt eine Jump List, welche zwei beispielhafte Einträge einer Kategorie namens *Benutzerdefiniert* beinhaltet: Der Eintrag *Notepad* öffnet *Notepad* und *ReadMe* öffnet *Notepad* mit einer Datei *readme.txt*.

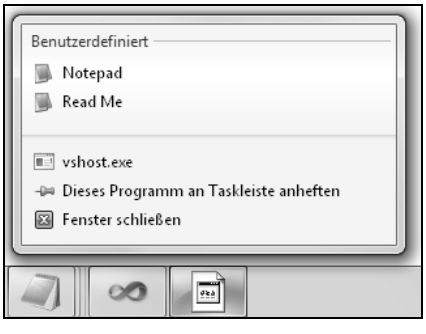

Abbildung 7.17 Verwendung einer Jump List

Listing 7.23 zeigt das dazugehörige XAML-Markup, welches innerhalb der Datei *App.xaml* definiert wurde. Die Sprungliste wird durch das Element `JumpList` repräsentiert. Die `JumpList` beinhaltet zwei `JumpTasks`. Die Eigenschaft *Title* definiert die anzuzeigende Bezeichnung; `Description` eine Beschreibung, `ApplicationPath` den Pfad zur startenden Applikation; `SymbolResourcePath` den Pfad zu jener Datei, deren Symbol in der `JumpList` angezeigt werden soll und `CustomCategory` den Namen der Kategorie, in welcher der Eintrag innerhalb der `JumpList` untergliedert werden soll. Der zweite Eintrag verwendet zusätzlich die Eigenschaft *Arguments*, über welche Befehlszeilenargumente definiert werden können.

```
<JumpList.JumpList>

    <JumpList>

        <JumpList.JumpItems>

            <JumpTask Title="Notepad"
                  Description="Open Notepad."
                  ApplicationPath="C:\Windows\notepad.exe"
                  IconResourcePath="C:\Windows\notepad.exe"
                  CustomCategory="Benutzerdefiniert"
                      />

            <JumpTask Title="Read Me"
                  Description="Open read me in Notepad."
                  ApplicationPath="C:\Windows\notepad.exe"
                  IconResourcePath="C:\Windows\notepad.exe"
                  WorkingDirectory="C:\temp"
                  CustomCategory="Benutzerdefiniert"
                  Arguments="readme.txt"
                      />

        </JumpList.JumpItems>

    </JumpList>
</JumpList.JumpList>
```

Listing 7.23 Definition einer *JumpList*

Listing 7.24 zeigt zusätzlich, wie Jump Lists zur Laufzeit erzeugt werden können. Zunächst wird ein `JumpTask` instanziiert; dann eine `JumpList`, zu welcher der `JumpTask` hinzugefügt wird. Anschließend wird diese `JumpList` mit der statischen Methode `JumpList.SetJumpList` als `JumpList` für die aktuelle Applikation festgelegt. Dabei gilt es zu beachten, dass jede Applikation zu einem Zeitpunkt lediglich eine `JumpList` besitzen kann – eine eventuell bereits bestehende `JumpList` wird durch den Aufruf von `SetJumpList` verdrängt. Änderungen an `JumpList` werden lediglich durch Aufruf dieser Methode bzw. durch den Aufruf der nicht-statischen Methode `JumpList.Apply` an Windows übergeben. Letztere gilt es deswegen aufzurufen, wenn die aktuelle `JumpList` aktualisiert wurde.

```
JumpTask jumpTask1 = new JumpTask();
jumpTask1.ApplicationPath = @"c:\windows\notepad.exe";
jumpTask1.IconResourcePath = @"c:\windows\notepad.exe";
jumpTask1.Title = "Notepad";
jumpTask1.Description = "Notepad";
jumpTask1.Arguments = @"c:\temp\test.txt";
jumpTask1.CustomCategory = "SomeCategory";
JumpList jumpList = new JumpList();
jumpList.JumpItems.Add(jumpTask1);
JumpList.SetJumpList(Application.Current, jumpList);
```

Listing 7.24 Programmatische Definition von *JumpTasks*

Multitouch

Wer zusätzlich zu Windows 7 auch ein Multitouch-fähiges Gerät sein eigen nennt, kann nun übrigens auch dessen Möglichkeiten nutzen. Informationen dazu finden sich zum Beispiel unter *http://blogs.msdn.com/jaimer/archive/2009/11/04/introduction-to-wpf-4-multitouch.aspx.*

Text, Grafik und Animationen

Dieser Abschnitt stellt Neuerungen in Hinblick auf Text, Grafiken sowie die Gestaltung von Animationen dar.

Neuerungen im Textstack

Ein neuer Formatierungsmodus namens *Display* sorgt ab Version 4.0 für klareren Text – vor allem bei Textgrößen bis ca. 15 Punkt, sofern der Text keiner Transformation unterworfen ist. In den Genuss dieses Features kommt man, indem die attached Property `TextOptions.TextFormattingMode` auf den Wert *Display* gesetzt wird. Auch der Antialiasing-Algorithmus kann nun bestimmt werden: Verwendet wird dazu die attached Property `TextOptions.TextRenderingMode`. Der Wert `ClearType` bewirkt, dass `ClearType Antialiasing` zum Einsatz kommt; `Grayscale` steht für Antialiasing mittels Graustufen und `Aliased` bewirkt, dass kein Antialiasing zum Einsatz kommt. Die Option `Auto` führt zur Verwendung von `ClearType`, sofern `ClearType` auf Systemebene aktiviert ist.

Für Aufsehen dürfte auch die Möglichkeit sorgen, die Farbe und Durchsichtigkeit von Textmarkierungen zu beeinflussen. Dazu steht nun die Eigenschaft `SelectionBrush` und `SelectionOpacity` zur Verfügung. Eingesetzt werden diese beispielsweise wie folgt:

```
<TextBox SelectionBrush="Green" SelectionOpacity="0.3">
```

Wem dies nicht genug ist, der kann auch über `SelectionBrush` einen `Brush` für Textmarkierungen definieren (siehe Abbildung 7.18).

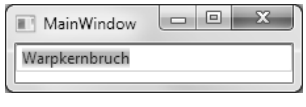

Abbildung 7.18 Verwendung von *SelectionBrush*

Darüber hinaus kann nun für die Rechtschreibkontrolle über `SpellCheck.CustomDictionaries` ein Wörterbuch mit benutzerdefinierten Worten angegeben werden. Dabei handelt es sich um eine `Uri`, welcher auf eine Textdatei, in welcher die einzelnen Wörter Zeile für Zeile gelistet sind, verweist. Listing 7.25 bietet ein Beispiel dafür sowie für benutzerdefinierte Textmarkierungen und die Verwendung der oben diskutierten neuen Textmodi. Der Prefix `sys1` stützt sich dabei auf den CLR-Namensraum `System` ab.

```
<TextBox
    TextWrapping="Wrap"
    TextOptions.TextRenderingMode="ClearType"
    TextOptions.TextFormattingMode="Display">

    <SpellCheck.IsEnabled>true</SpellCheck.IsEnabled>
```

```
    <SpellCheck.CustomDictionaries>
        <sys1:Uri>StarTrek.txt</sys1:Uri>
    </SpellCheck.CustomDictionaries>

    <TextBox.SelectionBrush>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
            <GradientStop Color="Transparent" Offset="-0.2" />
            <GradientStop Color="Green" Offset="0.25" />
            <GradientStop Color="Green" Offset="0.75" />
            <GradientStop Color="Transparent" Offset="1.2" />
        </LinearGradientBrush>

    </TextBox.SelectionBrush>
    Warpkernbruch
</TextBox>
```

Listing 7.25 Grafiken

Bezüglich der Darstellung von Grafiken wurde die attached Property `UseLayoutRounding` aus Silverlight übernommen. Wird sie auf *True* gesetzt, so wird bei der Umrechnung von plattformunabhängigen Pixeln in physische Pixel auf ganze Zahlen gerundet anstatt physische Teilpixel durch Farbübergänge zu simulieren. Für eine bessere Performance, unter anderem bei Transformationen und Animationen, soll die als *Cached Composition* bezeichnete Möglichkeit, Grafiken zwischenzuspeichern, sorgen.

Die bestehenden Bitmap-Effekte beinhalten keine Implementierung mehr. An ihre Stelle rücken neue, durch den Pixel Shader unterstützte und somit performantere Effekte. Diese sind daran zu erkennen, dass – im Gegensatz zu ihren seit Version 3.5 existierenden Pendants – das Wort Bitmap im Klassennamen fehlt. Der Ersatz für den `DropShadowBitmapEffect` lautet beispielsweise auf `DropShadowEffect`. Darüber hinaus wird ab Version 4.0 *Pixel Shader 3* unterstützt. Da es den Umfang dieses Werkes sprengen würde, folgen hierfür keine Beispiele.

Visual State Manager

Mit dem Visual State Manager (VSM) findet sich ab Version 4.0 ein Feature, welches es bereits seit einiger Zeit in Silverlight gibt, auch in WPF wieder. Er erlaubt es, das Aussehen eines Steuerelements abhängig von dessen aktuellem Zustand zu kontrollieren. So könnte zum Beispiel festgelegt werden, dass ein Button im Zustand `Pressed` eine andere Hintergrundfarbe als im Zustand `Normal` haben soll. Der VSM erlaubt auch die Beeinflussung von Zustandsübergängen. Beispielsweise könnte definiert werden, dass eine bestimmte Animation beim Übergang vom Zustand `MouseOver` in den Zustand `Pressed` stattfinden soll. Die einzelnen Zustände werden dabei in Zustandsgruppen (`StateGroups`) unterteilt. Dabei gilt, dass pro Zustandsgruppe zu einem Zeitpunkt immer genau ein Zustand aktiv ist.

Listing 7.26 bis Listing 7.28 veranschaulichen die Verwendung des VSM. Listing 7.26 zeigt ein `ControlTemplate` für einen Button. Dabei ist das Augenmerk auf die Eigenschaft `VisualStateManager.VisualStateGroups` zu richten. Diese beinhaltet einen Eintrag für jede `StateGroup`, für welche das Aussehen des Steuerelements beeinflusst werden soll. Der hier abgebildete Eintrag bezieht sich auf die *StateGroup* mit dem Namen `CommonStates` – jener `StateGroup`, welche die Zustände `Normal`, `MouseOver` und `Pressed` beinhaltet. Innerhalb von `VisualStateGroup.Transitions` wird das Verhalten definiert, welches bei Zustandsübergängen ausgelöst werden soll. Anschließend wird mittels `VisualState` pro Zustand das jeweils gewünschte Verhalten festgelegt. Am Ende folgt noch der »eigentliche« Inhalt des `ControlTemplates`: Ein `Grid` mit einem `Label`, welches den `ContentPresenter` beinhaltet.

Listing 7.27 zeigt den Inhalt von `VisualStateGroup.Transitions`. Zunächst wird mittels `VisualTransition` das Verhalten für einen Zustandswechsel von `MouseOver` zu `Pressed` definiert. Die Eigenschaft `From` benennt dabei den Ausgangszustand; *To* den Endzustand und `GeneratedDuration` die Dauer. Das Verhalten wird hierbei mit einem `StoryBoard`, welches eine Animation der Hintergrundfarbe des `Grid` im `ContentTemplate` veranlasst, festgelegt. Anschließend folgt eine weitere `VisualTransition` für den Übergang von `Pressed` zu `MouseOver`.

Listing 7.28 zeigt die Angaben, welche für die einzelnen Stati gemacht wurden. Für den Zustand *Normal* wurde kein vom Standard abweichendes Verhalten festgelegt; für den Zustand `Pressed` eine ständige Animation der Hintergrundfarbe von Rot zu Dunkelrot und wieder retour und für `MouseOver` eine Animation der Hintergrundfarbe nach Blau.

```
<ControlTemplate TargetType="Button">

    <Border x:Name="RootElement" Background="Black">

        <VisualStateManager.VisualStateGroups>

            <VisualStateGroup Name="CommonStates">

                <VisualStateGroup.Transitions>
                […]
                </VisualStateGroup.Transitions>

                <VisualState x:Name="Normal" />

                <VisualState x:Name="Pressed">
                    […]
                </VisualState>

                <VisualState x:Name="MouseOver">
                    […]
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>

        <Grid Margin="4" Background="{TemplateBinding Background}" x:Name="InnerGrid">
            <Label
                x:Name="InnerLabel"
                FontSize="{TemplateBinding FontSize}"
                HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
                VerticalAlignment="{TemplateBinding VerticalContentAlignment}"
                Foreground="{TemplateBinding Foreground}">

                <ContentPresenter />

            </Label>
        </Grid>
    </Border>
</ControlTemplate>
```

Listing 7.26 *ControlTemplate* mit *VisualStateManager*

```
<VisualStateGroup.Transitions>

    <VisualTransition
        From="MouseOver"
        To="Pressed"
        GeneratedDuration="0:0:1">

        <Storyboard>

            <ColorAnimationUsingKeyFrames
                Storyboard.TargetProperty="Background.Color"
                Storyboard.TargetName="InnerGrid"
                FillBehavior="HoldEnd">

                <ColorAnimationUsingKeyFrames.KeyFrames>

                    <LinearColorKeyFrame
                        Value="Black"
                        KeyTime="0:0:1" />

                    <LinearColorKeyFrame
                        Value="DarkRed"
                        KeyTime="0:0:2" />

                </ColorAnimationUsingKeyFrames.KeyFrames>


            </ColorAnimationUsingKeyFrames>

        </Storyboard>

    </VisualTransition>

    <VisualTransition
        From="Pressed"
        To="Mouseover"
        GeneratedDuration="0:0:1">

        <Storyboard>
            <ColorAnimation
            Storyboard.TargetName="InnerGrid"
            Storyboard.TargetProperty="Background.Color"
            To="Black" />
        </Storyboard>

    </VisualTransition>

</VisualStateGroup.Transitions>
```

Listing 7.27 Definition von Übergängen

```
<VisualState x:Name="Normal" />

  <VisualState x:Name="Pressed">

      <Storyboard RepeatBehavior="Forever">

          <ColorAnimationUsingKeyFrames
                  Storyboard.TargetProperty="Background.Color"
                  Storyboard.TargetName="InnerGrid"
                  FillBehavior="HoldEnd">
```

```
            <ColorAnimationUsingKeyFrames.KeyFrames>

                <LinearColorKeyFrame
                        Value="Red"
                        KeyTime="0:0:1" />


                <LinearColorKeyFrame
                        Value="DarkRed"
                        KeyTime="0:0:2" />

            </ColorAnimationUsingKeyFrames.KeyFrames>

        </ColorAnimationUsingKeyFrames>

    </Storyboard>

</VisualState>

<VisualState x:Name="MouseOver">
    <Storyboard>
        <ColorAnimation
            Storyboard.TargetName="InnerGrid"
            Storyboard.TargetProperty="Background.Color"
            To="Blue" />
    </Storyboard>
</VisualState>
```

Listing 7.28 Definition der Zustände

Animation Easing Functions

WPF 4.0 erlaubt das Erstellen von Animationen, welche mathematischen Formeln folgen. Dazu kommen die so genannten *Animation Easing Functions*, die aus Silverlight übernommen wurden, zum Einsatz. Listing 7.29 zeigt ein Beispiel eines Storyboards, welches die Animation Easing Function `ElasticEase`, welche eine elastische Bewegung simuliert, verwendet. Weitere Easing Functions finden sich im Namespace `System.Windows.Media.Animation`.

```
<Storyboard x:Key="AnimateTarget">
    <DoubleAnimationUsingKeyFrames Storyboard.TargetName="Transform"
                            Storyboard.TargetProperty="ScaleX">
        <EasingDoubleKeyFrame KeyTime="0:0:0"
                        Value="0.0" />
        <EasingDoubleKeyFrame KeyTime="0:0:3"
                        Value="5.0">
            <EasingDoubleKeyFrame.EasingFunction>
                <ElasticEase EasingMode="EaseOut" />
            </EasingDoubleKeyFrame.EasingFunction>
        </EasingDoubleKeyFrame>
    </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

Listing 7.29 Verwendung einer Animation Easing Function

Zur Implementierung von eigenen Animation Easing Functions kann von der Klasse `EasingFunctionBase` abgeleitet werden (siehe Listing 7.30). Im Zuge dessen ist unter anderem die Methode `EaseIn` zu überschreiben. Diese muss die mathematische Funktion, an der sich die Animation orientieren soll, widerspiegeln und wird während der Animation immer wieder aufgerufen. Übergeben wird dabei die bereits verstrichene Zeit als Wert zwischen 0 (=Start der Animation) und 1 (=Ende der Animation). Diese wird von der Funktion auf das Ergebnis der mathematischen Funktion abgebildet. In Listing 7.30 wird beispielsweise die Quadratwurzel (`Sqrt`) verwendet. Dadurch ergibt sich eine Animation, welche langsam startet und im Laufe der Zeit exponentiell schneller wird.

```
class SqrtEase : EasingFunctionBase
{
    protected override double EaseInCore(double normalizedTime)
    {
        return Math.Sqrt(normalizedTime);
    }

    protected override System.Windows.Freezable CreateInstanceCore()
    {
        return new SqrtEase();
    }
}
```

Listing 7.30 Benutzerdefinierte Animation Easing Function

WPF Designer in Visual Studio

Auch der in Visual Studio integrierte WPF-Designer wurde erweitert. Die Performance des Designers wurde im Gegensatz zur Vorgängerversion erheblich verbessert und er kann nun für Silverlight-Applikationen gleichermaßen wie für WPF-Applikationen herangezogen werden.

Unterstützt wird nun auch Datenbindung für WPF via Drag & Drop. Somit besteht wie bei Windows Forms die Möglichkeit, Datenfelder aus der Datenquellenansicht direkt in ein WPF-Fenster zu ziehen. Der Code für die Steuerelemente sowie für die Datenbindung wird dabei vom Designer erstellt.

Daneben existiert nun GUI-Unterstützung für das Definieren von Datenbindungen sowie für das Zuweisen von Ressourcen (Abbildung 7.19). Sowohl für die Auswahl von Ressourcen als auch das Festlegen von Datenbindungsausdrücken stehen nun Dialogfelder zur Verfügung. Abbildung 7.20 zeigt zum Beispiel ein Dialogfeld, mit welchem definiert wurde, dass die im Eigenschaftenfenster ausgewählte Eigenschaft `Text` an die gleichnamige Eigenschaft des Elements `textBox1` gebunden werden soll. Dazu wurde unter `SourceElementName` sowie `textBox1` und unter `Path` die Eigenschaft `Text` ausgewählt, wobei Letzteres in der betrachteten Abbildung minimiert dargestellt wird.

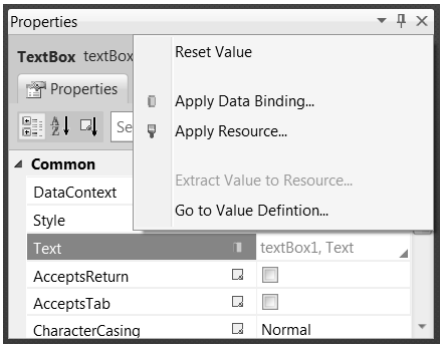

Abbildung 7.19 GUI-Unterstützung bei Definition der Datenbindung sowie beim Referenzieren einer Ressource

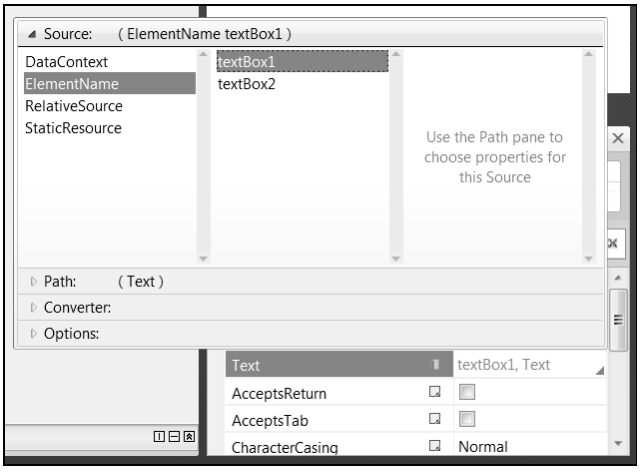

Abbildung 7.20 Definition der Datenbindung mit Unterstützung von Visual Studio

Weitere Neuerungen

Zu guter letzt widmet sich dieser Abschnitt kleineren Neuerungen, welche aufgrund ihres Umfangs kein eigenes Kapitel zugesprochen bekamen.

.NET 4.0. Client Profile

.NET 4.0 Client Profile stellt ein ca. 30 MB großes Subset von .NET Framework 4.0 dar. Es beinhaltet jene Elemente, welche in der Regel von Desktop-Applikationen benötigt werden. Somit soll die Größe der Installationsdateien so gering wie möglich gestaltet werden können. Im Gegensatz zu .NET Framework 3.5 SP1 ist das Client Profile in Version 4.0 für alle Plattformen verfügbar. Genau genommen besteht .NET Framework nun aus dem Client Profile sowie aus einem weiteren Teil, welcher den Namen *Extended* trägt. Damit sollen Konflikte zwischen dem Client Profile und dem vollständigen Framework vermieden werden. Darüber hinaus sehen viele Dokumentvorlagen in Visual Studio 2010, wie zum Beispiel jene für WPF-Applikationen, die Verwendung dieser Untermenge standardmäßig vor. Wer sich damit nicht zufrieden geben möchte, kann dies in den Projekteigenschaften sowie im Zuge der Erstellung von Setup and Deployment-Projekt ändern.

Datenbindung in WPF

Implementierungen von `ICommand` können nun an `InputBindings` gebunden werden (Listing 7.31). Dasselbe gilt für die Eigenschaft *text* des Elements *Run*.

```
<ListView.InputBindings>
    <KeyBinding Key="Delete" Command="{Binding RemoveIt}"/>
</ListView.InputBindings>
```

Listing 7.31 *InputBinding* und *Commands*

Auch dynamische Objekte können nun an WPF-Steuerelemente gebunden werden, sofern sie die Schnittstelle `IDynamicMetaObjectProvider` implementieren.

Aufruf von JavaScript-Funktionen aus Xbap

In Zukunft können XML Browser Applikationen (Xbap) mit jenen Seiten, in die sie eingebettet sind, kommunizieren. Listing 7.32 demonstriert dies, indem zunächst mittels `BrowserInteropHelper.HostScript` eine Referenz auf die Script-Umgebung der Webseite bezogen wird. Dabei ist zu beachten, dass diese Referenz vom Typ `dynamic` ist. Es handelt sich dabei um einen neuen Typ in C# 4.0, welcher für Interop-Szenarien mit dynamischen Sprachen gedacht ist. Anschließend wird die JavaScript-Methode `SayHello` aufgerufen. Listing 7.33 zeigt einen Ausschnitt der Html-Seite: Sie enthält die in Listing 7.32 aufgerufene Methode sowie die Xbap in einem IFrame.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    dynamic script = BrowserInteropHelper.HostScript;
    script.SayHello("World!!");
}
```

Listing 7.32 Aufruf der JavaScript-Funktion *SayHello*

```
[…]
<script>
  function SayHello(str) {
    alert("Hello " +str);
  }
</script>

[…]
<iframe src="Xbap.xbap" />
[…]
```

Listing 7.33 Website mit aufgerufener JavaScript-Funktion und Xpab

Änderungen

Neben den Neuerungen haben sich auch einige Details geändert. Da es sich hierbei großteils lediglich um Kleinigkeiten handelt, werden diese hier nicht beschrieben. Interessierte finden diese Details unter [WPF].

Fazit

Die Neuerungen in der WPF stellen in erster Linie eine Abrundung des bis dato vorhandenen Funktionsumfangs dar. Endlich befindet sich ein Data Grid im Lieferumfang der WPF und eine Unterstützung für Ribbons ist auch bereits in Greifweite. Der WPF-Designer wurde verbessert und in Punkte Animationen hat man Möglichkeiten, wie den Visual State Manager, aus Silverlight übernommen. Daneben können nun die neuen Möglichkeiten, die Windows 7 bietet, auf einfache Weise in WPF-Applikationen integriert werden.