BEREICHE IN WPF-GRID-LAYOUTS BENENNEN UND VERWENDEN

Zeilen und Spalten ade

Unter CSS gibt es benannte Bereiche im Grid-Layout. Wir übertragen das auf WPF.

Grid-Layouts in WPF haben ein Manko: Für jedes Element, das im Grid positioniert werden soll, müssen Zeilen- und Spaltenindizes angegeben werden. Fügt man im XAML-Code neue Zeilen oder Spalten ein oder löscht solche, dann müssen die Indizes für die nachfolgenden Elemente nachgeführt werden.

Zwar kann Visual Studio, sofern man die Operationen im Designer durchführt, die dazu nötigen Berechnungen automatisch vornehmen. Aber leider sind damit oft auch weitere, unerwünschte Einstellungsänderungen verbunden, sodass viele Programmierer es bevorzugen, die Anpassungen nur im XAML-Editor vorzunehmen. Hinzu kommt, dass die tatsächliche Anordnung im XAML-Code nicht gut erkennbar ist und der Designer nur helfen kann, wenn bereits zur Designzeit alles zur Verfügung steht, um das Layout anzeigen zu können.

Im Gegensatz zum WPF-Umfeld entwickeln sich die Layout-Systeme heutiger Webanwendungen unaufhörlich weiter. Moderne Browser unterstützen die aktuellen Neuentwicklungen von CSS, das inzwischen auch ein Grid-Layout zu bieten hat.

Zwangsläufig gibt es viele Ähnlichkeiten zum Layout-System des WPF-Grids – natürlich mit anderer Syntax. So lassen sich auch in CSS Zeilen und Spalten mit festen Größen, relativen Größenangaben oder automatischer Berechnung definieren. Und auch die Zuordnung kann über die Angabe von Zeilen- und Spaltenindizes erfolgen – aber sie muss es nicht. Denn CSS stellt noch eine Alternative bereit: *grid-template-areas* [1][2][3]. Ein Beispiel hierzu sehen Sie in Listing 1.

Listing 1: Grid-Layout in CSS

```
.container {
  display: grid;
  grid-template-columns: 80px 80px 80px 80px;
  grid-template-rows: auto;
        grid-gap: 5px;
  grid-template-areas:
    "header header header header  "
    "nav    main   main   sidebar1"
    "nav    main   main   sidebar2"
    "nav    footer footer footer  ";
}

.header{
  grid-area: header;
}

.footer{
  grid-area: footer;
}

nav{
  grid-area: nav;
}

.main{
   rid-area: main;
}

...
```

In CSS erfolgt über die Eigenschaft *grid-template-areas* eine textuelle Beschreibung des Layouts. Für jede Zeile wird ein String definiert, der den einzelnen Spalten Namen zuordnet. Ein Name kann mehrfach vorkommen, wenn sich der Bereich über mehrere Spalten beziehungsweise Zeilen erstreckt (Rowspan beziehungsweise Columnspan ist größer als 1). Ein Bereich muss aber immer rechteckig sein, also immer n x m Zellen umschließen.

Die Namen können anschließend verwendet werden, um einem Element eine Area zuzuordnen (Eigenschaft *grid-area*). Das Einstellen von Zeilen- und Spaltennummern sowie der Span-Eigenschaften entfällt dann komplett.

Ferner lässt CSS es auch zu, über die Eigenschaft *grid-gap* einheitliche Abstände zwischen den Elementen des Grids zu definieren. Auch das fehlt in WPF und muss für jedes Element einzeln eingestellt werden.

Umsetzung in WPF

Es finden sich im Web auch Diskussionen, wie man das WPF-Grid-Layout in dieser Hinsicht verbessern könnte (zum Beispiel [4]). Aber bei einigen Überlegungen stellt man fest, dass sich die Idee mit WPF-Bordmitteln bereits ganz gut umsetzen lässt. Die XAML-Syntax könnte beispielsweise so aussehen:

```
<Grid Margin="7" GridExtensions.Gap="7">
  <Grid.RowDefinitions>
    <RowDefinition Height="100*"/>
    <RowDefinition Height="100*"/>
    <RowDefinition Height="100*"/>
    <RowDefinition Height="100*"/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100*" />
```

```
    <ColumnDefinition Width="100*"/>
    <ColumnDefinition Width="100*"/>
    <ColumnDefinition Width="100*"/>
  </Grid.ColumnDefinitions>

  <GridExtensions.AreaDefinitions>
    <AreaRows>
      <AreaRow>eins zwei zwei drei</AreaRow>
      <AreaRow>eins zwei zwei drei</AreaRow>
      <AreaRow>eins fünf fünf fünf</AreaRow>
      <AreaRow>vier vier vier vier</AreaRow>
    </AreaRows>
  </GridExtensions.AreaDefinitions>

  <Button Content="1"
    GridExtensions.Area="eins" />
  <Button Content="2" GridExtensions.Area="zwei" />
  <Button Content="3" GridExtensions.Area="drei" />
  <Button Content="4" GridExtensions.Area="vier" />
  <Button Content="5" GridExtensions.Area="fünf" />

</Grid>
```

Dieses Layout ergibt sich aus der Definition der Areas im Grid (Bild 1)

Hier werden fünf Bereiche zeilenweise textuell beschrieben. So erstreckt sich beispielsweise Bereich „eins“ über drei Zeilen in der ersten Spalte und Bereich „zwei“ über zwei Zeilen und zwei Spalten, beginnend in der ersten Zeile und der zweiten Spalte.

Über eine Attached-Dependency-Property *GridExtensions.Gap* werden einheitliche Abstände festgelegt. Ein Beispiel für das resultierende Layout sehen Sie in Bild 1.

Die Implementierung hierzu ist ein wenig Fleißarbeit, aber keineswegs Hexenwerk. Zunächst werden einige Definitionen benötigt:

```
internal class AreaDefinition
{
  public int Row { get; set; }
  public int Column { get; set; }
  public int RowSpan { get; set; }
  public int ColumnSpan { get; set; }
}

// Definition einer Row
[TypeConverter(typeof(AreaRowConverter))]
public class AreaRow
{
  public string AreaNames { get; set; }
}

// Liste von Rows
public class AreaRows : List<AreaRow> { }

// Type converter zur Umwandlung von String -> AreaRow
internal class AreaRowConverter:TypeConverter
{
  public override bool CanConvertFrom(
    ITypeDescriptorContext context, Type sourceType)
  {
    return sourceType == typeof(string);
  }
  public override object ConvertFrom(
    ITypeDescriptorContext context,
    CultureInfo culture, object value)
  {
    return new AreaRow { AreaNames = (string)value };
  }
}
```

AreaDefinition kapselt die benötigten Werte für die Positionierung im WPF-Grid (*Row*, *Column*, *RowSpan*, *ColumnSpan*). Die textuelle Definition einer Zeile beschreibt die Klasse *AreaRow*. Zur einfacheren Verwendung im XAML-Code wird ihr ein *TypeConverter* zugeordnet (*AreaRowConverter*). Dieser nimmt einen String entgegen und setzt die Eigenschaft *AreaNames*.

Letztlich gibt es noch als Workaround wegen der fehlenden Möglichkeit, im XAML-Code Generics verwenden zu können, die Definition der Klasse *AreaRows*, die lediglich eine Auflistung von *AreaRow*-Objekten darstellt. Der Code reicht aus, um im XAML-Code Objekte vom Typ *AreaRows* wie im Listing gezeigt darstellen zu können.

Die Zuordnung zum Grid erfolgt über die Attached-Dependency-Property *GridExtensions.AreaDefinitions* (Listing 2). Beim Setzen dieser Eigenschaft (Methode *OnAreaDefinitionsChanged*) werden zunächst einige Plausibilitätsbedingungen überprüft und dann eine Analyse der Area-Definition durchgeführt. Die Daten werden dann in einer Instanz von *AreaDefinition* gespeichert.

Zu berücksichtigen ist, dass Attached-Dependency-Properties ausschließlich über statische Methoden und Eigenschaften beschrieben werden können. Eine Instanzierung der umschließenden Klasse erfolgt nicht, sodass natürlich das mehrfache Setzen der Property gesondert behandelt werden muss. Schließlich könnten im Programm ja mehrere Grids unabhängig voneinander den Mechanismus nutzen.

Daher werden die *AreaDefinition*-Objekte anschließend in einem Dictionary verlinkt (*AreaDefinitions*), das als Schlüs- ▶

sel jeweils die Referenz des Grid-Objekts verwendet. Eine zweite Attached-Dependency-Property wird benötigt, um einem Kind-Element des Grids einen Area-Namen zuzuweisen (*Area*, Listing 3).

Hier wird zunächst sichergestellt, dass das Parent-Element auch tatsächlich vom Typ *Grid* ist und es auch wirklich die gewünschte Area-Definition beinhaltet. Die Daten werden gelesen und die Attached-Dependency-Properties des Grids (das sind in diesem Fall die Eigenschaften *Column, Row, ColumnSpan, RowSpan*) gesetzt. Zu guter Letzt bleibt noch die Implementierung der *Gap*-Property, wie es das nun folgende Listing zeigt.

Listing 2: Setzen und Verwalten der AreaDefinitions mithilfe einer Attached Dependency Property

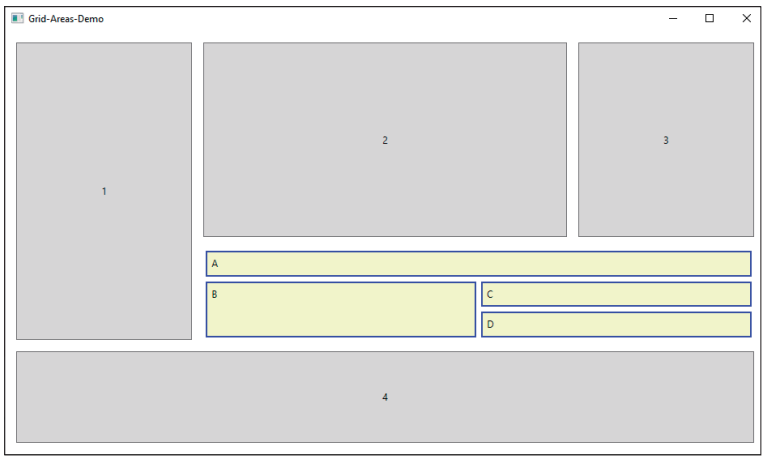
```
public class GridExtensions
{
  // Hashtable zum Verwalten aller
  // AreaDefinitions im Programm
  private static Dictionary<DependencyObject,
    Dictionary<string, AreaDefinition>>
    AreaDefinitions = new Dictionary<DependencyObject
    ,Dictionary<string, AreaDefinition>>();

  // Attached Dependency Property "AreaDefinitions"

  public static AreaRows GetAreaDefinitions(
    DependencyObject obj)...
  public static void SetAreaDefinitions(
    DependencyObject obj, AreaRows value)...

  public static readonly DependencyProperty
    AreaDefinitionsProperty = DependencyProperty
     .RegisterAttached("AreaDefinitions",
     typeof(AreaRows), typeof(GridExtensions),
     new FrameworkPropertyMetadata(
     OnAreaDefinitionsChanged));

  // Setzen der Property führt zur Analyse
  // und Anlegen der AreaDefinitions
  private static void OnAreaDefinitionsChanged(
  DependencyObject d,
    DependencyPropertyChangedEventArgs e)
  {
    // Zieltyp muss ein Grid sein
    if (!(d is Grid)) throw new ApplicationException(
      "Area definition can only be set on type Grid");

    // Definitionen dürfen nur einmal vorhanden sein
    if (AreaDefinitions.ContainsKey(d)) throw new
      ApplicationException(
        "Only one area definition allowed");

    var areaDefs = new Dictionary<string,
      AreaDefinition>();
    AreaDefinitions[d] = areaDefs;
    for (int row = 0;
      row < ((AreaRows)e.NewValue).Count; row++)
    {
      // In einer Row enthaltene Namen
      var names = ((AreaRows)e.NewValue)[row]
        .AreaNames.Split(' ');

      for (int col = 0; col < names.Length; col++)
      {
        var name = names[col];
        if (areaDefs.ContainsKey(name))
        // Name schon verwendet?
        {
          if (row == areaDefs[name].Row)
          // Definition erste Zeile des Vorkommens
          {
            if (col == areaDefs[name].Column +
              areaDefs[name].ColumnSpan)
              // Folgespalte?
              areaDefs[name].ColumnSpan++;
            else
              throw new ApplicationException(
              "Area must be rectangular");
          }
          else // Folgezeilen
          {
            // Weitere Zeile für diesen Namen?
            if (col == areaDefs[name].Column)
              areaDefs[name].RowSpan++;

            if (col < areaDefs[name].Column ||
              col >= areaDefs[name].Column +
                areaDefs[name].ColumnSpan)
              throw new ApplicationException(
                "Area must be rectangular");
          }
        }
        else
        {
          // Name ist neu -> neue Definition anlegen
          areaDefs[name] = new AreaDefinition {
            Row = row, Column = col, ColumnSpan = 1,
            RowSpan = 1 };
        }
      }
    }
  }
  ...
}
```

```
public class GridExtensions
{
  ...
  public static Thickness GetGap(
    DependencyObject obj)...
  public static void SetGap(DependencyObject obj,
    Thickness value)...

  public static readonly DependencyProperty GapProperty
    = DependencyProperty.RegisterAttached("Gap",
    typeof(Thickness), typeof(GridExtensions),
    new FrameworkPropertyMetadata(new Thickness(0),
    OnGapChanged));

  private static void OnGapChanged(DependencyObject d,
    DependencyPropertyChangedEventArgs e)
  {
    var grid = d as Grid;
    grid.Initialized += Grid_Initialized;
  }

  private static void Grid_Initialized(object sender,
    EventArgs e)
  {
    var grid = sender as Grid;
    grid.Initialized -= Grid_Initialized;

    var margin = GetGap(grid);
    foreach (FrameworkElement item in grid.Children)
    {
      item.Margin = margin;
    }
  }
}
```

▶

Listing 3: Einstellen der Grid-Parameter beim Setzen der Area-Eigenschaft

```
public class GridExtensions
{
  // Attached Dependency Property "Area"
  public static string GetArea(
    DependencyObject obj)...

  public static void SetArea(DependencyObject obj,
    string value)...

  public static readonly DependencyProperty
    AreaProperty = DependencyProperty
      .RegisterAttached("Area", typeof(string),
        typeof(GridExtensions),
        new FrameworkPropertyMetadata(OnAreaChanged));

  // Setzen der Property führt zum Setzen der Attached
  // Dependency Properties des Grids (Row, Column ...)
  private static void OnAreaChanged(
    DependencyObject d,
    DependencyPropertyChangedEventArgs e)
  {
    var fe = d as FrameworkElement;

    // Parent wird benötigt, daher nur Objekte vom
    // Typ FrameworkElement zulässig
    if (fe == null) throw new ApplicationException(
      "Area can only be set on Frameworkelements");

    var grid = fe.Parent as Grid;

    // Parent muss ein Grid sein
    if (grid == null)
      throw new ApplicationException(
        "Area can only be set on children of a Grid");

    // Die AreadDefinitions des Grids müssen
    // bereits definiert sein
    if (AreaDefinitions == null ||
      !AreaDefinitions.ContainsKey(grid))
        throw new ApplicationException(
          "Area definitions not set");

    var areaDefs = AreaDefinitions[grid];

    // Der gewünschte Area-Name muss bereits
    // definiert sein
    if (!areaDefs.ContainsKey((string)e.NewValue))
      throw new ApplicationException(
        $"Area name '{e.NewValue}' not defined");

      var areaDef = areaDefs[(string)e.NewValue];

      // Setzen der Attached Dependency Properties
      // des Grids
      d.SetValue(Grid.ColumnProperty, areaDef.Column);
      d.SetValue(Grid.RowProperty, areaDef.Row);
      d.SetValue(Grid.RowSpanProperty,
        areaDef.RowSpan);
      d.SetValue(Grid.ColumnSpanProperty,
        areaDef.ColumnSpan);
    }
    ...
  }
```



Auch geschachtelte Grid-Layouts können das Area-System nutzen (Bild 2)

Sie ist vom Typ *Thickness*, sodass sich bei Bedarf horizontal und vertikal unterschiedliche Abstände einstellen lassen.

Das Setzen der *Margin*-Eigenschaften der Kindelemente muss allerdings verzögert erfolgen, da beim Setzen dieser Eigenschaft in der Regel die Kindelemente noch nicht instanziert beziehungsweise in die Liste *Children* aufgenommen worden sind.

Daher erfolgt dieser Vorgang erst im *Initialize*-Event des Grid-Objekts. Bei dieser Implementierung werden individuelle Margin-Einstellungen der Kindelemente überschrieben.

Schachteln von Grids

Die Beispiel-Implementierung lässt auch die Schachtelung mehrerer Grids ineinander zu, wie am Beispiel von Bild 2 zu sehen ist. Das folgende Listing zeigt diese Verschachtelung im XAML-Code:

```
<Grid Margin="7" GridExtensions.Gap="7">

  <Grid.RowDefinitions>...
  <Grid.ColumnDefinitions>...

  <GridExtensions.AreaDefinitions>
    <AreaRows>
      <AreaRow>eins zwei zwei drei</AreaRow>
      <AreaRow>eins zwei zwei drei</AreaRow>
      <AreaRow>eins fünf fünf fünf</AreaRow>
      <AreaRow>vier vier vier vier</AreaRow>
    </AreaRows>
  </GridExtensions.AreaDefinitions>

  <Button Content="1" GridExtensions.Area="eins" />...

  <Grid GridExtensions.Area="fünf"
    GridExtensions.Gap="3">
    <Grid.RowDefinitions>...
    <Grid.ColumnDefinitions>...

    <GridExtensions.AreaDefinitions>
      <AreaRows>
        <AreaRow>aaa aaa</AreaRow>
        <AreaRow>bbb ccc</AreaRow>
        <AreaRow>bbb ddd</AreaRow>
      </AreaRows>
    </GridExtensions.AreaDefinitions>

    <Label  Content="A" GridExtensions
      .Area="aaa" Background=... />
    <Label  Content="B" GridExtensions
      .Area="bbb" Background=... />
    <Label  Content="C" GridExtensions
      .Area="ccc" Background=... />
    <Label  Content="D" GridExtensions
      .Area="ddd" Background=... />
  </Grid>

</Grid>
```

Auch für das untergeordnete Grid lassen sich wieder über die Eigenschaft *Gap* die Abstände der Kindelemente zueinander einstellen.

Fazit

Der Artikel hat gezeigt, dass sich mit vergleichsweise einfachen Handgriffen das Grid-Layout in WPF so erweitern lässt, dass ähnlich zur Definition in CSS benannte Bereiche definiert und genutzt werden können.

Die beschriebene Implementierung erhebt keinen Anspruch auf Vollständigkeit. Sie soll vielmehr als Basis für eigene Implementierungsideen dienen.

Die Möglichkeiten, die CSS zu bieten hat, sind noch wesentlich komplexer.

Aber bereits die Area- und die Gap-Einstellungen können im Umgang mit WPF-Grid-Layouts eine wertvolle Bereicherung darstellen. ■

[1] Grid Areas bei Mozilla,
www.dotnetpro.de/SL1909WPFGridAreas1
[2] CSS Grid Item,
www.dotnetpro.de/SL1909WPFGridAreas2
[3] A complete guide go Grid | CSS,
www.dotnetpro.de/SL1909WPFGridAreas3
[4] Feature Proposal: Make Grid Better,
www.dotnetpro.de/SL1909WPFGridAreas4
[5] Beispielimplementierung,
www.dotnetpro.de/SL1909WPFGridAreas5

Dr. Joachim Fuchs
ist begeisterter Anhänger von Microsofts .NET-Philosophie. Er arbeitet als Softwarearchitekt, Berater und Dozent im Expertennetzwerk www.it-visions.de. Seine Schwerpunkte liegen derzeit bei XAML- und Web-UI-Technologien.
dnp@fuechse-online.de

dnpCode A1909WPFGridAreas