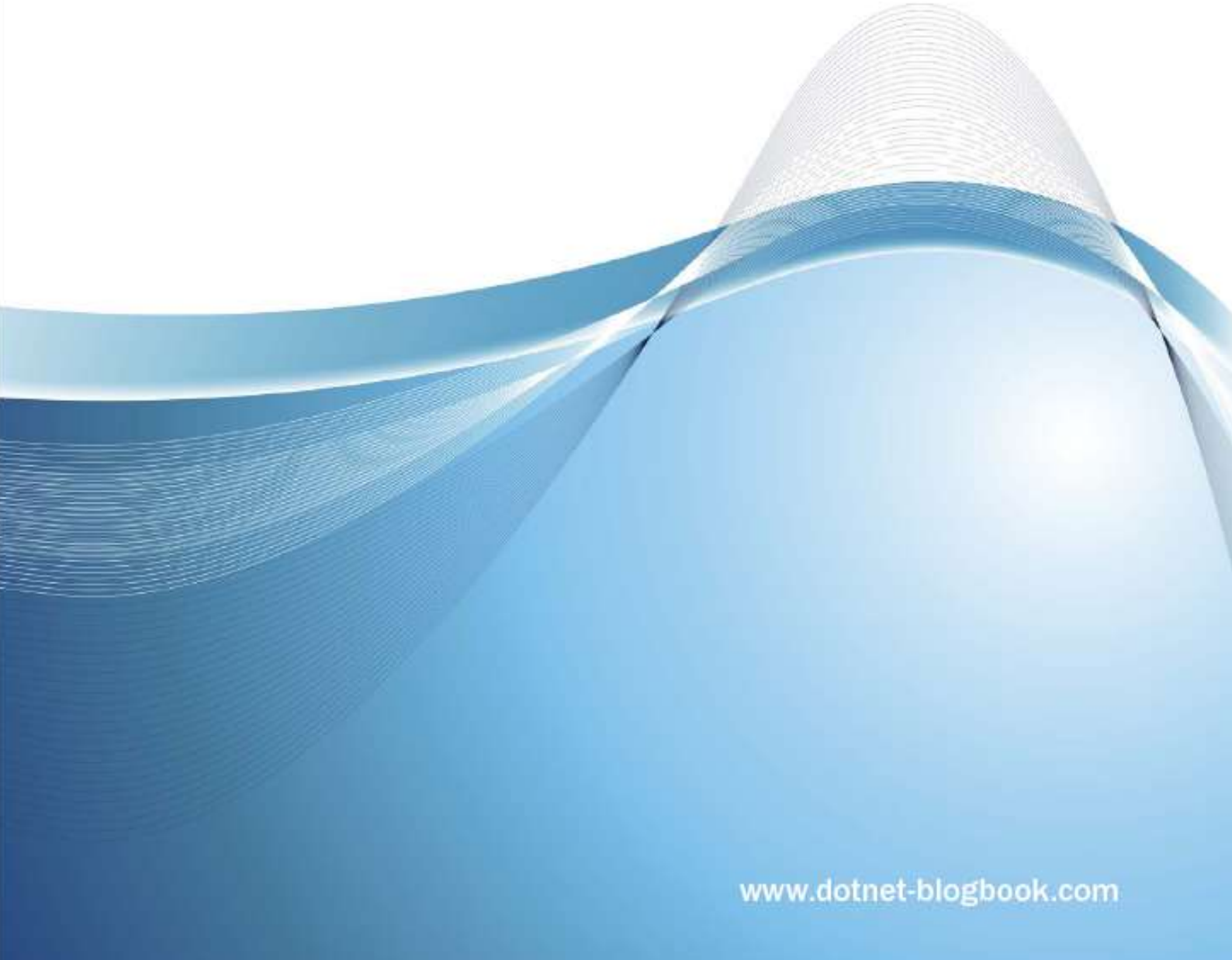
Norbert Eder und Kai Gloth

.NET BlogBook

Das Nachschlagewerk für den .NET Entwickler
mit Tipps, Tricks und Diskussionen

7. Ausgabe



www.dotnet-blogbook.com

Norbert Eder

Kai Gloth

Windows Presentation Foundation

.NET BlogBook Ausgabe 7

http://www.dotnet-blogbook.com

Hinweis

Inhaltsverzeichnis

1 Vorwort

In der Ausgabe 7 des .NET BlogBooks haben große Veränderungen Einzug gehalten. So erstrahlt die neue Ausgabe in einem komplett überarbeiteten Layout und eignet sich nun wesentlich besser für das Lesen am Bildschirm, als auch dem Ausdruck.

Damit wurden zahlreiche Anfragen erhört und entsprechend umgesetzt. Wir denken, dass mit den gemachten Änderungen eine spürbare qualitative Verbesserung eingetreten ist und werden auch weiterhin verstärkt daran arbeiten.

In diesem Sinne wünschen wir viel Spaß mit der aktuellen Ausgabe. Für Anregungen, Wünsche, Beschwerden und dergleichen können Sie uns natürlich jederzeit eine entsprechende Nachricht auf http://www.dotnet-blogbook.com hinterlassen.

2 Einführung

2.1 Tutorial – Teil 1 – Einführung

Der erste Teil einer Tutorials-Reihe gibt eine kleine Einführung in das Thema Windows Presentation Foundation (ehemals Avalon) und beschreibt die wichtigsten Punkte. Zudem wird anhand einer sehr simplen Testanwendung gezeigt, wie eine solche entwickelt werden kann.

Windows Presentation Foundation - Teil 1: Einführung (PDF)

2.2 Tutorial – Teil 2 – XAML und Layouts

Im zweiten Teil der Tutorials-Reihe über die Windows Presentation Foundation werden die Themen XAML und Layouts behandelt. So wird beschrieben, was XAML genau ist und wofür es da ist. Weiters wird gezeigt, welche grundlegenden Layout-Elemente zur Verfügung stehen. Natürlich ist auch dieser Teil wieder voll von Beispiel-Code und Screenshots zur Veranschaulichung.

Windows Presentation Foundation - Teil 2: XAML und Layouts (PDF)

2.3 WPF aus Sicht eines Unternehmens

Auch heute noch - WPF gibt es nun ja schon länger - wird die Windows Presentation Foundation als neue Technologie gehandelt. Beispiele gibt es dazu ja bereits einige und durch Silverlight hat XAML sicherlich einen neuen Hype erfahren. Dennoch scheuen sich sehr viele Unternehmen diese Technologie einzusetzen. Warum ist dem so?

Hier spielen sicherlich mehrere Faktoren eine wichtige Rolle:

Nicht jedes Unternehmen ist dazu gemacht, ein Innovator bzw. ein früher Adopter zu sein (siehe Erklärung). Dies bedeutet, nicht jede Firma setzt auf neueste Technologien. Aus unterschiedlichsten Gründen: Viele vertrauen auf Bewährtes. Es bestehen wenige Risiken (damit haben sich bereits Jahre zuvor andere auseinander gesetzt), Informationen sind breit verfügbar (Foren, Blogs, Bücher) und es gibt durchaus genügend Entwickler die sich mit Bestehendem auskennen und somit im Notfall eingesetzt werden können. Bei einem Innovator sieht es hingegen anders aus: Aktuellste Technologien werden eingesetzt um der Konkurrenz gegenüber einen technologischen Vorteil zu schaffen. Informationen sind rar (SDK Dokumentation, wenn verfügbar).

Know-How-Träger müssen kostenintensiv aufgebaut werden, wodurch die Produktivität anfangs sinkt und natürlich das Risiko besteht, das vorgesehene Projekt nie abzuschließen.

Eng mit dem ersten Punkt ist die Tatsache, dass Zeit geschaffen werden muss, um sich eine Technologie anzueignen. In Zeiten wie diesen - Microsoft veröffentlicht laufend neue Technologien - ist es sehr schwierig mit den aktuellen Entwicklungen Schritt zu halten und am aktuellen Stand zu bleiben. Gefordert sind hauptsächlich Entwickler, denn diese müssen dem Unternehmen bzw. dessen Führung die Vorteile der neuen Technologien schmackhaft machen (und sie selbst auch erlernen). Unternehmen, die die Möglichkeit bieten auf neue Technologien umzusteigen können daraus sicherlich Vorteile generieren. Dennoch ist der Faktor Mangelware an dieser Stelle Zeit. Zeit ist eine große Hürde die es zu überwinden gibt. Neue Technologien stellen ein große Herausforderung an das gesamte Unternehmen, vor allem an das Entwicklerteam, welches unter Zeitdruck steht und dennoch Erfolge bieten muss.

Umstiegskosten: Viele Unternehmen setzen auch heute noch Visual Studio 2003 ein. Dabei ist Visual Studio 2005 fast ein Jahr alt und die 2008er wartet bereits darauf fertig zu werden, um auf den Markt geworfen zu werden. Die Produktzyklen werden immer geringer und immer weniger machen den Schritt tatsächlich mit. Aus Kostengründen, denn Visual Studio ist nicht billig. Und WPF ist nun in der 2003er nicht verfügbar, vorhandene Projekte wollen nicht umgestellt werden, auf zwei Schienen entwickeln ist auch nicht unbedingt das Wahre usw.

Die Überwindung: Schließlich muss man sich überwinden, um tatsächlich ein Projekt auf Basis WPF durchzuziehen. Klar, wunderschön, man kann klar zwischen Designer und Entwickler trennen. Oft kann diese Trennung jedoch nicht vollzogen werden - aus Mangel an Designern. Allrounder sind also gefragt und diese müssen oft erst dazu bewogen werden. Sind Designer vorhanden, müssen diese erst Grundlagen der WPF erlernen, denn ganz ohne geht es dann auch nicht.

Wahrscheinlich ließe sich diese Liste noch fortführen. Tatsache ist, dass es viele Gründe gibt, warum Unternehmen WPF nicht sofort einsetzen, sondern auf einen günstigen Moment warten, auf das richtige Projekt, die richtigen Entwickler, oder möglicherweise gar nie diesen Schritt wagen. Fakt ist, dass dieser Schritt irgendwann vollzogen werden sollte. Die Möglichkeiten sind groß, aber eine vorhandene Anwendung wird nun eben nicht von Heute auf Morgen umgestellt - wobei auch die Sinnhaftigkeit einer Umstellung hinterfragt werden sollte. Doch auch neue Projekte werden lieber mit Windows-Forms entwickelt. Und warum? Vermutlich weil es einfach an allgemein bekannten Anwendungen fehlt, die mittels WPF entwickelt wurden. Entsprechende Patterns sind ebenfalls Mangelware (dazu kommen wir in einem anderen Teil).

Was also sollte in meinem Unternehmen passieren um auf die neue Technologie zu kommen? Diese Frage ist immer schwer zu beantworten. Grundsätzlich sollte durch Visual Studio 2005 (bald 2008) die Grundlage gelegt werden. D. h. auf das .NET

Framework 3.0 sollte umgestiegen werden. Diese muss nicht zwangsläufig für alle Anwendungen gelten. Es ist ok, sich nur eine kleine Anwendung (auf wenn diese nur für interne Verwaltungszwecke verwendet wird) heraus zu nehmen und diese quasi als Übung mit Hilfe der neuen Technologie umzusetzen. Mit den gewonnenen Erfahrungen kann man sich an größere Projekte wagen.

2.4 Windows Presentation Foundation oder doch lieber Windows Forms?

Im ersten Teil der WPF-Serie wurde das Thema **WPF und Unternehmen** behandelt. Nun stellt sich aber nicht nur in Unternehmen, sondern auch in privaten Projekten die Frage: Soll ich mein Projekt mittels der Windows Presentation Foundation umsetzen oder doch lieber zu den bekannten Windows Forms greifen? Dieser Artikel versucht auf diese Fragestellung genauer einzugehen.

In vielen Artikeln wird Windows Forms immer noch WPF vorgezogen. Die Gründe hierfür sind meist dieselben:

- Funktionsweisen sind bekannt
- Einschränkungen sind bekannt
- Windows Forms Designer ist ausgereift
- Entwicklungsaufwand geringer

Grundsätzlich stimmen diese Argumente. Auf der anderen Seite ist jedoch zu beachten, welche Möglichkeiten WPF mit sich bringt. Diese sind Windows Forms auf jeden Fall überlegen. Daher ist es sinnvoll die einzelnen Bereiche genauer zu hinterfragen.

2.4.1 Der Vergleich

Databinding

Die meisten Anwendungen sind sehr datenlastig. D.h. es werden Daten angezeigt, verarbeitet, verwaltet und dergleichen. Um diese Daten anzuzeigen wird oft Databinding verwendet. Hier sind die Möglichkeiten unter Windows Forms jedoch begrenzt. WPF hat diesbezüglich viele Erweiterungen erfahren, wodurch hier eindeutig ein Vorteil für die neue Technologie zu sehen ist.

Trennung Design - Code

Design, also das User Interface, kann unter WPF wesentlich leichter vom Code getrennt werden. Im Gegensatz zu Windows Forms ist es daher möglich, dass das

Oberflächendesign wirklich von einem Designer/Grafiker übernommen wird. Im Idealfall sollte dieser zwar WPF (XAML) Know-how haben, mit Hilfsmitteln á la Blend ist aber auch das nicht zwangsweise notwendig.

Probleme Designer

Unter Windows Forms kommt es immer wieder zu Problemen mit eigenen UserControls. Hier wurde zumindest von mir die Erfahrung gemacht, dass es durchaus vorkommen kann, dass der WinForm-Designer die Oberfläche komplett verwüstet. Controls, welche die Location bzw. Size komplett verlieren und daher im Designer nicht mehr aufscheinen, bis hin zu fehlenden Events etc. Diese Probleme sind in weiterer Folge im Designer kaum Herr zu werden, daher muss man wohl oder übel entweder auf ein Backup zurückgreifen, oder direkt im Code Hand anlegen. Auch hier sehe ich einen klaren Vorteil für WPF: XAML ist wesentlich einfacher und schneller editiert, als haufenweise Code im InitializeComponents etc. Zudem wird der Code nicht angerührt, wenn sich Änderungen im Design ergeben (sei es durch einen Fehler im Designer, oder durch einen menschlichen Eingriff).

Unterschiedliche Medien, Skins

Egal ob es um 3D-Effekte, um animierte Übergänge zwischen Grafiken, Videos etc. geht, ist WPF klar die Wahl Nummer 1. Hier hat Windows Forms kaum etwas entgegen zu setzen. Ebenfalls sind beispielsweise Skins mittels WPF wesentlich einfacher möglich.

2.4.2 Und wann jetzt Windows Forms?

Natürlich hat auch Windows Forms nach wie vor seine Berechtigung. Werden die neuen, von WPF mitgebrachten, Funktionalitäten nicht benötigt, gibt es keinen Grund, auf WPF umzusteigen. Vielmehr empfiehlt es sich dann, bei Windows Forms zu bleiben. Denn auch hier gibt es unzählige Vorteile:

- Große Community mit vielen Hilfestellungen
- Viele Online-Ressourcen, Artikel, Blogbeiträge, Beispiele, ...
- zahlreiche 3rd Party Controls

2.4.3 Fazit

Grundsätzlich bleibt es dem Entwickler überlassen, wann welche Technologie eingesetzt wird. Die aufgezählten Punkte sollten einen kleinen Überblick liefern, in welchen Bereichen WPF die Nase vorne hat. Jedoch bleibt immer zu bedenken, welche Möglichkeiten die zu erstellende Anwendung tatsächlich nutzen soll und wird. Aufgrund dieser Entscheidung lässt sich auch sehr einfach feststellen, welche Technologie zum Einsatz kommt.

Schließlich muss noch eines angemerkt werden: WPF lässt sich auch in Windows Forms Anwendungen verwenden.

2.5 Das Softwaredesign will gut überlegt sein

Im letzten Beitrag wurde die Frage geklärt, wann denn eine Umsetzung mit Hilfe der Windows Presentation Foundation als sinnvoll erscheint und in welchen Fällen dann doch zu den herkömmlichen Windows Forms gegriffen werden soll. Ist eine Entscheidung zugunsten der WPF gefallen, müssen einige Vorarbeiten erledigt werden. Unter anderem betrifft dies das Design der Software selbst. Einige Verhaltensweisen sind dann doch unterschiedlich, was sich unter anderem auch im Softwaredesign auswirkt. Nachfolgend werden diesbezüglich einige Varianten vorgestellt als auch einige allgemeine Punkte behandelt.

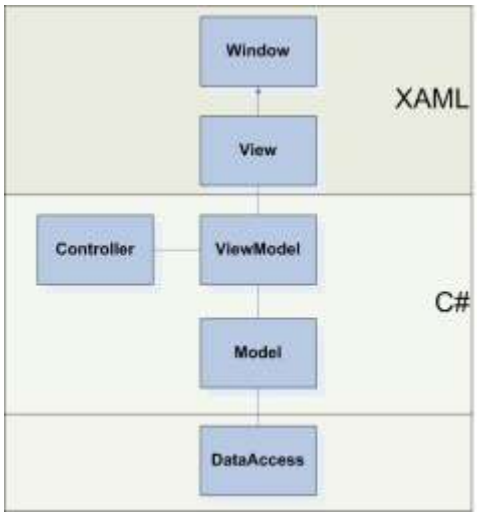
Bietet die WPF bereits ein entsprechendes Konzept?

In die Windows Presentation Foundation wurde einiges an Funktionalität und viele neue Konzepte hineingepackt. Auch für das Softwaredesign wurde etwas getan. So ist es wichtig, Geschäftslogik von der Darstellung zu trennen. Dies wird mit Hilfe der WPF-Commands ermöglicht. Wie der verlinkte Artikel zeigt, stehen mehrere Varianten zur Verfügung. Mal enger mit der UI gekoppelt, mal davon losgelöst.

Für kleinere Projekte mag dies durchaus eine gute Wahl darstellen. Für größere Projekte (beispielsweise von Anwendungen mit einigen Mannmonaten Aufwand) meine ich, dass dieses Konzept nicht vollkommen taugt. Hier bedarf es einer größeren und besseren Lösung.

Das Model-View ViewModel Pattern

Eine wesentlich bessere Lösung wird uns hier mit dem Model-View ViewModel in die Hand gelegt. Es ist dies eine Variante des Model-View-Controller Patterns.



Dies sieht auf den ersten Blick möglicherweise etwas verwirrend aus, tatsächlich gestaltet sich die Verwendung jedoch nicht schwierig. Im Gegensatz zum MVC wird der Controller hier mit einer ModelView ausgetauscht (wobei im Hintergrund weiterhin auch ein Controller seinen Dienst tun kann). Die ModelView besitzt eine enge Kopplung zur View -> DataBinding. Dadurch kann die UI auf sehr einfache Weise aktualisiert werden. Diese enge Bindung muss jedoch nicht an das Model weitergegeben werden. Letzteres hält die einzelnen Entitäten.

Durch die Trennung in diese einzelnen Bestandteile ist es nun möglich, Daten- und Geschäftsklassen vom User Interface zu trennen. Der Vorteil besteht darin, dass die gesamte Anwendungsarchitektur dadurch flexibler, austauschbarer und vor allem testbarer wird.

Zu diesem Thema gibt es eine Serie von Dan Crevier welche ich dem Leser nicht vorenthalten möchte:

- DataModel-View-ViewModel pattern: 1
- DataModel-View-ViewModel pattern: 2
- DM-V-VM part 3: A sample DataModel
- DM-V-VM part 4: Unit testing the DataModel
- DM-V-VM part 5: Commands
- DM-V-VM part 6: Revisiting the data model
- DM-V-VM part 7: Encapsulating commands
- DM-V-VM part 8: View Models

Ebenfalls sehr lesenswert ist der Artikel Tales from the Smart Client von John Gossman.

Weitere Patterns

Im WPF-Umfeld stehen derzeit nicht mehr wirklich andere Patterns für die Kommunikation zwischen dem User Interface und den darunter liegenden Schichten zur Verfügung. Natürlich muss hier erwähnt werden, dass selbst das MVVM Anpassungen erfahren kann und im Laufe der Entwicklung an einer eigenen Lösung auch erfahren wird.

Es empfiehlt sich jedoch auf jeden Fall, alle Anforderungen zu definieren und daraus weitere mögliche Patterns, die innerhalb des MVVMs zum Zuge kommen abzuleiten.

Fazit

Immer, wenn neue Technologien oder Konzepte ins Spiel kommen, müssen neue Mittel und Wege gefunden werden, ein Projekt erfolgreich abzuschließen. Das MVVM ist ein erster Schritt in die richtige Richtung. Mit Fortbestand der WPF werden weitere Lösungen, Abwandlungen des MVVMs und weitere Verbesserungen auf uns Entwickler zukommen.

Grundsätzlich empfehle ich, das erwähnte Pattern in kleinem Rahmen zu testen und einen kleinen Prototyp zu entwickeln, um ein erstes Gefühl zu bekommen. Darauf aufbauend können anschließend die ersten ernsthaften Anwendungen entwickelt werden.

Tipp: Setzen Sie sich mit den Eigenheiten der WPF auseinander, um vor unliebsamen Überraschungen geschützt zu sein. Gerade wenn unbekannte Patterns das erste Mal eingesetzt werden, kann es sehr leicht zu Designfehlern kommen, die im weiteren Zuge größere Umstellungen/Refactorings notwendig machen.

2.6 Fehlerbehandlung und Ressourcen

Bereits im dritten Teil wurde besprochen, dass natürlich die Anforderungen ein sehr wichtiges Thema sind und erst darauf aufbauend ein Design entwickelt werden kann. Nun ist ein Softwaredesign jedoch noch nicht alles. Weitere Punkte müssen unbedingt beachtet werden, um nicht später doch noch vor einem Problem zu stehen.

Einführung

Wie in allen Anwendungen (unabhängig, welche Technologien eingesetzt werden) sind Vorarbeiten notwendig, damit das Ergebnis den Wünschen entspricht. Dazu gehören unterschiedliche Dinge, unter anderem:

- Fehlerbehandlung
- Anpassbarkeit (Customization)
- usw.

Gerade diese Punkte können durch ein Softwaredesign nicht 100%ig (bzw. teilweise gar nicht) abgedeckt werden. Jedoch ist es unablässig, diesen Bereichen vor dem Start der Implementierung die notwendige Beachtung zu schenken. Hier soll auf diese Punkte näher eingegangen werden.

Globale Fehlerbehandlung

Fehler treten in jeder Anwendung und in jeder Library auf und müssen entsprechend behandelt werden. Kommt es bei der Framework-Entwicklung vor, dass teilweise Exceptions nicht behandelt werden (da dies vom Verwender vorgenommen werden sollte), ist es bei einer Anwendung, die mit Benutzern zu tun hat, erforderlich, dass alle Fehler abgefangen werden.

Hierzu ist zu entscheiden, welche Fehler der Benutzer tatsächlich zu Gesicht bekommen sollte und welche nicht. Nicht bis zum Benutzer vordringen sollten Exceptions, die vom Benutzer nicht beeinflusst bzw. behoben werden können und von interner Natur sind. Diese sollten in einen Logging-Mechanismus landen, dessen Einträge über ein Incident Management an den Hersteller übertragen werden können.

Treten Fehler (auf Basis von Fehleingaben etc.) auf, die für den Anwender von Relevanz sind, müssen diese den Benutzer entsprechend darauf hinweisen - und das in einer verständlichen und übersichtlichen Form. In diesen Fällen reicht es meist nicht, nur die Message-Eigenschaft einer Exception anzuzeigen.

Da grundsätzlich nicht alle Fehler abgefangen werden (können) bietet sich zudem ein globales Exception-Handling an. Dieses kann in der WPF sehr einfach realisiert werden:

Zu diesem Zwecke ist das Application.DispatcherUnhandledException-Event vorgesehen. Dieses kann entweder via XAML in der *App.xaml* angegeben werden (der Eventhandler befindet sich in der Codebehind-Datei), oder vollständig in der *App.xaml.cs*. Der Eventhandler würde wie folgt aussehen:

```
private void Application_DispatcherUnhandledException(
    object sender, DispatcherUnhandledExceptionEventArgs e)
{
    LoggingManager.GetLog().Error(e.Exception);
    LoggingManager.GetLog().Debug(
        CultureInfo.CurrentCulture,
        Environment.StackTrace);
}
```

In diesem Beispiel sorgt ein LoggingManager dafür, dass die Meldungen entsprechend aufgezeichnet werden, um zu einem späteren Zeitpunkt ausgewertet zu werden. Eine Erweiterung wäre hier, die möglichen Exceptions auszuwerten und gegebenenfalls gesondert darauf zu reagieren.

Ressourcen-Behandlung

Wie mit Ressourcen umgegangen wird, hängt sehr stark davon ab, wo und wie die resultierende Anwendung eingesetzt wird. Für eine rein interne Lösung, die für die Konfiguration von internen Systemen verwendet wird, haben Ressourcen meist keine sehr große Bedeutung. Bei Kundenprojekten sieht es schon anders aus.

Ressourcen können unterschiedlichste Dinge beinhalten: Unterschiedliche Sprachen, Grafiken, Konfigurationen, Templates, Styles und anderes ist möglich. Zu entscheiden ist in den einzelnen Fällen, was von externer Stelle manipulierbar sein soll und darf.

Wie bereits angesprochen, ist das Thema *Aussehen* und *Usability* bei kleinen internen Anwendungen meist nicht von Bedeutung. Jedoch bedeutet "intern" nicht unbedingt, dass die Anwendung nur von einem kleinen Personenkreis (z.B. nur Entwickler) verwendet wird. Ist dem der Fall gelten die gleichen Regeln wie für externe Anwendungen.

Bei externen Anwendungen gilt zu definieren, ob es sich dabei um Standard-Anwendungen handelt, oder um Grundgerüste, die auf den jeweiligen Kunden hin angepasst werden. Im ersteren Fall wird das Design meist vorgegeben und muss normalerweise nicht geändert werden. Vielmehr möchte man die Anwendung vor Änderungen schützen. Hierzu bietet es sich an, die Ressourcen zu integrieren. Im zweiteren Fall werden nicht nur spezifische Funktionalitäten für den Kunden entwickelt, sondern es müssen auch Anpassungen an die jeweilige Corporate Identity (CI) durchgeführt werden.

Da hierbei eventuell auch Änderungen am Aussehen, an der Platzierung der Elemente etc. direkt vor Ort vorgenommen werden können (ohne einen Entwickler bemühen zu müssen, oder weil der Kunde selbst Anpassungen durchführen möchte) bietet es sich an,

die Ressourcen nicht in die Anwendung zu kompilieren, sondern extern zur Verfügung zu stellen.

Dies kann in der WPF bequem mit Hilfe von ResourceDictionaries durchgeführt werden. Damit ist es möglich, Ressourcen aus Dateien einzubinden. Diese Dateien können in einem eigenen Verzeichnis ausgelagert sein und werden von dort eingebunden. Diese ResourceDictionaries können nun beispielsweise Styles und Templates für Elemente beinhalten (sowohl für einzelne Elemente als auch global wirkend). Damit ist das Aussehen einfach steuerbar.

Zu beachten ist weiters, wie die Ressourcen eingebunden werden. Dies kann grundsätzlich auf zwei Arten passieren:

- StaticResource
- DynamicResource

Der Unterschied dieser beiden Markup Extensions liegt darin, wie die Ressourcen geladen werden. Wird eine Ressource statisch gebunden, wird die Ressource beim ersten Zugriff darauf ausgewertet und auf das jeweilige Element angewandt. Bei der dynamischen Ressource kann sich die Ressource zur Laufzeit ändern. Das bedeutet, dass WPF selbst bestimmt, wann die Ressource neu eingelesen und angewandt werden muss.

Mit diesem Wissen gilt es nun zu entscheiden, wann welche Strategie zum Zug kommen soll. Als Faustregel: Was sich zur Laufzeit nie ändert, sollte als statische Ressource eingebunden werden. Was sich ändern kann, dynamisch. Generell sollte man mit dynamischen Ressourcen aber sparsam umgehen, da dies durchaus eine Auswirkung auf die Performance haben kann.

Fazit

Bevor die Implementierung eine Anwendung beginnt, sind zahlreiche Dinge zu klären. Fehlerbehandlung und Umgang mit den Ressourcen ist nur ein kleiner Teil davon. Doch gerade in diesem Bereich finden sich immer wieder Probleme in diversen Anwendungen bzw. Projekten. Klären Sie diese Punkte und etwaige Probleme werden sich zunehmends minimieren.

3 Layout und Styling

3.1 BringToFront und SendToBack

Unter Windows Forms gibt es die Methoden `BringToFront` und `SendToBack` um Steuerelemente in den Vordergrund zu bringen oder in den Hintergrund zu setzen. Diese Methoden sind in der Windows Presentation Foundation nicht mehr vorhanden. Bei der WPF verfügt jedes Element über eine ZOrder (quasi ein Index, welcher den Verlauf über die Z-Achse beschreibt). Elemente die später hinzugefügt werden, besitzen einen höheren Index und sind somit auf der Z-Achse höher oben angesiedelt und liegen daher über Elementen niedrigerer Ordnung.

Um nun `BringToFront` und `SendToBack` zur Verfügung zu stellen, muss die ZOrder manuell angepasst werden. Dies wird mit nachfolgender Methode gezeigt.

```
private void ChangeZOrder(bool bringToFront)
{
  int iNewIndex = -1;
  Canvas parentElement = (Canvas)_parentElement;
  if (bringToFront)
  {
    foreach (UIElement elem in parentElement.Children)
      if (elem.Visibility != Visibility.Collapsed)
        ++iNewIndex;
  }
  else
  {
    iNewIndex = 0;
  }

  int iOffset = (iNewIndex == 0) ? +1 : -1;
  int iElemCurIndex = Canvas.GetZIndex(this);

  foreach (UIElement child in parentElement.Children)
  {
    if (child == this)
      Canvas.SetZIndex(this, iNewIndex);
    else
    {
      int iZIndex = Canvas.GetZIndex(child);

      if (bringToFront && iElemCurIndex < iZIndex &&
        !bringToFront && iZIndex < iElemCurIndex)
      {
        Canvas.SetZIndex(child, iZIndex + iOffset);
      }
    }
  }
}
```

Hierbei ist folgendes zu beachten: In dieser Methode wird auf das Element

`_parentElement` zugegriffen. Hintergrund ist der, dass die Methode (in meinem Fall) in einer Basisklasse für spezielle Controls definiert wurde und mittels der beiden nachfolgenden Methoden in allen entsprechenden Controls zur Verfügung steht. Genauso könnte es eine Ableitung des Containers (in diesem Fall ein Canvas) geben, dem mitgeteilt wird, welches Child manipuliert werden soll.

```
public void BringToFront()
{
  ChangeZOrder(true);
}

public void SendToBack()
{
  ChangeZOrder(false);
}
```

Gegebenenfalls muss hier an die eigenen Bedürfnisse angepasst werden, es passiert jedoch nichts Aufregendes. Im Falle von `SendToBack` wird als ZIndex 0 gesetzt, d. h. das Element ist anschließend das erste auf dem Container und wird ganz hinten angezeigt, im Falle von `BringToFront` geht es in die andere Richtung.

Noch ein kleiner Hinweis: muss das `Parent`-Objekt gesucht werden, kann dies damit erledigt werden:

```
UIElement parent = VisualTreeHelper.GetParent(myElement);
```

3.2 Element innerhalb eines Canvas per Sourcecode positionieren

Von den Windows-Forms ist man ja noch die Eigenschaft `Location` gewohnt. Diese gibt es bei WPF nicht mehr, stattdessen muss man einen anderen Weg gehen.

Per XAML wird ein Element in einem Canvas recht einfach exakt positioniert:

```
<Canvas x:Name="LayoutRoot" Background="Yellow">
  <ListBox
    Width="100"
    Height="100"
    Canvas.Left="352"
    Canvas.Top="192"
    IsSynchronizedWithCurrentItem="True"/>
</Canvas>
```

Doch wie wird das per Code gemacht? Eigentlich auch nicht wirklich schwieriger, man muss eben nur wissen wie:

```
Canvas canvas=new Canvas();
canvas.Background=Brushes.Yellow;

ListBox lbx=new ListBox();
lbx.Width=100;
lbx.Height=100;
lbx.IsSynchronizedWithCurrentItem=true;
Canvas.SetLeft(lbx,352);
Canvas.SetTop(lbx,192);
canvas.Children.Add(lbx);
```

3.3 ListBox um Grafiken/Images erweitern

Die Standard-ListBox unter WPF ist dann ein doch eher fades Steuerelement. In den meisten Fällen wird man mit den angebotenen Möglichkeiten nicht auskommen, daher muss es angepasst bzw. erweitert werden. Dieser Artikel zeigt, welche Schritte vorgenommen werden müssen, um folgendes Aussehen zu erhalten:

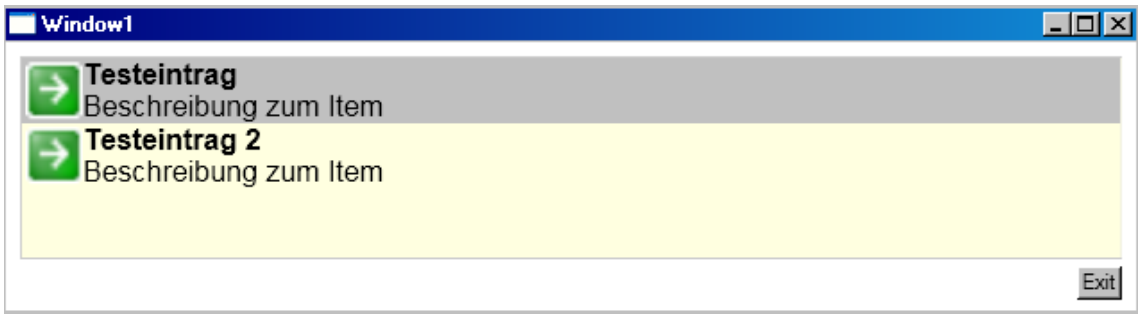


Abbildung 3-1: Grafiken in einer ListBox anzeigen

Normalerweise werden mit Hilfe der ListBox ListBoxItems (es können auch andere Child-Elemente sein) angezeigt. Diese besitzen eine Eigenschaft Content. Darüber kann nun der anzuzeigende Text dargestellt werden. Für dieses Beispiel werden jedoch zwei Strings benötigt und ein Image.

Betreffend der Strings können wir ein simples Datenobjekt erstellen und an die ListBox binden.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ListBoxSampleApp
{
  public class DataItem
  {
    private string _title = null;
    private string _description = null;

    public string Title
    {
      get { return this._title; }
      set { this._title = value; }
    }

    public string Description
    {
      get { return this._description; }
      set { this._description = value; }
    }

    public DataItem() { }

    public DataItem(
      string title,
      string description)
    {
      _title = title;
      _description = description;
    }
  }
}
```

Die Bindung der Daten als auch das Einbinden der Grafik erfolgt an der gleichen Stelle. Hierzu muss eine Ressource erstellt werden, in der ein **DataTemplate** definiert wird. Dieses Template beschreibt in unserem Fall das Aussehen eines Eintrages in der ListBox.

```
<Window.Resources>
  <DataTemplate x:Key="ListItemTemplate">
    <StackPanel>
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="32"/>
          <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
          <RowDefinition/>
          <RowDefinition/>
        </Grid.RowDefinitions>
        <Image
          Source="GoLtrHS.png"
          Grid.Column="0"
          Grid.RowSpan="2"/>
        <TextBlock
          Text="{Binding Path=Title}"
          Grid.Column="1"
          Grid.Row="0"
          FontWeight="Bold"/>
        <TextBlock
          Text="{Binding Path=Description}"
          Grid.Column="1"
          Grid.Row="1" />
      </Grid>
    </StackPanel>
  </DataTemplate>
</Window.Resources>
```

Wie zu sehen ist wird der Eintrag für unsere ListBox aus einem StackPanel und einem Grid erzeugt. Das Grid besitzt insgesamt zwei Zeilen und zwei Spalten. In der ersten Spalte wird nun unsere Grafik angezeigt (die Grafik wurde in das Projekt eingefügt und steht somit zur Verfügung). Durch die Angabe der `RowSpan` erstreckt sich die Grafik über beide Zeilen. In der zweiten Spalte ist für jede Zeile ein `TextBlock` zu finden. Diese enthalten jeweils ein Binding auf Eigenschaften aus dem zuvor definierten Datenobjekt: einmal für den Titel und einmal für die Beschreibung.

Wenn dieser Schritt erledigt ist, muss nur noch der ListBox mitgeteilt werden, dass das erstellte Template für die Anzeige der Items verwendet werden soll. Dies geschieht in der Eigenschaft **ItemTemplate** der ListBox.

```
<ListBox
  RenderTransformOrigin="0.5,0.5"
  Cursor="Arrow"
  x:Name="MainListBox"
  Background="#FFFFFFE0"
  BorderBrush="#FFCCCCCC"
  BorderThickness="1,1,1,1"
  FontFamily="Arial"
  FontSize="16"
  IsSynchronizedWithCurrentItem="True"
  Margin="8,8,8,29"
  d:LayoutOverrides="GridBox"
  ItemTemplate="{StaticResource ListItemTemplate}">
</ListBox>
```

Um dies testen zu können, müssen nur noch Testdaten erstellt und an die `ItemSource`-Eigenschaft der ListBox gebunden werden.

```
public Window1()
{
  this.InitializeComponent();

  Fillup();
}

private void Fillup()
{
  List<DataItem> itemList = new List<DataItem>();
  DataItem di1 = new DataItem("Testeintrag", "Beschreibung zum Item");
  DataItem di2 = new DataItem("Testeintrag 2", "Beschreibung zum Item");
  itemList.Add(di1);
  itemList.Add(di2);
  MainListBox.ItemsSource = itemList;
}
```

Fertig ist die angepasste WPF-ListBox.

3.4 Bewegliche Grid-Spalten - GridSplitter Beispiel

Wer braucht sie nicht? Vergrößerbare Bereiche. Mit Hilfe des `GridSplitters` kann dies sehr einfach gelöst werden. Hierzu sind die entsprechenden Spalten (funktioniert auch auf Zeilen) zu definieren und ein GridSplitter zu verwenden. Dem GridSplitter muss nun noch mitgeteilt werden, zwischen welchen Spalten er sich befindet. Hier gleich ein Beispiel:

```
<Window x:Class="GridSplitterSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="GridSplitterSample"
  Height="300"
  Width="300">
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>
  <Label Background="Yellow" Grid.Column="0"/>
  <GridSplitter Grid.Column="0" Width="2" />
  <Label Background="Green" Grid.Column="1"/>
</Grid>

</Window>
```

Wer dies testen möchte: eine leere WPF-Anwendung erstellen und den obigen Sourcecode in die `Window1.xaml` kopieren, starten und probieren. Das Ergebnis sieht übrigens so aus:

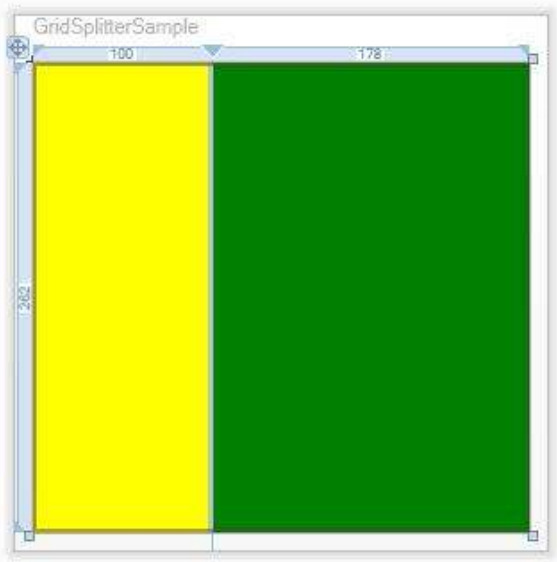


Abbildung 3-2: GridSplitter Demo

Durch den Splitter können nun die einzelnen Spaltengrößen verändert werden:

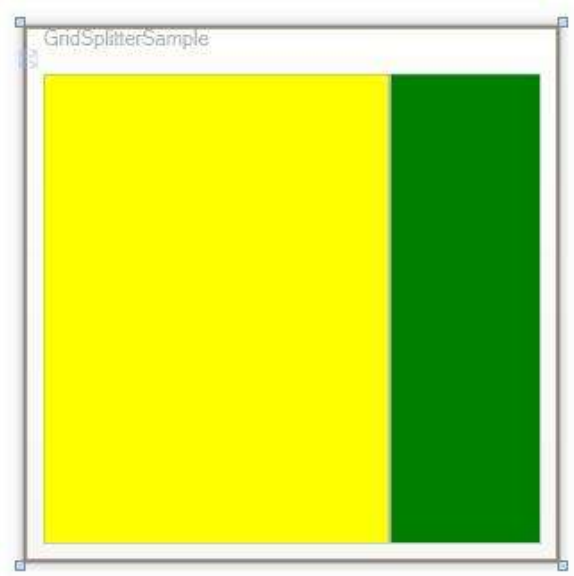


Abbildung 3-3: GridSplitter Demo 2

3.5 CornerRadius - nicht alle Ecken abrunden

Mit Hilfe von `CornerRadius` ist es beispielsweise sehr einfach möglich, die Ecken eines Rechtecks abzurunden. Ein mögliches Beispiel ist hier gegeben:

```
<Window x:Class="CornerRadiusSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1"
  Height="300"
  Width="300">

<Grid>
  <Label
    Background="Blue"
    Name="TestLabel"
    Width="100"
    Height="100">
    <Label.Template>
      <ControlTemplate>
        <Border
          BorderThickness="1"
          Padding="0,10,0,10"
          Background="{TemplateBinding Background}"
          BorderBrush="{TemplateBinding BorderBrush}"
          CornerRadius="15"/>
      </ControlTemplate>
    </Label.Template>
  </Label>
</Grid>
</Window>
```

Und so sieht es aus:

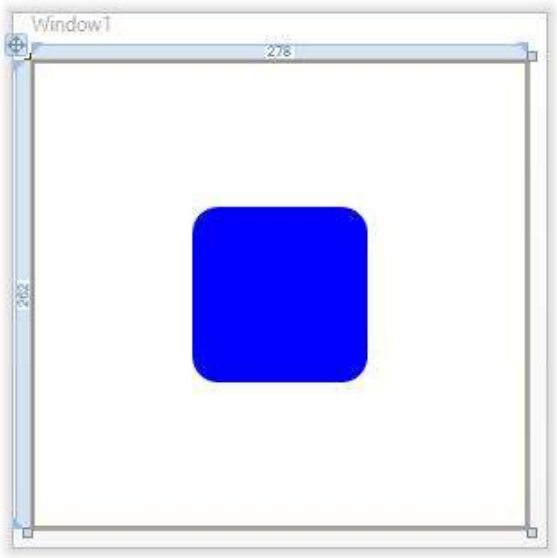


Abbildung 3-4: CornerRadius auf 4 Ecken

Nun kommt es irgendwann zu dem Fall, dass nicht alle Ecken abgerundet werden sollen, sondern nur bestimmte. Hierfür kann man der Eigenschaft `CornerRadius` die Werte für jede einzelne Ecke setzen:

```
<Window x:Class="CornerRadiusSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1"
  Height="300"
  Width="300">
<Grid>
  <Label
    Background="Blue"
    Name="TestLabel"
    Width="100"
    Height="100">
    <Label.Template>
      <ControlTemplate>
        <Border
          BorderThickness="1"
          Padding="0,10,0,10"
          Background="{TemplateBinding Background}"
          BorderBrush="{TemplateBinding BorderBrush}"
          CornerRadius="0,15,0,15"/>
      </ControlTemplate>
    </Label.Template>
  </Label>
</Grid>
</Window>
```

Das sieht dann so aus:

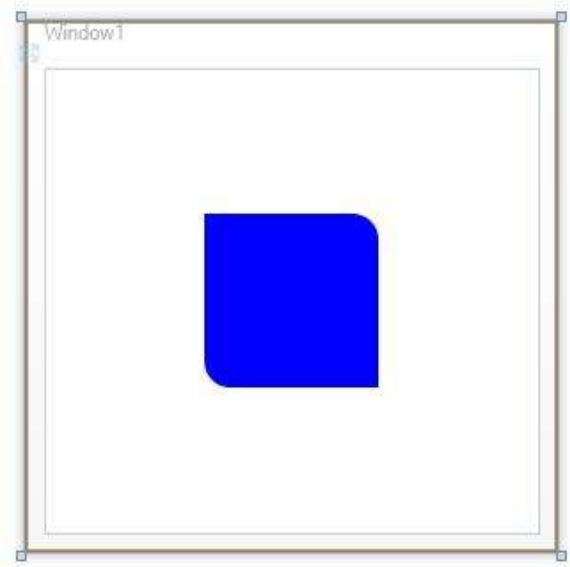


Abbildung 3-5: CornerRadius auf zwei Ecken

Dieser Schritt kann natürlich auch auf andere Controls, wie beispielsweise der ListBox angewandt werden.

3.6 Logischer und visueller Baum

Für einfache Anwendungen unter WPF ist es sicherlich nicht erforderlich, über den logischen und visuellen Baum Bescheid zu wissen. Sobald jedoch eigene Elemente erstellt werden, kommt man unweigerlich damit in Berührung. Daher ist es gut und sinnvoll, den Unterschied zu kennen.

3.6.1 Logischer Baum

Im logischen Baum unter WPF befinden sich alle Elemente, die via XAML bzw. Code erzeugt wurden. Dies ist notwendig, damit nicht nur der Entwickler, sondern WPF selbst über sämtliche Child-Objekte informiert ist und darüber iterieren kann. Ebenfalls wird darüber ein Nachrichtensystem (Notifications) abgebildet. Hilfsmethoden werden über die Klasse LogicalTreeHelper angeboten.

3.6.2 Visueller Baum

Der logische Baum kann nun Elemente enthalten, die nicht unbedingt sichtbar sein müssen. Der visuelle Baum hingegeben (wie der Name schon sagt) beinhaltet alle Elemente, die sichtbar sind und somit von Visual ableiten. Dieser Baum ist vor allem für Entwickler interessant, die ihre eigenen Controls entwickeln und daher genau wissen müssen, wie der visuelle Baum dahinter aussieht. Hilfsmethoden werden über die Klasse VisualTreeHelper angeboten.

Für weitere Informationen empfiehlt sich der Artikel Trees in WPF im MSDN.

3.7 Windows System Fonts in einer ComboBox darstellen

Viele kennen die ComboBox mit den Systemschriften aus den diversen Office-Produkten und auch anderen Anwendungen. Nicht nur, dass der Name der Schrift angezeigt wird, man bekommt auch gleich eine Vorschau.

Dass die Umsetzung nicht aufwändig ist, zeigt nachfolgendes Beispiel. Hierzu das notwendige XAML:

```
<ComboBox x:Name="FontChooser">
  <ComboBox.ItemsPanel>
    <ItemsPanelTemplate>
      <VirtualizingStackPanel />
    </ItemsPanelTemplate>
  </ComboBox.ItemsPanel>
  <ComboBox.ItemTemplate>
    <DataTemplate>
      <TextBlock Text="{Binding}" FontFamily="{Binding}"/>
    </DataTemplate>
  </ComboBox.ItemTemplate>
</ComboBox>
```

Hier wird eine ComboBox angelegt, das ItemsPanel überschrieben (aus Performancegründen) und ein DataTemplate für die Darstellung der einzelnen Items definiert.

Das DataTemplate besteht aus einem TextBlock, welchem via Data Binding der Text (Name der Schriftart) als auch die FontFamily (die zu verwendende Schriftart zur Darstellung) übergeben werden.

Im Source ist dann nur mehr folgende Zeile notwendig und das Ergebnis kann bewundert werden:

```
FontChooser.ItemsSource = Fonts.SystemFontFamilies;
```

Die System-Schriften sind in *Fonts.SystemFontFamilies* aufgelistet und können direkt an die *ItemsSource*-Eigenschaft der ComboBox übergeben werden. Damit werden automatisch die einzelnen Items hinzugefügt und die Werte mittels dem oben definierten Data Binding zugewiesen.

Fertig ist die Fonts-ComboBox. Und so sieht sie aus:

Abbildung 3-6: ComboBox zur Anzeige der Systemschriften

3.8 Unterschied zwischen DataTemplate und ControlTemplate

Sehr oft wird in Foren bzw. per Email gefragt, wo denn genau der Unterschied zwischen einem DataTemplate und einem ControlTemplate liegt. Dies ist relativ einfach erklärt.

3.8.1 DataTemplate

Ein DataTemplate wird verwendet, um das Aussehen bzw. die Zusammensetzung eines Daten-Items zu beschreiben. D.h. darüber wird der Visual Tree eines Daten-Items definiert. Relevant ist dies bei Items für eine ListBox, ListView usw. Dabei kann es sich um eine Ableitung eines ListBoxItems (je nach Steuerelement) oder um eine CLR-Objekt handeln.

3.8.2 ControlTemplate

Ein ControlTemplate hingegen beschreibt den Visual Tree und damit den Aufbau eines Steuerelements.

3.8.3 Fazit

Sollen also Items einer "Auflistung" beschrieben werden, dann ist die Verwendung eines DataTemplates zielführend. Andernfalls ist ein ControlTemplate zu verwenden.

3.9 UniformGrid - das einfache Grid

Im Namespace **System.Windows.Controls.Primitives** verbirgt sich ein Grid namens **UniformGrid**. Dabei handelt es sich um ein wirklich sehr einfaches Grid, welches (wie der Name schon sagt) im Endeffekt nicht mehr tut, als alle enthaltenen Elemente in gleicher Größe darzustellen. Dabei wandert jedes Kindelement in eine eigene Zelle.

Sehen wir uns gleichein Beispiel an:

```
<UniformGrid>
    <Button Content="Button 1"/>
    <Button Content="Button 2"/>
    <Button Content="Button 3"/>
    <Button Content="Button 4"/>
</UniformGrid>
```

Definiert wird ein UniformGrid mit insgesamt vier Buttons als Kindelemente. Und so sieht's aus:

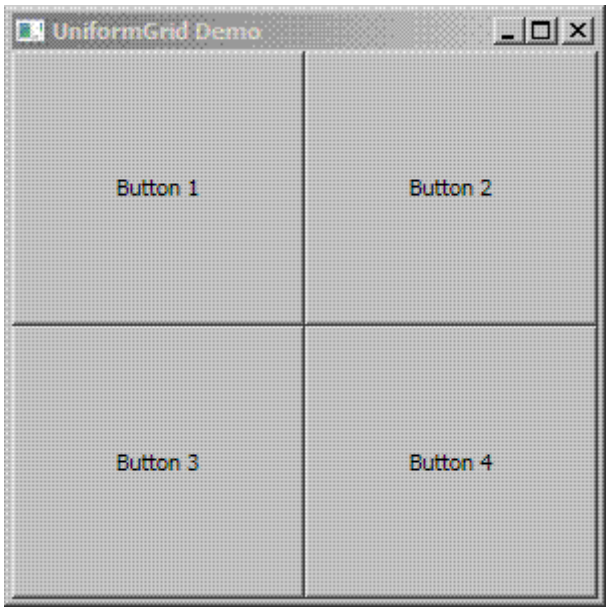


Abbildung 3-7: UniformGrid Beispiel 1

Wird nun ein weiterer Button hinzugefügt, ändert sich das UniformGrid wie folgt:

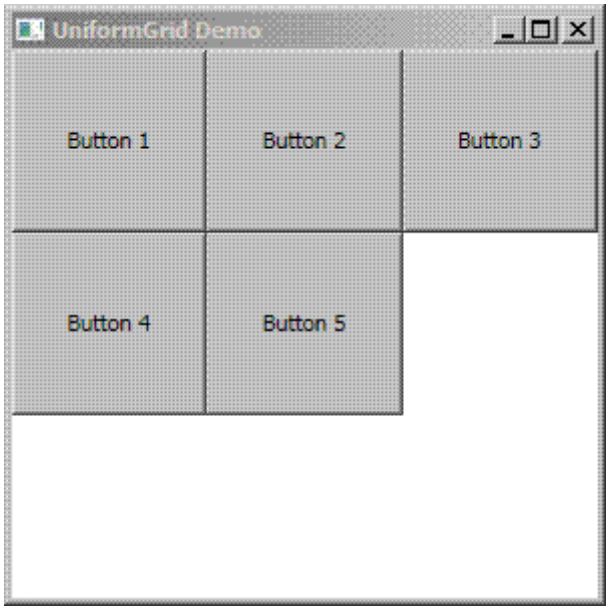


Abbildung 3-8: UniformGrid Beispiel 2

Wie also zu sehen ist, werden die notwendigen Spalten und Reihen automatisch berechnet. Diese können aber mit den Attributen Columns und Rows beeinflusst werden:

```
<UniformGrid Columns="2" Rows="3">
    <Button Content="Button 1"/>
    <Button Content="Button 2"/>
    <Button Content="Button 3"/>
    <Button Content="Button 4"/>
    <Button Content="Button 5"/>
</UniformGrid>
```

Definiert sind zwei Spalten und drei Reihen. Hier das Ergebnis:

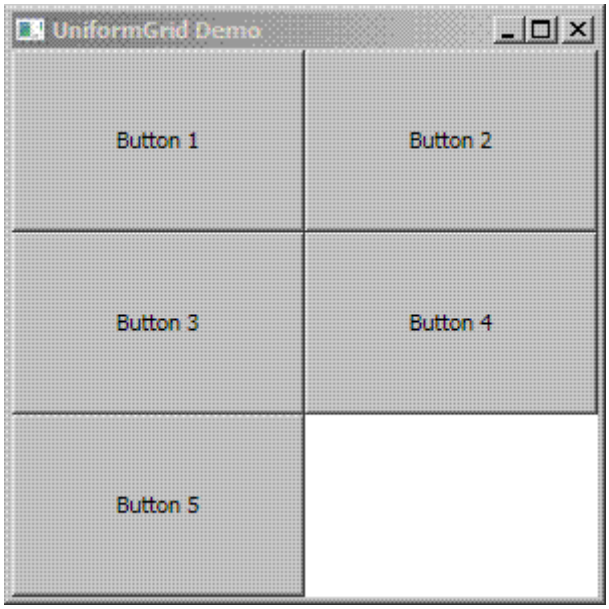


Abbildung 3-9: UniformGrid Beispiel 3

Aber aufgepasst: angezeigt wird nur, was auch tatsächlich auf die vorhandene Fläche passt:

```
<UniformGrid Columns="2" Rows="2">
    <Button Content="Button 1"/>
    <Button Content="Button 2"/>
    <Button Content="Button 3"/>
    <Button Content="Button 4"/>
    <Button Content="Button 5"/>
</UniformGrid>
```

Definiert sind zwei Spalten und zwei Reihen, vorhanden sind jedoch fünf Elemente. Das wird daraus:

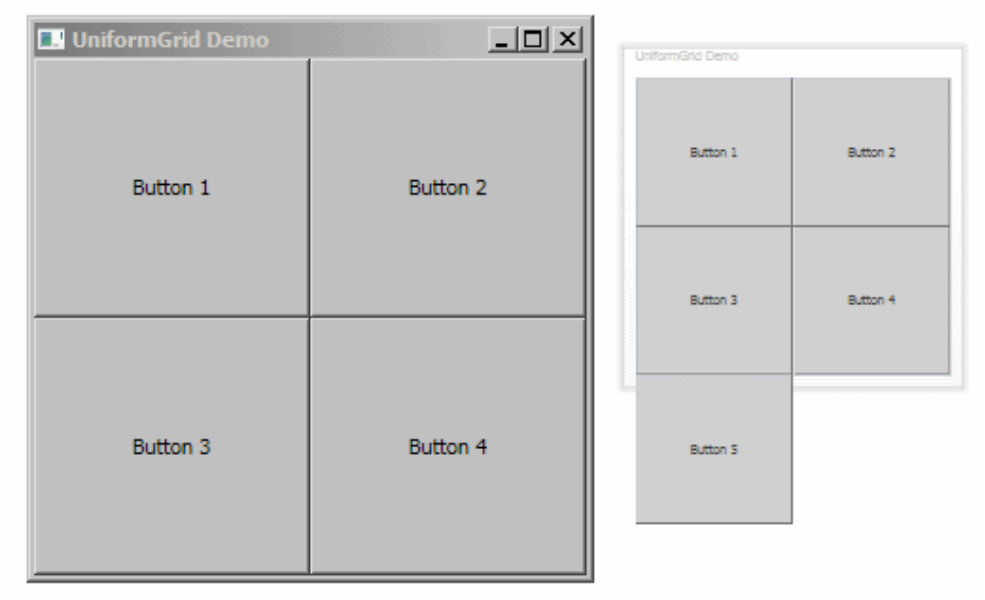


Abbildung 3-10: UniformGrid Beispiel 4

3.10Layout-System und Performance

Müssen viele Elemente erstellt und dargestellt werden, kann die Performance schon mal sehr schnell in den Keller gehen. Dabei müssen die darzustellenden Elemente keineswegs komplex sein. Vielmehr reichen oft kleine Fallen, die vermieden werden wollen.

Oftmals führt das Verwenden eines **falschen Panels** zu einem Performance-Problem. Meist ist es so, dass genau das Panel verwendet wird, welches die gewünschte Aufgabe mit dem geringsten Aufwand (für uns als Entwickler) erfüllt. So wird also beispielsweise ein StackPanel verwendet, sollen Elemente untereinander platziert werden - und zwar in der Reihenfolge, wie sie hinzugefügt wurden. Werden ein paar wenige Elemente hinzugefügt, ist keine Auswirkung zu spüren. Sind es hunderte oder gar tausende, dann wird dieser Vorgang spürbar langsamer.

Hierfür verantwortlich ist die unterschiedliche Funktionalität der Panels. Während ein Canvas relativ *dumm* ist und Elemente basierend auf einer Positionsangabe platziert, ist diese Logik in einem StackPanel oder einem DockPanel wesentlich komplexer.

3.10.1 Layout System

Vom Layout-System werden pro Kindelement zwei Schritte durchgeführt:

- Measure
- Arrange

Jedes dieser Kindelemente besitzt seine eigene Implementierung von Measure und Arrange, damit die an das Element gestellten Anforderungen bezüglich des Layout-Verhaltens erfüllt werden können.

Ablauf Layouting

- Ein Kindelement (UIElement) beginnt den Layoutprozess durch das Erfassen der Basiseigenschaften
- Dann werden die Eigenschaften bezüglich Höhe, Breite, Randstärke ausgewertet
- Ausführung von Panel-spezifischer Logik (siehe Orientation-Eigenschaft des StackPanels)
- Der Inhalt wird ausgerichtet bzw. positioniert, nachdem die Größe etc. aller Kindelemente bemessen wurde.
- Die Liste der Kindelemente wird am Bildschirm ausgegeben

Wann wird es haarig?

Beim Eintreten bestimmter Aktionen, wird der gesamte Layout-Prozess erneut durchlaufen. Folgende Aktionen sind dafür maßgeblich:

- Hinzufügen eines neuen Kindelementes
- Ausführung der Methode **UpdateLayout** eines Kindelementes
- Zuweisung von **LayoutTransform** eines Kindelementes

Bei Änderung einer Dependency Property mit Auswirkung auf Measure und Arrange (gesetzt über die Metadaten)

3.10.2 Fazit

Durch das Wissen wie das Layout-System funktioniert und wann es erneut durchlaufen werden muss, lässt sich feststellen, in welchen Fällen worauf verzichtet werden sollte. So kann es beispielsweise effizienter sein, eine ListBox anstatt eines StackPanels zu verwenden (durch das Definieren von Templates kann das auch schön aussehen), oder vielleicht doch einen Canvas.

Ist man sich nicht sicher, ob die geplante Variante die beste Performance bietet, bietet sich auch an, vor der konkreten Implementierung dann eventuell doch eine kleine Beispielanwendung zu schreiben. Auch eingesetzte Profiler sind hilfreich.

Siehe auch: **WPF: Performance messen und verbessern**

3.11 WPF, Layouts und verschwundene Scrollleiste

3.11.1 Das Problem

Ich habe auf meinem Fenster einige Elemente angeordnet. Darunter auch eine ListView. Diese hatte anfangs eine Scrollleiste, aber jetzt ist diese verschwunden? Warum?

3.11.2 Die Lösung

Diese und ähnliche Fragen werden sehr häufig gestellt. In den meisten Fällen liegt es daran, dass die verwendeten Layout-Elemente nicht gut genug bekannt sind und daher ein Fehlverhalten interpretiert wird, wo eigentlich gar keines ist.

Ein sehr verdächtiges Element ist in diesem Zuge das StackPanel. Es verändert seine Größe abhängig der eingebetteten Elemente.

Hier ein kleiner Beispielfall um dies näher zu beschreiben: Wir gehen davon aus, dass es ein StackPanel gibt. Dieses hat als Kindelement eine ListView. Letztere ist an eine Liste gebunden, welche einige hundert Elemente besitzt. Wird nun die Liste gebunden, vergrößert sich die ListView mit jedem Element das hinzugefügt wird. Bei einigen hundert Elementen kann man davon ausgehen, dass nicht mehr alle am Bildschirm angezeigt werden können und deswegen eine Scrollbar durchaus sehr sinnvoll wäre. Tatsächlich ist es nun so, dass sich das Elternelement, also das StackPanel, an die Größe der ListView anpasst und es somit gar nie notwendig wird, die Scrollbar anzuzeigen.

```
<DockPanel x:Name="ParentDock" Margin="8"
           LastChildFill="True">
    <StackPanel DockPanel.Dock="Top">
        <Button Content="Click me first"/>
        <Button Content="Do not click me"/>
        <ListView x:Name="DemoBox">
            <ListView.View>
                <GridView>
                    <GridViewColumn/>
                    <GridViewColumn/>
                    <GridViewColumn/>
                </GridView>
            </ListView.View>
        </ListView>
    </StackPanel>
</DockPanel>
```

In diesem Beispiel ist das StackPanel zwar in ein DockPanel eingebettet, verändert aber dennoch seine Größe, da der Inhalt wesentlich größer ist. So sieht es aus:

Abbildung 3-11: ListView ohne Scrollbar

In diesem Fall muss auch gar nicht viel verändert werden. Damit die ListView zu ihrer Scrollbar kommt, ist diese lediglich aus dem StackPanel heraus zu nehmen:

```
<DockPanel x:Name="ParentDock" Margin="8"
           LastChildFill="True">
    <StackPanel DockPanel.Dock="Top">
        <Button Content="Click me first"/>
        <Button Content="Do not click me"/>
    </StackPanel>
    <ListView x:Name="DemoBox">
        <ListView.View>
            <GridView>
                <GridViewColumn/>
                <GridViewColumn/>
                <GridViewColumn/>
            </GridView>
        </ListView.View>
    </ListView>
</DockPanel>
```

Damit ist nun das **DockPanel** für die ListView zuständig und beschränkt deren Größe. Dadurch wird die Scrollbar sichtbar. So sieht's aus:

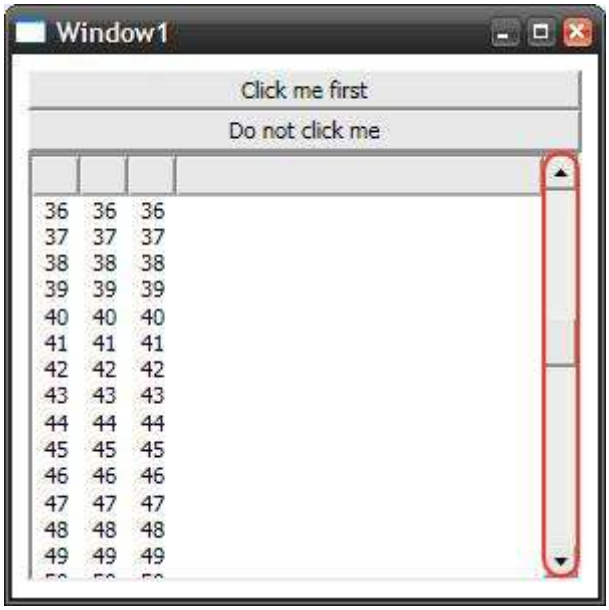


Abbildung 3-12: ListView mit Scrollbar

3.11.3 Fazit

Durch dieses Problem muss sich jeder durchkämpfen, der mit WPF beginnt. Die Verhaltensweisen sind unterschiedlich zu Windows Forms, in einigen Fällen ähnlich zu ASP.NET. Wichtig ist, sich mit den jeweiligen Layout-Elementen zu beschäftigen und zu lernen, wie sie mit Inhalten umgehen. Mit ein wenig Spielerei sollten dann zukünftig derartige Probleme vermieden werden können.

3.12 Data Template zur Laufzeit ändern

Wer WPF-Anwendungen entwickelt, kommt sehr schnell zur Problematik, dass dich Data Templates zur Laufzeit ändern sollen. Dies kann passieren, da bei einem MouseOver zusätzliche Informationen angezeigt werden sollen, oder es soll auf eine andere Aktion eine Reaktion gezeigt werden. Die Realisierung ist recht einfach, aber häufig gefragt.

Nehmen wir also ein kleines Beispiel. Gegeben sei ein Button, der eine Beschriftung anzeigt, die durch ein standardmäßig gesetztes Data Template erweitert wird. Beim MouseOver (also quasi als Hover-Effekt), sollt dieses Data Template ersetzt werden und vor der eigentlichen Beschriftung des Buttons einen anderen Text anzeigen, welcher beim Verlassen wieder zurückgesetzt wird.

Dafür erstellen wir ein Fenster, welches die jeweiligen Data Templates enthält, sowie die Darstellung des Buttons. Hier das Innenleben des Fensters als XAML:

```
<Window.Resources>
    <DataTemplate x:Key="DefaultTemplate">
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="Default: "/>
            <TextBlock Text="{Binding}"/>
        </StackPanel>
    </DataTemplate>
    <DataTemplate x:Key="SpecialTemplate">
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="Special: "/>
            <TextBlock Text="{Binding}"/>
        </StackPanel>
    </DataTemplate>
</Window.Resources>

<Grid>
    <Button
        x:Name="TestButton"
        Content="Test"
        ContentTemplate="{StaticResource DefaultTemplate}"/>
</Grid>
```

Damit wären nun die beiden Data Templates definiert und der anzuzeigende Button. Das Template namens *DefaultTemplate* ist dem Button standardmäßig zugewiesen und setzt der eigentlichen Beschriftung des Buttons ein *Default:* voraus. Das Template, auf welches bei einem Hover gewechselt werden soll, setzt der eigentlichen Besschriftung ein *Special:* voraus.

In der Codebehind-Datei werden nun die notwendigen Events abonniert:

- MouseOver
- MouseLeft

Befindet sich die Maus über dem Button, soll das spezielle Template angewendet

werden. Beim Verlassen der Maus, muss das ursprüngliche Template (also das DefaultTemplate) wieder hergestellt werden. Sehen wir uns dazu den C#-Code an:

```
public partial class Window1 : Window
{
    public Window1()
    {
        InitializeComponent();

        this.TestButton.MouseMove +=
            new MouseEventHandler(TestButton_MouseMove);
        this.TestButton.MouseLeave +=
            new MouseEventHandler(TestButton_MouseLeave);
    }

    private void TestButton_MouseMove(
        object sender,
        MouseEventArgs e)
    {
        TestButton.ContentTemplate =
            this.FindResource("SpecialTemplate") as DataTemplate;
    }

    private void TestButton_MouseLeave(
        object sender,
        MouseEventArgs e)
    {
        TestButton.ContentTemplate =
            this.FindResource("DefaultTemplate") as DataTemplate;
    }
}
```

Im Konstruktor des Fensters werden die beiden benötigten Events abonniert. In den jeweiligen Events wird nichts anderes gemacht, als in den Ressourcen nach dem gewünschten DataTemplate zu suchen und es der Eigenschaft *ContentTemplate* des Buttons zuzuweisen. Idealerweise sollte noch eine Sicherheitsabfrage bezüglich der Existenz der Ressource untergebracht werden.

Damit ist es nun möglich, zur Laufzeit das Data Template eines Elements zu wechseln.

4 Data Binding

4.1 Objekte manuell an Controls binden

DataBinding ist eine wichtige Sache. Schließlich wollen Daten auf der Programmoberfläche ja auch angezeigt werden. Nun stellt sich die Frage, wie dies per Sourcecode-Anweisungen geschehen kann. Hier ein kleines Beispiel dazu. Gegeben sei folgende Oberfläche:

```
<Window x:Class="DataBindingDemo.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="DataBindingDemo"
  Height="150"
  Width="300">

<Grid ShowGridLines="True">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="75"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
  </Grid.RowDefinitions>
  <Label x:Name="FirstnameLabel"
    Content="Firstname"
    Grid.Column="0"
    Grid.Row="0"/>
  <Label x:Name="LastnameLabel"
    Content="Lastname"
    Grid.Column="0"
    Grid.Row="1"/>
  <TextBox x:Name="FirstnameTextBox"
    Grid.Column="1"
    Grid.Row="0"/>
  <TextBox x:Name="LastnameTextBox"
    Grid.Column="1"
    Grid.Row="1"/>
  <Button x:Name="ShowValuesButton"
    Grid.Column="1"
    Grid.Row="2"
    Content="Show Values"/>
  </Grid>
</Window>
```

Also nicht wirklich etwas Besonderes, zwei TextBoxen und ein Button, um zu überprüfen, ob die Änderungen auch tatsächlich übernommen wurden.

Nun benötigen wir für unser Beispiel noch eine Klasse `Person`. Ein Objekt dieses Typs wird anschließend an die beiden Textfelder gebunden:

```
public class Person
{
  public string Firstname { get; set; }
  public string Lastname { get; set; }

  public Person(string firstname, string lastname)
  {
    Firstname = firstname;
    Lastname = lastname;
  }
}
```

Die Klasse ist erstellt, nun kommt das eigentliche Binding. Im Konstruktor von Window1 rufen wir eine Methode `Init` auf, die uns einen Eventhandler für das `Click`-Ereignis des Buttons erstellt, damit die Änderungen angezeigt werden und somit überprüft werden kann, ob das dahinterliegende Objekt auch tatsächlich die korrekten Werte enthält. Weiters wird das Binding definiert.

```
public partial class Window1 : Window
{
  private Person p;

  public Window1()
  {
    InitializeComponent();

    Init();
  }

  private void Init()
  {
    ShowValuesButton.Click +=
      new RoutedEventHandler(ShowValuesButton_Click);

    p = new Person("Norbert", "Eder");
    Binding firstBinding = new Binding();
    firstBinding.Source = p;
    firstBinding.Path = new PropertyPath("Firstname");

    FirstnameTextBox.SetBinding(TextBox.TextProperty, firstBinding);

    Binding lastBinding = new Binding();
    lastBinding.Source = p;
    lastBinding.Path = new PropertyPath("Lastname");

    LastnameTextBox.SetBinding(TextBox.TextProperty, lastBinding);
  }

  void ShowValuesButton_Click(object sender, RoutedEventArgs e)
  {
    MessageBox.Show(
      String.Format("Lastname: {0} -- Firstname: {1}",
      p.Lastname,
      p.Firstname));
  }
}
```

Was wir also für das Binding benötigen ist ein Objekt des Typs `Binding`. Diesem Objekt muss eine `Source` übergeben werden. Dies kann wiederum ein beliebiges Objekt sein - in unserem Fall ein Objekt des Typs Person. Mit Hilfe der Eigenschaft `Path` kann nun festgelegt werden, welche Eigenschaft des an die Source übergebenen Objektes gebunden werden soll.

Schließlich muss am Ziel-Control noch das Binding mit Hilfe der Methode `SetBinding` gesetzt werden. Hierzu ist die gewünschte `DependencyProperty` (beschreibt welche Eigenschaft des Controls befüllt werden soll) anzugeben und das zuvor erstellte `Binding`-Objekt. Das gleiche wird nun auch für das zweite Textfeld durchgeführt.

Zur Laufzeit sieht das dann so aus:

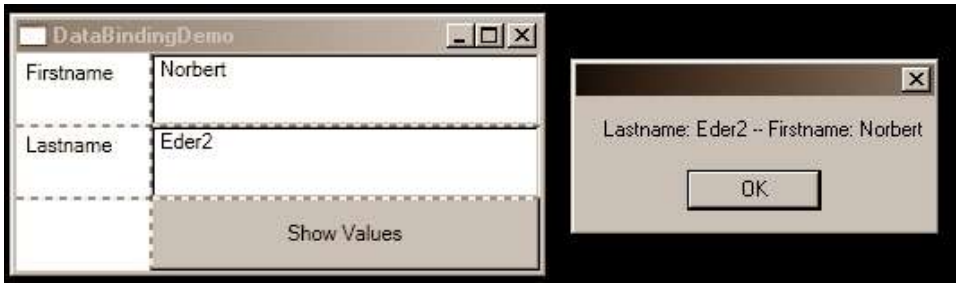


Abbildung 4-1: Einfaches Data Binding

Nach jeder Änderung eines der Textfelder sollte die `MessageBox` diese Änderung auch tatsächlich anzeigen. Wenn dem so ist, dann kann man sich zu einem erfolgreich durchgeführten DataBinding gratulieren.

4.2 Demo zu Data Binding, DataTemplate usw.

Mit Hilfe einer kleinen Demoanwendung sollen folgende Punkte veranschaulicht werden:

- Data Binding
- Data Templates
- CLR Namespaces
- Styles
- usw.

Grundsätzlich tut die Anwendung nicht viel. Sie zeigt lediglich Personendaten innerhalb einer ListBox in Form einer Visitenkarte an. Zusätzlich kann das angehängte Foto (in diesem Fall nur Beispielgrafiken) vergrößert dargestellt werden. Das ganze sieht dann so aus:

Abbildung 4-2: DataBinding: Demo-Anwendung

Designtechnisch begabte Personen können die Darstellung sicherlich nach belieben verbessern, der WPF-Einsteiger hat die Möglichkeit, sich ein paar Dinge genauer anzusehen. Sicherlich für den Einsteiger ein überschaubares Beispiel.

Da ich es niemanden vorenthalten wollte (würde sonst lediglich auf meiner Festplatte verstauben), hier der Download:

Download WPF Demo

4.3 Externes XML mit der WPF an Elemente binden

Data Binding (siehe hier und hier) ist ein sehr wichtiges und häufig genutztes Konzept in der Windows Presentation Foundation.

4.3.1 Kurze Data Binding Einführung

Damit kann eine Datenquelle an ein Element gebunden werden. Je nachdem wie das Binding definiert wird, werden Änderungen in beide Richtungen durchgeschliffen oder nur in eine. Soweit in aller Kürze.

4.3.2 Ein einfaches Beispiel

In vielen Fällen (vor allem kleinere Anwendungen) kommt man mit einer XML-Datei als Quelle aus. In einigen Fällen werden darin nicht nur "unsichtbar" Informationen (Konfiguration etc.) gespeichert, sondern müssen auch einmal angezeigt werden.

Idealerweise erledigt man diesen Punkt per Data Binding, da auf damit einiges an Arbeit (und somit auch Zeit) gespart werden kann.

Aber gehen wir ein Beispiel Schritt für Schritt durch.

Wichtig ist, dass ein entsprechendes XML definiert wird. In diesem Fall wird einem neuen WPF Application Projekt ein neues Item *XmlData.xml* hinzugefügt und ins Unterverzeichnis *Resources* gelegt. Wichtig ist, in den Eigenschaften der Datei, die *Build Action* auf *Content* zu stellen und *Copy to Output Directory* auf *Copy if newer*. Damit wird sichergestellt, dass die XML-Datei bei der Ausführung der Anwendung auch tatsächlich vorhanden ist, also in das Output-Verzeichnis kopiert wird.

In diesem einfachen Beispiel enthält die XML-Datei Personendaten und sieht so aus:

```
<Persons>
  <Person>
    <Firstname>Norbert</Firstname>
   <Lastname>Eder</Lastname>
  </Person>
  <Person>
    <Firstname>Hugo</Firstname>
    <Lastname>Tester</Lastname>
  </Person>
</Persons>
```

Damit wir auf die XML-Datei als Ressource zugreifen können, müssen wir in unserem Fenster (Window) diese als Ressource definiert werden. Möchte man die XML-Datei anwendungsweit zur Verfügung haben, muss sie als Ressource in der *App.xaml* definiert werden.

Hierfür ist ein XmlDataProvider zu verwenden. Diesem muss per *x:Key* ein eindeutiger Schlüssel zugewiesen werden, mit dem auf diese Ressource zugegriffen werden kann. Anschließend ist ein *XPath* zu definieren, der den Pfad auf die Einträge in der XML-Datei angibt, die angezeigt werden sollen. Schlussendlich ist noch die Eigenschaft *Source* zu definieren. Diese gibt den Pfad zur XML-Datei an. Und so sieht es aus:

```
<XmlDataProvider
    x:Key="PersonsData"
    XPath="Persons/Person"
    Source="Resources/XmlData.xml"/>
```

Damit wäre nun die Ressource definiert und dadurch kann damit gearbeitet werden. Als nächsten Schritt werden alle Einträge an eine ListBox gebunden. Dazu wird im Hauptfenster eine ListBox definiert, deren *ItemsSource* an die XML-Datenquelle gebunden wird:

```
<ListBox
    x:Name="PersonsListBox"
    Grid.Column="0"
    Grid.Row="0">
    <ListBox.ItemsSource>
        <Binding Source="{StaticResource PersonsData}"/>
    </ListBox.ItemsSource>
</ListBox>
```

Dazu wird ein Binding für ItemsSource definiert, welches auf auf statische Art und Weise die Resource mit dem definierten Schlüssel *PersonsData* zugreift. Diese wird durch den definierten *XmlDataProvider* repräsentiert.

Wenn nun bei der Auswahl eines Eintrags der ListBox die einzelnen XML-Elemente in eigenen Feldern angezeigt werden sollen, dann kann dies ebenfalls mit einem Binding gelöst werden.

In diesem Beispiel werden die einzelnen Elemente, welche die einzelnen Werte anzeigen sollen, in einem StackPanel angezeigt. Da alle Elemente im StackPanel die gleiche Datenquelle haben, kann die Eigenschaft *DataContext* des StackPanels genutzt werden.

```
<StackPanel
    Orientation="Vertical"
    DataContext="{Binding ElementName=PersonsListBox,
                    Path=SelectedItem}"
    Grid.Column="1"
    Grid.Row="0">
```

Was passiert hier? Das StackPanel erhält als Datenkontext den ausgewählten Eintrag der ListBox. Damit nun die Kinder-Elemente des StackPanels die einzelnen Werte anzeigen können, muss bei ihnen ebenfalls ein Binding definiert werden:

```
<Label Content="{Binding XPath=Firstname}"/>
```

Im Binding wird ein *XPath* angegeben und dieser verweist auf das Child-Element *Firstname*. Dies bedeutet, dass im definierten Label der Vorname des in der ListBox ausgewählten Elements angezeigt werden soll.

4.3.3 Fazit

Diese Beispiel zeigte, wie einfach ein XML Data Binding realisiert werden kann. Notwendig sind dazu natürlich Daten in Form einer XML-Datei und ein wenig Wissen rund um das Thema Data Binding. Mit Hilfe dieses Beispiels sollten erste Binding-Versuche erfolgreich abgeschlossen werden können.

Download WPF Xml Data Binding Demo

4.4 Formatierung der Daten bei einem Data Binding in WPF

Das Data Binding (siehe hier, hier und hier) wurde ja unter der Windows Presentation Foundation stark verbessert. Daten können nun wirklich vielseitig und einfach an Elemente gebunden werden. Was bisher jedoch fehlte war die Möglichkeit, die gebundenen Daten auch einfach zu formatieren. Dies konnte bisher über Converter erledigt werden.

Mit der Einführung von `StringFormat` mit .NET Framework 3.5 SP1 kann dies nun einfacher durchgeführt werden. Nehmen wir an, es soll ein Personen-Objekt mit den Eigenschaften `FirstName`, `LastName` und `Birthday` an Eingabefelder gebunden werden und der Fokus an der Formatierung des Geburtsdatums liegen, dann kann dies wie folgt aussehen:

```
<TextBlock>First Name</TextBlock>
<TextBox Text="{Binding FirstName}"/>
<TextBlock>Last Name</TextBlock>
<TextBox Text="{Binding LastName}"/>
<TextBlock>Birth Day - Long Format</TextBlock>
<TextBox Text="{Binding Path=BirthDay, StringFormat=D}"/>
<TextBlock>Birth Day - Short Format</TextBlock>
<TextBox Text="{Binding Path=BirthDay, StringFormat=d}"/>
```

Ein wenig angepasst, könnte das Ergebnis folgendermaßen aussehen:

Abbildung 4-3: StringFormat

4.5 DataSet und Data Binding

Gerade zum Thema Data Binding habe ich mittlerweile einige Beispiele erstellt (siehe hier, hier und hier). Was noch fehlt (und oft gefragt wird) ist, wie ein DataSet gebunden werden kann. Dem möchte sich dieser Artikel widmen.

Auch in diesem Fall ändert sich relativ wenig. Anstatt eines konkreten Objektes steht nun ein DataSet zur Verfügung. Dieses kann Tabellen enthalten (0 - n) und jede Tabelle kann Spalten enthalten. Das Grundprinzip besteht nun einfach darin:

4.5.1 Vorgehensweise anhand einer TreeView

Das DataSet wird als DataContext bei der TreeView gesetzt. Das Root-Element wird beispielsweise per Hand (Markup) angelegt. Diesem wird nun eine Tabelle des DataSets als `ItemsSource` gesetzt. In weiterer Folge wird ein `HierarchicalDataTemplate` erstellt, welches nun dafür zuständig ist, die einzelnen Felder (oder auch nur ein paar wenige davon) darzustellen.

In weiterer Folge kann nun eine Eingabemaske erstellt werden, welche an das `SelectedItem` der TreeView gebunden ist und den ausgewählten Datensatz zur Anzeige bringt und somit bearbeitbar macht.

4.5.2 Konkretes Beispiel

Ob nun das DataSet aus einer Datenbank geladen oder manuell erstellt wird, macht keinen Unterschied. In diesem Beispiel wird ein DataSet per Code generiert und zur Verfügung gestellt:

```
public static class DataSetMock
{
    public static DataSet CreateDataSet()
    {
        DataSet ds = new DataSet();

        ds.Tables.Add("Person");

        ds.Tables[0].Columns.Add("FirstName");
        ds.Tables[0].Columns.Add("LastName");

        ds.Tables[0].Rows.Add("Norbert", "Eder");
        ds.Tables[0].Rows.Add("Hugo", "Tester");

        return ds;
    }
}
```

Dies geschieht in diesem Fall durch eine statische Klasse. Die Methode `CreateDataSet` kann nun über einen ObjectDataProvider via XAML zur Verfügung gestellt werden. Damit beschränken wir uns beim Schreiben von Sourcecode auf die Implementierung eben dieser statischen Klasse.

Wie bereits angesprochen, wird das DataSet in diesem Beispiel über ObjectDataProvider zur Verfügung gestellt:

```
<ObjectDataProvider
    x:Key="PeopleProvider"
    MethodName="CreateDataSet"
    ObjectType="{x:Type local:DataSetMock}"/>
```

Der Provider selbst muss in den entsprechenden Ressourcen (in diesem Fall in den Ressourcen des Fensters) definiert werden.

In weiterer Folge wird nun ein TreeView-Element auf dem Fenster positioniert. Als DataContext wird der zuvor erstellte Provider gesetzt, da dieser die generierten Daten (DataSet) zurück liefert:

```
<TreeView
    DataContext="{StaticResource PeopleProvider}"
    x:Name="PeopleTreeView"
    DockPanel.Dock="Left"
    Width="200">
    <TreeViewItem
        x:Name="PeopleRoot"
        Header="People"
        ItemsSource="{Binding Person}"
        ItemTemplate="{StaticResource PersonTemplate}"/>
</TreeView>
```

Im Source ist zusätzlich zu sehen, dass eine Root-Node über das Markup erstellt wird. Diesem Root-Node wird als ItemsSource nun per Binding die Tabelle Person aus dem DataSet zugewiesen. Dadurch wird der Gültigkeitsbereich des Bindings für diese Node festgelegt. Somit kann nun ein ItemTemplate definiert werden, welches für die Darstellung der einzelnen Datensätze zuständig ist:

```
<HierarchicalDataTemplate x:Key="PersonTemplate">
    <StackPanel Orientation="Horizontal">
        <TextBlock Text="{Binding LastName}"/>
        <TextBlock Text=", "/>
        <TextBlock Text="{Binding FirstName}"/>
    </StackPanel>
</HierarchicalDataTemplate>
```

Dieses Template wird ebenfalls in den Ressourcen des Fensters abgelegt. Zusätzlich muss es der Root-Node als `ItemTemplate` zugewiesen werden.

Das wären bezüglich `TreeView` alle notwendigen Schritte. Möchte man nun ein Eingabeformular erstellen, welches die Daten des aktuell selektierten Datensatzes anzeigen soll, dann kann nun folgendes definiert werden:

```
<StackPanel Orientation="Vertical">
    <TextBlock Text="Firstname"/>
    <TextBox
        Text="{Binding
            ElementName=PeopleTreeView,
            Path=SelectedItem.FirstName}"/>
    <TextBlock Text="Lastname"/>
    <TextBox
        Text="{Binding
            ElementName=PeopleTreeView,
            Path=SelectedItem.LastName}"/>
</StackPanel>
```

Was passiert hier? Es wird ein `StackPanel` definiert, welches alle Kind-Elemente untereinander anordnet. Die Elemente vom Typ TextBlock stellen die Überschriften dar. Für die eigentlichen Daten wird jeweils eine TextBox verwendet. Diese enthalten ein Binding auf die jeweilige Spalte des aktuell selektierten Datensatzes. Änderungen schlagen sich natürlich sofort auf die `TreeView` durch.

4.5.3 Fazit

Das Data Binding funktioniert grundsätzlich immer gleich. Wichtig ist, dass bewusst ist, wann welcher Kontext gesetzt wird und was dieser Kontext tatsächlich zur Verfügung stellt. Darauf kann gebunden werden. Dies verhält sich bei einem DataSet gleich wie bei einer Objekthierarchie.

Das gezeigte Beispiel steht natürlich als **Download** zur Verfügung.

5 Commands

5.1 Commands – Eine Einführung

Einige mögen bereits wissen, dass unter der Windows Presentation Foundation die Möglichkeit besteht, Commands zu verwenden. Für andere mag es neu sein. Wem ein Command gar nichts sagt, der möge sich unter Patterns: Command Pattern zuvor einen kurzen Überblick verschaffen.

Das Command-System wurde in die Windows Presentation Foundation integriert, um durchzuführende Aufgaben/Tasks besser vom User Interface (Oberfläche) zu trennen.

5.1.1 Warum UI von Aufgaben trennen

Ein oft geschilderter Vorteil der WPF ist, dass das grafische Design nicht zwingend vom Entwickler gemacht werden muss. So kann hier in der Tat ein Designer Hand anlegen (jedoch muss er etwas von XAML verstehen). Das ist jedoch nicht der Punkt, auf den ich hinaus möchte.

Ein Grundsatz der Softwareentwicklung ist es, stabile und robuste Software zu bauen. Dazu muss natürlich getestet werden. Manuelle Tests, die alles abdecken sind aber einer bestimmten Projektgröße kaum durchzuführen. Automatisierte Tests müssen her. Mit Hilfe von Unit Tests können nun Teile der Anwendung getestet werden. Dies funktioniert jedoch nur dann, wenn diese Teile vom grafischen Aufbau komplett getrennt sind. Sinnvoller Weise werden nun sämtliche Aufgaben von der UI getrennt und werden/bleiben so testbar.

5.1.2 Wie kann die WPF hier helfen?

Mit Hilfe der WPF Commands muss hier kein großer Aufwand betrieben werden, um obiges Ziel auch tatsächlich zu erreichen. Nun gibt es jedoch zwei abzudeckende Möglichkeiten:

- Commands müssen das UI aktualisieren
- Commands können komplett frei von allen UI-Einflüssen ausgeführt werden

Muss der Command mit der Oberfläche interagieren, sind `RoutedUICommands` genau der richtige Ansatz. Im zweiten Fall ist das Interface `ICommand` zu implementieren.

Bezüglich der `RoutedUICommands` ist es nun so, dass durch die WPF bereits einige fertige Implementierungen vorhanden sind:

- ApplicationCommands
- ComponentCommands
- NavigationCommands

- MediaCommands
- EditingCommands

Der Vorteil liegt darin, dass diese Commands sehr einfach an UIElemente gebunden und ausgeführt werden können. Hier ein kleines (unvollständiges) Beispiel zur Veranschaulichung:

```
<UserControl.CommandBindings>
  <CommandBinding
    Command="MediaCommands.Play"
    CanExecute="OnQueryExecute"
    Executed="Execute"/>
</UserControl.CommandBindings>

<StackPanel Grid.Row="2" Orientation="Horizontal">
  <Button
    Command="MediaCommands.Play"
    Content="{Binding RelativeSource={RelativeSource Self},
    Path=Command.Text}" />
</StackPanel>
```

Da diese Commands jedoch ein spezifisches Verhalten anbieten, welches nicht immer benötigt wird, ist es mitunter notwendig, eigene Commands zu implementieren, um die gewünschte Funktionalität zu erhalten.

5.1.3 Eigene Commands via ICommand implementieren

Müssen eigene Funktionalitäten als `Command` abgebildet werden, kann dies mit Hilfe einer Implementierung des `ICommand`-Interfaces passieren. Dieses Interface schreibt zwei Methoden vor und ein Event vor:

- CanExecute
- Execute
- CanExecuteChanged

Ein simpler Beispiel-Command, welcher ein bestimmtes Objekt entgegen nimmt und dieses manipuliert, könnte wie folgt aussehen:

```
public class MySaveCommand : ICommand
{
  #region ICommand Members

  public bool CanExecute(object parameter)
  {
    return true;
  }

  public event EventHandler CanExecuteChanged;

  public void Execute(object parameter)
  {
    if (parameter is EasySum)
    {
      EasySum es = (EasySum)parameter;
      es.Result = es.Summand1 + es.Summand2;
    }
  }

  #endregion
}
```

Eine weitere - und vor allem übliche - Möglichkeit besteht darin, einen `RoutedCommand` zu erstellen.

5.1.4 RoutedCommand erstellen

Mit Hilfe von RoutedCommands ist es sehr einfach, einen Command für das User Interface zu erstellen. Dazu benötigt es nicht sehr viel. Das nachfolgende Beispiel zeigt, wie anhand eines RoutedCommands die Hintergrundfarbe eines Panels verändert werden kann.

Hierzu muss unser neuer Command definiert werden:

```
public static RoutedCommand ChangeBackgroundCommand = new RoutedCommand();
```

Des Weiteren müssen zwei Eventhandler implementiert werden. Zum einen für das `CanExecute`-Event und zum anderen für das `Executed`-Event.

```
private void ChangeBackgroundCommandCanExecute(
  object sender, CanExecuteRoutedEventArgs e)
{
  if (sender is Panel)
    e.CanExecute = true;
  else
    e.CanExecute = false;
}

private void ChangeBackgroundCommandExecuted(
  object sender, ExecutedRoutedEventArgs e)
{
  Panel target = e.Source as Panel;
  if (target != null)
  {
    if (target.Background == Brushes.AliceBlue)
    {
      target.Background = Brushes.LemonChiffon;
    }
    else
    {
      target.Background = Brushes.AliceBlue;
    }
  }
}
```

Im Eventhandler für den `CanExecute`-Event überprüfen wir lediglich, ob das übergebene UIElement auch tatsächlich ein Panel ist. Ist dem so, lassen wir eine Manipulation der Hintergrundfarbe zu. Andernfalls wird eine Ausführung des Commands verwehrt.

Im Eventhandler für das `Executed`-Event wird dann die Hintergrundfarbe geändert bzw. wieder auf die Ursprungsfarbe zurück gestellt.

Damit der Command nun auch verwendet werden kann, muss ein `CommandBinding` erstellt werden:

```
<StackPanel Name="MyStackPanel" Background="AliceBlue" Focusable="True">
  <StackPanel.CommandBindings>
    <CommandBinding
      Command="{x:Static custom:Window1.ChangeBackgroundCommand}"
      Executed="ChangeBackgroundCommandExecuted"
      CanExecute="ChangeBackgroundCommandCanExecute"/>
  </StackPanel.CommandBindings>

  <Label>A StackPanel</Label>
  <Button Command="{x:Static custom:Window1.ChangeBackgroundCommand}"
    CommandParameter="ButtonOne"
    CommandTarget="{Binding ElementName=MyStackPanel}"
    Content="Change Background" />
</StackPanel>
```

Wie zu sehen ist, wird hier der aufzurufende Command definiert, als auch die beiden

Eventhandler, die bereits implementiert wurden. `custom` definiert hier lediglich einen eigenen Namespace.

Das `CommandBinding` ist an einem `StackPanel` definiert, welches ein `Label` und einen `Button` enthält, welcher dann schlussendlich den Command auslöst.

5.1.5 Fazit

Dieser Artikel hat das der WPF zugrunde liegende Konzept der Commands vorgestellt und eine kleine Übersicht inklusive einiger Beispiele gegeben. Ebenso wurden die Unterschiede erläutert. Damit sollte nun jeder angehende WPF-Entwickler das notwendigste Wissen mitbringen, um eigene Commands zu entwickeln und diese in seinen Anwendungen einzusetzen. Natürlich wird es durchaus noch notwendig sein, das eine oder andere nachzulesen. Der Start sollte damit allerdings gemacht sein.

6 Sonstiges

6.1 Rotation und Skalierung einfach gemacht

Beschäftigt man sich näher mit der Materie Windows Presentation Foundation, sieht man schon an sehr einfachen Beispielen, dass die Möglichkeiten schon sehr mächtig sind. Beispielhaft zeige ich an dieser Stelle, wie einfach ein Button skaliert und gedreht werden kann.

Dazu wird im Beispiel ein simpler Button erstellt und platziert. Über Schieberegler ist es möglich, die Werte für die Rotation bzw. dem Zoomfaktor zu setzen. Dies sieht dann folgendermaßen aus:

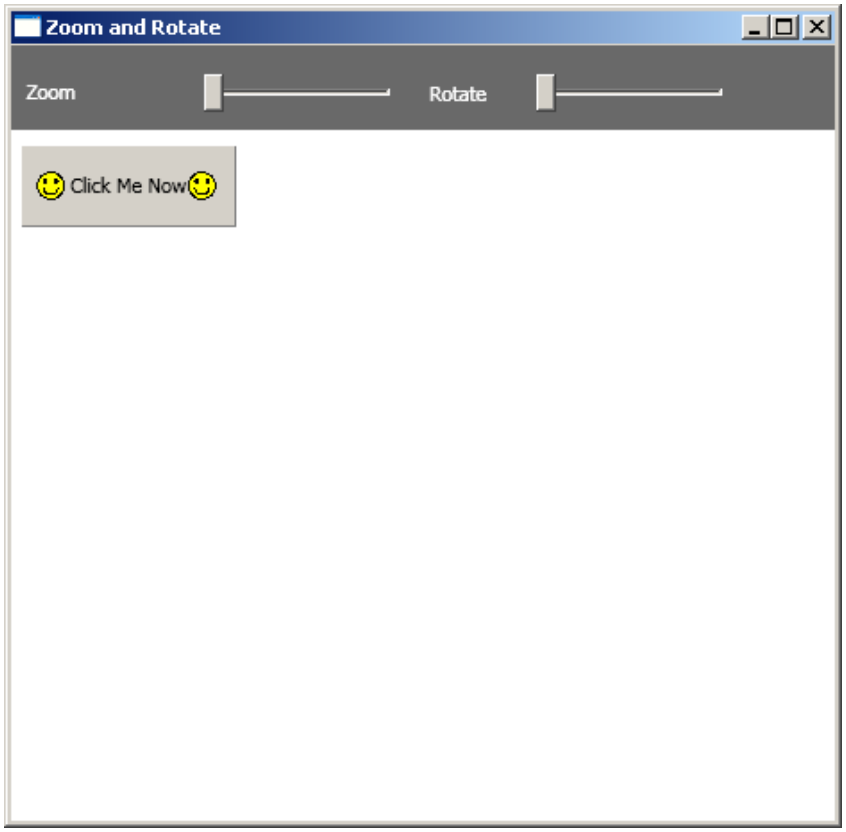


Abbildung 6-1 Zoom und Rotation

Um nun die Funktionalität zu implementieren ist nichts weiter zu machen, als die beiden notwendigen Eventhandler zu setzen:

Zoomfaktor

```
public void sliderZoom_ValueChanged(object sender, RoutedEventArgs e)
{
  ScaleTransform st =
    new ScaleTransform(sliderZoom.Value, sliderZoom.Value);
  this.btnTest.LayoutTransform = st;
}
```

Rotationsfaktor

```
public void sliderRotate_ValueChanged(object sender, RoutedEventArgs e)
{
  RotateTransform rt = new RotateTransform(sliderRotate.Value);
  this.btnTest.RenderTransform = rt;
}
```

Das Ergebnis sieht dann wie folgt aus:

Abbildung 6-2 Gezoomt und rotiert

Dies kann jetzt nicht nur auf einen Button angewandt werden, sondern auf die gesamte Oberfläche, oder auch nur einzelne Bereiche. Das dahinter liegende XAML-Markup etc. ist im gesamten Projekt enthalten, welches zum Download bereit steht.

Download WPF Rotation Test Beispiel

6.2 WPF in Windows Forms verwenden

WPF Steuerelemente können durchaus auch in (bestehenden) Windows Forms – Anwendungen verwendet werden. Damit kann sowohl neue Funktionalität genutzt werden, als mit deren Hilfe Inhalte verständlicher präsentiert werden.

6.2.1 WPF-Control erstellen

Im ersten Schritt muss ein entsprechendes WPF-Control erstellt oder ein vorhandenes benutzt werden. In meinem Beispiel habe ich ein einfaches UserControl erstellt.

```
<UserControl x:Class="AddinTestApp.LoginControl"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="118" Width="300">
<Grid Height="120">
  <Label Height="23" HorizontalAlignment="Left"
         Margin="9,19,0,0" Name="label1"
         VerticalAlignment="Top" Width="120">
         Username
  </Label>
  <Label Height="23" HorizontalAlignment="Left"
         Margin="9,40,0,0" Name="label2"
         VerticalAlignment="Top" Width="120">
         Password
  </Label>
  <TextBox Height="21" Margin="91,21,20,0"
           Name="UsernameTextBox"
           VerticalAlignment="Top" />
  <TextBox Height="21" Margin="91,44,20,0"
           Name="PasswordTextBox"
           VerticalAlignment="Top" />
  <Button Height="23" HorizontalAlignment="Right"
          Margin="0,77,20,0" Name="LoginButton"
          VerticalAlignment="Top" Width="75">
          Login
  </Button>
</Grid>
</UserControl>
```

6.2.2 Windows Formular erstellen

Im zweiten Schritt wird ein normales Windows-Forms-Formular erstellt. Im selben Projekt müssen nun einige Referenzen hinzugefügt werden:

- PresentationCore
- PresentationFramework
- UIAutomationProvider
- UIAutomationTypes
- WindowsBase

Zusätzlich ist noch die Assembly WindowsFormsIntegration.dll zu laden. Diese befindet sich für gewöhnlich im Ordner:

```
%programfiles%\Reference Assemblies\Microsoft\

Framework\v3.0\WindowsFormsIntegration.dll
```

6.2.3 WPF-Control zur Anzeige bringen

Wenn nun sowohl das WPF-Control als auch das Windows Formular erstellt wurden, kann das WPF-Control folgendermaßen eingebunden werden:

```
private void MainForm_Load(object sender, EventArgs e)
{
  ElementHost host = new ElementHost();
  LoginControl lc = new LoginControl();
  host.Child = lc;

  host.Dock = DockStyle.Fill;
  this.Controls.Add(host);
}
```

Bei einem Start der Anwendung sollte nun das WPF-Control sichtbar sein. Wichtig hierbei ist, dass das Host-Element via `DockStyle.Fill` am Windows Formular ausgerichtet werden muss.

6.3 Oberstes Element bei Mausklick mittels HitTest feststellen

Unter der Windows Presentation Foundation kommt es immer wieder vor, dass bei einem Mausklick, auf beispielsweise einem Panel, festgestellt werden muss, welches das oberste von vielen übereinander liegenden Elementen ist (siehe auch WPF: BringToFront und SendToBack).

Hierzu ist die Methode VisualTreeHelper.HitTest sehr praktisch. Diese ermittelt auf Basis der übergebenen Koordination automatisch das in der Z-Order am höchsten angesiedelte Child-Element und liefert als Ergebnis ein HitTestResult zurück, welches eine Referenz auf das entsprechende Element enthält. So kann nun wieder mit dem entsprechenden Objekt weiter gearbeitet werden.

Zur Veranschaulichung noch ein kleines Beispiel. Gegeben sei eine Anwendung, die durch folgendes XAML beschrieben wird:

```
<Window x:Class="HitTestDemo.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="HitTestDemo" Height="300" Width="300">

<Grid>
  <Canvas
     x:Name="BaseCanvas"
     Background="LightYellow">
    <Canvas
      x:Name="Canvas1"
      Canvas.Left="50"
      Canvas.Top="50"
      Background="Blue"
      Width="50"
      Height="50"/>
    <Canvas
      x:Name="Canvas2"
      Canvas.Left="200"
      Canvas.Top="200"
      Background="Red"
      Width="50"
      Height="50"/>
    <Canvas
      x:Name="Canvas3"
      Canvas.Left="40"
      Canvas.Top="75"
      Background="Black"
      Width="50"
      Height="50"/>
    </Canvas>
</Grid>

</Window>
```

In der Code-Behind-Datei findet sich der nachfolgende Code:

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;

namespace HitTestDemo
{
  public partial class Window1 : Window
  {
    public Window1()
    {
      InitializeComponent();

      BaseCanvas.MouseDown +=
        new MouseButtonEventHandler(BaseCanvas_MouseDown);
    }

    private void BaseCanvas_MouseDown(
      object sender, MouseButtonEventArgs e)
    {
      HitTestResult htr =
      VisualTreeHelper.HitTest(this, e.GetPosition(this));
      MessageBox.Show("Clicked item: " +
        ((Canvas)htr.VisualHit).Name);
    }
  }
}
```

Dies ausgeführt zeigt zuerst dieses Fenster:

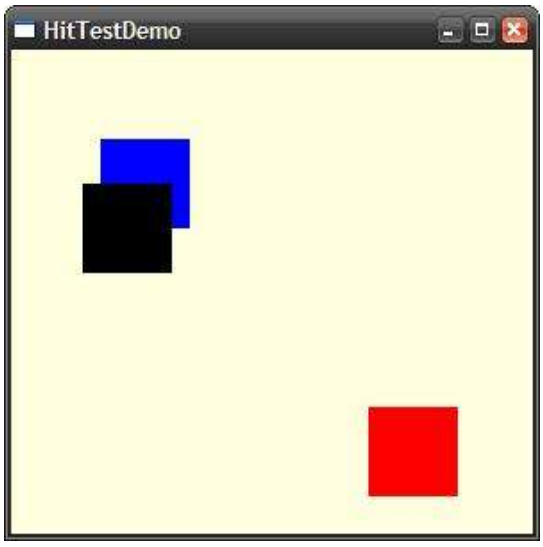


Abbildung 6-3: HitTest Demo 1

Je nachdem, auf welchen Child-Canvas nun geklickt wird, erscheint der Name des jeweiligen Canvas in einer `MessageBox`, wie hier gezeigt:

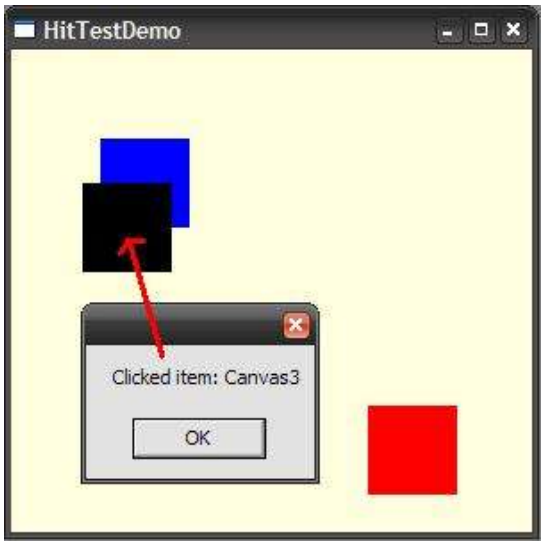


Abbildung 6-4: HitTest Demo 2

6.4 Einführung in die Routed Events

Bisher waren Events genau einem einzigen Element zugewiesen und wurden am angehängten Eventhandler behandelt. Mit WPF wurden die so genannten **Routed Events** eingeführt, welche sich je nach Strategie durch den WPF Elementbaum bewegen. Insgesamt gibt es drei unterschiedliche Arten dieser Weiterleitungen

- Bubbling
- Tunneling
- Direkt

6.4.1 Bubbling

Zuerst wird der Eventhandler der Quelle aufgerufen. Wird also beim Button geklickt wird der direkt angehängte Eventhandler aufgerufen. Anschließend wird der Event an alle Elternelemente weitergereicht, bis das Wurzelelement erreicht ist.

6.4.2 Tunneling

Der Eventhandler des Wurzelelements wird als erstes aufgerufen. Dieser gibt den Event anschließend an alle Kinder weiter, bis der Auslöser erreicht wird.

6.4.3 Direkt

Die direkte Event-Strategie entspricht der Funktionsweise, wie wir sie bis dato gewohnt waren. D.h. nur das Element selbst erhält den Event.

6.4.4 Warum Routed Events

Als Entwickler ist es nicht immer notwendig zu wissen, welche Strategie nun für den Event implementiert wurde, da spezielles Verhalten entsprechend versteckt wurde.

Notwendig wird das Wissen um Routed Events bei der Erstellung von eigenen Steuerlementen usw.

Listener und Quellen von Routed Events müssen keinen eigenen Event in ihrer Hierarchie teilen. Jedes `UIElement` bzw. `ContentElement` kann ein Listener sein. Vielmehr agieren die Routed Events als "Interface" worüber Informationen ausgetauscht werden können.

Ebenfalls ist es möglich, über den Elementbaum hinweg zu kommunizieren, da die Event-Daten an jedes Element im Baum weitergereicht werden und somit für alle sichtbar sind. So kann beispielsweise ein Element die Daten verändern, worauf ein anderes Element entsprechend reagieren kann.

6.4.5 Weitere Informationen

Weitere Informationen können über das MSDN bezogen werden. Ein guter Einstiegspunkt findet sich im Artikel Routed Events Overview.

6.5 Code-Only Anwendungen mit WPF entwickeln

Eine WPF-Anwendung muss nicht immer mit Hilfe von XAML geschrieben werden, auch ohne XAML kann eine WPF-Anwendung erstellt werden. Warum dies notwendig ist könnte mehrere Gründe haben, der wohl oft vorkommendste: XAML ist nicht bekannt und es fehlt die Einarbeitungszeit. Eine andere Möglichkeit besteht natürlich auch darin, dass zur Laufzeit ein Fenster dynamisch erstellt werden sollen.

Wie ist also vorzugehen: Grundsätzlich kann eine ganz normale WPF-Anwendung erstellt werden. Die Dateien `App.xaml` und `Window1.xaml` sind anschließend zu löschen (die Codebehind-Dateien werden automatisch mit gelöscht). Der Vorteil dieser Variante liegt darin, dass die notwendigen Referenzen alle dem Projekt hinzugefügt werden und somit diesbezüglich keinerlei Aufwand entsteht (natürlich kann dies auch manuell geschehen).

Wer dies manuell machen möchte, muss folgende Referenzen seinem Projekt hinzufügen:

- PresentationCore
- PresentationFramework
- WindowsBase

Nun wird im ersten Schritt eine neue Klasse *App.cs* dem Projekt hinzugefügt. Diese erbt von *Application* und enthält den notwendigen Einstiegspunkt.

```
public class App : Application
{
  [STAThread]
  static void Main(string[ args)
  {
    Application app = new Application();
    app.Run(new MainWindow());
  }
}
```

Im Grunde passiert nichts anderes, als dass eine neue *Application*-Instanz erstellt wird und deren Methode *Run* aufgerufen wird. Als Parameter erhält sie eine Instanz des Typs *MainWindow*. MainWindow ist so aktuell noch nicht verfügbar, werden wir jedoch im nächsten Schritt erstellen.

```
public class MainWindow : Window
{
  StackPanel mainPanel = null;
  Label label1 = null;
  Button button1 = null;
  private void InitializeComponent()
  {
    mainPanel = new StackPanel();
    mainPanel.Name = "MainPanel";
    mainPanel.Orientation = Orientation.Vertical;
    label1 = new Label();
    label1.Content = "Code Only App";
    label1.FontSize = 18;
    button1 = new Button();
    button1.Content = "Close";
    mainPanel.Children.Add(label1);
    mainPanel.Children.Add(button1);
    this.Content = mainPanel;
  }
  public MainWindow()
  {
    InitializeComponent();
    Init();
    button1.Click +=
    new RoutedEventHandler(button1_Click);
  }
  void button1_Click(
    object sender,
    RoutedEventArgs e)
  {
    Application.Current.Shutdown(0);
  }
  private void Init()
  {
    this.Title = "Code Only App";
    this.Height = 300;
    this.Width = 300;
  }
}
```

Die Klasse *MainWindow.cs* erbt von *Window* und stellt unser Hauptfenster dar. Wie zu sehen ist, werden nun lediglich durch Code (kein XAML) die einzelnen anzuzeigenden Elemente erstellt. Sämtliche Methoden wurden manuell erstellt, d.h. es gibt hier keinen Code, der irgendwie automatisch erstellt wurde.

Das Ergebnis kann in der nachfolgenden Grafik *bewundert* werden. Natürlich kann auf diese Variante ebenfalls ein hübsches Design erzeugt werden. Diese Demo sollte jedoch nur grundsätzlich zeigen, wie Code-Only-Applikationen erstellt werden können.

Abbildung 6-5: WPF-Anwendung per Sourcecode

6.6 Offene Fenster im Überblick behalten

Wir kennen es wohl alle: Das "Window"-Menü mit allen offenen Fenstern derselben Applikation. Vor allem in den Office-Produkten war dies immer wieder zu finden. Doch wie kann dies unter WPF implementiert werden? Dieser Artikel zeigt wie's geht. Und so kann das Endresultat aussehen (es ist zwar nicht hübsch, aber es funktioniert):

Zu Beginn stellt sich natürlich dir Frage ob es da nicht schon etwas Fertiges gibt. Ja. Gibt es. Unter **Application.Current.Windows** findet sich die aktuelle Auflistung aller geöffneten Fenster. Einziger Nachteil: Diese stecken in einer eigenen **WindowCollection**, welche lediglich **ICollection** und **IEnumerable** implementiert.

Damit jedoch Data Binding möglich ist, wäre eine `ObservableCollection` notwendig.

Damit es uns Entwicklern nicht fad wird, müssen wir daher selbst in die Tasten klopfen und unser eigenes System entwickeln, wollen wir doch Data Binding verwenden!

Zuerst benötigen wir eine simple Datenklasse, welche Informationen für uns hält und auch später wieder zugänglich macht:

```
public class WindowInformation
{
    private Window _window;

    public WindowInformation(Window window)
    {
        _window = window;
    }

    public Window Window
    {
        get { return _window; }
    }

    public int WindowHashCode
    {
        get { return _window.GetHashCode(); }
    }

    public String Title
    {
        get { return _window.Title; }
    }

    public ImageSource Icon
    {
        get { return _window.Icon; }
    }
}
```

Die Datenklasse fungiert in diesem Fall lediglich als Wrapper-Klasse, könnte aber ohne großen Aufwand um weitere Informationen angereichert werden.

Damit nun Data Binding verwendet werden kann, benötigen wir eine - wie schon angesprochen - `ObservableCollection`:

```
public class WindowInformationCollection :
    ObservableCollection<WindowInformation>
{
}
```

Nun wird eine **Manager-Klasse** benötigt, welche die Liste der geöffneten Fenster hält und zusätzliche Funktionen wie Entfernen eines Fensters oder das Zurückgeben eines speziellen Fensters zur Verfügung stellt. Dieser Manager wurde mit dem Singleton-

Pattern umgesetzt, damit sichergestellt ist, dass nur eine Instanz davon pro gestartete Anwendungsinstanz vorhanden ist.

```
public class WindowManager
{
    private static WindowManager _manager;

    private WindowInformationCollection _openWindows =
        new WindowInformationCollection();

    private WindowManager() { }

    public static WindowManager GetInstance
    {
        get
        {
            if (_manager == null)
                _manager = new WindowManager();
            return _manager;
        }
    }

    public WindowInformationCollection OpenWindows
    {
        get { return _openWindows; }
    }

    public Window GetOpenWindow(int hashCode)
    {
        foreach (WindowInformation wi in _openWindows)
        {
            if (wi.WindowHashCode == hashCode)
            {
                return wi.Window;
            }
        }
        return null;
    }

    public bool RemoveOpenWindow(int hashCode)
    {
        WindowInformation windowToRemove = null;
        foreach (WindowInformation wi in _openWindows)
        {
            if (wi.WindowHashCode == hashCode)
            {
                windowToRemove = wi;
                break;
            }
        }
        if (windowToRemove != null)
        {
            _openWindows.Remove(windowToRemove);
            return true;
        }
        return false;
    }
}
```

Zu guter Letzt wird noch ein Handler implementiert, der schlussendlich in das **Window**-Element des XAMLs eingebunden werden kann und somit ein Fenster in das System einbindet:

```
public class WindowInformationHandler
{
    public static readonly DependencyProperty
        IsHandledProperty =
    DependencyProperty.RegisterAttached("IsHandled",
        typeof(bool), typeof(WindowInformationHandler),
        new FrameworkPropertyMetadata((bool)false,
            new PropertyChangedCallback(OnIsHandledChanged)));

    private static void OnIsHandledChanged(
        DependencyObject dObj,
        DependencyPropertyChangedEventArgs e)
    {
        Window openWindow = dObj as Window;
        if (openWindow != null)
        {
            openWindow.Loaded +=
                new RoutedEventHandler(WindowLoaded);
            openWindow.Closed +=
                new EventHandler(WindowClosed);
        }
    }

    public static bool GetIsManaged(DependencyObject dObj)
    {
        return (bool)dObj.GetValue(IsHandledProperty);
    }

    public static void SetIsManaged(
        DependencyObject dObj,
        bool value)
    {
        dObj.SetValue(IsHandledProperty, value);
    }

    private static void WindowClosed(
        object sender, EventArgs e)
    {
        Window closedWindow = sender as Window;
        if (closedWindow != null)
        {
            WindowManager.GetInstance.RemoveOpenWindow(
                closedWindow.GetHashCode()
                );
            openWindow.Loaded -=
                new RoutedEventHandler(WindowLoaded);
            openWindow.Closed -=
                new EventHandler(WindowClosed);
        }
    }

    private static void WindowLoaded(
        object sender, RoutedEventArgs e)
    {
        Window openWindow = sender as Window;
        if (openWindow != null)
        {
            WindowInformation winInfo =
                new WindowInformation(openWindow);
            WindowManager.GetInstance.OpenWindows.Add(winInfo);
        }
    }
}
```

Wird die Eigenschaft **IsHandled** auf **true** gesetzt, werden zwei Events (Loaded und Closed) registriert. Auf diese wird in weiterer Folge reagiert, um das Fenster nach erfolgtem Laden in die Liste der geöffneten Fenster aufzunehmen bzw. davon auch wieder zu entfernen.

In XAML wird dies folgendermaßen in das Window-Element eingebunden:

```
local:WindowInformationHandler.IsManaged="true"
```

Damit ist dieses System fertig und kann angewandt werden.

Und hier noch der Inhalt aus dem Code Behind:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        mnuWindows.ItemsSource =
            WindowManager.GetInstance.OpenWindows;
    }

    private void NewWindow(object sender,
        RoutedEventArgs e)
    {
        MainWindow window = new MainWindow();
        window.Title = String.Format("Open Window {0}",
            Application.Current.Windows.Count);
        window.Show();
    }

    private void ShowWindow(object sender,
        RoutedEventArgs e)
    {
        MenuItem mi = sender as MenuItem;
        if (mi != null)
        {
            WindowInformation wi = mi.DataContext
                as WindowInformation;
            if (wi != null)
            {
                wi.Window.Focus();
            }
        }
    }
}
```

6.7 Eigenen XML-Namespace erstellen

Wer bereits etwas mit WPF gemacht hat, wird den WPF und den XAML Namespace kennen. Beide werden automatisch beim Erstellen eines Elementes in das vordefinierte XAML eingetragen.

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Zum Einen stehen uns darüber alle möglichen WPF-Elemente zur Verfügung, als auch ein paar XAML-Erweiterungen (Präfix x:) für das Erstellen von Arrays, Inline-Code, Typenangaben usw.

Wer nun eigene Elemente einbinden möchte, greift für gewöhnlich auf einen CLR-Namespace zurück:

```
xmlns:demo="clr-namespace:WpfNamespaceDemo.Lib;assembly=WpfNamespace.Lib"
```

Wurden nun jedoch zahlreiche Elemente entwickelt, liegen diese in unterschiedlichen Namespaces und sollen viele davon in einem Window verwendet werden, entsteht eine nette Liste von CLR-Namespace-Definitionen. Viel einfacher geht dies über die Erstellung eines eigenen XML-Namespaces.

Hierfür kann das Attribute *XmlnsDefinition* verwendet werden. Dieses wird in der *AssemblyInfo.cs* definiert und sieht so aus:

```
[assembly: XmlnsDefinition(
    "http://www.norberteder.com/2008/wpf",
    "WpfNamespaceDemo")]
```

Was wird hier gemacht? Der erste Parameter definiert den Wert des eigenen XML-Namespaces, der zweite Parameter gibt an, welcher CLR-Namespace in zum XML-Namespace hinzugefügt werden soll. Dies kann nun für mehrere CLR-Namespaces gemacht werden.

Schlussendlich kann darauf nun folgendermaßen zugegriffen werden:

```
xmlns:local="http://www.norberteder.com/2008/wpf"
```

Es werden nun sämtliche CLR-Namespaces eingebunden, welche unter dem XML-Namespace zusammengefasst wurden. Dies fördert die Übersichtlichkeit und vermindert Fehler. Allerdings muss dies auch entsprechend gewartet werden und kann aufwendiger werden, wenn zig unterschiedliche Assemblies ins Spiel kommen.

Hinweis: Wird in der aktuellen Assembly ein XML-Namespace definiert, kann dieser nicht verwendet werden. Dies funktioniert nur, wenn eine Assembly mit einem definierten XML-Namespace eingebunden wird.

6.8 Webgrafik einfach und schnell mit WPF anzeigen

Wie geht man ans Werk, wenn in einer WPF-Anwendung Grafiken aus dem Internet geladen und angezeigt werden sollen? Nun ja, herunterladen, zwischenspeichern (Dateisystem, oder Arbeitsspeicher), eine `BitmapSource` erstellen und dann der Source eines Image-Elementes zuweisen.

Aber das geht auch einfacher!

```
private void HandleImage(Image image, Uri webUri)
{
  BitmapDecoder bDecoder = BitmapDecoder.Create(
    webUri,
    BitmapCreateOptions.PreservePixelFormat,
    BitmapCacheOption.None);

  if (bDecoder != null && bDecoder.Frames.Count > 0)
    image.Source = bDecoder.Frames[0];
}
```

Das ist auch schon alles was man tun muss. Stichwort ist der `BitmapDecoder`. Dieser bekommt in der statischen Methode *Create* ein Uri-Objekt (daher muss es nicht unbedingt ein Image im Web sein) übergeben. Wenn alles gut geht, können wir das Resultat aus der Frames-Auflistung beziehen und zur Anzeige bringen.

7 Tools

7.1 Unterstützung beim Debuggen von WPF-Anwendungen

Das Debuggen von WPF-Anwendungen kann sehr anstrengend und aufwendig sein. Quasi manuell müssen Eigenschaftswerte überprüft werden, der genaue Ablauf will genau durchlaufen werden. Hier kommen kleine Helferleins zum Einsatz, die das Leben als WPF-Entwickler wesentlich vereinfachen.

An dieser Stelle möchte ich zwei Tools erwähnen, die ein visuelles Debuggen von WPF-Anwendungen erlauben. D. h. es wird der VisualTree dargestellt und die entsprechenden Einstellungen können einfach gesichtet werden. Zudem ist es auch eine kleine Lernhilfe bezüglich des Ablaufes und des Aufbaus.

7.1.1 Snoop

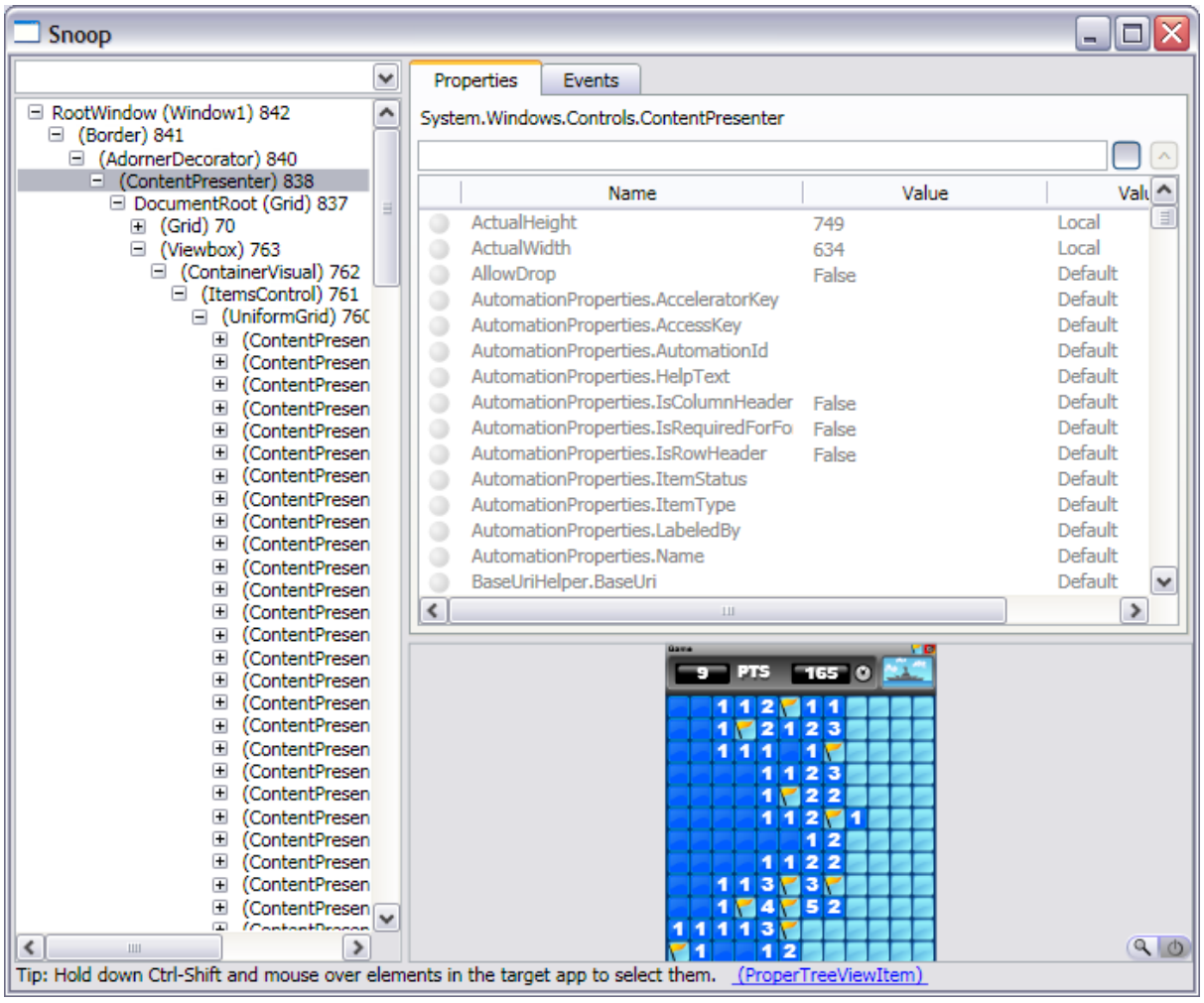


Abbildung 7-1: Screenshot Snoop

Link: http://www.blois.us/Snoop/

7.1.2 Woodstock

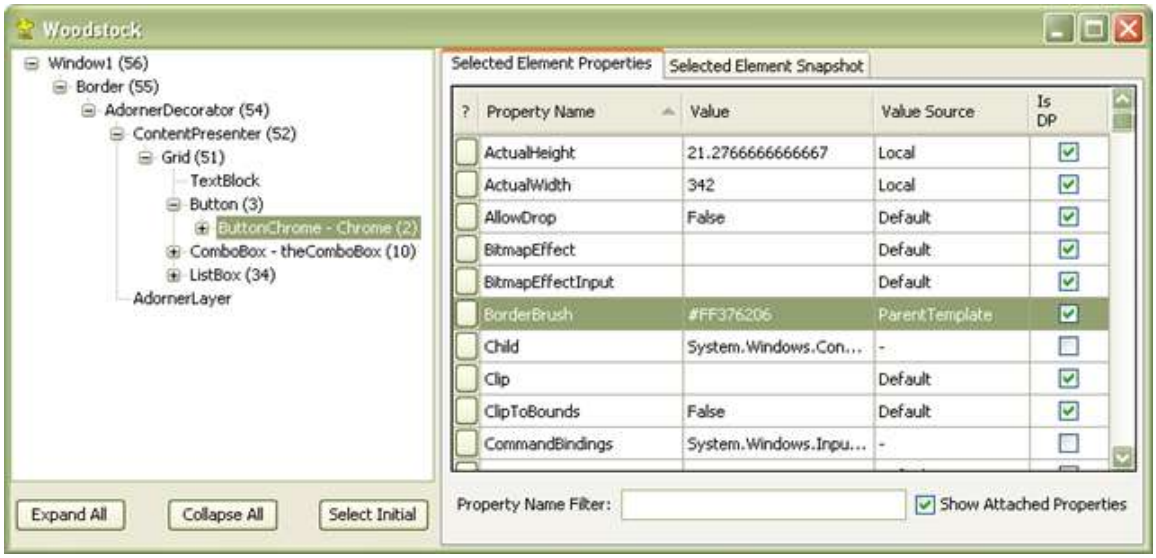


Abbildung 7-2: Screenshot Woodstock

Link: http://www.codeproject.com/useritems/WoodstockForWPF.asp

Grundsätzlich bieten beide Tools einen ähnlichen Funktionsumfang. Der große Unterschied liegt jedoch darin, dass Snoop eine eigene kleine Anwendung ist, die sich an die gewählte Anwendung anhängt, während Woodstock direkt ins Visual Studio eingebunden ist und daher das Handling einfacher ist.

7.2 WPF: Performance messen und verbessern

Hier und hier befinden sich bereits Informationen zum Thema WPF und Performance .

Nun möchte ich auf den MSDN Artikel Performance Profiling Tools for WPF verweisen. Dieser Artikel beschäftigt sich mit WPFPerf aus dem Windows SDK und beschreibt die inkludierten 5 Performance Profiling Tools:

- Perforator
- Visual Profiler
- Working Set Analyzer
- Event Trace
- ETW Trace Viewer

Auf jeden Fall ein sehr hilfreicher Artikel, der eine gute Übersicht der Tools aus dem SDK bietet. Damit ist man durchaus in der Lage, sich mit Performancetests rund um WPF Applikationen zu beschäftigen.

Abbildungsverzeichnis