Ein pragmatischer Ansatz zur Globalisierung von WPF-Anwendungen

Unternehmen-Babelfisch

Anwendungen mehrsprachig und für verschiedene Kulturen zu gestalten geht nicht ohne Mehraufwand. Mit WPF-Bordmitteln kann man sich recht einfach eine flexible Infrastruktur für diese Aufgabe schaffen.

Auf einen Blick

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 WPF und Silverlight. Er ist Autor mehrerer Bücher und zahlreicher Fachartikel.

Inhalt

dnpCode
A1412SpracheWPF

In vielen WPF-Schulungen und -Beratungsgesprächen stellt sich immer wieder dieselbe Frage, nämlich wie man es sinnvoll angeht, Sprachumschaltungen in der Anwendung vorzusehen. Microsofts Ansatz [1] ist sehr umständlich und steif, für die Praxis ungeeignet. dotnetpro hat in der Vergangenheit auch bereits Lösungsansätze vorgestellt [2, 3]. Viele Firmen wünschen sich jedoch wesentlich mehr Flexibilität in Bezug auf das Bereitstellen der übersetzten Texte und deren Einbindung ins Programm. Insbesondere werden immer wieder die folgenden Anforderungen genannt:

- Freie Wahl der Datenquelle (Ressourcen, Datenbank, XML-Datei, Ini-Datei, Webservice et cetera).
- Zentrale Austauschbarkeit der Datenquelle (heute XML, demnächst DB).
- Umschalten der Sprache im laufenden Betrieb (weil verschiedensprachige Anwender mit demselben Rechner arbeiten müssen).
- Möglichst einfache Einbindung im XAML-Code.
- Erkennen und Loggen fehlender Daten (Bezüge auf IDs, die in der Datenquelle nicht vorhanden sind).

Wie lassen sich nun diese Anforderungen in WPF umsetzen? Die klassischen Ansätze, wie beispielsweise Bezüge im XAML-Code über *{x:Static ...}* oder das Setzen der *XmlLanguage*-Eigenschaft über *OverrideMetadata* im statischen Konstruktor der *Application*-Klasse, sind hier wenig hilfreich, da sie keine dynamische Umschaltung zulassen. Auch Vorgehensweisen aus Windows Forms, Delphi oder ähnlichen Systemen, bei denen im Code auf die betreffenden Controls zugegriffen wird und die Inhalte imperativ gesetzt werden, lassen sich nicht gut nach WPF übertragen. Dieses Konzept widerspricht meist den gängigen Entwurfsmustern wie Model-View-ViewModel, bei dem die Oberflächenobjekte ihre Informationen über Datenbindungen beziehen.

Was das Austauschen von Inhalten zur Laufzeit betrifft, drängen sich in WPF zwei leistungsfähige Mechanismen förmlich auf: *Binding* und *DynamicResource*. Im ersten Fall stellt man die Daten über Objekteigenschaften bereit, im zweiten über Resource Dictionaries. Bevor man sich jedoch für eine der Varianten entscheidet und den XAML-Code nun mit Tausenden Bindungsausdrücken übersät, sollte man noch eine weitere Abstraktionsebene einfügen, damit spätere Änderungen leichter möglich sind.

Abstraktion durch eine Markup Extension

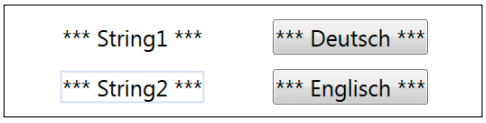
Das Konzept der Markup Extensions bietet in WPF eine Möglichkeit, XAML-Code zu vereinfachen. Sicher kennen Sie Extension-Ausdrücke wie *{Binding}* oder *{StaticResource}*. Solche Erweiterungen lassen sich aber auch recht einfach selbst erstellen. Dazu muss lediglich eine Klasse von *MarkupExtension* abgeleitet und die Methode *ProvideValue* überschrieben werden. Hinzu kommt noch, dass man möglichst die Namenskonvention einhalten sollte, die besagt, dass der Name der Klasse auf *Extension* enden soll. Der Rückgabewert von *ProvideValue* stellt dann das Ergebnis der *MarkupExtension* dar, das der betreffenden Eigenschaft zugewiesen wird. Einen ersten Ansatz hierfür sowie eine Einbindung im XAML-Code sehen Sie in **Listing 1**. Die Einbindung im XAML-Code folgt dabei folgender Syntax:

```
{Extension Eigenschaft1=Wert1...}
```

Hier wird der parameterlose Konstruktor der Klasse aufgerufen und anschließend werden die Eigenschaften gesetzt. Wenn wie im vorliegenden Fall nur eine Eigenschaft zum Einsatz kommt, bietet es sich an, eine Konstruktorüberladung vorzusehen, die diese Eigenschaft als Parameter entgegennimmt (zweiter Konstruktor in **Listing 1**). Dann kann alternativ die folgende Syntax verwendet werden:

```
{Extension Eigenschaftswert}
```

Der Konstruktor muss in diesem Fall selbst für die Zuweisung der Eigenschaft sorgen. Microsoft hat dies bei den gängigen Markup-Erweiterungen so implementiert.



[Abb. 1] Texte aus einer Markup Extension.

In **Abbildung 1** sehen Sie das Ergebnis der Benutzung der Markup Extension. Die im XAML-Code definierten IDs werden in der Anzeige zunächst zur Demonstration mit Sternchen eingeschlossen. Dieser Code muss später noch durch eine passende Datenbindung ersetzt werden.

Kulturinformationen kapseln

Neben den sprachabhängigen Texten sind für die korrekte Darstellung weitere kulturspezifische Informationen wichtig. Diese sollten in Form einer Klasse gekapselt werden, damit sie einerseits für die Sprachauswahl und andererseits für die korrekte Darstellung in der Oberfläche schnell und einfach zugreifbar sind. Die Klasse *SupportedCulture* **(Listing 2)** erfüllt diesen Zweck. Die Eigenschaft *CultureInfo* bietet den Zugriff auf das Objekt vom gleichnamigen Typ aus dem Framework, das alle Kulturinformationen wie Sprachnamen, Textformate oder Kalender enthält, *Flag* verweist auf ein Bild, das für die leichtere Wiedererkennung der Kultur in der Oberfläche eingesetzt werden kann. Das Symbol wird aus einem Resource Dictionary entnommen. Auf die Eigenschaften *XmlLanguage* und *FlowDirection* wird später noch eingegangen. Die Klasse verfügt über einen Indexer, der als Parameter die ID aus der Markup Extension durchgereicht bekommen wird. Er greift auf das Repository zu, das die zur Kultur gehörigen Texte enthält. Wie dieses aussehen kann, wird ebenfalls weiter unten behandelt. Zusätzlich verfügt die Klasse noch über einen Selektionsmechanismus, damit eine möglichst einfache Listenauswahl über XAML-Bindungen möglich ist.

Zentrale Vermittlung

Eine Singleton-Implementierung der Klasse *GlobalizationUtilities* stellt eine Liste der verfügbaren Kulturen sowie die ausgewählte Kultur als Eigenschaften bereit **(Listing 3)**. Bei Änderung der Kulturauswahl wird indirekt *SelectCulture* aufgerufen. Hier werden die Eigenschaften *CurrentCulture* sowie *CurrentUICulture* des GUI-Threads auf die ausgewählte Kultur gesetzt. Dies ist notwendig, damit zum Beispiel Formatierungen, die im C#-Code vorgenommen werden, auch passend zur Kultur durchgeführt werden. Ferner werden dadurch auch die Sprachen für Systemmeldungen und Standarddialoge in der richtigen Sprache zur Anzeige gebracht (Installation des Language Packs des Frameworks vorausgesetzt).

Listing 1

Erste Implementierung und Einsatz einer Markup Extension.

```
public class CultureResourceExtension : MarkupExtension
{
  public string Id { get; set; }

  public CultureResourceExtension()
  {
  }

  public CultureResourceExtension(string id)
  {
    this.Id = id;
  }

  // Erster Ansatz für die Implementierung
  public override object ProvideValue(IServiceProvider serviceProvider)
  {
    return "*** " + this.Id + " ***";
  }
}

XAML:
<Window xmlns:local="clr-namespace:...">
  <TextBlock ... Text="{local:CultureResource Id=String1}" />
  <TextBlock ... Text="{local:CultureResource String2}" />
  <Button Content="{local:CultureResource Deutsch}" .../>
  <Button Content="{local:CultureResource Englisch}" ..."/>
```

Listing 2

Kapseln kulturspezifischer Informationen.

```
public class SupportedCulture : SelectableItem
{
  // Alle Informationen zur Kultur
  public CultureInfo CultureInfo { get; set; }

  // Ein Symbol für die grafische Darstellung
  public ImageSource Flag { get; set; }

  // Wert für die Spracheinstellung im XAML-Code
  public XmlLanguage XmlLanguage { get; set; }

  // Textausrichtung
  public FlowDirection FlowDirection { get; set; }

  // Indexer für den Zugriff auf die Texte über eine Id
  public string this[string id]
  {
    get
    {
      return GlobalizationRepository.GetString(CultureInfo.Name,id);
    }
  }

  // Konstruktor zur initialen Einrichtung der Eigenschaften auf Basis
  // der vorgegebenen Kultur
  public SupportedCulture(string cultureId, EventHandler isSelectedChangedEventHandler)
  {
    CultureInfo = CultureInfo.GetCultureInfo(cultureId);
    Flag = (ImageSource)App.Current.FindResource(cultureId);
    XmlLanguage = XmlLanguage.GetLanguage(CultureInfo.Name);
    FlowDirection = CultureInfo.TextInfo.IsRightToLeft ?
      FlowDirection.RightToLeft : FlowDirection.LeftToRight;
    IsSelectedChanged += isSelectedChangedEventHandler;
  }
}
```

Listing 3

Die Schaltzentrale für die Sprachumschaltung.

```
public class GlobalizationUtilities : INotifyPropertyChanged
{
  // Singleton-Referenz
  public static readonly GlobalizationUtilities TheInstance =
    new GlobalizationUtilities();
  static GlobalizationUtilities() { }

  // Liste der unterstützten Kulturen
  public List<SupportedCulture> SupportedCultures { get; set; }

  // Ausgewählte Kultur
  public SupportedCulture SelectedCulture { get; set; }
  public event EventHandler SelectedCultureChanged;

  // Konstruktor. Einrichten der verfügbaren Kulturen
  public GlobalizationUtilities()
  {
    SupportedCultures = new List<SupportedCulture>();
    SupportedCultures.Add(new SupportedCulture("de-DE",
      GlobalizationUtilities_IsSelectedChanged));
    SupportedCultures.Add(new SupportedCulture("en-GB",
      GlobalizationUtilities_IsSelectedChanged));
    SupportedCultures.Add(new SupportedCulture("en-US",
      GlobalizationUtilities_IsSelectedChanged));
    SupportedCultures.Add(new SupportedCulture("fr-FR“,
      GlobalizationUtilities_IsSelectedChanged));
    SupportedCultures.Add(new SupportedCulture("fa-IR",
      GlobalizationUtilities_IsSelectedChanged));

    this.SelectedCulture = SupportedCultures.FirstOrDefault();

  }

  // Umschalten der Kultur
  private void SelectCulture(SupportedCulture culture)
  {
    ...
    SelectedCulture = culture;

    // GUI-Thread passend einstellen
    Thread.CurrentThread.CurrentCulture = culture.CultureInfo;
    Thread.CurrentThread.CurrentUICulture = culture.CultureInfo;

    // Melden, dass sich alle Eigenschaften geändert haben
    if (PropertyChanged != null) PropertyChanged(this, new PropertyChangedEventArgs(null));
    if (SelectedCultureChanged != null) SelectedCultureChanged(this, EventArgs.Empty);
  }
  ...
}
```

Auf die Singleton-Instanz der Klasse kann über das statische Feld *TheInstance* zugegriffen werden. Damit lässt sich nun die Implementierung der oben beschriebenen Markup Extension vervollständigen (Listing 4). Hier wird eine Instanz der Klasse *Binding* (abgeleitet ebenfalls von *MarkupExtension*, nur leider ohne Einhaltung der Namenskonvention) erzeugt und dem Konstruktor der Ausdruck für den Binding-Path mitgegeben. Dieser bezieht sich auf die Eigenschaft *SelectedCulture* und greift auf den oben beschriebenen Indexer zu. Die Datenquelle, das Singleton-Objekt von *GlobalizationUtilities*, wird über die Eigenschaft *Source* zugewiesen. Da Binding ebenfalls eine Markup Extension ist, kann deren Methode *ProvideValue* für den Rückgabewert verwendet werden. So schließt sich der Kreis von der Definition im XAML-Code:

```
Text="{glob:CultureResource Begrüßung}"
```

bis zum Zugriff auf das Repository im Indexer der Klasse *SupportedCulture*.

Wie bereits oben beschrieben, könnte man alternativ statt der Datenbindung auch eine Verknüpfung mittels *DynamicResource* an dieser Stelle in der Markup Extension implementieren. *SupportedCulture* müsste dann das entsprechende *ResourceDictionary* erstellen und/oder auswählen. Im XAML-Code würde sich nichts ändern, sodass die Umstellung nur wenige Codestellen beträfe.

Die Datenquelle

Wie der bislang gezeigte Code bereits vermuten lässt, sind Sie bei der Implementierung der Klasse *GlobalizationRepository* völlig frei. Eine Beispielimplementierung zeigt Listing 5. Hier ist die Methode *GetString* so implementiert worden, dass sie die Texte aus einer XML-Struktur bezieht. Einen Ausschnitt dieser Struktur sehen Sie in Listing 6.

An dieser Stelle kann auch festgestellt werden, ob die angeforderte ID für die eingestellte Sprache unterstützt wird. Im Beispiel wird im Fehlerfall die ID, eingerahmt mit Ausrufezeichen, zurückgegeben. Aber auch zusätzliche Schritte sind denkbar. Ein Eintrag in eine Log-Datei könnte darauf aufmerksam machen, dass entweder eine ID fehlt oder der Programmierer sich verschrieben hat. Auch das automatische Erstellen eines Eintrags im Repository wäre leicht möglich, sodass später nur die Texte nachgetragen werden müssten.

Implementieren Sie diese Klasse so, dass sie Ihren Anforderungen genügt. Sie können die Texte aus Assembly-Ressourcen, aus Datenbanken, Textdateien oder sonstigen Ablagen beziehen. Alternativ lassen sich auch Webdienste für die automatische Übersetzung aufrufen, um die Texte online abzurufen. Oder ein Helpdesk in Indien, wo ein „sprachkundiger“ Mitarbeiter die Übersetzung direkt eintippt. Alles eine Frage der Abwägung, ob man mehr Wert auf Qualität oder auf den Spaßfaktor legt.

Mit diesem Ansatz lassen sich bestehende Datenquellen nutzen. Niemand wird dazu gezwungen, Assembly-Ressourcen zu verwenden, wie es in anderen Lösungen teilweise der Fall ist.

Zwischenbilanz

In Abbildung 2 sehen Sie den bislang erreichten Stand. Ein Menü bietet die ver-

fügbaren Kulturen an, jeweils angezeigt durch das Flaggensymbol und den Namen der Kultur in der jeweiligen Landessprache. Dies ist nur eine Beispielimplementierung, die Sie auf der Heft-CD finden können und die hier aus Platzgründen auch nicht näher beschrieben werden soll. Das *CultureInfo*-Objekt bietet den Kulturnamen auch in der Windows-Sprache (Eigenschaft *DisplayName*) oder in Englisch (Eigenschaft *EnglishName*) an. Die Symbole helfen beim Wiederfinden der eigenen Kultur, falls eine für den Anwender völlig unverständliche Sprache ausgewählt worden ist.

Die Umschaltung der Texte über die Markup Extension mittels ID mit verknüpften Texten in einer XML-Struktur funktioniert nun schon so weit. Im Demo-Fenster gibt es noch zwei Textblöcke, die wie folgt mit einem ViewModel verbunden sind:

```
<TextBlock Text="{Binding AktuellesDatum,
 StringFormat=G}"/>
<TextBlock Text="{Binding DatumAlsText}"/>
```

Der zweite Textblock bekommt seine Daten von einer Eigenschaft vom Typ *String*, die das aktuelle Datum bereits im ViewModel mittels *ToString("G")* formatiert hat. Diese Formatierung funktioniert bereits korrekt dank der oben beschriebenen Umschaltung der Thread-Culture, da sie ja vom C#-Code durchgeführt wird. Der erste Textblock hingegen ist direkt an eine Eigenschaft vom Typ *DateTime* gebunden und stellt die Formatierung im Bindungsausdruck im XAML-Code ein. Hier funktioniert die Formatierung nicht korrekt, wie am Beispiel der Auswahl für *Deutsch* zu sehen ist.

XAML und die Kulturen

Dass sich die Formatierung im XAML-Code anders verhält als im C#-Code, ist auf den ersten Blick verwunderlich. Der Grund dafür ist, dass sich Microsoft seinerzeit dazu entschieden hat, Bindungsausdrücke im XAML-Code nicht mit der Culture-Einstellung des laufenden Threads zu verknüpfen, sondern mit der des XML-Codes. Und die Standardeinstellung der Kultur in XML ist – wie könnte es auch anders sein – *en-US*.

Diese Kultureinstellung muss nun bei der Sprachumschaltung auch noch angepasst werden. Die eingangs erwähnte Vorgehensweise über die Metadaten hilft hier leider nicht, da sich diese nur umstellen lassen, wenn die betreffenden Datentypen noch nicht verwendet worden sind.

Listing 4

Die Markup Extension formuliert die benötigte Datenbindung.

```
public class CultureResourceExtension : MarkupExtension
{
  ...
  public override object ProvideValue(IServiceProvider serviceProvider)
  {
    Binding binding = new Binding("SelectedCulture[" + Id + "]");
    binding.Source = GlobalizationUtilities.TheInstance;
    return binding.ProvideValue(serviceProvider);
  }
}
```

Listing 5

Beispiel für ein Repository: Texte aus einer XML-Struktur.

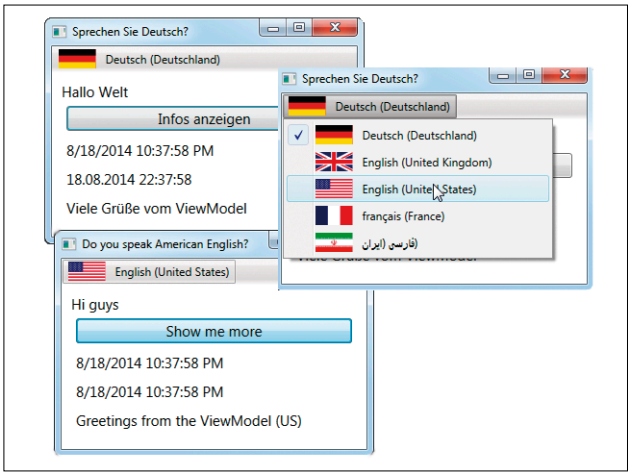
```
public class GlobalizationRepository
{
  // Beispielimplementierung für in einer XML-Struktur abgelegte Texte
  public static string GetString(string cultureId, string id)
  {
    XDocument doc = XDocument.Load("data/texts.xml");
    var xCulture = doc.Root.Elements("Culture").FirstOrDefault
      (c => c.Attribute("id").Value == cultureId);
    if (xCulture == null) return "!!" + id + "!!";
    var xEntry = xCulture.Elements("Entry").FirstOrDefault(
      e => e.Attribute("id").Value == id);
    if (xEntry == null) return "!" + id + "!";
    return xEntry.Value;
  }
}
```

Listing 6

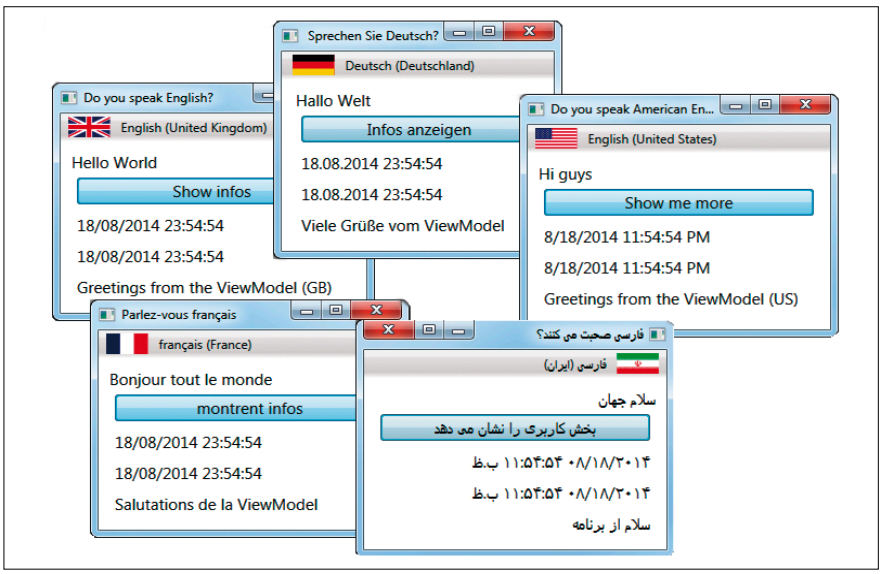
Die Datenbasis für die Texte.

```
<?xml version="1.0" encoding="utf-8" ?>
<Texts>
  <Culture id="de-DE">
    <Entry id="Begrüßung">Hallo Welt</Entry>
    <Entry id="InfosAnzeigen">Infos anzeigen</Entry>
    <Entry id="BegrüßungAusVM">Viele Grüße vom ViewModel</Entry>
    <Entry id="Fenstertitel">Sprechen Sie Deutsch?</Entry>
  </Culture>
  <Culture id="en-GB">
    <Entry id="Begrüßung">Hello World</Entry>
    <Entry id="InfosAnzeigen">Show infos</Entry>
    <Entry id="BegrüßungAusVM">Greetings from the ViewModel (GB)</Entry>
    <Entry id="Fenstertitel">Do you speak English?</Entry>
  </Culture>
    ...
</Texts>
```

Nun ist die dynamische Anpassung der XML-Kultureinstellung aber auch kein Hexenwerk. Auf WPF-Seite ist hierfür die Eigenschaft *Language* aus der Klasse *FrameworkElement* zuständig. Sie ist vom Typ *XmlLanguage* und kann jederzeit umgeschaltet werden. Die Daten hierfür stehen in *SupportedCulture* (siehe **Listing 2**) bereits zur Verfügung. Sie müssen nur noch zentral im Hauptfenster bezie-



[Abb. 2] Die ersten Fortschritte bei der Lokalisierung.



[Abb. 3] Übersetzte Texte und kulturspezifische Formatierungen.

Listing 7

Einstellen von Language und FlowDirection über Datenbindungen in einer Window-Basisklasse.

```
public class LocalizedWindow : Window
{
  // Einrichten der für die Lokalisierung notwendigen Datenbindungen
  public LocalizedWindow()
  {
    Binding binding = new Binding("SelectedCulture.XmlLanguage");
    binding.Source = GlobalizationUtilities.TheInstance;
    this.SetBinding(LanguageProperty, binding);

    binding = new Binding("SelectedCulture.FlowDirection");
    binding.Source = GlobalizationUtilities.TheInstance;
    this.SetBinding(FlowDirectionProperty, binding);

  }
}
```

hungsweise allen anderen derzeit sichtbaren Fenstern umgeschaltet werden. Das lässt sich wiederum einfach über eine Datenbindung einrichten.

Für die Einrichtung dieser Datenbindung können verschiedene Wege eingeschlagen werden. Gibt es nur ein einziges Fenster, dann reicht die einmalige Bindung im XAML-Code oder im Konstruktor der Fensterklasse. Müssen mehrere Fenster angepasst werden, dann empfiehlt sich der Einsatz einer Basisklasse für alle betreffenden Fenster (siehe **Listing 7**). Bei Anpassung bestehender Fensterklassen ist zu berücksichtigen, dass der Datentyp sowohl im XAML-Code als auch im Code-behind angepasst werden muss.

Ist das Anpassen der Fensterklasse nicht gewünscht oder nicht möglich, kann man alternativ auch ein Behavior programmieren, das man dann an die jeweiligen Fenster knüpft. Dieses Behavior implementiert die Datenbindung in der *OnAttached*-Methode und erfüllt damit denselben Zweck. Auch mit einer Attached Dependency Property ließe sich die Aufgabe lösen. Viele Wege sind hier denkbar.

Bei der Gelegenheit wird, wie ebenfalls in **Listing 7** zu sehen, gleich noch eine weitere Eigenschaft gebunden: *FlowDirection*. Diese regelt den Textfluss – von links nach rechts oder von rechts nach links. Auch das ist beim Umschalten der Kultur zu berücksichtigen. Wie bei der XML-Kultureinstellung reicht auch hier die Bindung im obersten Knoten des Visual Trees. Beide Eigenschaften werden an die Kindelemente vererbt.

Abbildung 3 zeigt das Ergebnis für die unterstützten Kulturen. Die Datumsformate sind nun für beide Fälle (Formatierung in C# und Formatierung in XAML) korrekt. Auch der Schreibfluss von rechts nach links, hier für die iranische Kultur, wird richtig eingestellt.

Das ViewModel und die Kulturen

Oft wird es so sein, dass ein ViewModel Informationen kulturspezifisch aufbereitet und über Eigenschaften der View zur Verfügung stellt, wie im Beispiel anhand der Zeitangabe geschehen. Dann müssen solche Daten beim Wechsel der Kultureinstellung aufgefrischt werden. Insofern bietet es sich an, in einer ViewModel-Basisklasse die notwendige Infrastruktur bereits vorzusehen. Einen Ansatz hierfür zeigt **Listing 8**.

Die Klasse abonniert das Event *SelectedCultureChanged* von *Globalization-Utilities* und ruft eine abstrakte Methode (*OnCurrentCultureChanged*) auf, die in abgeleiteten ViewModel-Klassen überschrieben werden muss **(Listing 9)**. Je nachdem, wie das Programm mit der Instanzierung der ViewModels verfährt, kann es notwendig sein, die eingerichteten Eventhandler wieder zu entfernen, wenn

das ViewModel nicht mehr gebraucht wird. Ansonsten verweist das automatisch erzeugte Delegate-Objekt weiterhin auf das ViewModel-Objekt, sodass es nicht von der Garbage Collection entsorgt werden kann.

Aus diesem Grund implementiert *View-ModelBase* auch *IDisposable*. Die Programmierer müssen dann selbst daran denken, *Dispose* aufzurufen, wenn sie das ViewModel nicht weiter benötigen. Eventuell ist es auch sinnvoll, hier das vollständige Disposable-Pattern nebst Finalizer zu implementieren. Darauf wurde im Beispiel verzichtet.

Was ist sonst noch zu beachten?

Die vorgestellte Infrastruktur liefert lediglich die Technik, um sprachabhängige Inhalte korrekt darstellen zu können. Aber auch bei der Gestaltung der Oberfläche sind entsprechende Vorkehrungen zu treffen. Ein Text für ein und denselben Begriff kann in unterschiedlichen Sprachen erhebliche Differenzen für den Platzbedarf aufweisen. Ein Begriff, der in einer Sprache sehr kurz und prägnant formuliert werden kann, muss in einer anderen Sprache vielleicht umständlich umschrieben werden.

Das Layout sollte daher möglichst dynamisch aufgebaut sein, sodass Steuerelemente ihre Größe an den Inhalt anpassen können. Ein übersetzter Text nützt nichts, wenn er nach der Hälfte abgeschnitten wird, weil die Schaltfläche zu schmal ist.

Umso wichtiger ist es, das Layout der Anwendung später noch einmal für die verschiedenen Sprachen zu kontrollieren. Werden die Texte richtig dargestellt? Gibt es unschöne Umbrüche, weil einige Inhalte zu lang sind?

Auch sollte für jede Sprache eine sprachkundige Person hinzugezogen werden, die die Übersetzungen im Zusammenhang beurteilen kann. Die Texte im Beispiel, insbesondere die persischen, wurden mittels Google Translate übersetzt und mangels Sprachkenntnissen des Autors nicht weiter gegengelesen. Was für westliche Programmierer hübsch aussieht, muss für persische Anwender aber nicht unbedingt verständlich sein. Solche Erfahrung hat sicher jeder schon einmal mit Bedienungsanleitungen gemacht, die stümperhaft ins Deutsche übersetzt wurde.

Um Daten in Texte formatieren zu können, stellt .NET eine Reihe von Format-String-Definitionen bereit. Aber nicht alle Varianten sind sinnvoll, wenn die Anwendung verschiedene Kulturen unterstützen soll. Ein Format wie *dd.MM.yyyy* für eine Datumsanzeige mag für Deutschland goldrichtig sein, würde aber in den USA eher zu Verwirrung führen. Verwenden Sie daher grundsätzlich nur die allgemeinen Formate wie *g*, *G*, *T* oder *D* oder deren Funktionsäquivalente wie *ToLongDateString* et cetera. Notfalls können kulturspezifische Formate mit eigener ID im Repository abgelegt und dann später passend zur eingestellten Kultur abgerufen werden. Aber feste kulturspezifische Formatierungen haben weder im XAML noch im C#-Code etwas zu suchen.

ID-Benennung

Bevor jedes Teammitglied nun kreativ wird und eigenständig IDs erfindet, soll-

Listing 8

Auch ViewModels benötigen etwas Infrastruktur.

```
public abstract class ViewModelBase : INotifyPropertyChanged, IDisposable
{
  // Die eingestellte Kultur
  public SupportedCulture CurrentCulture{...}

  // Konstruktor. Richtet Kultur und Änderungsbenachrichtigung ein
  public ViewModelBase()
  {
    currentCulture = GlobalizationUtilities.GlobalizationUtilities
      .TheInstance.SelectedCulture;
    GlobalizationUtilities.GlobalizationUtilities.TheInstance
      .SelectedCultureChanged += TheInstance_SelectedCultureChanged;
  }

  // Die Kultur hat sich geändert
  void TheInstance_SelectedCultureChanged(object sender, EventArgs e)
  {
    CurrentCulture = GlobalizationUtilities.GlobalizationUtilities
      .TheInstance.SelectedCulture;
  }

  // Diese Methode wird bei Änderungen aufgerufen und muss von
  // abgeleiteten ViewModels implementiert werden
  protected abstract void OnCurrentCultureChanged();

  // INotifyPropertyChanged-Implementierung
  protected void OnPropertyChanged([CallerMemberName]string propertyName = null)
  {
    var handler = PropertyChanged;
    if (handler != null) handler(this,
      new PropertyChangedEventArgs(propertyName));
  }

  public event PropertyChangedEventHandler PropertyChanged;

  // Verknüpfung der Event-Handler wieder entfernen
  public virtual void Dispose()
  {
    GlobalizationUtilities.GlobalizationUtilities.TheInstance
      .SelectedCultureChanged -= TheInstance_SelectedCultureChanged;
  }
}
```

Listing 9

Aktualisieren der Daten beim Ändern der Kultureinstellung.

```
public class MainViewModel : ViewModelBase
{
  public DateTime AktuellesDatum {...}
  public string DatumAlsText {...}
  public string BegrüßungAusVM {...}

  ...

  private void InfosAnzeigen()
  {
    AktuellesDatum = DateTime.Now;
    EigenschaftenAktualisieren();
  }

  protected override void
    OnCurrentCultureChanged()
  {
    EigenschaftenAktualisieren();
  }

  private void EigenschaftenAktualisieren()
  {
    DatumAlsText =
      aktuellesDatum.ToString("G");
    BegrüßungAusVM =
      CurrentCulture["BegrüßungAusVM"];
  }
}
```

ten Richtlinien und Regeln vereinbart werden. Dabei sind die Anforderungen allerdings wesentlich lockerer als bei den von Windows Forms oder Delphi bekannten Vorgehensweisen.

Früher hat man die IDs meist Control-spezifisch vergeben, weil der Code auf das Steuerelement zugreifen musste, um die Texte eintragen zu können. Etwa:

```
Hauptfenster.GruppeXYZ.OkButton
```

Bei den Datenbindungsmechanismen in WPF ist dies nicht erforderlich. Hier ist es oftmals besser, die IDs nach fachlichen Kriterien und Kategorien zu vergeben. Viele Texte wie *OK*, *Abbrechen*, *Speichern* und so weiter werden immer wieder vorkommen und müssen nicht redundant im Repository gehalten werden.

Grundsätzlich sind Sie bei der Wahl der Syntax für die IDs völlig frei und können nach eigenem Ermessen Trennzeichen für eine Strukturierung festlegen. Auch Parameterangaben lassen sich leicht umsetzen, falls beispielsweise Texte zu einem Begriff in unterschiedlichen Längen und Abkürzungen vorgehalten werden sollen.

Fazit

WPF bietet out of the box viele Mechanismen, die sich nutzen lassen, um Anwendungen für unterschiedliche Sprachen und Kulturen einzurichten. Dieser Artikel hat verschiedene Ansätze aufgezeigt, die sich an ganz unterschiedliche Anforderungen anpassen lassen, um zur Laufzeit Spracheinstellungen umschalten zu können.

Es sind weder externe Werkzeuge notwendig, noch wird die Ablageform der kulturspezifischen Informationen vorgeschrieben. **[bl]**

[1] MSDN, Die Microsoft-Lösung, www.dotnetpro.de/SL1412WPFSprache1
[2] Martin Kleine, Multikulti für XAML, Lokalisieren von WPF-, Silverlight- oder Windows-Phone-7-Anwendungen, dotnetpro 05/2012, Seite 36 ff., www.dotnetpro.de/A1205Lokalisierung
[3] Mathias Raacke, Do you speak Deutsch?, WPF-Anwendungen lokalisieren, dotnetpro 08/2010, Seite 36 ff., www.dotnetpro.de/A1008Lokalisierung