Walter Doberenz
Thomas Gewinnus
Jürgen Kotz
Walter Saumweber

Visual C# 2017 – Grundlagen, Profiwissen und Rezepte

Bonuskapitel

HANSER

Die Autoren:
Prof. Dr.-Ing. habil. Werner Doberenz, Wintersdorf
Dipl.-Ing. Thomas Gewinnus, Frankfurt/Oder
Jürgen Kotz, München
Walter Saumweber, Ratingen

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Lektorat: Sylvia Hasselbach
Sprachlektorat: Walter Doberenz
Layout: Kösel Media GmbH, Krugzell

Print-ISBN: 978-3-446-45359-3
E-Book-ISBN: 978-3-446-45370-8

Inhalt

Teil I: WPF-Anwendungen

- Einführung in WPF
- Übersicht WPF-Controls
- Wichtige WPF-Techniken
- Grundlagen WPF-Datenbindung
- Drucken/Druckvorschau in WPF

1 Einführung in WPF

Wer innovative und optisch anspruchsvollere Windows Desktop-Anwendungen entwerfen will, kommt nicht um die Verwendung von WPF, dem Nachfolger/Pendant der Windows Forms, herum.

WPF ist die Abkürzung für *Windows Presentation Foundation*. Hierbei handelt es sich im weitesten Sinne um eine weitere Windows Präsentations-Schnittstelle bzw. ein Framework für die Entwicklung interaktiver Anwendungen. Eines der wesentlichsten Merkmale ist die strikte Trennung von Präsentations- und Geschäftslogik, basierend auf der Beschreibungssprache XAML.

Wer den Umstieg von Windows Forms nach WPF wagt, muss mit beträchtlichem Portierungsaufwand rechnen, da vielfach gänzlich andere Konzepte zum Einsatz kommen. Eine Teilmigration Ihrer Anwendung wird von Microsoft durch das *ElementHost*-Control unterstützt, dieses kann WPF-Elemente innerhalb einer Windows Forms-Anwendung darstellen. Im Umkehrschluss lassen sich Windows Forms-Elemente per *WindowsFormsHost* auch in WPF-Anwendungen nutzen.

Bei Neuentwicklungen für den Desktop sollten Sie jedoch in jedem Fall die Verwendung von WPF in Betracht ziehen, dies auch im Hinblick auf eine Wiederverwendbarkeit Ihres Codes z. B. im Rahmen einer Windows-App.

1.1 Einführung

Bevor wir uns im Gestrüpp der WPF-Programmierung verirren, wollen wir zunächst einen Blick auf die Grundkonzepte, Vor- und Nachteile sowie die möglichen Anwendungstypen werfen.

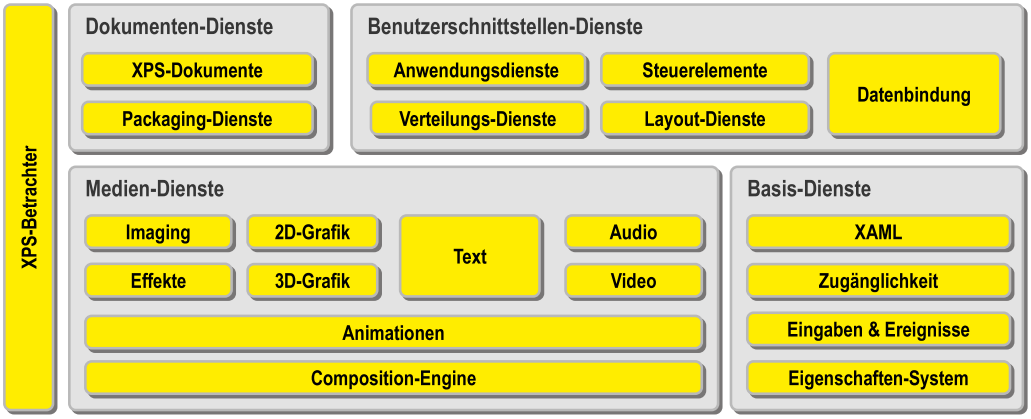
1.1.1 Was kann eine WPF-Anwendung?

Nachdem Sie die letzten 10-15 Jahre mit fast den gleichen Bibliotheken und APIs (GDI32, User32) gekämpft haben, stellt sich bei einer so radikalen Änderung zunächst die Frage, was kann WPF bzw. was macht den Unterschied zur bisherigen Vorgehensweise aus?

Statt vieler Worte also zunächst eine kurze Liste der Highlights:

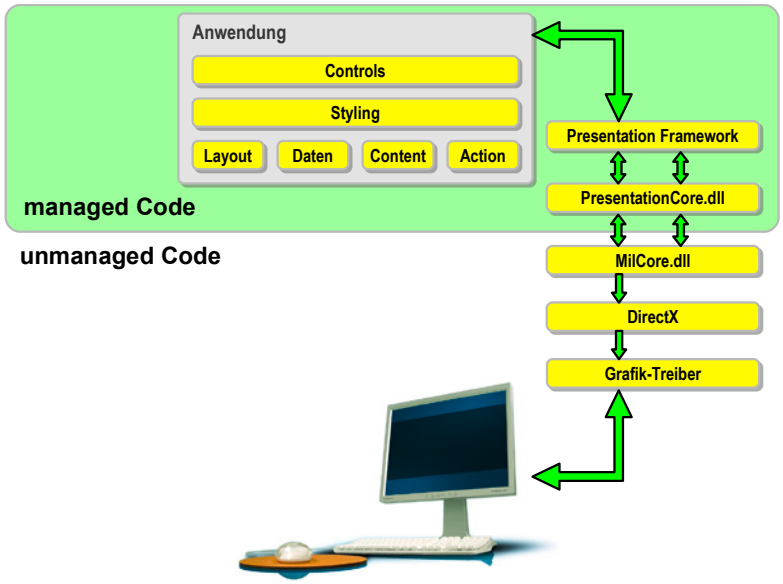
- Hierarchische Oberflächenbeschreibung mit XAML (XML), alternativ können die Oberflächen auch per Code beschrieben werden,
- Codebehind-Modell, ähnlich wie bei ASP.NET-Anwendungen,
- Strikte Trennung von Design und Logik,
- Desktop- (Fenster-basiert) oder Browser-Anwendung (Seiten-basiert) möglich,
- vektorbasierte Grafikausgabe (Fließkomma-Arithmetik), damit frei skalierbare Oberflächen unabhängig vom Ausgabegerät,
- schnelle Grafikausgabe dank Hardwarebeschleunigung und Ausgabe per DirectX,
- umfangreiche Unterstützung für 2D- und 3D-Grafik,
- hervorragende Layout-Optionen für Texte und Steuerelemente,
- umfangreiche Unterstützung für Grafik-Effekte (Schatten, Transparenz, Rotation, Scherung etc.) bei der Oberflächengestaltung,
- komplexe grafische Animationen für Elemente,
- Unterstützung von Medien (Videos, Bilder, Audio),
- einfache Datenbindung für fast alle Eigenschaften möglich
- ClickOnce oder XCopy-Deployment,
- teilweise Abwärtskompatibilität durch Windows Forms-Integration,

WPF soll all diese Möglichkeiten nicht nur unter einem Dach vereinigen, sondern die einzelnen Konzepte auch sinnvoll miteinander verzahnen. Microsoft teilt die einzelnen Funktionen dazu in eine Reihe von Diensten auf, die Sie als Entwickler in Anspruch nehmen können:



Alle obigen Dienste liegen als managed Code vor, verwenden das .NET-Framework und setzen auf DirectX für die Grafikausgabe auf.

Die folgende Abbildung zeigt Ihnen in einer Übersicht, welche Teile von WPF als managed Code vorliegen und welche Abhängigkeiten zwischen den einzelnen Ebenen bestehen.



HINWEIS: Aus Performance-Gründen ist die *MilCore.dll* in unmanged Code geschrieben, hier laufen alle Grafikausgaben der Anwendung durch.

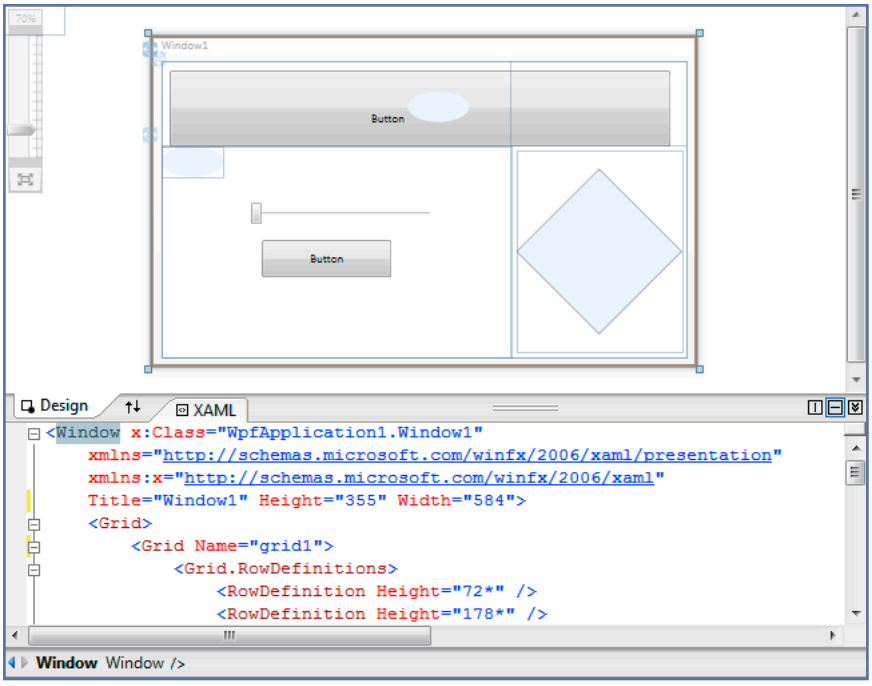
1.1.2 Die eXtensible Application Markup Language

Wie schon kurz erwähnt, basieren WPF-Anwendungen im Normalfall[1] auf einer Trennung von Oberflächenbeschreibung und Quellcode, wie Sie es auch von ASP.NET-Anwendungen her kennen. Die Oberfläche selbst wird mit einer XML-basierten Beschreibungssprache namens XAML (*eXtensible Application Markup Language*, gesprochen „Xemmel“) definiert, die Programmlogik schreiben Sie wie gewohnt in C# oder VB.

Diese strikte Trennung ermöglicht es auch, die Oberfläche von einem Designer und die Logik von einem Programmierer erstellen zu lassen. Wer jetzt denkt, als Programmierer arbeitslos zu werden, braucht keine Sorge zu haben. Das grundsätzliche Programmgerüst werden Sie nach wie vor entwerfen, die optische Gestaltung (Aussehen von Controls, Animationen Grafiken etc.) kann dann ein Grafiker übernehmen.

[1] Sie könnten auch eine Anwendung nur per Code erzeugen, aber dann gehen Sie vermutlich auch nur zu Fuß und verwenden kein Auto.

Über grundlegende XAML-Kenntnisse sollten Sie als Programmierer ebenfalls verfügen. Auch wenn Ihnen der Visual Studio-Designer viel Arbeit abnimmt, ist es häufig wesentlich einfacher, ein paar Tags in das XAML-Dokument einzufügen, als mühevoll die Oberfläche zusammenzuklicken und per Eigenschafteneditor zu konfigurieren. Diese Tatsache scheint auch Microsoft nicht entgangen zu sein, der XAML-Code wird im Editor parallel zum Designer angezeigt, Änderungen in einem der beiden Editoren wirken sich wechselseitig aus (siehe folgende Abbildung).



HINWEIS: Da Microsoft zwei Zielgruppen (Designer und Entwickler) mit WPF ansprechen möchte, sind auch zwei getrennte Entwicklungsumgebungen verfügbar. Vor einer sitzen Sie vermutlich gerade (Visual Studio), die zweite nennt sich *Blend for Visual Studio* und richtet sich vornehmlich an Designer.

Unsere erste XAML-Anwendung

Statt vieler Worte wollen wir uns zunächst mit einem einfachen XAML-Beispiel beschäftigen, bevor wir uns in den Details verlieren.

Beispiel 1.1: Eine erste XAML-Anwendung

XAML

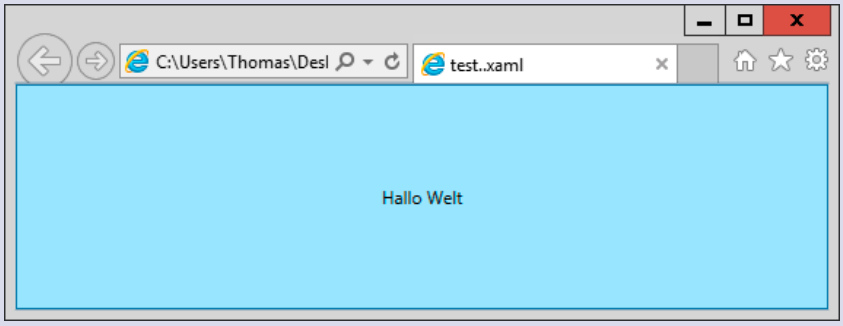
Öffnen Sie einen Texteditor (*Notepad*) und tippen Sie folgenden Code ein.

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
   Hallo Welt
</Button>
```

Speichern Sie nachfolgend die "Anwendung" unter dem Namen *Test1.xaml* ab.

Ergebnis

Mit einem Doppelklick können wir den ersten Test starten. Ist das Framework ab Version 3 installiert, sollte im Internet-Explorer eine Schaltfläche angezeigt werden, die den gesamten Clientbereich ausfüllt:



HINWEIS: Die Angabe des Namespaces (*xmlns=...*) in obigem Beispiel macht dem Interpreter klar, um was für einen Button es sich eigentlich handelt. „Hallo Welt“ ist der im Button enthaltenen Content, darauf kommen wir später noch zurück.

So, das ging ja schon recht schnell, auch wenn wir noch keine Funktionalität implementiert haben. Ein paar Worte zu den Hintergründen der neuen XAML-Anwendung:

- Obige Datei wird als ungebundenes bzw. stand-alone XAML bezeichnet. Dateien mit dieser Extension sind automatisch mit dem *PresentationHost* verknüpft und werden durch diesen interpretiert.
- Da es sich um kein komplettes Programm handelt, wird automatisch ein Objekt vom Typ *Page* erzeugt, dem obiger XAML-Code als Content (Inhalt) zugewiesen wird.
- Die so erzeugten *Page*-Objekte können im Internet-Explorer angezeigt werden.

Probleme mit dem Stammelement

Mutig geworden, wollen wir nun versuchen, noch einen zweiten Button in unser Prográmmchen einzufügen.

Beispiel 1.2: Darstellung von zwei Schaltflächen

XAML

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
   Hallo Welt
</Button>
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
   Hallo XAML
</Button>
```

Statt der erwarteten Schaltflächen taucht im Browser eine Fehlermeldung auf, die uns mit vielen Worten den knappen Sachverhalt erklären will, dass mehr als ein Stammelement vorhanden ist.

Das scheint zwar auf den ersten Blick richtig zu sein (siehe obiges Listing), aber kurz vorher hatten wir ja erklärt, dass diese Daten als Content in ein *Page*-Objekt eingefügt werden, damit ist ja wohl ein Stammelement vorhanden. Dies ist auch korrekt, jedoch schreibt die XAML-Spezifikation vor, dass sowohl ein *Window* als auch eine *Page* nur **ein untergeordnetes Element** enthalten dürfen[2].

Dieses Regel ist für einen Windows Forms-Programmierer sicher gewöhnungsbedürftig, kann er doch dort beliebig viele Controls in ein Fenster einfügen.

Umgehen können wir diese Einschränkung, indem wir zunächst ein sogenanntes Container-Control definieren, in das wir unsere Schaltflächen einfügen. WPF bietet einen reichhaltigen Fundus an Container-Klassen an, wir werden uns in Abschnitt 1.2 ausführlich damit beschäftigen.

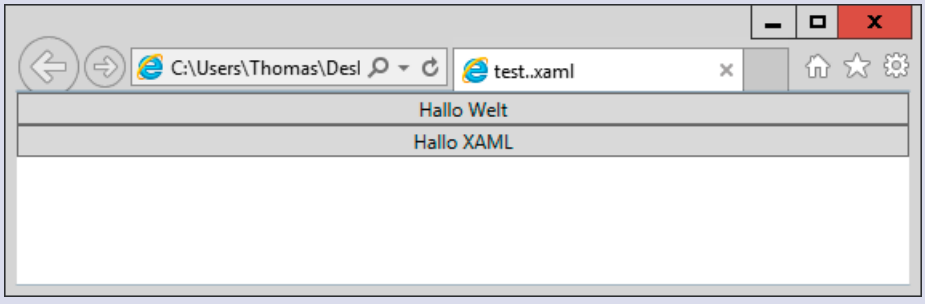
Beispiel 1.3: Darstellung von zwei Schaltflächen (zweiter Versuch)

XAML

```
<StackPanel  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Button>Hallo Welt</Button>
  <Button>Hallo XAML</Button>
</StackPanel>
```

Ergebnis

Ein Testlauf bringt jetzt auch die erwarteten zwei Schaltflächen auf den Bildschirm:



[2] Darüber werden Sie sicher mehr als einmal stolpern ...

HINWEIS: Da der Namespace bereits beim *StackPanel* festgelegt wurde, können wir bei den untergeordneten Schaltflächen darauf verzichten.

Aus obigem Beispiel können wir auch noch zwei weitere Erkenntnisse ableiten:

- XAML-Oberflächen sind hierarchisch aufgebaut (*Page → StackPanel → Button, Button)*
- Der Content (Inhalt) bestimmt die Abmessungen des Controls (Höhe der Schaltflächen[3])

Ein kleines Experiment

Nach so viel Erkenntnissen trauen wir uns an ein kleines Experiment. XAML-Oberflächen sind hierarchisch aufgebaut. Das wollen wir auf etwas eigenwillige Art überprüfen, indem wir einen Button in den anderen einfügen.

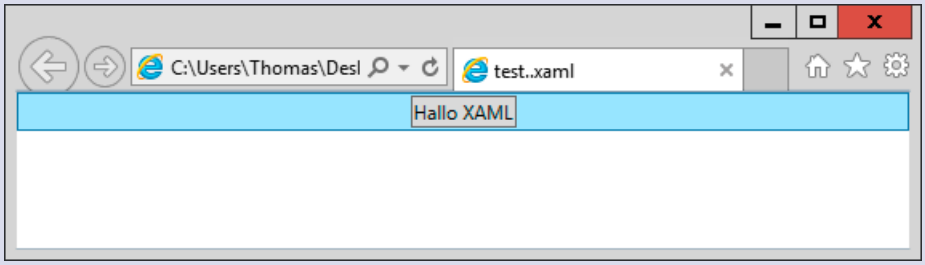
Beispiel 1.4: Button im Button

XAML

```
<StackPanel  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" >
   <Button>
      <Button>
         Hallo XAML
      </Button>
   </Button>
</StackPanel>
```

Ergebnis

Ein Testlauf zeigt das gewünschte Ergebnis:



Für alle Ungläubigen: Der innere Button ist wirklich im Content des äußeren Buttons enthalten, nicht darüber. Das merken Sie schon daran, dass eine Mausbewegung über den inneren Button auch die äußere Schaltfläche aktiviert. Dies bedeutet auch, dass ein Mausklick sich auf beide Controls auswirkt (mehr dazu später).

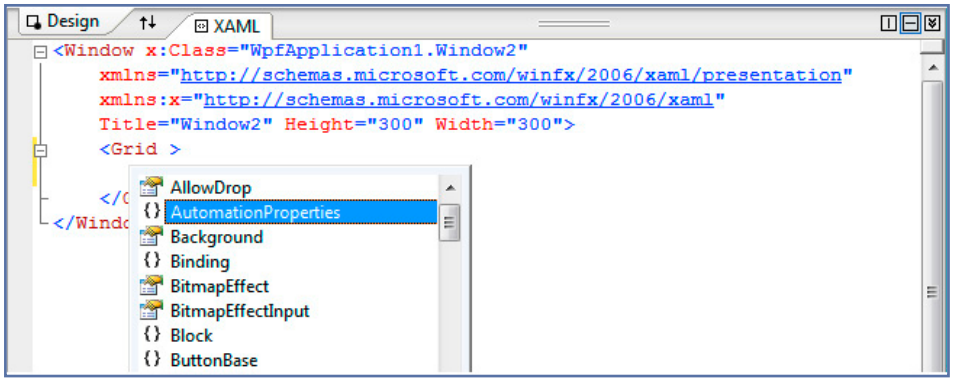
Soll ich das alles von Hand eintippen?

Nein, natürlich nicht, wir wollten nur demonstrieren, dass Sie WPF-Oberflächen auch mit einem einfachen Editor programmieren können. Für die Eingabe stehen Ihnen in Visual Studio entweder die XAML-Ansicht oder die bekannte Design-Ansicht zur Verfügung.

[3] Die Breite wird in diesem Fall durch das umgebenden *StackPanel* bestimmt.

Der XAML-Editor bietet mit der integrierten IntelliSense bereits eine gute Grundlage, um Controls und deren Eigenschaften (Attribute) zu definieren bzw. Ereignismethoden zuzuweisen.

Sie werden im Laufe der Zeit feststellen, dass Sie mit dem XAML-Editor recht häufig arbeiten, da dies wesentlich intuitiver als die mühsame Klickerei mit dem Designer ist.



1.1.3 Verbinden von XAML und C#-Code

Bisher haben wir uns nur mit der XAML-Oberflächenbeschreibung beschäftigt, doch so ganz ohne Code wird Ihr Programm wohl kaum auskommen. An dieser Stelle wollen wir deshalb unsere bisherigen Experimente abbrechen und uns der Projektverwaltung in Visual Studio zuwenden.

Öffnen Sie ein neues Projekt und wählen Sie den Projekttyp „WPF-Anwendung“. Visual Studio erstellt daraufhin vier Dateien, zwei XAML-Dateien (*App.xaml*, *MainWindow.xaml*) und zwei dazugehörige Klassendefinitionen (*App.xaml.cs*, *MainWindow.xaml.cs*) in C#.

Schauen wir uns diese Dateien einmal im Detail an.

App.xaml

In dieser Datei wird, neben der Einbindung der beiden obligatorischen Namespaces, die Klasse *App* konfiguriert. Im vorliegenden Fall wird mit dem Attribut *StartupUri* festgelegt, welches Fenster als erstes angezeigt wird.

Beispiel 1.5: *App.xaml*

XAML

```
<Application x:Class="WpfApplication1.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

HINWEIS: Der Bezug zur Hintergrundcodedatei wird über das Attribut *x:Class* hergestellt (*Namespace.<Klassennamen>*).

Ganz nebenbei nimmt diese Datei auch noch die Definition von Anwendungsressourcen (globale Ressourcen) auf, wir kommen ab Abschnitt 3.2 darauf zurück.

App.xaml.cs

Hierbei handelt es sich um die Hintergrundcodedatei zu obiger XAML-Datei mit der noch leeren Klassendefinition:

Beispiel 1.6: *App.xaml.cs*

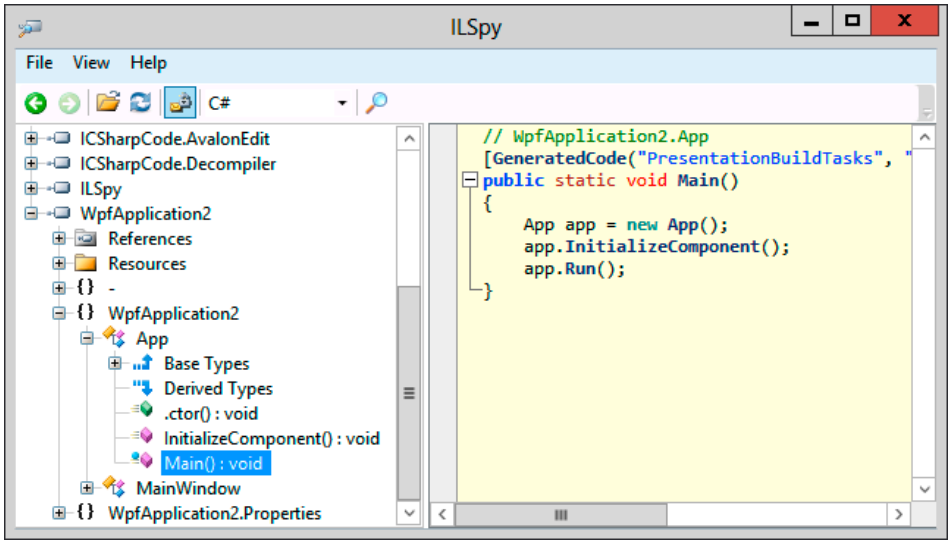
C#

```
using System;
using System.Collections.Generic;
...
using System.Windows;

namespace WpfApplication1
{
...
    public partial class App : Application
    { }
}
```

Die derart definierte *App*-Klasse können Sie zur Laufzeit über *App.Current* abrufen. Die Klasse selbst stellt bereits einen umfangreichen Satz an Ereignissen und Eigenschaften zur Verfügung (mehr dazu ab Abschnitt 1.3.1).

Wer das eigentliche Hauptprogramm in dieser Datei vermutet hat, liegt falsch, denn dieses wird von Visual Studio automatisch erzeugt. Zusätzlich wird noch eine Methode *InitializeComponent* angelegt, in der das Startformular entsprechend der Optionen in *App.xaml* festgelegt wird.

Wer es nicht glaubt, der kann sich mit dem ILSpy davon überzeugen, dass die finale Assembly über ein Hauptprogramm verfügt (siehe folgende Abbildung).



HINWEIS: Das obige *STAThread*-Attribute gibt an, dass das COM-Threading-modell für die Anwendung *Singlethread-Apartment* ist. Dieses Attribut muss vorhanden sein, andernfalls funktionieren die Controls/Windows nicht richtig.

Beispiel 1.7: Der Inhalt der Methode *InitializeComponent*

C#

```
[DebuggerNonUserCode]
public void InitializeComponent()
{
    base.StartupUri = new Uri("MainWindow.xaml", UriKind.Relative);
}
```

MainWindow.xaml

Hier haben wir es mit der XAML-Beschreibung des ersten bereits automatisch erzeugten Fensters zu tun. Den Bezug zum C#-Sourcecode stellt das x:Class-Attribut her, die Einbindung der beiden obligaten Namespaces dürfte Ihnen bereits aus der *App.xaml* bekannt sein:

Beispiel 1.8: *MainWindow.xaml*

C#

```
<Window x:Class="WpfApplication2.MainWindow"
    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>
    </Grid>
</Window>
```

Mit dem Attribut *Title* wird der Inhalt der Kopfzeile definiert, *Width*, und *Height* dürften selbsterklärend sein.

Das *Grid*-Element definiert bereits den grundlegenden Container (und damit das Layout) für das neue Fenster, dieses Element können Sie bei Bedarf löschen und durch eines der anderen Container-Elemente ersetzen (siehe Abschnitt 1.2).

MainWindow.xaml.cs

Die letzte der vier Dateien dürfte die „Spielwiese“ für Sie als Programmierer sein, hier haben Sie es mit der Code-Hintergrunddatei für das mit *MainWindow.xaml* definierte Fenster zu tun.

Beispiel 1.9: *MainWindow.xaml.cs*

C#

```
using System;
...
using System.Windows.Shapes;

namespace WpfApplication2
{
...
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

In dieser Datei werden die Ereignis-Handler, zusätzliche Methoden und lokale Variablen abgelegt.

Ein erster Ereignis-Handler

Sicher wollen Sie endlich auch etwas Aktivität in unser Programm bringen, fügen Sie dazu einen einfachen *Button* in den Designer und damit in das *Grid*!!! von *MainWindow* ein. Alternativ können Sie auch die XAML-Datei von *MainWindow* bearbeiten:

Beispiel 1.10: Ein erster Ereignis-Handler

XAML

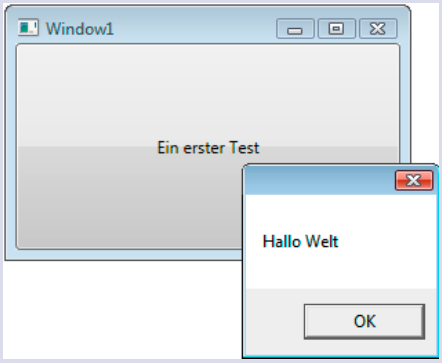
```
...
    <Grid>
        <Button>Ein erster Test</Button>
    </Grid>
...
```

C#

Nach einem Doppelklick auf den Button erzeugt Visual Studio für Sie bereits den Ereignis-Handler. Fügen Sie jetzt noch eine *MessageBox.Show ...* ein, und Sie haben ein funktionstüchtiges Programm, bei dem XAML-Oberfläche und C#-Code miteinander verbunden sind.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
   MessageBox.Show("Hallo Welt");
}
```

Ergebnis



Sicher nicht ganz uninteressant ist jetzt die XAML-Datei, hier muss ja ein Verweis auf die zugehörige Ereignis-Routine eingefügt werden. Und siehe da, dem Attribut *Click* wird der Name der Ereignis-Methode übergeben:

```
...
    <Grid>
      <Button Click="Button_Click">Ein erster Test</Button>
   </Grid>
...
```

Und wo ist mein Button-Objekt?

Versuchen Sie jetzt einmal per Code auf den Button zuzugreifen, um zum Beispiel dessen Beschriftung (Content) zu ändern. Haben Sie den Button per XAML erzeugt, dürfte Ihnen dies schwerfallen, denn ein derartiges Objekt ist nicht zu finden. Ursache ist die fehlende Zuweisung des Name-Attributs in der XAML-Datei.

HINWEIS: Ein XAML-Element muss nur dann benannt werden, wenn Sie es auch per Code ansprechen wollen.

Erweitern Sie also die XAML-Definition des Buttons um ein *Name*-Attribut mit eindeutigem Namen:

```
...
 <Grid>
```

```
    <Button Name="button1" Click="Button_Click">Ein erster Test</Button>
 </Grid>
...
```

Jetzt können Sie in der Codedatei wie gewohnt mit dem Objekt *button1* arbeiten und zum Beispiel dessen Beschriftung ändern:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    button1.Content = "Das funktioniert also auch ...";
}
```

Die folgende Abbildung zeigt noch einmal die Zusammenhänge zwischen Code und XAML:

Brauche ich unbedingt eine Trennung von Code und XAML?

Im Prinzip nein, aber eine Mischung von Code und XAML dürfte der Übersichtlichkeit Ihrer Programme sicher nicht zuträglich sein, ganz abgesehen davon, dass in diesem Fall eine getrennte Bearbeitung von Oberfläche und Logik nicht mehr sinnvoll realisierbar ist.

Aus diesen Gründen gehen wir auf die Möglichkeit, Code und XAML in einer Datei unterzubringen, nicht ein.

Kann ich Oberflächen auch per Code erzeugen?

Im Prinzip ja, der Aufwand ist jedoch teilweise recht beträchtlich, da sich die Eigenschaften und Zuweisungen leichter per XAML konfigurieren lassen.

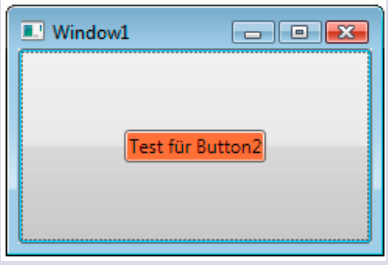
Beispiel 1.11: Einen neuen *Button* in *button1* erzeugen

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Button btn = new Button();
    btn.Content = "Test für Button2";
    btn.Name = "button2";
    btn.Background = Brushes.Coral;
    button1.Content = btn;
}
```

Ergebnis

Das Resultat zeigt die folgende Abbildung:



Dabei wollen wir es an dieser Stelle belassen, auf weitere Details werden wir im Verlauf des Kapitels zu sprechen kommen.

1.1.4 Zielplattformen

Für den Entwickler sicher nicht ganz uninteressant dürfte die Frage nach den Zielplattformen für die erstellten WPF-Anwendungen sein. Hier müssen Sie sich auf Plattformen beschränken, auf denen das .NET-Framework mindestens ab der Version 3.0 läuft.

HINWEIS: In diesem Fall müssen Sie die Zielplattform in den Projekteigenschaften explizit auf 3.0 festlegen. Möchten Sie neuere Features in Ihrer Anwendung unterstützen, wählen Sie als Zielplattform mindestens „.NET Framework 4.5 Client“ oder „.NET Framework 4.5“, wie wir es auch für dieses Buch voraussetzen.

Folgende unterstützte Betriebssysteme kommen nach obiger Forderungen infrage:

- Windows 10
- Windows 8/8.1
- Windows Server 2012
- Windows 7
- Windows Server 2008
- Windows Vista
- Windows XP (ab SP 3) und Windows 2003 Server (hier müssen Sie das .NET-Framework erst noch installieren)

1.1.5 Applikationstypen

Microsoft bietet verschiedene Templates im Zusammenhang mit WPF an, u a.:

- WPF-Anwendung
- WPF-Browseranwendung, teilweise auch als Express-Anwendung bezeichnet
- WPF-Benutzersteuerelementebibliothek
- Benutzerdefinierte WPF-Steuerelementebibliothek

Im Folgenden wollen wir uns kurz mit den Unterschieden der beiden erstgenannten Anwendungstypen auseinandersetzen, um Ihnen die Entscheidung für das eine oder andere Template zu erleichtern.

WPF-Anwendung

Dieser Anwendungstyp entspricht im Wesentlichen den bisherigen Windows Forms-Anwendungen mit folgenden Möglichkeiten:

- Die Installation erfolgt per Setup (MSI) oder ClickOnce auf dem Zielcomputer.
- Ein Verweis kann im Start-Menü bzw. in der Systemsteuerung unter *Software* erfolgen.
- Die Anwendung läuft per Default im Full Trust-Mode, d. h., Sie können auf die Registrierdatenbank, das Dateisystem, WCF etc. zugreifen.
- Die Anwendung läuft in einem eigenen Fenster (Windows).
- Die Anwendung läuft offline.
- Updates müssen explizit installiert werden (per ClickOnce automatisierbar).

WPF-Browseranwendungen

Auch wenn sie so heißt, handelt es sich doch um keine Web-Anwendung, der Name bezieht sich lediglich auf den Anzeigeort der Anwendung.

- Diese Anwendungen laufen nur im Internet Browser mit wizardartiger Oberfläche (Navigation zwischen Pages[4]).
- Die Anwendung läuft mit stark eingeschränkten Rechten in einer Sandbox.
- Die Anwendung wird **nicht** auf dem Zielcomputer installiert.
- Ein Verweis im Start-Menü bzw. in der Systemsteuerung unter Software ist **nicht** möglich.
- Die Anwendungen werden automatisch per ClickOnce verteilt.
- Obwohl der überwiegende Teil der WPF-Features genutzt werden kann, gibt es einige Einschränkungen bei der Darstellung.

1.1.6 Vor- und Nachteile von WPF-Anwendungen

Über die Vorteile von WPF-Anwendungen wissen Sie ja bereits eine ganze Menge (siehe Abschnitt 1.1.1). Der wichtigste Vorteil dürfte für die meisten sicher die phantastische Oberflächengestaltung sein. Hier handelt es sich um einen echten Quantensprung gegenüber den teilweise recht tristen Windows Forms-Anwendungen.

Doch wo viel Licht ist, da ist auch Schatten, und so werden Sie früher oder später auch einige „Haare in der Suppe“ finden:

- WPF-Anwendungen sind nicht auf älteren Betriebssystemen (Windows 2000 etc.) lauffähig, was jedoch mittlerweile vernachlässigbar ist.
- Die Anwendungen erfordern etwas leistungsfähigere Hardware.
- Komplexere Anwendungen erfordern schnell umfangreiche Kenntnisse (steile Lernkurve).
- Nicht alle Windows Forms-Steuerelemente sind vorhanden, einige Standarddialoge fallen schnell durch ihr „altbackenes“ Äußeres auf.
- Viele Konzepte in WPF richten sich an den Designer und nicht an den Entwickler.
- Das Setzen von Eigenschaften per Code ist teilweise recht aufwändig.

[4] Die freie Navigation durch den Anwender kann den Entwickler schnell in den Wahnsinn treiben.

- Trennung von Code und Oberfläche kann Probleme beim dynamischen Erstellen von Oberflächenelemente bereiten.
- WPF-Anwendungen werden schnell zum „Selbstbedienungsladen“, sowohl der C#-Code als auch die XAML-Oberflächenbeschreibung können relativ einfach aus den Anwendungen extrahiert werden (Decompiler).
- Last, but not least, ist nicht jeder Anwender davon begeistert, wenn er eine zwar bunte aber kaum funktionale Anwendung vorgesetzt bekommt. WPF „erleichtert“ es dem Entwickler, konfuse und wenig intuitive Oberflächen zu gestalten, aber das haben Sie ja selbst in der Hand.

Tja, wann also sollten Sie WPF verwenden und die Lektüre dieses Kapitels fortsetzen? Immer dann, wenn Sie obige Einschränkungen nicht stören, Sie sowieso ein neues Projekt beginnen und, das sollte wohl der Hauptgrund sein, Sie einen echten Nutzen mit WPF erzielen.

1.1.7 Weitere Dateien im Überblick

Sicher sind Ihnen bei Nachforschungen in den Projektverzeichnissen auch einige Dateien aufgefallen, mit denen Sie auf die Schnelle nichts anfangen können. Wir wollen uns mit *.baml* und *.g.cs* zwei der wichtigsten Typen herausgreifen und näher anschauen.

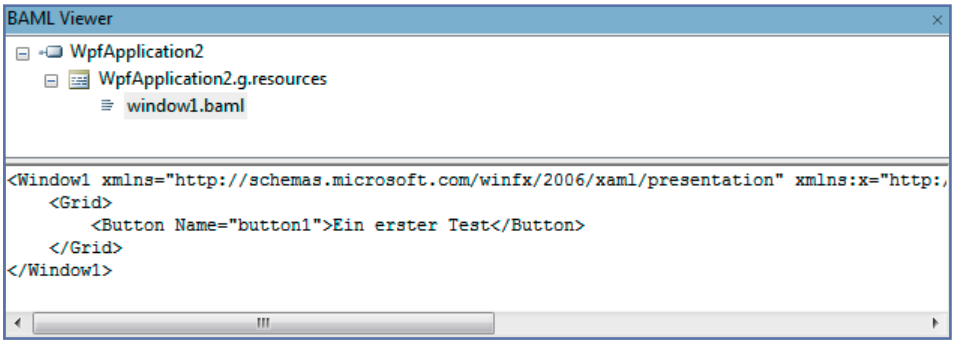
Was sind .BAML-Dateien und was passiert damit?

Bei den im Verzeichnis *\obj\...\Debug* enthaltenen Dateien handelt es sich um binäre Repräsentationen (*Binary Application Markup Language*) der einzelnen XAML-Dateien (Windows/Pages).

Diese sind nicht für die Betrachtung oder Bearbeitung vorgesehen, sondern werden als Ressourcen in die finale Assembly gelinkt.

.BAML-Dateien liegen bereits in einem Zwischenformat vor, was zur Laufzeit eine schnellere Verarbeitung des Objekt-Graphen ermöglicht.

HINWEIS: Mit einem extra Add-In ist der .NET-Reflector in der Lage, diese Ressourcen sichtbar zu machen und als lesbaren XAML-Code anzuzeigen:



Den BAML-Viewer können Sie unter *http://reflectoraddins.codeplex.com/releases/view/1805* herunterladen.

Worum handelt es sich bei den .G.CS-Dateien?

Das „.G“ steht zunächst für „Generated“. Im Detail handelt es sich um die Klassendefinition in der jeweiligen Projekt-Programmiersprache, die automatisch zu jedem XAML-File (Window/Page) angelegt wird. Diese Datei stellt den genauen Zusammenhang zwischen XAML-Elementen (Controls) und den jeweiligen C#-Klassen her.

Beispiel 1.12: Eine .G.CS-Datei für *Windows1.xaml*

C#

```
#pragma checksum "..\..\Window1.xaml" "{406ea660-64cf-4c82-b6f0-42d48172a799}"
"871E64C0DAEA9297F82991CACE5FBBA3"
...
using System;
...
using System.Windows.Shapes;
```

Der definierte Namespace:

```
namespace WpfApplication2 {
```

Die bereits in XAML definierte Klasse:

```
    public partial class Window1 : System.Windows.Window,
                                   System.Windows.Markup.IComponentConnector {
```

Haben Sie in die XAML-Datei benannte Controls eingefügt (*Name*-Attribut), finden Sie hier den C#-Pendant:

```
        internal System.Windows.Controls.Button button1;

        private bool _contentLoaded;
```

Der eigentliche Initialisierungsvorgang:

```
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        public void InitializeComponent() {
            if (_contentLoaded) {
                return;
            }
            _contentLoaded = true;
            System.Uri resourceLocater = new
                System.Uri("/WpfApplication2;component/window1.xaml",
                           System.UriKind.Relative);
            #line 1 "..\..\Window1.xaml"
            System.Windows.Application.LoadComponent(this, resourceLocater);
        }
```

Mit der folgenden Methode werden die Objekte und die Ereignisse zugeordnet:

```
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [System.ComponentModel.EditorBrowsableAttribute(
           System.ComponentModel.EditorBrowsableState.Never)]
        [System.Diagnostics.CodeAnalysis.SuppressMessageAttribute("Microsoft.Design",
            "CA1033:InterfaceMethodsShouldBeCallableByChildTypes")]
        void System.Windows.Markup.IComponentConnector.Connect(int connectionId,
                object target)
        {
            switch (connectionId)
            {
            case 1:
            this.button1 = ((System.Windows.Controls.Button)(target));

            #line 6 "..\..\Window1.xaml"
            this.button1.Click += new
System.Windows.RoutedEventHandler(this.Button_Click);

            #line default
            #line hidden
            return;
            }
            this._contentLoaded = true;
        }
    }
}
```

1.2 Alles beginnt mit dem Layout

Vielleicht erscheint Ihnen diese Überschrift etwas merkwürdig, aber nachdem Sie sich über den Typ der WPF-Anwendung (siehe Abschnitt 1.1.5) im Klaren sind, sollten Sie sich in Erinnerung rufen, dass sowohl eine *Page* als auch ein *Window* nur über **ein** untergeordnetes Stammelement verfügen dürfen – ganz im Gegensatz zu einer Windows Forms-Anwendung, bei der Sie recht unbekümmert Controls in den Formularen verteilen können.

Aus diesem Grund werden wir uns in diesem Abschnitt zunächst mit einigen recht speziellen Controls, den Panel-Elementen oder auch Containern beschäftigen, bevor wir auf die eigentlichen Elemente der Bedienoberfläche zu sprechen kommen (Schaltflächen, Listenfelder etc.).

1.2.1 Allgemeines zum Layout

Erster Schritt nach dem Erstellen eines neuen *Window* oder einer neuen *Page* ist die Auswahl einer passenden Panel-Klasse[5], die das spätere Layout der Seite vorgibt. WPF bietet hier eine reiche Auswahl an, die folgende Tabelle zeigt einige wichtige Panels:

Layout-Control	Kurzbeschreibung
Canvas	In diesem Panel können Sie eine absolute Positionierung realisieren.
Dockpanel	Controls können im Panel andocken (*Top*, *Bottom, Left, Right, Fill).*
Grid	Ein Tabellen-Layout wie es auch bei HTML-Seiten verwendet wird.
Stackpanel	Controls werden übereinander (vertikal) oder nebeneinander (horizontal „gestapelt“. Die Richtung kann vorgegebenen werden.
TabPanel	Entspricht einem *TabControl*, enthält untergeordnete *TabItem*s, die wiederum andere Panel-Controls aufnehmen können.
UniformGrid	Formatiert enthaltene Controls in ein festes Raster aus Zeilen und Spalten, alle Controls sind gleich groß.
ViewBox	Enthaltene Grafiken können verschiedenartig skaliert werden.
WrapPanel	Dieses Panel verhält sich ähnlich wie ein *StackPanel,* allerdings werden die enthaltenen Controls bei Bedarf in die nächste „Zeile“ umgebrochen.

Wie Sie sehen, besitzt jedes diese Elemente spezielle Verhaltensweisen für die Anordnung der darin abgelegten Controls. Doch warum brauchen wir überhaupt ein Layout?

Die Antwort findet sich in der Möglichkeit von WPF-Anwendungen, Dialoge frei zu skalieren, d. h., eine Größenänderung des Formulars soll sich auch **sinnvoll** auf die darin enthaltenen Controls auswirken. Absolute Positions- und Größenangaben sind in diesem Zusammenhang ungeeignet, vielmehr soll das Layout durch geschickte Auswahl und Parametrierung von Panel-Controls erzeugt werden. Dabei ist es auch möglich, die Panel-Controls ineinander zu verschachteln, um die gewünschten Effekte zu erreichen.

Beispiel 1.13: Darstellung eines einfachen Taschenrechners (auszugsweise)

XAML

Zunächst das komplette Fenster definieren:

```
<Window x:Class="WpfApplication2.Window2"
...
```

Ein *Grid* mit zwei Zeilen, einer Spalte teilt die Grundfläche im Verhältnis 1:4:

```
<Grid>
 <Grid.RowDefinitions>
   <RowDefinition Height="*" />
   <RowDefinition Height="4*" />
</Grid.RowDefinitions>
```

[5] Standardmäßig ist bereits ein *Grid* enthalten (keine Zeile/Spalten).

In die obere Zelle des Grids wird eine Textbox zur Ausgabe der Werte eingefügt, diese definiert ihre Randabstände über das Attribut *Margin*:

```
<TextBox Grid.Row="0" Margin="4,4,4,10" Name="textBox1" />
```

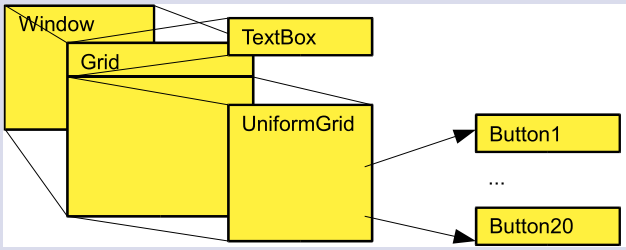
In die untere Zelle des Grids fügen wir ein *UniformGrid* ein, enthaltene Steuerelemente werden in ihrer Größe automatisch an ein per *Columns* bzw. *Rows* festgelegtes Raster angepasst:

```
<UniformGrid Name="uniformGrid1" Columns="5" Rows="4" Grid.Row="1">
```

Was fehlt sind unsere Schaltflächen, die durch das *UniformGrid* automatisch skaliert und positioniert werden. Wir können also auf all diese Angaben verzichten:

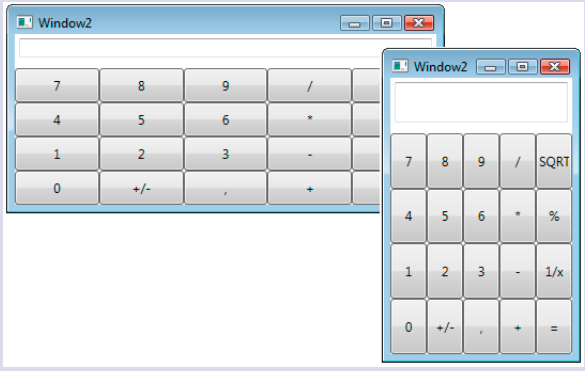
```
      <Button  Name="button1" >7</Button>
      <Button  Name="button2" >8</Button>
...
      <Button  Name="button20" >=</Button>
    </UniformGrid>
  </Grid>
</Window>
```

Dieses noch recht einfache Beispiel hat bereits vier Hierarchieebenen, wie sie in der folgenden Abbildung noch einmal gezeigt werden:



Ergebnis

Zur Laufzeit können Sie das Fenster jetzt beliebig vergrößern/verkleinern bzw. im Seitenverhältnis verändern, das Grundlayout des Fensters bleibt erhalten:



HINWEIS: Vergessen Sie in diesem Zusammenhang nicht die Regel, dass Controls in einigen Panels ohne explizite Größenangabe ihre Abmessungen nach dem Inhalt (Content) bestimmen.

1.2.2 Positionieren von Steuerelementen

Wer sich obiges Beispiel bereits genauer angesehen hat, wird festgestellt haben, dass wir trotz allem einige absolute Maßangaben im XAML-Code versteckt haben. Die Rede ist hier von der *TextBox*, deren äußere Ränder wir mit *Margin* fest definiert haben.

Was sind das überhaupt für Maßangaben?

Um was für Maßangaben bzw. Einheiten (Pixel, Inch ...) handelt es sich eigentlich? Die Lösung: In WPF werden Koordinatenangaben in geräteunabhängigen Pixeln angegeben! Wer jetzt auch nicht schlauer ist, dem sei gesagt, dass Microsoft die Größe eines derartigen Pixels mit genau 1/96 Zoll festgelegt hat.

Dieser Wert leitet sich aus der „angenommenen" Standard-Bildschirmauflösung von 96 DPI (Pixel/ Inch) ab, im Idealfall entspricht also 1 logischer Pixel einem Pixel auf dem Bildschirm. Dass diese Annahme für absolute Maßangaben ziemlicher Blödsinn ist, dürfte spätestens nach einem Wechsel der Bildschirmauflösung klar sein, ein Pixel ist jetzt viel größer/kleiner als vorher.

Denken sie also nicht allzu viel darüber nach und nutzen Sie diesen Wert einfach so wie die allseits bekannten Pixel.

Wer Probleme mit der Umrechnung zwischen dieser schönen neuen Einheit und den konventionellen Einheiten hat, kann sich mit folgenden Umrechnungsfaktoren behelfen:

Masseinheit	Von WPF-Pixel nach ...	Von ... in WPF-Pixel
Inch	0,01041666	96
Millimeter	0,26458333	3,779527
Point	0,75	1,333333

Top/Left/Width/Height

Wenn Sie sich ein Control bzw. dessen Eigenschaften näher ansehen, werden Sie schnell feststellen, dass weder *Top* noch *Left* vorhanden sind. Wozu auch, in den meisten Layout-Controls findet sowieso eine automatische Positionierung statt.

Einzige Ausnahme: das *Canvas*-Control, wo eine absolute Positionierung möglich ist. In diesem Fall helfen Ihnen so genannte „angehängte Eigenschaften" (*Attached Properties*) weiter. Diese beziehen sich jeweils auf das übergeordnete Control, in diesem speziellen Fall auf das *Canvas*-Control.

Beispiel 1.14: Positionieren einer Schaltfläche in einem *Canvas*-Control

C#

```
<Canvas Name="canvas1" >
  <Button Canvas.Left="74" Canvas.Top="70" Height="45" Name="button1"
          Width="89">Beschriftung
  </Button>
</Canvas>
```

Zu *Width* und *Height* brauchen wir Ihnen sicher nicht viel zu sagen, beachten Sie jedoch, dass in den meisten Fällen die Breite/Höhe des Controls durch den umgebenden Container bestimmt wird. Dies ist auch im Sinne einer layout-orientierten Programmierung, in der die Gestaltung vom übergeordneten Layout und nicht vom einzelnen Control bestimmt wird.

MinWidth/MaxWidth/MinHeight/MaxHeight

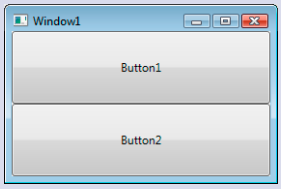
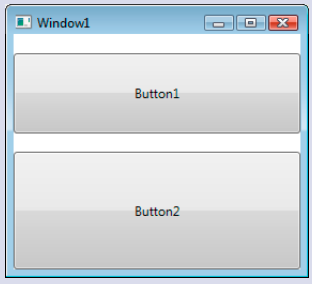
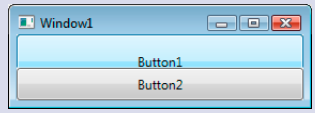
Mit diesen Attributen/Eigenschaften können Sie Mindestgrößen für das Control vorgeben. Sind diese nicht realisierbar (zu wenig Platz), wird das Control abgeschnitten, ist mehr Platz, wird das Control in seiner Größe beschränkt.

Beispiel 1.15: Höhenbeschränkung einer Schaltfläche

XAML

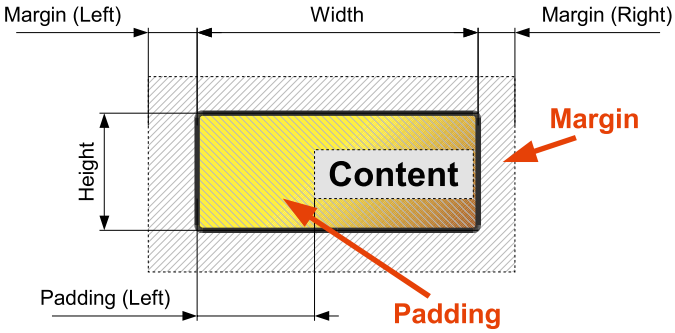
```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Button Grid.Row="0" MinHeight="50" MaxHeight="75">Button1</Button>
  <Button Grid.Row="1" >Button2</Button>
</Grid>
```

Ergebnis

Defaultanzeige	*MaxHeight* dominiert (es entstehen Freiflächen)	*MinHeigh* dominiert (der Button wird beschnitten)
		

Margin/Padding

Im Zusammenhang mit der Ausrichtung von Controls sind auch zwei Eigenschaften interessant, die zum einen den Abstand des Controls zum umgebenden Elternfenster/-Control angeben (Margin), zum anderen den Abstand des Control-Contents zu den Control-Außengrenzen (Padding). Die folgende Abbildung zeigt die entsprechenden Bereiche:



Beide Eigenschaften können Sie vollständig (4 Werte) oder auch vereinfacht angeben (1 Wert, 2 Werte).

Werte	Beispiel	Bedeutung
1 Wert	*Margin=„5“*	der Wert gilt für alle Ränder
2 Werte	*Margin=„5,10“*	der erste Wert bestimmt den linken und rechten Rand, der zweite Wert bestimmt den oberen und unteren Rand
4 Werte	*Margin=„1,5,10,20“*	Einzelangaben für linken, oberen, rechten, unteren Rand

HINWEIS: Die Angabe von 3 oder mehr als vier Werten führt zu einem Laufzeitfehler.

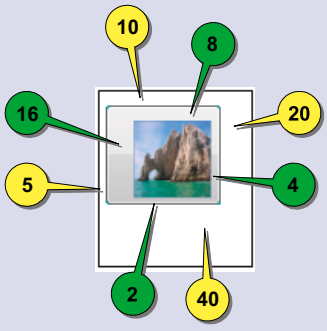
Beispiel 1.16: Verwendung von *Padding* und *Margin* für die Gestaltung einer Schaltfläche

XAML

```
...
   <Button  Margin="5,10,20,40" Padding="16,8,4,2" Name="button2">
     <Image Width="50" Height="50" Source="Bilder/Cabo.jpg" />
   </Button>
...
```

Ergebnis

Die Ausgabe mit Angabe der Abstände:



HorizontalAlignment/VerticalAligment

Vielleicht wundern Sie sich an dieser Stelle, warum wir uns mit der horizontalen bzw. vertikalen Ausrichtung von Controls beschäftigen, wenn diese doch meist in die übergeordneten Layout-Controls hineinskaliert werden.

Der Grund für dieses Verhalten ist die Defaulteinstellung für beide Eigenschaften, diese ist mit *Stretch* festgelegt. Alternativ können Sie für *HorizontalAlignment* auch *Left, Right, Center* auswählen bzw. für *VerticalAlignment* die Werte *Top*, *Bottom* oder *Center*.

Ist ein Wert ungleich *Stretch* festgelegt, bestimmt der enthaltene Content des Controls die Größe bzw. die Angaben von *Width* und *Height*.

1.2.3 Canvas

Mit diesem Control haben Sie die Möglichkeit, die klassische Variante für die Positionierung von Steuerelementen zu realisieren. Dazu stehen Ihnen mit *Top, Left, Right, Bottom* vier angehängte Eigenschaften zur Verfügung.

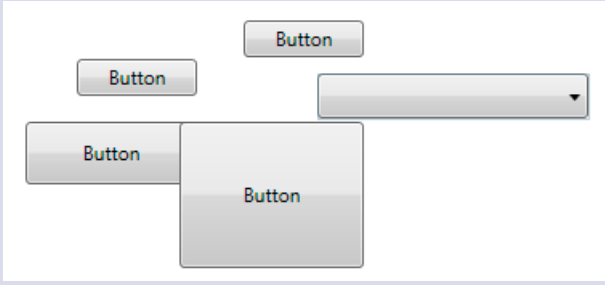
Beispiel 1.17: Freies Positionieren von Controls in einem *Canvas* mit *Left* und *Top*

XAML

```
  <Canvas Height="196" Width="396">
    <Button Canvas.Left="56" Canvas.Top="46" Height="23" Width="75">Button</Button>
    <Button Canvas.Left="24" Canvas.Top="85" Height="39" Width="107">Button</Button>
    <Button Canvas.Left="160" Canvas.Top="22" Height="23" Width="75">Button</Button>
    <Button Canvas.Left="120" Canvas.Top="85" Height="91"
Width="115">Button</Button>
    <ComboBox Canvas.Left="206" Canvas.Top="55" Height="28" Width="169" />
  </Canvas>
```

Ergebnis

Die Laufzeitansicht:



Beachten Sie, dass bei einer Verankerung am rechten oder unteren Rand, im Gegensatz zur *Anchor*-Eigenschaft bei den Windows Forms, die Breite und Höhe des Controls unverändert bleibt:

Beispiel 1.18: Auswirkung einer Fenstergrößenänderung bei Verankerung an der rechten unteren Ecke

XAML

```
  <Canvas >
    <Button Canvas.Right="10" Canvas.Bottom= "20" Height="23"
Width="75">Button</Button>
  </Canvas>
```

Ergebnis

1.2.4 StackPanel

Das *StackPanel* ermöglicht das einfache „Stapeln“ der enthaltenen Controls. Die Richtung dieses Stapels können Sie über die *Orientation*-Eigenschaft steuern.

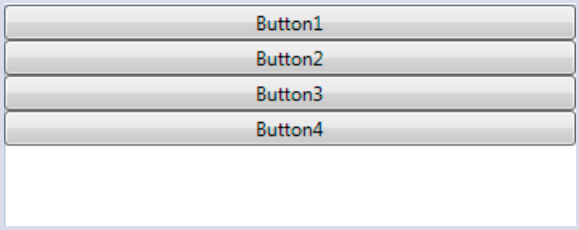
Beispiel 1.19: *StackPanel* mit vertikaler Ausrichtung (Default)

XAML

```
  <StackPanel>
    <Button>Button1</Button>
    Button>Button2</Button>
    <Button>Button3</Button>
    <Button>Button4</Button>
  </StackPanel>
```

Ergebnis

Da keine der enthaltenen Schaltflächen eine Größenangabe enthält, werden die Schaltflächen automatisch an die Breite des *StackPanel*s angepasst, die Höhe ergibt sich aus dem Content der jeweiligen Schaltfläche, wie es auch die folgende Abbildung zeigt:



Änderungen der Breite des *StackPanel*s wirken sich unmittelbar auf die Breite der enthaltenen Controls aus. Wird die Defaultgröße der enthaltenen Controls unterschritten, werden diese abgeschnitten.

Bei horizontaler Ausrichtung kehrt sich das Verhalten um, jetzt wird die Höhe der Controls durch das *StackPanel* bestimmt, die Breite bestimmt sich aus dem Content.

Beispiel 1.20: Horizontale Ausrichtung im *StackPanel*

XAML

```
<StackPanel Orientation="Horizontal" >
  <Button>Button1</Button>
  <Button>Button2</Button>
  <Button>Button3</Button>
  <Button>Button4</Button>
</StackPanel>
```

Ergebnis

HINWEIS: Über die Eigenschaft *FlowDirection* steuern Sie bei horizontaler Ausrichtung die Fließrichtung (*RightToLeft, LeftToRight*), d. h. die Anzeigereihenfolge.

Beispiel 1.21: Änderung der *FlowDirection*

XAML

```
<StackPanel FlowDirection="RightToLeft" Orientation="Horizontal">
  <Button>Button1</Button>
  <Button>Button2</Button>
  <Button>Button3</Button>
  <Button>Button4</Button>
</StackPanel>
```

Ergebnis

1.2.5 DockPanel

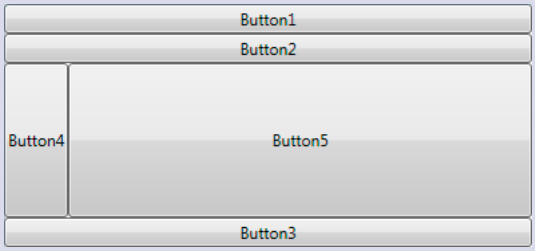
Eines der wichtigsten Controls für das Erzeugen Explorer-ähnlicher Oberflächen dürfte das *DockPanel* sein. Hier können Sie festlegen, an welcher Seite des *DockPanel*s die enthaltenen Controls ausgerichtet werden sollen. Die Ausrichtung wird bei jedem eingelagerten Control über die angehängte Eigenschaft *DockPanel.Dock* bestimmt.

Beispiel 1.22: Einige Schaltflächen in einem *DockPanel* ausrichten

XAML

```
<DockPanel >
  <Button DockPanel.Dock="Top">Button1</Button>
  <Button DockPanel.Dock="Top">Button2</Button>
  <Button DockPanel.Dock="Bottom">Button3</Button>
  <Button DockPanel.Dock="Left">Button4</Button>
  <Button DockPanel.Dock="Right">Button5</Button>
</DockPanel>
```

Ergebnis



Drei Regeln lassen sich bereits aus obigem Beispiel erkennen:

- Controls mit gleicher Ausrichtung werden gestapelt (*Button1*, *Button2*).
- Die Reihenfolge der Controls bestimmt, welches Control „dominant“ ist (siehe folgendes Beispiel).
- Das letzte Control in der Liste füllt den verbliebenen Platz komplett aus, egal welche Angaben gemacht wurden (*Button5*).

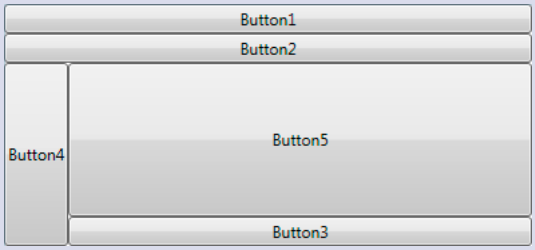
Beispiel 1.23: Einfluss der Reihenfolge auf die Anzeige (wir tauschen lediglich *Button3* und *Button4* miteinander)

XAML

```
<DockPanel >
  <Button DockPanel.Dock="Top">Button1</Button>
  <Button DockPanel.Dock="Top">Button2</Button>
  <Button DockPanel.Dock="Left">Button4</Button>
  <Button DockPanel.Dock="Bottom">Button3</Button>
  Button DockPanel.Dock="Right">Button5</Button>
</DockPanel>
```

Ergebnis

Wie Sie sehen, ist jetzt *Button4* dominanter und bestimmt damit auch die Breite von *Button3* und *Button5*:



Soll das letzte enthaltene Control nicht per Default den gesamten verbliebenen Platz ausfüllen, können Sie dieses Verhalten mit *LastChildFill=False* abschalten.

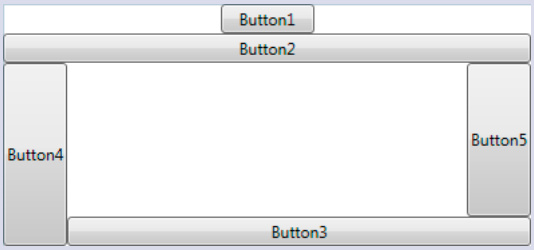
Beispiel 1.24: Änderung des Füllverhaltens und der Breite von Elementen

XAML

```
<DockPanel LastChildFill="False">
  <Button DockPanel.Dock="Top" Width="70">Button1</Button>
  <Button DockPanel.Dock="Top">Button2</Button>
  <Button DockPanel.Dock="Left">Button4</Button>
  <Button DockPanel.Dock="Bottom">Button3</Button>
  <Button DockPanel.Dock="Right">Button5</Button>
</DockPanel>
```

Ergebnis

Geben Sie bei den enthaltenen Controls *Height* oder *Width* an, überschreiben diese Werte die automatisch vorgegebenen Werte des *DockPanel*s (*Button1*):



Im obigen Beispiel wurde *Button1* per Default horizontal zentriert dargestellt. Mit *HorizontalAligment* bzw. *VerticalAligment* können Sie dieses Verhalten ändern.

HINWEIS: Controls in einem *DockPanel* können sich nicht überlappen, notfalls werden diese abgeschnitten.

1.2.6 WrapPanel

Hier haben wir es mit einem nahen Verwandten des *StackPanels* zu tun, das Verhalten ist recht ähnlich, mit dem Unterschied, dass bei fehlendem Platz die enthaltenen Elemente in die nächste „Zeile“ umgebrochen werden.

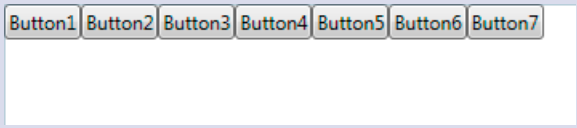
Beispiel 1.25: Ein *WrapPanel* mit Buttons füllen

XAML

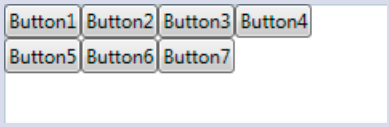
```
<WrapPanel>
  <Button>Button1</Button>
  <Button>Button2</Button>
  <Button>Button3</Button>
  <Button>Button4</Button>
  <Button>Button5</Button>
  <Button>Button6</Button>
  <Button>Button7</Button>
</WrapPanel>
```

Ergebnis

Die erste Ansicht:



Eine Breitenänderung des *WrapPanel*s führt zu folgender Ansicht:



Ist nicht mehr genügend Platz für einen Umbruch vorhanden, werden die betreffenden Elemente abgeschnitten.

1.2.7 UniformGrid

Bevor wir uns dem wesentlich komplexeren Verwandten zuwenden, wollen wir noch einen kurzen Blick auf das *UniformGrid* werfen. Dieses dient dem einfachen Ausrichten von Elementen, wenn diese alle die gleiche Größe erhalten und in einer Rasterstruktur angeordnet werden sollen.

Die Verwendung ist relativ einfach: Definieren Sie über die Eigenschaften *Columns* und *Rows* zunächst die Anzahl der Zeilen und Spalten und fügen Sie die gewünschten Elementen ein.

HINWEIS: Eine gezielte Zuordnung zu den einzelnen Gridzellen ist nicht möglich, hier entscheidet die Reihenfolge der Definition über die Position im *UniformGrid*. Die Zellen werden von links nach rechts und dann von oben nach unten gefüllt.

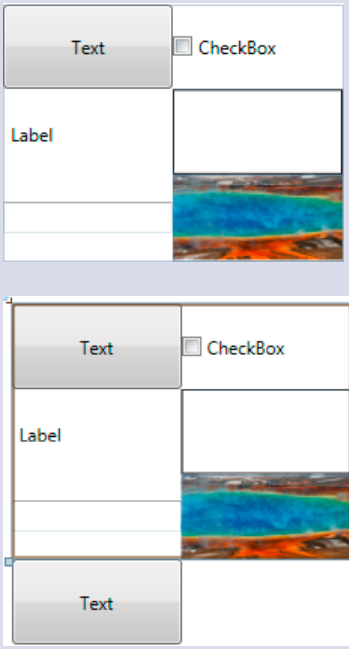
Beispiel 1.26: Verwendung des *UniformGrid*

XAML

```
<UniformGrid Name="uniformGrid1" Columns="2" Rows="3" Grid.Row="1" Height="170"
             Width="226">
  <Button>Text</Button>
  <CheckBox Height="16" Name="checkBox1" Width="120">CheckBox</CheckBox>
  <Label Height="23" Name="label1" Width="120">Label</Label>
  <Rectangle Name="rectangle1" Stroke="Black" />
  <TextBox Height="21" Name="textBox1" />
  <Image Name="image1" Source="Crater.jpg" />
</UniformGrid>
```

Ergebnis

Das angezeigte *UniformGrid:*



HINWEIS: Ist für das *UniformGrid* eine Höhe bzw. Breite vorgegeben, werden Kind-Elemente auch dann dargestellt, wenn sie die Anzahl der vorgegebenen Zeilen und Spalten überschreiten.

1.2.8 Grid

Hier haben wir es mit einem recht komplexen Control zu tun, bei dem Kind-Elemente in Zeilen und Spalten angeordnet werden können. Im Unterschied zum *UniformGrid* ist das *Grid* jedoch wesentlich flexibler, was die Definition von Zeilen und Spalten anbelangt. Auch

die enthaltenen Elemente können freier positioniert werden, es ist sogar möglich, Elemente zellübergreifend zu definieren. Doch der Reihe nach:

Definition des Grundlayouts

Fügen Sie ein neues *Grid* in ein *Window* ein, verfügt dieses zunächst nur über eine einzige Zelle. Neue Zeilen bzw. Spalten definieren Sie über einen gesonderten Bereich *RowDefinitions* bzw. *ColumnDefinitions*.

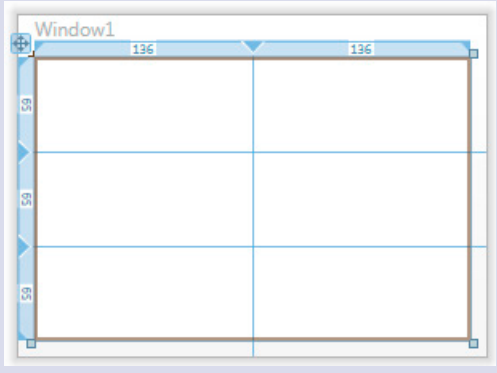
Beispiel 1.27: Ein *Grid* mit zwei Spalten und drei Zeilen definieren

XAML

```
<Window x:Class="WpfApplication2.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="215" Width="294"  >
 <Grid>
   <Grid.RowDefinitions>
     <RowDefinition/>
     <RowDefinition/>
     <RowDefinition/>
   </Grid.RowDefinitions>
   <Grid.ColumnDefinitions>
     <ColumnDefinition/>
     <ColumnDefinition/>
   </Grid.ColumnDefinitions>
 </Grid>
</Window>
```

Ergebnis

Wie Sie sehen, sind bei dieser Art der Definition alle Zellen gleich groß, vergrößern Sie das *Grid*, werden die Spalten und Zeilen im gleichen Verhältnis vergrößert.



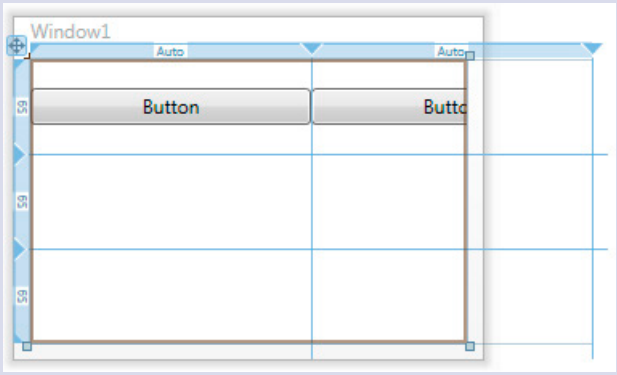
Für die Definition spezifischer Zeilenhöhen bzw. Spaltenbreiten hat sich Microsoft einige Varianten ausgedacht:

Beispiel 1.28: Spaltenbreite entsprechend dem Inhalt festlegen (das breiteste bzw. höchste Element bestimmt die Breite bzw. Höhe)

XAML

```
<Grid>
...
   <Grid.ColumnDefinitions>
     <ColumnDefinition Width="Auto"/>
     <ColumnDefinition Width="Auto"/>
   </Grid.ColumnDefinitions>
   <Button Grid.Column="0" Height="23" Name="button1" Width="175"
           Grid.Row="0">Button</Button>
   <Button Grid.Column="1" Height="23" Name="button2" Width="175"
           Grid.Row="0">Button</Button>
</Grid>
```

Ergebnis



Beispiel 1.29: Festlegen einer bestimmten Breite

XAML

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

Beispiel 1.30: Festlegen eines Anteils am verfügbaren Platz

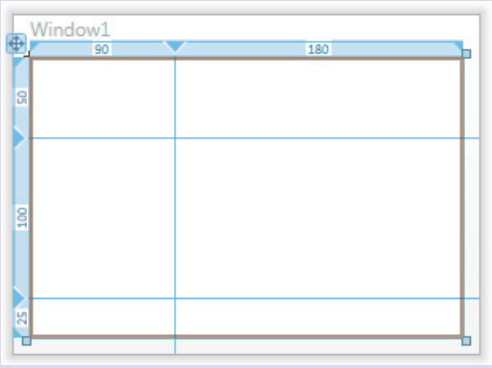
XAML

```
<Grid>
   <Grid.RowDefinitions>
     <RowDefinition Height="*"/>
     <RowDefinition Height="2*"/>
     <RowDefinition Height="0.5*"/>
   </Grid.RowDefinitions>
   <Grid.ColumnDefinitions>
```

```
      <ColumnDefinition Width="*"/>
      <ColumnDefinition Width="2*"/>
    </Grid.ColumnDefinitions>
  </Grid>
```

Ergebnis

In diesem Fall kommen Sie um etwas Rechnerei nicht herum. Für obiges Beispiel gilt: Es gibt 3,5 Zeilenanteile (entspricht 175 Einheiten) und 3 Spaltenanteile (entspricht 270 Einheiten). Die daraus resultierenden Spaltenbreiten bzw. Zeilenhöhen können Sie der folgenden Laufzeitansicht entnehmen:



Wie Sie sehen, können die Zellgrößen im *Grid* recht flexibel definiert werden.

Zuordnen von Kind-Elementen

Die Zuordnung der Kindelemente zu den einzelnen Gridzellen erfolgt über die angehängten Eigenschaften *Grid.Column* und *Grid.Row*, die Sie bei jedem der Kindelemente setzen können.

Beispiel 1.31: Erstellen einer einfachen Oberfläche

XAML

Zunächst definieren wir das *Grid* und dessen Hintergrundfarbe:

```
<Grid Background="AliceBlue">
```

Die Zeilen festlegen:

```
  <Grid.RowDefinitions>
    <RowDefinition Height="*"/>
    <RowDefinition Height="2*"/>
    <RowDefinition Height="0.5*"/>
  </Grid.RowDefinitions>
```

Die Spalten festlegen:

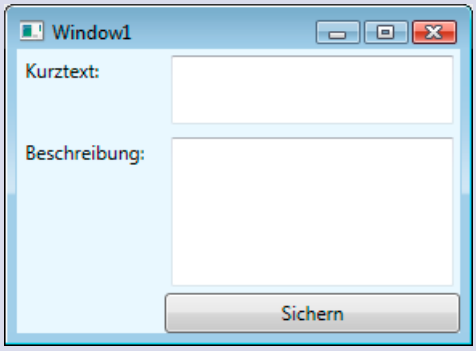
```
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"/>
```

```
      <ColumnDefinition Width="2*"/>
    </Grid.ColumnDefinitions>
```

Und hier kommen die Inhalte:

```
    <Label Grid.Column="0" Grid.Row="0">Kurztext:</Label>
    <Label Grid.Column="0" Grid.Row="1">Beschreibung:</Label>
    <TextBox Grid.Column="1" Grid.Row="0" Margin="4" Name="textBox1" />
    <TextBox Grid.Column="1" Grid.Row="1" Margin="4" Name="textBox2"
 AcceptsReturn="True"/>
    <Button Grid.Column="1" Grid.Row="2"  Name="button1">Sichern</Button>
  </Grid>
```

Ergebnis



HINWEIS: Für das Ausrichten der Inhalte können Sie die Eigenschaften *VerticalAlignment* und *HorizontalAlignment* verwenden.

Doch was ist, wenn sich ein Element über zwei oder drei Spalten bzw. Zeilen erstrecken soll? Auch das ist kein Problem, verwenden Sie dafür die Eigenschaften *Grid.ColumnSpan* bzw. *Grid.RowSpan*, um für das jeweilige Element mehrere Spalten/Zeilen zusammenzufassen.

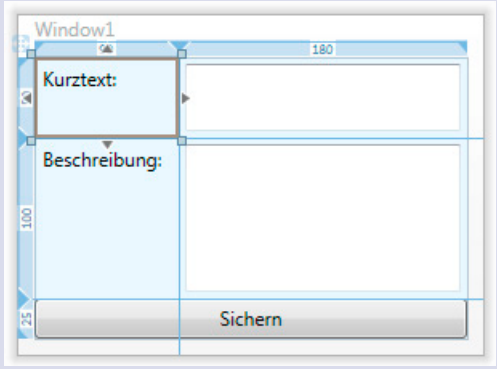
Beispiel 1.32: Die Schaltfläche aus obigem Beispiel soll sich über zwei Spalten erstrecken

XAML

```
  <Grid Background="AliceBlue">
...
    <Button Grid.Column="0" Grid.Row="2"  Grid.ColumnSpan="2"
Name="button1">Sichern</Button>
  </Grid>
```

Ergebnis

In der Entwurfsansicht können Sie deutlich sehen, dass sich die Schaltfläche jetzt über die Spaltentrennlinie erstreckt (siehe folgende Abbildung):



Verwendung des GridSplitters

Möchten Sie zur Laufzeit Einfluss auf die Spaltenbreiten bzw. Zeilenhöhen nehmen, können Sie einen so genannten *GridSplitter* in eine oder auch mehrere Zellen (*ColumnSpan*, *RowSpan*) einfügen. Dessen Breite bestimmen Sie mit *Width*, die Farbe können Sie über das *Background*-Attribut festlegen.

HINWEIS: Per Default ist der *GridSplitter* am rechten Rand der Zelle verankert, Sie können dieses Verhalten aber auch über die *HorizontalAlignment*-Eigenschaft ändern.

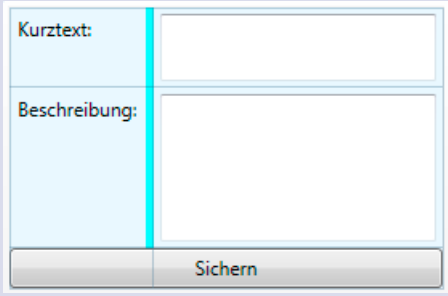
Beispiel 1.33: Verwendung *GridSplitter* für obiges Beispiel

XAML

```
  <Grid Background="AliceBlue">
...
   <GridSplitter Grid.Column="0" Grid.RowSpan="2" HorizontalAlignment="Right"
Width="5"
                 Background="Cyan" />
 </Grid>
```

Ergebnis

Die Entwurfsansicht zeigt bereits, dass der *GridSplitter* nur in den ersten beiden Zeilen des Grids angezeigt wird:



HINWEIS: Der *GridSplitter* überdeckt den Inhalt der jeweiligen Zelle etwas, legen Sie also für die enthaltenen Elemente einen entsprechend großen Randabstand fest (*Margin*).

1.2.9 ViewBox

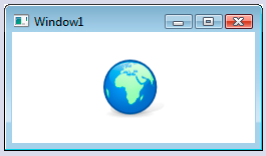
Geht es um die Anzeige von Grafiken in bestimmten Seitenverhältnissen bzw. mit automatischer Anpassung an das übergeordnete Element, ist die *ViewBox* die erste Wahl.

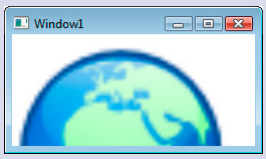
Beispiel 1.34: Platzieren Sie einfach die gewünschte Grafik im Clientbereich und legen Sie per *Stretch*-Attribut das Verhalten fest.

XAML

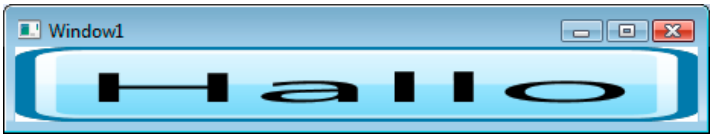
```
<Viewbox Name="viewbox1" Stretch="UniformToFill" >
  <Image Source="browser.png"/>
</Viewbox>
```

Die Tabelle zeigt die Auswirkung der *Stretch*-Eigenschaft auf das Aussehen der enthaltenen Grafik:

Stretch	Beispiel	Beschreibung
None		Grafik wird in Originalgröße angezeigt. Reicht der Platz nicht, wird die Grafik abgeschnitten. Ist die Grafik kleiner als die *ViewBox*, wird die Grafik per Default zentriert.

Stretch	Beispiel	Beschreibung
Fill		Die Grafik wird ohne Rücksicht auf die Proportionen in die *ViewBox* skaliert.
Uniform		Die Grafik wird proportional skaliert, sodass diese vollständig in der Clientfläche angezeigt wird.
UniformToFill		Die Grafik wird proportional skaliert, jedoch so, dass die *ViewBox* vollständig gefüllt ist. Hierbei werden meist Teile der Grafik abgeschnitten

Dass auch beliebige andere Elemente problemlos mit der *ViewBox* skaliert werden können, zeigt das folgende Beispiel eines Buttons:



HINWEIS: Da es sich um Vektorgrafik handelt, sind auch starke Vergrößerungen ohne Qualitätsverlust möglich.

1.2.10 TextBlock

Im Gegensatz zu den bisher vorgestellten Elementen nimmt der *TextBlock* eine Sonderstellung ein. Auch wenn dieser prinzipiell weitere Steuerelemente aufnehmen kann, ist doch seine Hauptaufgabe die Ausgabe von formatierten Texten. Damit empfiehlt sich dieses Control als „großer Bruder“ des guten alten *Label*-Controls, mit wesentlich erweiterten Möglichkeiten.

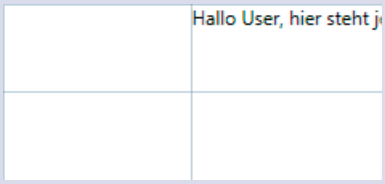
Beispiel 1.35: Ausgabe von Text in einer Gridzelle mit Hilfe eines *TextBlock*-Elements

XAML

```
...
 <Grid>
   <Grid.ColumnDefinitions>
     <ColumnDefinition/><ColumnDefinition/>
   </Grid.ColumnDefinitions>
   <Grid.RowDefinitions>
     <RowDefinition/><RowDefinition/>
   </Grid.RowDefinitions>
   <TextBlock Grid.Column="1" Grid.Row="0">
       Hallo User, hier steht jede Menge Text!
     </TextBlock>
 </Grid>
...
```

Ergebnis

Ein Ergebnis ist bereits zu sehen, dürfte aber noch nicht ganz den Erwartungen entsprechen, da der Text nicht umgebrochen wird



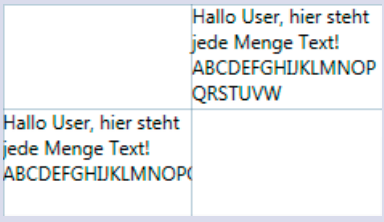
Beispiel 1.36: Umbrechen des Textes mit dem Attribut *TextWrapping*

XAML

```
...
   <TextBlock TextWrapping="Wrap" Grid.Column="1" Grid.Row="0">
       Hallo User, hier steht jede Menge Text! ABCDEFGHIJKLMNOPQRSTUVW
     </TextBlock>
   <TextBlock TextWrapping="WrapWithOverflow" Grid.Column="0" Grid.Row="1">
     Hallo User, hier steht jede Menge Text! ABCDEFGHIJKLMNOPQRSTUVW
   </TextBlock>
...
```

Ergebnis

Die beiden möglichen Umbruch-Varianten unterscheiden sich nur beim Umbruch langer Wörter, die nicht in eine Zeile passen (*Wrap* trennt das Wort, *WrapWithOverflow* schneidet es ab):



Textformatierungen

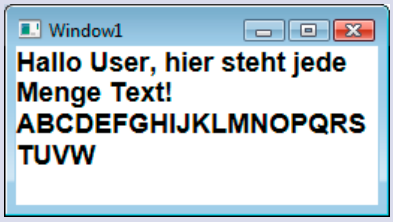
Der auffälligste Unterschied zum *Label* besteht in den umfangreichen Formatierungsmöglichkeiten, die ein *TextBlock* anbietet:

Beispiel 1.37: Einfache Vorgabe eines Textformats für den gesamten *TextBlock*

XAML

```
...
   <TextBlock FontFamily="Arial" FontSize="18" FontWeight="Bold"
TextWrapping="Wrap">
       Hallo User, hier steht jede Menge Text! ABCDEFGHIJKLMNOPQRSTUVW
   </TextBlock>
...
```

Ergebnis



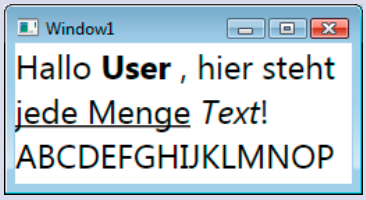
Beispiel 1.38: Spezielle Textformatierungen im *TextBlock*

XAML

```
...
            <TextBlock FontSize="24" TextWrapping="Wrap">
            Hallo <Bold>User</Bold>, hier steht <Underline>jede Menge</Underline>
            <Italic>Text</Italic>! ABCDEFGHIJKLMNOPQRSTUVW
   </TextBlock>
...
```

Ergebnis

Der formatierte Text:



Besonderheit von Leerzeichen/Zeilenumbrüchen

Auf ein Thema der Textausgabe müssen wir unbedingt noch eingehen, da es sich hier um eine Besonderheit des XAML-Compilers handelt. Die Rede ist von der Ausgabe mehrfacher Leerzeichen und von Zeilenumbrüchen, die standardmäßig nicht berücksichtigt werden.

Mit dem Attribut *xml:space* können Sie das Verhalten des Compilers beeinflussen, sodass auch mehrfache Leerzeichen und Zeilenumbrüche richtig interpretiert werden.

HINWEIS: Doch Achtung: In diesem Fall werden auch die Formatierungen im XAML-Quelltext (Einrückungen) berücksichtigt.

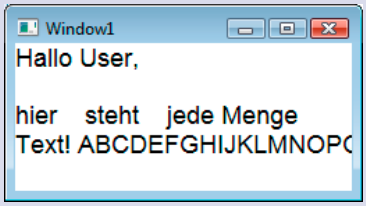
Beispiel 1.39: Ausgabe mehrfacher Leerzeichen und Zeilenumbrüche

XAML

```
<TextBlock xml:space="preserve" FontFamily="Arial" FontSize="18">Hallo User,

hier    steht    jede Menge
Text! ABCDEFGHIJKLMNOPQRSTUVW
</TextBlock>
```

Ergebnis

Die Ausgabe (beachten Sie, dass Wrapping nicht aktiviert ist):



HINWEIS: Zeilenumbrüche könne Sie auch mit einem *<LineBreak/>*-Element erzeugen.

Textausrichtung

Neben dem Textformat ist meist auch eine spezielle Ausrichtung der Texte gewünscht. Der *TextBlock* verwendet dafür, wie wohl nicht anders zu erwarten, das Attribut *TextAligment*. Die möglichen Werte sind *Left, Right, Center* (mittig zentriert), *Justify* (Blocksatz).

1.3 Das WPF-Programm

Da zeigen wir Ihnen im vorhergehenden Abschnitt schon diverse WPF-Elemente und kümmern uns gar nicht um das eigentliche Programmgerüst! Das wollen wir nun nachholen, bevor wir uns mit weiteren Details der WPF-Controls beschäftigen.

Wie bereits im Abschnitt 1.1.5 erwähnt, müssen Sie zwischen zwei grundsätzlichen WPF-Anwendungstypen unterscheiden:

- WPF-Anwendung
- WPF-Browseranwendung

Erstere ist eine Ansammlung von einzelnen Windows, wie Sie es auch aus den altgedienten Windows Forms-Anwendungen kennen[6], WPF-Browseranwendungen verwenden statt einzelner Fenster so genannte Pages, zwischen denen navigiert werden kann (ähnlich ASP.NET).

HINWEIS: Im Folgenden wollen wir uns aus Platzgründen auf den WPF-Anwendungstyp beschränken.

1.3.1 Die App-Klasse

Auf die in der Datei *App.xaml.cs* abgeleitete *App*-Klasse sind wir bereits in Abschnitt 1.1.3 eingegangen. Eine zugehörige Instanz dieser Klasse können Sie zur Laufzeit über die statische Methode *App.Current* abrufen.

Interessant ist diese Klasse vor allem für den Lebenszyklus des Programms:

- Festlegen des Startobjekts
- Auswerten von Kommandozeilenparametern
- Beenden der Anwendung
- Auswerten von Anwendungsereignissen

Wir wollen uns die einzelnen Punkte im Folgenden etwas näher ansehen.

1.3.2 Das Startobjekt festlegen

Wie schon erwähnt, wird das Hauptprogramm der WPF-Anwendung automatisch vom Visual Studio-Compiler erzeugt. Doch wo legen wir dann das Startobjekt bzw. das Startfenster fest?

Hier hilft uns die Datei *App.xaml* weiter, in dieser wird mit dem *StartupUri*-Attribut das erste Fenster festgelegt:

```
<Application x:Class="WpfApplication2.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

[6] SDI-Anwendung, MDI-Anwendungen werden noch nicht unterstützt.

Diesen Eintrag können Sie ohne Probleme auf ein beliebiges anderes Fenster festlegen. Der Compiler nutzt diese Information für die später automatisch erzeugte Methode *InitializeComponent:*

```
[DebuggerNonUserCode]
public void InitializeComponent()
{
    base.StartupUri = new Uri("Window1.xaml", UriKind.Relative);
}
```

HINWEIS: Sie können diesen Wert auch erst zur Laufzeit ändern, überschreiben Sie dazu beispielsweise die *OnStartup*-Methode.

Beispiel 1.40: Zur Laufzeit ein Startformular festlegen *(App.xaml.cs)*

C#

```
...
    public partial class App : Application
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            if (e.Args.Count() == 0)
                this.StartupUri = new Uri("Window2.xaml", UriKind.Relative);
            else
                this.StartupUri = new Uri("Window1.xaml", UriKind.Relative);
        }
    }
```

Beispiel 1.41: Alternative zum Überschreiben der *OnStartup*-Methode

XAML

Alternativ können Sie auch das entsprechende Ereignis *Startup* verwenden. Fügen Sie dazu in der Datei *App.xaml* einen entsprechenden Tag hinzu:

```
<Application x:Class="WpfApplication2.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml" Startup="Application_Startup">
...
    <Application.Resources>
    </Application.Resources>
</Application>
```

C#

Anschließend editieren Sie die Datei *App.xaml.cs*:

```
...
namespace WpfApplication2
{
    public partial class App : Application
    {
```

```
        private void Application_Startup(object sender, StartupEventArgs e)
        {
 ...
        }
    }
}
```

HINWEIS: Sie können in diesem Ereignis bzw. in der überschriebenen Methode auch weitere Fenster erzeugen und anzeigen, diese erscheinen zeitgleich mit dem Hauptformular, das per *App.xaml* definiert wurde.

1.3.3 Kommandozeilenparameter verarbeiten

Auch wenn die Verwendung von Kommandozeilenparametern immer mehr aus der Mode kommt, ist es in dem einen oder anderen Fall doch erforderlich.

In der *OnStartUp*-Methode bzw. im *Startup*-Ereignis wird ein Parameter vom Typ *StartupEventArgs* übergeben. Über dessen Eigenschaft *Args* können Sie auf die Liste der übergebenen Parameter zugreifen.

Beispiel 1.42: Überschreiben der Methode *OnStartUp*, um die Kommandozeilenparameter auszuwerten *(App.xaml.cs)* und anzuzeigen

C#

```
...
    public partial class App : Application
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            if (e.Args.Count() == 0)
            {
                MessageBox.Show("Das Programm muss mit Parametern gestartet
werden!");
                this.Shutdown();
            }
            foreach (string arg in e.Args)
                MessageBox.Show(arg);
        }
    }
```

1.3.4 Die Anwendung beenden

Eigentlich eine dumme Aufgabenstellung, werden die meisten denken, wird doch die Anwendung standardmäßig geschlossen, wenn das letzte Fenster vom Anwender geschlossen wird. Dies ist zunächst korrekt, doch dieses Verhalten ist nicht immer gewünscht, und es soll auch Fälle geben, wo die Anwendung aus sich heraus geschlossen wird.

Die Art, wie eine Anwendung beendet wird, beeinflussen Sie über die Eigenschaft *ShutdownMode* des *App*-Objekts. Neben dem Standardwert *OnLastWindowClose* (wenn kein offenes Fenster mehr vorhanden ist) können Sie auch *OnMainWindowClose* (beim Schließen des Startfensters) oder *OnExplicitShutdown* wählen.

Die letzte Variante erwartet den expliziten Aufruf der *Shutdown*-Methode des *Application*-Objekts.

Beispiel 1.43: Explizites Beenden der Anwendung per Schaltfläche

C#

```
private void button1_Click(object sender, RoutedEventArgs e)
{
   App.Current.Shutdown();
}
```

1.3.5 Auswerten von Anwendungsereignissen

Für den Programmierer hält die *App*-Klasse noch einige interessante Ereignisse bereit, mit denen er auf bestimmt Anwendungszustände reagieren kann.

Ereignis	Beschreibung
Activated	Die Applikation wird zur Vordergrundanwendung.
Deactivated	Die Applikation ist nicht mehr die Vordergrundanwendung.
DispatcherUnhandledException	Ein nicht behandelter Fehler ist aufgetreten.
Exit	Das Anwendungsende ist ausgelöst.
SessionEnding	Windows wird beendet (Logout oder Shutdown). Sie können den Grund über den Parameter *e.ReasonSessionEnding* ermitteln. Mit *e.Cancel* können Sie versuchen, den Shutdown aufzuhalten, dies ist jedoch nicht in allen Fällen möglich.
Startup	Die Anwendung wurde gestartet.

Beispiel 1.44: Eine zentrale Fehlerbehandlung implementieren

C#

In der Datei *App.xaml.cs* erzeugen Sie einen neuen Ereignishandler für *DispatcherUnhandledException*:

```
...
    public partial class App : Application
    {
        private void Fehlerbehandlung(object sender,
DispatcherUnhandledExceptionEventArgs e)
        {
            MessageBox.Show("Mal wieder ein Fehler: " + e.Exception.Message);
            e.Handled = true;
```

```
        }
    }
...
```

XAML

In der Datei *App.xaml* fügen Sie das Attribut *DispatcherUnhandledException* hinzu, um Ereignis und Ereignismethode miteinander zu verknüpfen:

```
<Application x:Class="WpfApplication2.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml"
    DispatcherUnhandledException="Fehlerbehandlung" >
...
```

Tritt jetzt in Ihrem Programm ein unbehandelter Fehler auf, wird obige Ereignisprozedur ausgeführt und das Programm anschließend fortgesetzt.

1.4 Die Window-Klasse

Die eigentliche „Spielwiese“ des Programmierers ist sicherlich das einzelne Fenster innerhalb der Anwendung. In WPF-Anwendungen handelt es sich dabei um Instanzen der Klasse *Window*, dem Pendant zu *Form* von Windows Forms-Anwendungen.

Bevor wir jetzt der Versuchung erliegen, alle Eigenschaften, Methoden und Ereignisse detailliert aufzulisten, wollen wir uns lieber einige spezifische Aufgabenstellungen herauspicken, vieles kennen Sie ja bereits von den Windows Forms.

1.4.1 Position und Größe festlegen

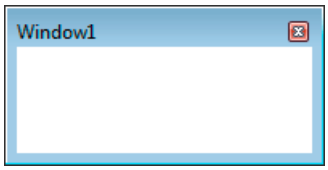
Haben Sie per Visual Studio ein neues Window erzeugt, sind bereits die Attribute für Breite und Höhe vorhanden (*Width, Height*) und müssen nur noch angepasst werden.

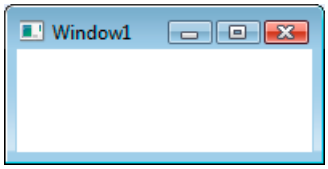
Die Position des Fensters können Sie zum einen mit dem Attribut *WindowStartupLocation* festlegen (*CenterOwner*, *CenterScreen, Manual*), zum anderen können Sie *Top* oder *Left* für die Ausrichtung verwenden.

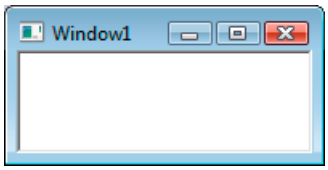
Für die Position in der Liste der angezeigten Windows ist die Eigenschaft *Topmost* verantwortlich.

1.4.2 Rahmen und Beschriftung

Die Art des Rahmens beeinflussen Sie mit dem Attribut *WindowStyle (None, ToolWindow*, *SingleBorderWindow, ThreeDBorderWindow*):







Die Beschriftung des Fensters lässt sich mit dem Attribut Title festlegen:

```
<Window x:Class="WpfApplication3.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="100" Width="200">
```

1.4.3 Das Fenster-Icon ändern

Über das Attribut *Icon* lässt sich dem *Window* auch ein neues Icon (linke obere Ecke bzw. Taskleistenansicht) zuordnen.

Beispiel 1.45: Das Fenster-Icon festlegen

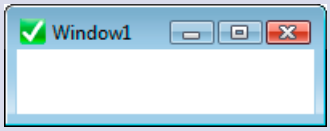
XAML

Ziehen Sie per Drag&Drop eine Icon-Datei in Ihr Projekt und legen Sie die *Icon*-Eigenschaft des *Window* auf deren Namen fest:

```
<Window x:Class="WpfApplication3.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="100" Width="200" Icon="Sign.ico">
...
```

Ergebnis

Öffnen Sie jetzt das Fenster, wird das neue Icon angezeigt:



HINWEIS: Es ist besser, wenn Sie derartige Ressourcen in einem Unterverzeichnis Ihres Projekts speichern, so geht der Überblick nicht verloren. In diesem Fall müssen Sie für *Icon* auch den relativen Pfad angeben (z. B. „*\Images\Sign.ico*“).

1.4.4 Anzeige weiterer Fenster

WPF-Windows zeigen Sie, wie auch die Windows Forms, nach dem Instanziieren mit den Methoden *Show* oder *ShowDialog* (modal) an.

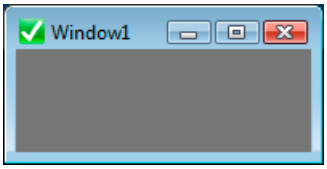
Beispiel 1.46: Aufruf eines zweiten Fensters

C#

```
...
        private void button1_Click(object sender, RoutedEventArgs e)
        {
            Window2 w2 = new Window2();
            w2.ShowDialog();
...
        }
```

1.4.5 Transparenz

Legen Sie die Eigenschaft *Opacity* auf den gewünschten Wert der Transparenz (0 ... 1) fest, werden Sie sicher zunächst enttäuscht sein, denn es wird lediglich eine Graustufe angezeigt:



Der Grund: dieses Attribut wird nur im Zusammenhang mit dem Attribut *AllowsTransparency* berücksichtigt.

HINWEIS: Zusätzlich muss der Rahmentyp (*WindowStyle*) auf *None* festgelegt sein.

Beispiel 1.47: Teiltransparentes Window

XAML

```
<Window x:Class="WpfApplication3.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="100" Width="200" Top="10" Left="10"
    Opacity="0.7" AllowsTransparency="True" WindowStyle="None">
```

Ergebnis

Das erzeugte Fenster über dem Desktop:

1.4.6 Abstand zum Inhalt festlegen

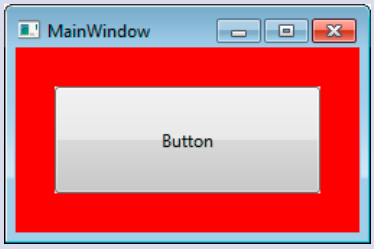
Über die Eigenschaft *BorderThickness* legen Sie fest, wie groß der Rahmen um den Clientbereich des Formulars ist. Die Rahmenfarbe legen Sie mit *BorderBrush* fest.

Beispiel 1.48: Verwendung von *BorderThickness*

XAML

```
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525" Loaded="Window_Loaded"
        BorderThickness="27"  BorderBrush="Red">
    <Grid>
        <Button Content="Button" Name="button1"  />
    </Grid>
</Window>
```

Ergebnis



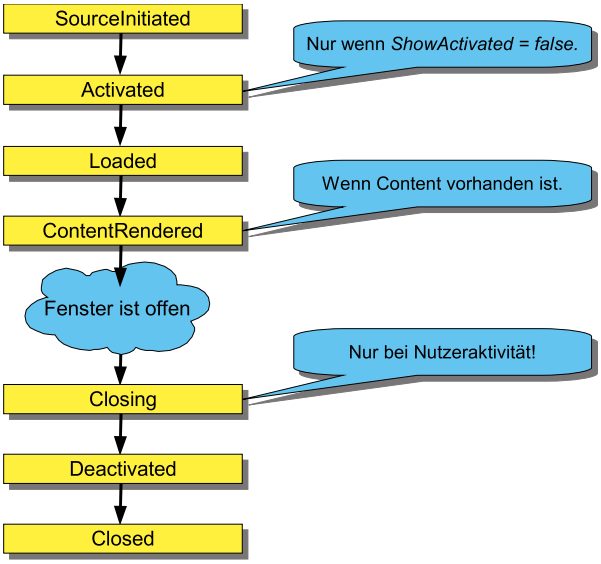
1.4.7 Fenster ohne Fokus anzeigen

Nicht in jedem Fall möchten Sie mit dem Einblenden eines neuen Fensters diesem auch den Eingabefokus zuweisen. Hier sei nur an Tool-Windows oder andere Statusfenster erinnert, die ohne Useraktivität auskommen.

Bevor Sie sich jetzt in endlose Programmierorgien stürzen, sollten Sie sich besser mit der *Window*-Eigenschaft *ShowActivated* vertraut machen. Setzen Sie diese auf *true*, wird das Fenster zwar geöffnet, aber es erhält nicht den Eingabefokus. Sie erkennen dies an der Rahmenfarbe, nicht aktivierte Fenster sind etwas heller.

1.4.8 Ereignisfolge bei Fenstern

Wie auch bei den Windows Forms möchten Sie als Programmierer auf diverse Ereignisse im „Leben“ eines Windows reagieren können. Die WPF-Entwickler haben auch hier nicht gespart, für fast jede denkbare Situation steht ein Ereignis zur Verfügung:



HINWEIS: Weitere Informationen zum Ereignismodell von WPF finden Sie in Abschnitt 3.3.

1.4.9 Ein paar Worte zur Schriftdarstellung

Nachdem die ersten WPF-Versionen noch mit einer rudimentären Textausgabe aufgewartet haben[7], hatten die Entwickler wohl ein Einsehen und spendierten ab Version 4.0 eine verbesserte Textausgabe, die vor allem bei kleinen Schriften nicht gleich zu Kopfschmerzen führt.

Ursache des Übels war der Algorithmus für das Rendern der Schriftarten. Dieser war zwar aus Programmierersicht akkurat (geräteneutral), auf TFT-Bildschirmen mit ihrer beschränkten Auflösung führte diese Darstellung jedoch zu fast unlesbaren Ergebnissen.

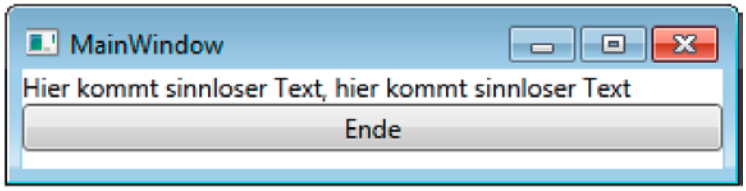
Über die neue Eigenschaft *TextOptions* bietet sich nun die Möglichkeit, die Darstellung von Schriftarten in Ihren Formularen zu optimieren. Zwei Optionen sind in diesem Zusammenhang von Bedeutung:

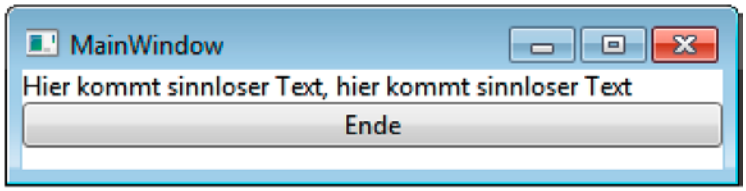
- *TextOptions.TextFormattingMode* und
- *TextOptions.TextRenderingMode*

HINWEIS: Beide Optionen können Sie für das gesamte Formular (wie im Weiteren beschrieben) oder auch nur für einzelne Controls zuweisen.

TextOptions.TextFormattingMode

Bevor wir lange um den heißen Brei herumreden, hier ein Beispiel für die Auswirkung von *TextOptions.TextFormattingMode* auf die Darstellung (Abbildung vergrößert):

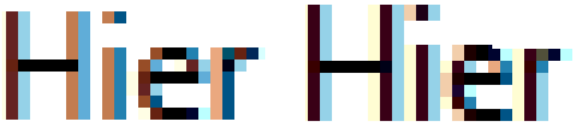




[7] Für einige Entwickler war die schlechte und verschwommene Textausgabe der alten Version **das** Killerargument gegen WPF.

Die obere Abbildung zeigt die Standarddarstellung von Schriften („Ideal“), bei der unteren ist die Eigenschaft *Options.TextFormattingMode* auf „Display“ gesetzt.

Beachten Sie bei der Darstellung insbesondere die senkrechten Linien (bei „H“, „i“ etc.). In der unteren Abbildung werden die teils grausigen Antialiasing-Artefakte drastisch entschärft, dies allerdings zu Lasten der Genauigkeit (siehe folgende Vergrößerung: links „Ideal“, rechts „Display“):



Das folgende Beispiel zeigt, wie Sie die Option für das gesamte Formular setzen:

Beispiel 1.49: *TextOptions.TextFormattingMode* setzen

XAML

```
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="84" Width="340" Loaded="Window_Loaded"
        TextOptions.TextFormattingMode="Display">
    <StackPanel>
        <TextBlock Text="Hier kommt sinnloser Text, hier kommt sinnloser Text"/>
        <Button>Ende</Button>
    </StackPanel>
```

Doch wann sollten Sie welche Option verwenden? Eine allgemeingültige Antwort lässt sich kaum geben, entweder Sie experimentieren etwas (unterschiedliche Bildschirme), oder Sie halten sich an folgende Pauschalaussagen:

- Verwenden Sie „Display“ grundsätzlich bei kleinen Texten (< 15pt).
- Verwenden Sie die Standardeinstellung bzw. „Ideal“ bei großen Texten, bei Transformationen, Zoom etc.

TextOptions.TextRenderingMode

Nach der bisherigen Kritik an der Textdarstellung wollten es die Entwickler wohl besonders gut machen und haben gleich noch eine weitere „Stellschraube“ spendiert, die Sie für die Textoptimierung einsetzen können. Die Rede ist von *TextOptions.TextRenderingMode*, für das die drei Optionen *ClearType*, *Grayscale* und *Aliased* zur Verfügung stehen.

HINWEIS: Alle drei Optionen sind nur von Bedeutung, wenn die Eigenschaft *TextOptions.TextFormattingMode* auf *Display* gesetzt ist.

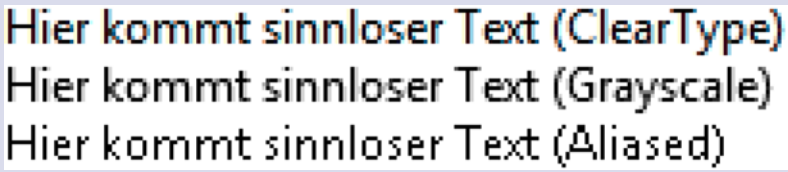
Ein Beispiel soll Ihnen die Unterschiede demonstrieren.

Beispiel 1.50: Verwendung von *TextOptions.TextRenderingMode*

XAML

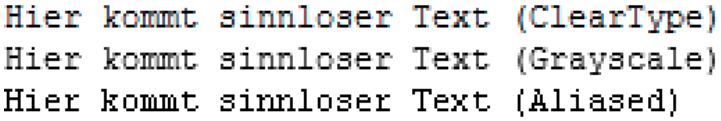
```
Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="84" Width="340" Loaded="Window_Loaded"
        TextOptions.TextFormattingMode="Display">
    <StackPanel>
        <TextBlock>
            Hier kommt sinnloser Text (ClearType)
        </TextBlock>
        <TextBlock TextOptions.TextRenderingMode="Grayscale">
            Hier kommt sinnloser Text (Grayscale)
        </TextBlock>
        <TextBlock TextOptions.TextRenderingMode="Aliased">
            Hier kommt sinnloser Text (Aliased)
        </TextBlock>
    </StackPanel>
```

Ergebnis



Bei der Standard-Option „ClearType“ wird Antialiasing mit diversen Farbabstufungen (Grau-, Brauntöne) realisiert, „Grayscale“ nutzt hingegen lediglich Graustufen für die Kantenglättung. „Aliased“ kommt ganz ohne Kantenglättung aus.

Wer jetzt auf die Idee kommt, „Cleartype“ sei die Ideallösung für alle Einsatzfälle, sollte sich einmal die folgende Darstellung der Schriftart „Courier New“ ansehen:



In diesem Fall dürfte der „klare“ Sieger wohl eindeutig feststehen.

1.4.10 Ein paar Worte zur Darstellung von Controls

Nachdem wir uns bereits einige Seiten über die Schriftdarstellung ausgelassen haben, sollten wir auch noch kurz auf die Antialiasing-Effekte bei der Darstellung von Controls durch die Layoutengine eingehen. Auch hier gilt, dass die Auflösung von Bildschirmen begrenzt ist, und so bleibt der Layoutengine manchmal nichts anderes übrig, als eine berechnete Position/Breite durch einen Antialiasing-Effekt anzudeuten. So weit so gut, aber nicht jeder Anwender (insbesondere bei TFT-Bildschirmen) honoriert es, wenn Kanten nicht mehr klar, sondern verschwommen dargestellt werden, wie es das folgende Beispiel zeigt:

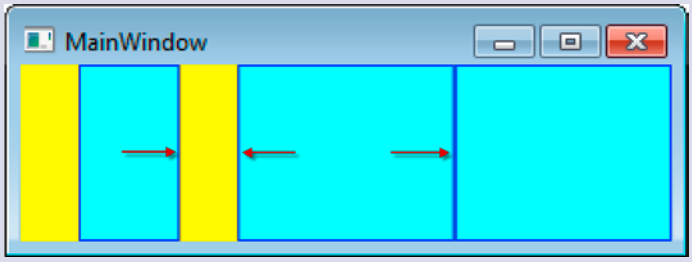
Beispiel 1.51: Antialiasing-Effekte bei Controls

XAML

```
...
    <Grid Background="Yellow">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" /><ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Rectangle Stroke="Blue" StrokeThickness="1" Fill="Aqua" Width="50.5"/>
        <Rectangle Stroke="Blue" StrokeThickness="1" Fill="Aqua" Grid.Column="1"/>
        <Rectangle Stroke="Blue" StrokeThickness="1" Fill="Aqua" Grid.Column="2"/>
    </Grid>
...
```

Ergebnis

Beachten Sie die verschwommenen senkrechten Kanten der Rechtecke:

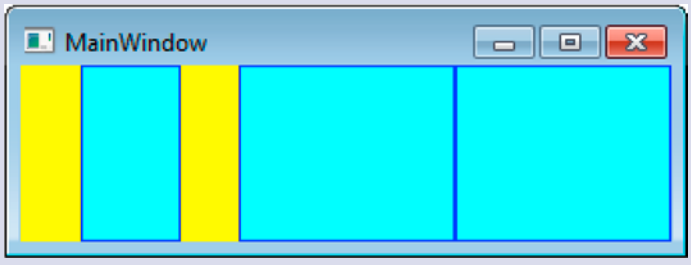


Beispiel 1.52: Ausschalten der Effekte mit *UseLayoutRounding*

XAML

```
...
    <Grid UseLayoutRounding="True" Background="Yellow">
...
```

Ergebnis



Wie Sie dem zweiten Beispiel entnehmen können, führt das Setzen von *UseLayoutRounding* auf *True* dazu, dass kritische Kanten so verschoben werden, dass sie mit den Bildschirmpixeln übereinstimmen. Dies geht zwar etwas zu Lasten der Genauigkeit, im Interesse einer klaren Darstellung können wir hier aber sicher „ein Auge zudrücken“.

1.4.11 Wird mein Fenster komplett mit WPF gerendert?

Nein, Rahmen und Kopfzeile werden nach wie vor durch GDI-Anweisungen generiert, was auch erklärt, warum viele WPF-Features (Styles, Transparenz, Rotation ...) nicht auf das eigentliche Fenster angewendet werden können.

Möchten Sie dennoch den Fenster-Kopf oder -Rand mit WPF-Mitteln gestalten, bleibt Ihnen nichts anderes übrig, als ein rahmenloses Window zu erstellen und die Buttons bzw. die Reaktion auf die Mausereignisse mit WPF-Controls nachzustellen.

Damit haben wir uns mit den notwendigsten Grundkenntnissen über die WPF-Anwendung versorgt.

HINWEIS: In den weiteren Kapiteln wollen wir uns zunächst den wichtigsten Controls zuwenden (Kapitel 2), bevor wir zu speziellen Themen wie Ereignisbehandlung Styles etc. kommen (Kapitel 3). Das Spezialthema „Datenbindung“ finden Sie im Kapitel 4, Kapitel 5 widmet sich abschließend der Druckausgabe in WPF.

2 Übersicht WPF-Controls

Bei den WPF-Controls handelt es sich, ähnlich wie bei den Windows Forms-Steuerelementen, um ein recht komplexes Thema. Angesichts der erdrückenden Vielfalt unterschiedlichster Controls und deren Features können wir auch hier keinen Anspruch auf Vollständigkeit erheben.

2.1 Allgemeingültige Eigenschaften

Da die meisten Controls direkt oder indirekt von der Klasse *Control* (*System.Windows.Controls*) abgeleitet sind (und diese wiederum von *FrameworkElement* ...), verfügen sie auch über einige gemeinsame Eigenschaften, die wir an dieser Stelle kurz besprechen wollen.

Eigenschaft	Beschreibung
Background	die Füll-/Hintergrundfarbe, z. B. `<Button Background="Blue">`
BorderBrush	die Rahmenfarbe, z. B. `<Button BorderBrush="Blue">`
Cursor	der Mauszeiger für das Control, z. B. `<Button Cursor="Wait">`
FontFamily, *FontSize*, *FontStyle*, *FontWeight*	Schriftart, -größe, -schnitt (*Normal*, *Italic*, *Oblique*), -breite (*Light*, *Normal*, *UltraBold*), z. B. `<Button FontFamily="Arial" FontSize="12" FontStyle="Italic" FontWeight="UltraBold">`
Foreground	Vordergrundfarbe, z. B. `<Button Foreground="Coral">`
HorizontalAlignment, *VerticalAlignment*	Horizontale/vertikale Ausrichtung bezüglich des übergeordneten Elements, z. B. `<Button HorizontalAlignment="Left">`

(Fortsetzung nächste Seite)

(Fortsetzung)

Eigenschaft	Beschreibung
HorizontalContentAlignment, VerticalContentAlignment	Horizontale/vertikale Ausrichtung der Inhalte, z. B. `<Button HorizontalContentAlignment="Left">`
Height/Width	Höhe und Breite des Controls, siehe auch Abschnitt 1.3.1
Margin	Außenabstände, siehe auch Abschnitt 1.3.2
MaxHeight, MaxWidth, MinHeight, MinWidth	maximale bzw. minimale Abmessungen des Controls
Name	Name der Komponente, z. B. `<Button Name="button1">`
Opacity	Transparenz (ein Wert zwischen 0 und 1), z. B. `<Button Opacity="0.5">`
Padding	Innenabstände, siehe auch Abschnitt 1.3.2
Parent	das übergeordnete Element
Resources	Verweis auf Ressourcen, dazu mehr ab Abschnitt 3.2
Style	Verweis auf einen Style, dazu mehr ab Abschnitt 3.5
TabIndex	Index in der Tabulatorreihenfolge
Tag	Frei definierbare Eigenschaft vom Typ *object*, hier können Sie eigene Informationen speichern
ToolTip	Die beliebten kleinen Fähnchen, die dem Endanwender „unnötige“ Informationen aufdrängen
Visibility	Die Sichtbarkeit des Controls (*Collapsed*, *Hidden*, *Visible*), Achtung: Bei *Hidden* beansprucht das Control nach wie vor seinen Platz, ist nur nicht sichtbar. Mit *Collapsed* verschwindet auch die „Leerstelle“.

Ergänzend wollen wir noch auf zwei Eigenschaften eingehen, die über einen anderen Namespace in den XAML-Code importiert werden, aber häufig benötigt werden:

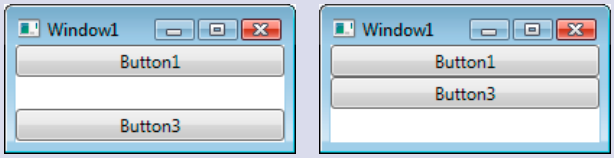
Eigenschaft	Beschreibung
x:Name	Ermöglicht es, Objekten die nicht von *FrameworkElement* abgeleitet sind, einen Bezeichner zu geben, um zur Laufzeit auf dieses Objekt zuzugreifen.
x:FieldModifier	Mit dem Zugriffsmodifizierer steuern Sie die Sichtbarkeit des Objekts außerhalb der Klasse (*public* oder *internal*).

Beispiel 2.1: Der Unterschied zwischen *Hidden* (links) und *Collapsed* (rechts) für *Button2*

XAML

```
<StackPanel>
  <Button>Button1</Button>
  <Button Visibility="Collapsed">Button2</Button>
  <Button>Button3</Button>
</StackPanel>
```

Ergebnis



HINWEIS: Umsteiger werden sicher darauf hereinfallen: Die Höhe und Breite des Controls rufen Sie nicht über *Height* bzw. *Width,* sondern über *ActualHeight* und *ActualWidth* ab. Der Grund ist in der Verwendung der Layouts zu finden, d. h., die übergeordneten Elemente bestimmen die Abmessungen des Controls. Eine Vorgabe für Breite und Höhe wird jedoch nach wie vor über *Width* und *Height* realisiert.

2.2 Label

Dieses innovative Control kennen Sie sicher noch zur Genüge aus Ihrer „Vor"-WPF-Ära, der Verwendungszweck ist zum einen das Beschriften von Controls (z. B. *TextBox*), zum anderen lassen sich so zum Beispiel Access-Keys einer *TextBox* zuordnen. Die Zuordnung zwischen *Label* und Control erfolgt mit dem Attribut *Target*, wie es das folgende Beispiel zeigt.

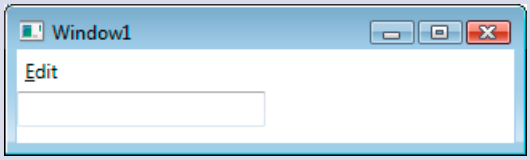
Beispiel 2.2: Festlegen eines Access-Key (Alt+E) und Zuordnen der *TextBox1* (Ziel des Eingabefokus)

XAML

```
...
   <StackPanel Grid.Column="0" Grid.Row="0" >
     <Label Target="{Binding ElementName=textBox1}" Name="label1" >
     _Edit
     </Label>
     <TextBox Name="textBox1" />
   </StackPanel>
...
```

Ergebnis

Die Laufzeitansicht (nach dem Drücken der Alt-Taste):

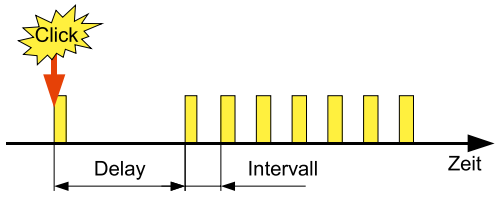


2.3 Button, RepeatButton, ToggleButton

Der Verwendungszweck von *Button*, *RepeatButton* und *ToggleButton* als Schaltfläche mit *Click*-Ereignis dürfte schnell klar sein. Doch es gibt einige Unterschiede, wie die folgende Tabelle zeigt:

Element	Beschreibung
Button	Ein einzelnes *Click*-Ereignis nach Betätigung.
RepeatButton	Mehrfache *Click*-Ereignisse bei Betätigung, die Frequenz wird mit der Eigenschaft *Intervall* (Millisekunden) bestimmt, der Abstand zwischen dem ersten Klick (resultiert aus dem Niederdrücken) und der Wiederholung kann mit *Delay* (Millisekunden) festgelegt werden (siehe folgende Abbildung).
ToggleButton	Einzelnes *Click*-Ereignis nach Betätigung, die Schaltfläche schaltet jedoch mit jedem Klick zwischen den Zuständen „gedrückt“ und „nicht gedrückt“ um. Zur Auswertung des Status stehen die beiden Ereignisse *Checked* und *UnChecked* sowie die Eigenschaft *IsChecked* zur Verfügung.

Das Verhalten des *RepeatButton*:



Beispiel 2.3: Einige Schaltflächendefinitionen

XAML

```
<StackPanel>
  <Button Margin="6" Click="Button_Click">Button</Button>
  RepeatButton Delay="2000" Interval="500" Click="RepeatButton_Click">RepeatButton
  </RepeatButton>
  <ToggleButton  Name="tb" Click="RepeatButton_Click">ToggleButton</ToggleButton>
</StackPanel>
```

2.3.1 Schaltflächen für modale Dialoge

Eine Standardaufgabe für den Programmierer ist häufig das Erstellen von modalen Dialogen, deren Ergebnis (OK, Abbruch) im übergeordneten Fenster ausgewertet werden muss. Wird das Window mit *Alt+F4*, der Schließen-Schaltfläche oder per Systemmenü geschlossen, wird die *DialogResult*-Eigenschaft des betreffenden Windows auf *False* gesetzt. Den Wert *True* müssen Sie z. B. über die OK-Schaltfläche explizit setzen.

Die standardmäßige Zuordnung der beiden Tasten (OK/Abbruch) können Sie mit den Eigenschaften *IsDefault* (Verknüpfung mit Enter-Taste) und *IsCancel* (Verknüpfung mit *ESC*-Taste) realisieren. *IsCancel=True* führt zum automatischen Schließen des Fensters.

Beispiel 2.4: Aufruf eines modalen Dialogs und Auswertung von OK/Abbruch

XAML

Das modale Window:

```
<Window x:Class="WpfApplication3.Window2"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window2" Height="300" Width="300">
    <Grid>
     <Grid.RowDefinitions>
       <RowDefinition Height="216*" />
       <RowDefinition Height="46*" />
     </Grid.RowDefinitions>
    <Button IsDefault="True" Grid.Row="1" Margin="89,6,104,8" Name="button1"
      Click="button1_Click">Ok</Button>
     <Button IsCancel="True" Grid.Row="1" HorizontalAlignment="Right"
Margin="0,6,14,10"
      Name="button2" Width="79">Abbruch</Button>
    </Grid>
</Window>
```

C#

Die Ereignisprozedur für den OK-Button:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    this.DialogResult = true;
    this.Close();
}
```

Der Aufruf des modalen Window und Auswertung:

```
Window2 w2 = new Window2();
w2.ShowDialog();
if ((bool)w2.DialogResult)
    MessageBox.Show("OK");
else
    MessageBox.Show("Abbruch");
```

2.3.2 Schaltflächen mit Grafik

Wer jetzt verzweifelt nach einer Image-Eigenschaft Ausschau hält, dürfte enttäuscht sein. Keine der Schaltflächen verfügt darüber, was jedoch kein Problem ist, da Sie in WPF-Anwendungen den Content des Controls selbst bestimmen können (*ContentControl*).

Doch wie müssen wir vorgehen? Ausschlaggebend für die Zusammenstellung unserer Schaltfläche ist das zukünftige Layout (Grafik oben, Texte unten oder anders herum). Mit

den im Abschnitt 1.3 vorgestellten Layout-Controls können Sie zunächst ein geeignetes Layout für den Inhalt der Schaltfläche entwerfen und dann die eigentlichen Image- und Text-Elemente[1] anordnen.

Beispiel 2.5: Schaltfläche mit Text und Grafik

C#

```
    <Button Click="Button_Click">
```

Ein *StackPanel* bestimmt das innere Layout der Schaltfläche:

```
      <StackPanel Orientation="Horizontal" Margin="10">
```

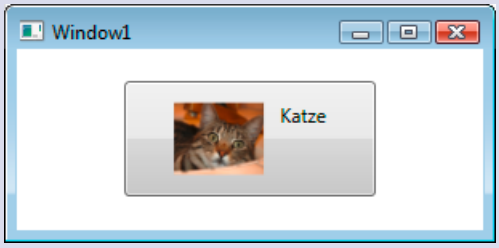
Die Grafik festlegen (ziehen Sie vorher einfach eine Grafik per Drag&Drop in Ihr Projekt):

```
        <Image Source="Annie in the Sink.jpg" Width="56" Height="46"
 Margin="0,0,10,0"/>
```

Der Beschriftungstext (hier könnten Sie auch mit Textformatierungen arbeiten):

```
        <TextBlock VerticalAlignment="Center">Katze</TextBlock>
      </StackPanel>
    </Button>
```

Ergebnis



HINWEIS: Statt dieser Lösung können Sie natürlich auch die *Button*-Klasse ableiten, oder Sie nutzen die Möglichkeit eines UserControls. Beide Varianten sollen aber nicht im Mittelpunkt dieses Abschnitts stehen.

[1] Statt einer Grafik könnten Sie auch gleich ein komplettes Video in der Schaltfläche anzeigen. Ob dies sinnvoll ist, ist eine andere Frage.

2.4 TextBox, PasswordBox

Beide Controls dürften Ihnen als wichtige Möglichkeiten für die Texteingabe sicher bekannt sein. An dieser Stelle wollen wir deshalb nur auf einige Eigenschaften zur Konfiguration dieser Eingabefelder eingehen.

2.4.1 TextBox

Die wichtigsten Eigenschaften auf einen Blick:

Eigenschaft	Beschreibung
Text	Der Inhalt der *TextBox*.
TextWrapping	Einzeilige (*NoWrap*) oder mehrzeilige (*Wrap, WrapWithOverFlow*) Darstellung. *Wrap* bricht bei jedem Zeichen um, *WrapWithOverFlow* nur an Leerzeichen.
AcceptReturn	Wenn *True,* erzeugt die *Enter*-Taste eine neue Zeile, andernfalls wird Enter ignoriert.
IsReadOnly	Schreibschutz ja/nein.
VerticalScrollBarVisibility, HorizontalScrollBarVisibility	Sichtbarkeit der Scrollbars vorgeben. Mögliche Werte: *Auto, Disabled, Hidden, Visible*
MaxHeight, MaxWidth, MinHeight, MinWidht	Maximale/minimale Abmessungen der *TextBox* vorgeben.
MaxLines, MinLines	Minimale/maximale Anzahl von angezeigten Zeilen in der *TextBox*. Nur sinnvoll mit *TextWrapping*.
MaxLength	Die maximale Zeichenzahl.
CaretIndex	Position des Eingabekursors.
SpellCheck.IsEnabled	Ein-/Ausschalten der integrierten Rechtschreibkorrektur (siehe Beispiel).

Beispiel 2.6: Setzen der *Text*-Eigenschaft per XAML

XAML

```
<TextBox Text="Meine Vorgabe">
</TextBox>
```

oder

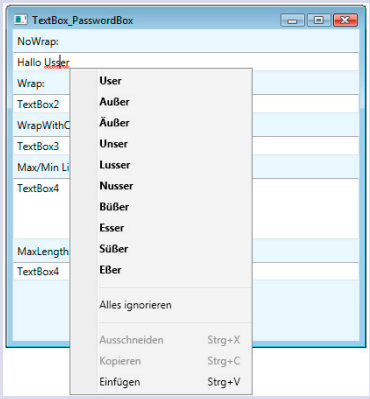
```
<TextBox>
  Meine Vorgabe
</TextBox>
```

Beispiel 2.7: Verwendung der Rechtschreibkorrektur

XAML

```
<TextBox SpellCheck.IsEnabled="True" Name="TextBox1" >TextBox1</TextBox>
```

Ergebnis



Zur Laufzeit wird jetzt, je nach aktueller Landeseinstellung, eine automatische Rechtschreibprüfung durchgeführt, Fehler werden rot unterstrichen und Alternativen per Kontextmenü angezeigt.

Und da Deutsch eine Sprache mit andauernden behördlichen Änderungen ist, findet sich auch eine Möglichkeit, die aktuelle Rechtschreibreform zu berücksichtigen. Verwenden Sie einfach die Eigenschaft *SpellingReform* und setzen Sie einen der folgenden Werte (*Prerefrom*, *Postrefrom*, *PreAndPostrefrom*)[2].

Wahrscheinlich haben Sie dieses Feature bis jetzt noch gar nicht vermisst: seit WPF 4 können Sie sowohl für den Eingabekursor als auch für die Kursorauswahl einen extra *Brush* verwenden, was Ihren gestalterischen Fähigkeiten natürlich jede Menge Raum gibt.

Beispiel 2.8: Texteingabe- und Auswahlkursor ändern

XAML

Hier zunächst der geänderte Eingabekursor:

```
<TextBox FontSize="20" CaretBrush="Green">
```

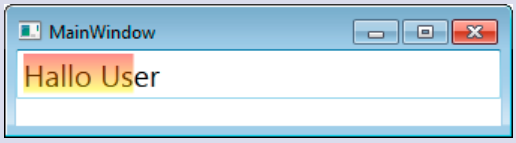
Und hier ein etwas komplexerer Auswahlkursor:

```
<TextBox.SelectionBrush>
    <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
        <GradientStop Color="Transparent" Offset="-.2" />
        <GradientStop Color="Red" />
        <GradientStop Color="Yellow" Offset=".9" />
        <GradientStop Color="Transparent" Offset="1.2" />
    </LinearGradientBrush>
```

[2] In .NET 5.x findet sich dann sicher auch ein *PostPostNextPreReform*

```
            </TextBox.SelectionBrush>
            Hallo User
        </TextBox>
```

Ergebnis



2.4.2 PasswordBox

Der Sinn dieses Controls ist die verdeckte Eingabe von Zeichenketten (Passwörtern). Für Sie als Programmierer sind die beiden Eigenschaften

- *PasswordChar* (das Maskierungszeichen) und
- *Password* (die eingegebene Zeichenkette)

interessant.

Beispiel 2.9: Verwendung *PasswordBox*

XAML

```
<PasswordBox Name="Pwd1" MaxLength="6" Password="geheim" PasswordChar="?"
             KeyUp="PasswordBox_KeyUp"/>
```

Ergebnis

Die Ereignis-Routine:

```
private void PasswordBox_KeyUp(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Enter)
        MessageBox.Show(Pwd1.Password);
}
```

2.5 CheckBox

Geht es um die Auswahl bzw. Darstellung boolescher Werte (*True/False),* bietet sich neben dem *ToggleButton* die gute alte *CheckBox* an.

Die Beschriftung des Controls bestimmen Sie über die *Content*-Eigenschaft, hier ist stattdessen z. B. auch ein *Image* möglich, was optisch sicher viel hergibt (siehe Beispiel am Abschnittsende).

Den aktuellen Status können Sie über die Eigenschaft *IsChecked* bestimmen.

HINWEIS: Achtung: Hier sind drei Zustände (*Null, True, False*) möglich, wenn Sie die Eigenschaft *IsThreeState* auf *True* setzen (unbestimmte Angabe).

Die Auswertung einer Änderung kann in den Ereignissen *Checked* bzw. *UnChecked* erfolgen, besser ist jedoch *Click*, da Sie hier beide Zustände auswerten können.

Beispiel 2.10: Verwendung der *CheckBox*

XAML

```
  <StackPanel Margin="5">
```

Variante 1:

```
    <CheckBox Checked="CheckBox_Checked" Unchecked="CheckBox_Unchecked"
 Click="CheckBox_Click"
              Name="checkBox1" Foreground="Blue">Ich kann lesen</CheckBox>
```

Variante 2:

```
    <CheckBox Name="checkBox2" Foreground="Green">Ich kann schreiben</CheckBox>
```

Hier mit drei Zuständen:

```
    <CheckBox Name="checkBox3" IsThreeState="True" IsChecked="">Schwanger</CheckBox>
```

CheckBox mit Grafik statt Text:

```
    <CheckBox Name="checkBox4" Foreground="Green">
      <Image Source="Images\rudi.gif" Width="50" Height="50"/>
    </CheckBox>
```

Natürlich könnte es auch jedes andere Control sein, das Sie als Content der *CheckBox* festlegen, aber einen Sinn sollte es schon machen.

```
  </StackPanel>
```

C#

Die Ereignisroutinen:

```
        private void CheckBox_Checked(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("True");
        }

        private void CheckBox_Unchecked(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("False");
        }
```

Nur *True/False* auswerten:

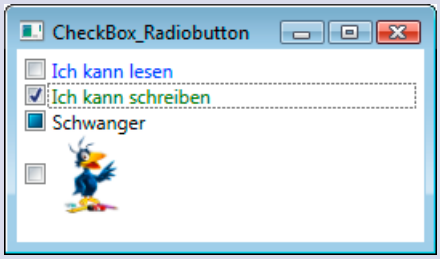
```
        private void CheckBox_Click(object sender, RoutedEventArgs e)
        {
```

```
            if ((bool)checkBox1.IsChecked)
                MessageBox.Show("True");
            else
                MessageBox.Show("False");
        }
```

Unbestimmten Zustand auswerten:

```
        private void checkBox3_Click(object sender, RoutedEventArgs e)
        {
            if (checkBox3.IsChecked == null) MessageBox.Show("Etwas Schwanger!");
        }
```

Ergebnis



Beachten Sie die obige Option „Schwanger“: angezeigt wird der unbestimmte Zustand, d.h. in diesem Fall „etwas schwanger“.

2.6 RadioButton

Möchten Sie mehr als nur eine *True/False*-Auswahl anbieten (siehe *CheckBox),* können Sie eine Reihe von Radiobuttons verwenden, bei der jeweils nur ein *RadioButton* markiert sein kann.

Um die einzelnen Radiobuttons miteinander zu verbinden, steht Ihnen die Eigenschaft *GroupName* zur Verfügung. Alle Controls mit übereinstimmenden Namen werden zu einer Gruppe zusammengefasst, nur ein Element der Gruppe kann markiert sein (*IsChecked*).

Beispiel 2.11: Verwendung *RadioButton*

XAML

```
  <StackPanel Margin="5">
    <Label Content="----------------RadioButton ----------------------"
Margin="4,10"/>
```

Die einfachste Variante:

```
    <RadioButton Name="RB1">Option1</RadioButton>
```

Mit gesetzter *IsChecked*-Option:

```
<RadioButton Name="RB2" IsChecked="True">Option2</RadioButton>
```

Als Content ist auch ein Image oder jedes andere Control zulässig:

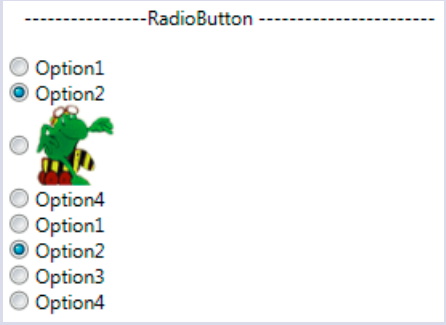
```
<RadioButton Name="RB3">
     <Image Source="Images\frosch.gif" Width="50" Height="50"/>
</RadioButton>
<RadioButton Name="RB4">Option4</RadioButton>
```

Und hier beginnt eine neue Gruppe, da anderer Parent:

```
<StackPanel>
  <RadioButton Name="RB5">Option1</RadioButton>
  <RadioButton Name="RB6" IsChecked="True">Option2</RadioButton>
  <RadioButton Name="RB7">Option3</RadioButton>
  <RadioButton Name="RB8">Option4</RadioButton>
</StackPanel>
</StackPanel>
```

Ergebnis

Wie Sie sehen können, sind bei diesem Beispiel zwei *RadioButton*s markiert:



HINWEIS: Geben Sie keinen *GroupName* an, werden alle *RadioButton*s mit gemeinsamem Parent (z. B. *StackPanel*) zu einer Gruppe zusammengefasst.

2.7 ListBox, ComboBox

Für die Auswahl aus Listen mit mehreren Einträgen bieten sich statt der *RadioButton*s besser eine *ListBox* oder eine *ComboBox* an. Erstere gestattet die gleichzeitige Anzeige mehrerer Werte, die *ComboBox* stellt nur einen Wert aus der Liste dar.

2.7.1 ListBox

Jede *ListBox* kann mehrere Einträge, d. h. *ListBoxItems*, enthalten, von denen wiederum ein oder mehr Einträge ausgewählt sein können.

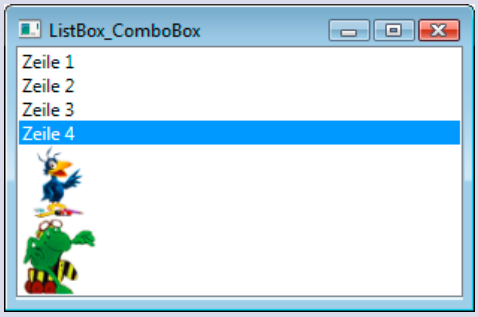
Sollen mehrere Einträge ausgewählt werden können, setzen Sie *SelectionMode* auf *Multiple* (Auswahl durch einfachen Klick) oder *Extended* (Auswahl durch Klick + Shift-Taste).

Beispiel 2.12: Definition einer *ListBox* mit 6 Einträgen inklusive zweier Grafiken

XAML

```
<ListBox SelectionMode="Extended">
  <ListBoxItem IsSelected="True">Zeile 1</ListBoxItem>
  <ListBoxItem>Zeile 2</ListBoxItem>
  <ListBoxItem>Zeile 3</ListBoxItem>
  <ListBoxItem>Zeile 4</ListBoxItem>
  <ListBoxItem>
     <Image Source="Images\rudi.gif" Width="50" Height="50"/>
</ListBoxItem>
<ListBoxItem>
     <Image Source="Images\frosch.gif" Width="50" Height="50"/>
  </ListBoxItem>
</ListBox>
```

Ergebnis



Alternativ können Sie die Beschriftung auch mit einer etwas kürzeren Schreibweise zuweisen:

```
<ListBoxItem Content="Zeile 4"/>
```

HINWEIS: Im Gegensatz zu den Windows Forms-Listboxen können Sie jeden Eintrag individuell (Farbe, Hintergrundfarbe, Schrift etc.) formatieren. Ob dies sinnvoll und der Übersicht dienlich ist, sei dahingestellt.

Wem diese Möglichkeiten nicht ausreichen, der kann auch andere Controls der *ListBox* als Einträge hinzufügen.

Beispiel 2.13: Controls in der *ListBox* (eine einfache „CheckListBox“)

XAML

```
<ListBox SelectionMode="Multiple" Height="75">
 <CheckBox Name="Item1">Zeile 1</CheckBox>
  <CheckBox Name="Item2">Zeile 2</CheckBox>
  <CheckBox Name="Item3">Zeile 3</CheckBox>
  <CheckBox Name="Item4">Zeile 4</CheckBox>
</ListBox>
<ListBox >
  <Image Source="Images\frosch.gif" Width="50" Height="50"/>
  <CheckBox Name="Item7">Zeile 2</CheckBox>
  <Image Source="Images\frosch.gif" Width="50" Height="50"/>
</ListBox>
```

Ergebnis

Die beiden erzeugten *ListBox*en:

Doch wie können Sie die aktuelle Markierung bzw. die Auswahl ermitteln? Die *ListBox* stellt zu diesem Zweck eine Reihe von Eigenschaften zur Verfügung, von denen *SelectedIndex* (die erste markierte Zeile, nullbasiert) und *SelectedItems* (Collection von *ListBoxItems*) am nützlichsten sind.

HINWEIS: Prüfen Sie vor der Verwendung gegebenenfalls, ob überhaupt ein Eintrag markiert wurde. In diesem Fall muss *SelectedIndex* einen Wert ungleich -1 aufweisen.

Beispiel 2.14: *ListBox* mit Einträgen füllen und später die markierten abfragen

XAML

XAML-Code zum Füllen der *ListBox:*

```
<ListBox Name="ListBox1" SelectionMode="Extended">
  <ListBoxItem>Zeile 1</ListBoxItem>
  <ListBoxItem>Zeile 2</ListBoxItem>
  <ListBoxItem>Zeile 3</ListBoxItem>
  <ListBoxItem>Zeile 4</ListBoxItem>
```

Eintrag mit Grafik:

```
    <ListBoxItem>
       <Image Source="Images\rudi.gif" Width="50" Height="50"/>
    </ListBoxItem>
    <ListBoxItem>
       <Image Source="Images\frosch.gif" Width="50" Height="50"/>
    </ListBoxItem>
  </ListBox>
```

C#

Auswerten per Ereigniscode:

```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            if (ListBox1.SelectedIndex != -1)
            {
                MessageBox.Show(ListBox1.SelectedValue.ToString());
                MessageBox.Show("Es ist mindestens ein Eintrag markiert!");
                MessageBox.Show("Zeile " + ListBox1.SelectedIndex.ToString() +
                                " ist markiert!");
                foreach (ListBoxItem item in ListBox1.SelectedItems)
                    MessageBox.Show(item.Content.ToString());
            }
        }
```

HINWEIS: Für normale Textzeilen wird obiger Code die sichtbare Beschriftung zurückgeben, für markierte Grafik-Einträge kommt der Text „System.Windows.Controls.Image“.

Beispiel 2.15: ListBoxeintrag zur Entwurfszeit markieren

XAML

```
...
   <ListBoxItem IsSelected="True">Four</ListBoxItem>
...
```

Natürlich müssen Sie nicht unbedingt auf XAML zurückgreifen, wenn Sie eine *ListBox* füllen wollen. Dies funktioniert genauso gut auch per Code.

Beispiel 2.16: *ListBox* zur Laufzeit füllen

C#

```
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            ListBox2.Items.Clear();             // Alles löschen
            CheckBox cb = new CheckBox();       // zur Probe mal eine CheckBox einfügen
            cb.Content = "Ein Test";
            ListBox2.Items.Add(cb);
            for (int i = 1; i < 50; i++)        // und jetzt noch fünfzig Zeilen
Text ...
                ListBox2.Items.Add("Zeile" + i.ToString());
        }
```

Ergebnis

HINWEIS: Möchten Sie die Einträge einzeln formatieren, verwenden Sie *ListBoxItems* beim Aufruf von *Add*.

2.7.2 ComboBox

Das Handling aus Sicht des Programmierers entspricht weitgehend der *ListBox*, mit dem wesentlichen Unterschied, dass der Anwender nur ein Element der Auswahlliste markieren kann. Dieses wird nachfolgend in der *ComboBox* angezeigt (Eigenschaft *Text).*

Neben der reinen Auswahl von vorgegebenen Werten können Sie auch neue Einträge zulassen. Dazu müssen Sie die Eigenschaft *IsEditable* auf *True* setzen. In diesem Fall erscheint eine Textbox, die der Nutzer nach Wunsch füllen kann.

Beispiel 2.17: Einfache *ComboBox*

XAML

```
<ComboBox Name="ComboBox1"  SelectionChanged="ComboBox1_SelectionChanged">
  <ComboBoxItem>Zeile 1</ComboBoxItem>
  <ComboBoxItem>Zeile 2</ComboBoxItem>
  <ComboBoxItem>Zeile 3</ComboBoxItem>
  <ComboBoxItem>Zeile 4</ComboBoxItem>
  <ComboBoxItem>Zeile 5</ComboBoxItem>
  <ComboBoxItem>
    <Image Source="Images\rudi.gif" Width="50" Height="50"/>
  </ComboBoxItem>
  <ComboBoxItem>
    <Image Source="Images\frosch.gif" Width="50" Height="50"/>
  </ComboBoxItem>
</ComboBox>
```

C#

Ereignisauswertung:

```
        private void ComboBox1_SelectionChanged(object sender,
SelectionChangedEventArgs e)
        {
            MessageBox.Show("Markierte Zeile: " +
ComboBox1.SelectedIndex.ToString());
        }
```

Ergebnis

Die *ComboBox* in Aktion:

HINWEIS: In obiger Ereignisprozedur ist die *Text*-Eigenschaft noch **nicht** auf den neuen Wert gesetzt, dies erfolgt (aus unerfindlichen Gründen) erst zu einem späteren Zeitpunkt.

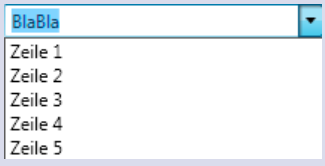
Beispiel 2.18: *ComboBox* mit Texteingabe

XAML

```
<ComboBox Name="ComboBox2" IsEditable="True">
  <ComboBoxItem>Zeile 1</ComboBoxItem>
  <ComboBoxItem>Zeile 2</ComboBoxItem>
  <ComboBoxItem>Zeile 3</ComboBoxItem>
  <ComboBoxItem>Zeile 4</ComboBoxItem>
  <ComboBoxItem>Zeile 5</ComboBoxItem>
</ComboBox>
```

Ergebnis

Die Texteingabe zur Laufzeit:



HINWEIS: Auf das Füllen per Datenbindung gehen wir erst in Kapitel 4 ein.

2.7.3 Den Content formatieren

Gut und schön werden Sie sicher sagen, vieles davon kann auch Windows Forms. Doch da täuschen Sie sich vermutlich etwas. Die einzelnen *ComboBoxItem*- bzw. *ListBoxItem*-Einträge verfügen nicht nur über die *Content*-Eigenschaft sondern bieten darüber hinaus auch noch reichlich Optionen für das gezielte Formatieren des Inhalts an.

Statt endloser Aufzählungen soll ein Beispiel einige Anregungen bieten:

Beispiel 2.19: Formatieren von *ComboBox*-Einträgen

XAML

```
        <ComboBox Width="150" FontSize="16" BorderThickness="2" BorderBrush="Blue">
            <ComboBoxItem HorizontalAlignment="Right" Foreground="Green">Zeile 1
            </ComboBoxItem>
            <ComboBoxItem FontWeight="Bold" FontFamily="Courier New">Zeile
2</ComboBoxItem>
            <ComboBoxItem Width="50" Background="Yellow">Zeile 3</ComboBoxItem>
            <ComboBoxItem FontSize="8">Zeile 4</ComboBoxItem>
            <ComboBoxItem Background="Gray" Content="Zeile 5" />
        </ComboBox>
```

Ergebnis

Unter bestimmten Umständen können Sie statt der ausführlichen Syntax auch eine verkürzte Form verwenden, allerdings müssen Sie in diesem Fall einen zusätzlichen Namespace einbinden und die Einträge typisieren:

Beispiel 2.20: String-Formatierung in einer *ComboBox*

XAML

```
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib" Height="127"
Width="340">
    <StackPanel>
        <ComboBox Width="150" HorizontalContentAlignment="Center"
ItemStringFormat="0.00 €">
            <sys:String>Kein Wert</sys:String>
            <sys:Decimal>0.1</sys:Decimal>
            <sys:Decimal>1</sys:Decimal>
            <sys:Decimal>1.1</sys:Decimal>
            <sys:Decimal>1.11</sys:Decimal>
            <sys:Decimal>1.111</sys:Decimal>
```

```
        </ComboBox>
    </StackPanel></Window>
```

Ergebnis

2.8 Image

Ganz nebenbei haben wir in den vorhergehenden Beispielen bereits einen Blick auf das *Image*-Control geworfen. Wie Sie sicher schon festgestellt haben, können Sie damit Grafiken auf einfache Weise sowohl in einem *Window*, als auch in ListBoxen, ComboBoxen, Buttons etc. anzeigen.

Zur Verfügung stehen neben den altbekannten Formaten BMP, GIF, ICO JPG, PNG, TIFF auch das WDP-Format[3]. Leider werden sowohl das WMF als auch das EMF-Format nicht unterstützt, hier hilft nur die Verwendung von Windows Forms.

2.8.1 Grafik per XAML zuweisen

Zugewiesen wird die Grafik über das *Source*-Attribut. Die genaue Form der Adressierung von Programmressourcen besprechen wir ausführlich erst in Abschnitt 3.2, an dieser Stelle soll uns ein Beispiel genügen.

Beispiel 2.21: Grafik aus Ressource laden

XAML

Erstellen Sie zunächst per Kontextmenü einen neuen Ordner[4] (*Images)* für Ihr WPF-Projekt. Fügen Sie nachfolgend per Drag&Drop eine Grafik (z. B. *Frosch.gif*) in dieses Verzeichnis ein.

Der XAML-Code:

```
<Window x:Class="BSP_Controls.Image_Bsp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Image_Bsp" Height="300" Width="300">
       <Image Source="Images\frosch.gif"/>
</Window>
```

[3] Windows Media Photo-Format

[4] Dies ist nicht zwingend erforderlich, verbessert aber die Übersicht im Projekt.

2.8.2 Grafik zur Laufzeit zuweisen

Im Gegensatz zu den guten alten Windows Forms ist das Laden von Grafiken per Code in WPF etwas mühevoller geworden.

Beispiel 2.22: Laden der im vorhergehenden Beispiel erzeugten Ressource

C#

```
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            BitmapImage bi = new BitmapImage();
            bi.BeginInit();
            bi.UriSource = new Uri("pack://application:,,,/images/frosch.gif");
```

Alternativ:

```
            bi.UriSource = new Uri("images/frosch.gif",UriKind.Relative);
            bi.EndInit();
            Image5.Source = bi;
        }
```

Der *Source*-Eigenschaft können Sie entweder ein initialisiertes *BitmapImage*-Objekt übergeben, in diesem Fall muss das eigentlich Laden der Datei (egal ob extern oder intern) mit den Methoden *BeginInit* und *EndInit* eingeleitet/beendet werden, oder Sie erzeugen ein BitmapFrame-Objekt, wie im folgenden Beispiel gezeigt:

Beispiel 2.23: Laden der Ressource per *BitmapFrame*

C#

```
Image5.Source = BitmapFrame.Create(new Uri("pack://application:,,,/images/frosch.gif"));
```

HINWEIS: In den Microsoft-Dokumentationen findet Sie beim *BitmapImage* noch die Eigenschaften *DecodePixelWidth* und *DecodePixelHeight.* Damit können Sie schon beim Laden des Bildes eine Skalierung durchführen (z. B. entsprechend der Anzeigefläche). Im Speicher wird jetzt nur ein Bild dieser Größe gehalten, eine dauernde Skalierung ist nicht mehr notwendig (weniger Speicher, weniger Rechenzeit).

Beispiel 2.24: Maximale Anzeigebreite ist 50, die Originalgröße ist 200

C#

```
...
            BitmapImage bi = new BitmapImage();
            bi.BeginInit();
            bi.DecodePixelWidth = 50;
            bi.UriSource = new Uri("images/frosch.gif",UriKind.Relative);
...
```

HINWEIS: Die Anzeige sollte in diesem Fall mit *Stretch* = *None* erfolgen.

2.8.3 Bild aus Datei laden

Hier müssen wir auf die uralten Dateidialoge zugreifen, WPF hat derzeit noch keine eigenen Dateidialoge.

Beispiel 2.25: Laden einer Grafik per Dateidialog

C#

Zunächst den Namespace einbinden:

```
using Microsoft.Win32;
...
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
```

Instanz erzeugen:

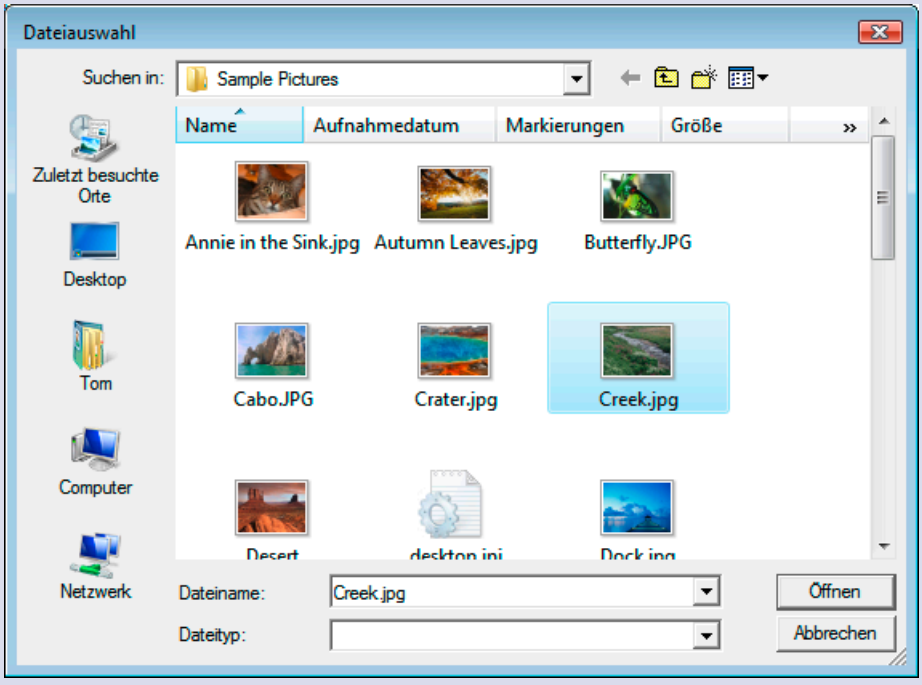
```
            OpenFileDialog Dlg = new OpenFileDialog();
            Dlg.Title = "Dateiauswahl";
            Dlg.DefaultExt = "jpg";
```

Anzeige und Auswertung, wenn Öffnen angeklickt wurde:

```
            if ((bool)Dlg.ShowDialog())
                Image6.Source = BitmapFrame.Create(new Uri(Dlg.FileName));
        }
```

Ergebnis

Der Dateiauswahldialog erscheint beim Öffnen des Fensters:



2.8.4 Die Grafikskalierung beeinflussen

Nicht jede Grafik hat bereits die Größe, die wir für die Anzeigefläche benötigen. Das *Image*-Control stellt aus diesem Grund zwei Eigenschaften zur Verfügung, mit denen Sie das Skalieren konfigurieren können:

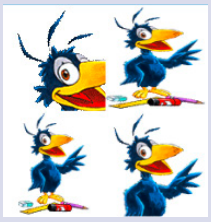
- *Stretch* (*None, Fill, Uniform, UniformToFill*)
- *StretchDirection* (*Both, DownOnly, UpOnly*)

Beispiel 2.26: Verwendung von *Stretch*

XAML

```
<Grid>
 <Grid.RowDefinitions>
   <RowDefinition Height="*" />
   <RowDefinition Height="*" />
 </Grid.RowDefinitions>
 <Grid.ColumnDefinitions>
   <ColumnDefinition Width="*" />
   <ColumnDefinition Width="*" />
 </Grid.ColumnDefinitions>

Originalgröße:

    <Image Grid.Column="0" Grid.Row="0" Source="Images\Rudi.gif" Stretch="None"/>

Skalierung ohne Rücksicht auf Proportionen:

    <Image Grid.Column="1" Grid.Row="0" Source="Images\rudi.gif" Stretch="Fill"/>

Skalierung mit Rücksicht auf Proportionen, längste Seite bestimmt die Größe:

    <Image Grid.Column="0" Grid.Row="1" Source="Images\rudi.gif" Stretch="Uniform"/>

Skalierung mit Rücksicht auf Proportionen, kürzeste Seite bestimmt die Größe:

    <Image Grid.Column="1" Grid.Row="1" Source="Images\rudi.gif"
Stretch="UniformToFill"/>
 </Grid>
```

Ergebnis



Wird das *Image* frei positioniert (z. B. *Canvas),* sollten Sie für eine vorgegebene Größe entweder die Höhe **oder** die Breite angeben, nie beides, da es in diesem Fall zu Verzerrungen kommen kann.

HINWEIS: Beachten Sie in diesem Fall auch die Eigenschaften *ActualWidth* und *ActualHeight*, beide liefern erst nach dem Laden des Bildes einen sinnvollen Wert.

2.9 MediaElement

Für die Anzeige von Videos bzw. die Wiedergabe von Audiodateien bietet sich in WPF das *MediaElement* an. Für die Steuerung verwenden Sie die Methoden *Play, Pause* und *Stop*, zuweisen können Sie das Video über die *Source*-Eigenschaft. Doch Achtung:

HINWEIS: Sie können keine als Ressourcen eingebetteten Videos wiedergeben. Kopieren Sie die Videos stattdessen in das Ausgabeverzeichnis (*Build Action=Content, CopyToOutputDirectory=PreserveNewest*).

Nach dem Öffnen der Videodatei stehen Ihnen einige wichtige Eigenschaften zur Verfügung:

- *NaturalDuration* (Laufzeit des Videos)
- *NaturalVideoHeight*, *NaturalVideoWidth* (die Originalabmessungen des Videos)
- *ActualWidth, ActualHeight* (die im Window realisierten Abmessungen des Controls)

Über die Eigenschaft *SpeedRatio* können Sie Einfluss auf die Wiedergabegeschwindigkeit nehmen, Werte größer eins beschleunigen die Wiedergabe, Werte kleiner eins verlangsamen sie.

Wer es gern ruhig mag, der kann *IsMuted* auf *True* setzen, das schont die Ohren.

HINWEIS: Die *LoadedBehavior*-Eigenschaft müssen Sie auf *Manual* setzen, wenn Sie die obigen Methoden zur Steuerung des Controls verwenden wollen.

Beispiel 2.27: Verwendung der *MediaElements*

XAML

```
  <StackPanel>
    <StackPanel Orientation="Horizontal">
      <Button Click="Button_Click">Start</Button>
      <Button Click="Button_Click_1">Pause</Button>
      <Button Click="Button_Click_2">Stop</Button>

Der Lautstärkeregler:

      <Slider Name="VSlider" VerticalAlignment="Center"
ValueChanged="VSlider_ValueChanged"
                    Minimum="0" Maximum="1" Value="0.5" Width="70"/>
```

Die Geschwindigkeit:

```
      <Slider Name="SSlider" VerticalAlignment="Center" ValueChanged=
                          "SSlider_ValueChanged"
                    Value="1" Maximum="5" Minimum="0.1" Width="70" />
    </StackPanel>
```

Das eigentliche *MediaElement*:

```
    <MediaElement LoadedBehavior="Manual" UnloadedBehavior="Stop" Width="400"
 Name="Media1"
     Stretch="Uniform" Source="butterfly.wmv" MediaOpened="Media1_MediaOpened"
     MediaEnded="Media1_MediaEnded">
    </MediaElement>
```

C#

Der Ereigniscode:

```
        private void Button_Click(object sender, RoutedEventArgs e)
        {  Media1.Play(); }

        private void Button_Click_1(object sender, RoutedEventArgs e)
        { Media1.Pause(); }

        private void Button_Click_2(object sender, RoutedEventArgs e)
        { Media1.Stop();  }
```

Lautstärke regeln:

```
        private void VSlider_ValueChanged(object sender,
                  RoutedPropertyChangedEventArgs<double> e)
        { if (Media1 != null) Media1.Volume = (double)VSlider.Value; }
```

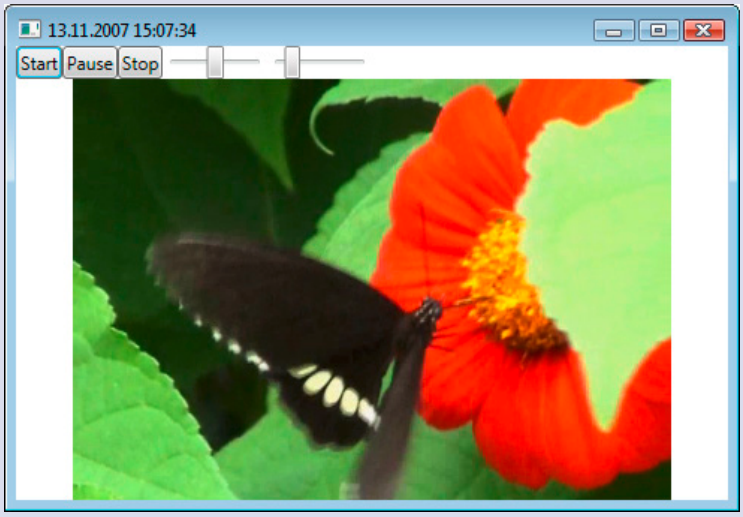
Geschwindigkeit regeln:

```
        private void SSlider_ValueChanged(object sender,
                   RoutedPropertyChangedEventArgs<double> e)
        { if (Media1!=null) Media1.SpeedRatio = (double)SSlider.Value; }
```

Nach dem Öffnen des Videos bzw. am Ende:

```
        private void Media1_MediaOpened(object sender, RoutedEventArgs e)
        { }

        private void Media1_MediaEnded(object sender, RoutedEventArgs e)
        {  }
```

Ergebnis



2.10 Slider, ScrollBar

Im Folgenden wollen wir uns mit zwei Controls beschäftigen, die für die Eingabe von Werten innerhalb eines bestimmten Wertebereichs geeignet sind.

2.10.1 Slider

Im vorhergehenden Abschnitt hatten wir bereits auf das *Slider*-Control zurückgegriffen, um die Lautstärke und die Abspielgeschwindigkeit zu regeln (siehe obige Abbildung). Damit dürfte auch schon der Verwendungszweck erkennbar sein:

- Auswahl eines Wertes durch Verschieben des Reglers (*Value*)
- Vorgabe eines minimalen Wertes (*Minimum)*
- Vorgabe eines maximalen Wertes (*Maximum)*

Beachten Sie, dass es sich beim *Value* um einen *Double*-Wert handelt, Sie also gegebenenfalls eine Typisierung/Rundung durchführen müssen, um den Wert in Ihrem Programm sinnvoll nutzen zu können. Änderungen der Auswahl können Sie über das *ValueChanged*-Ereignis auswerten.

HINWEIS: Natürlich können Sie den *Slider* auch als Ausgabemedium zur Wertanzeige missbrauchen.

Neben den drei genannten Eigenschaften bieten sich auch noch weitere Möglichkeiten zur Konfiguration an:

- **Ausrichtung**
  Verwenden Sie *Orientation*, um die Ausrichtung des Controls zu beeinflussen (*Vertical, Horizontal*).
- **Anzeige von Werten**
  Hier bietet sich die Verwendung von Tooltips an. Setzen Sie *AutoToolTipPlacement* auf *BottomRight* oder *TopLeft,* um den jeweils aktuellen Wert beim Verschieben des Sliders anzuzeigen. Die Genauigkeit bestimmen Sie mit *AutoToolTipPrecision*.
- **Werteverlauf**
  Standardmäßig finden Sie die kleinsten Werte links bzw. unten. Sollen sich diese rechts bzw. oben befinden, müssen Sie IsDirectionReversed auf *True* setzen.
- **Markierungsbereich**
  Mit *IsSelectionRangeEnabled* aktivieren Sie die Anzeige eines Markierungsbereichs. Den Bereich selbst definieren Sie über *SelectionStart* und *SelectionEnd.*
- **Markierung, Skala**
  Mit *TickPlacement* (*Both, BottomRight, TopLeft*) können Sie die Anzeige von Markierungsstrichen einschalten. Da standardmäßig die Striche für jeden ganzzahligen Wert gesetzt werden, sollten Sie mit *TickFrequency* ein anderes Intervall festlegen.
- **Schrittweite**
  Haben Sie *TickFrequency* festgelegt, können Sie mit *IsSnapToTickEnabled=True* erreichen, dass nur Werte genau an den Markierungsstrichen ausgewählt werden können.

Beispiel 2.28: Horizontaler *Slider* mit Markierungsbereich und Tooltip

XAML

```
<Slider  AutoToolTipPlacement="TopLeft"  IsSelectionRangeEnabled="True"
SelectionStart="10"
         SelectionEnd="20" Minimum="1" Maximum="50" Value="25" Canvas.Left="11"
         Canvas.Top="55" Height="26" Name="slider1" Width="179" />
```

Ergebnis

Beispiel 2.29: Vertikaler *Slider* mit Markierungsstrichen und zwangsweiser Auswahl (0, 10, 20 .. 100)

XAML

```
<Slider  TickPlacement="Both" TickFrequency="10" IsSnapToTickEnabled="True"
         AutoToolTipPlacement="BottomRight" Width="43"  Orientation="Vertical"
         Canvas.Left="220" Canvas.Top="21" Height="111" Name="slider2"
Maximum="100" />
```

Ergebnis

2.10.2 ScrollBar

Ähnlich wie beim *Slider* bietet ein *ScrollBar* die Möglichkeit, Werte (*Value*) aus einem Wertebereich (*Minimum, Maximum)* auszuwählen. Scrollbars können mit *Orientation* ebenfalls horizontal oder vertikal ausgerichtet werden. Die Auswertung kann mit dem Ereignis *Scroll* erfolgen, alternativ kommt jedoch meist eine direkte Datenbindung (siehe Kapitel 4) in Frage.

Im Gegensatz zum *Slider* fehlen jedoch alle weiteren Möglichkeiten zur Konfiguration (Wertanzeige, Skala etc.).

Beispiel 2.30: Verwendung *ScrollBar*

XAML

```
<ScrollBar Maximum="10" Scroll="ScrollBar_Scroll" Orientation="Horizontal"
Canvas.Left="42" Canvas.Top="182" Height="25" Name="scrollBar1" Width="214" />
```

Ergebnis

2.11 ScrollViewer

Nicht genug der Scroller, mit dem *ScrollViewer* können Sie direkt ein im *Content* enthaltenes Container-Control horizontal und vertikal verschieben, wenn es nicht in den sichtbaren Clientbereich des ScrollViewers passt. Ob und wann die ScrollBars angezeigt werden, entscheiden Sie mit den Eigenschaften *HorizontalScrollBarVisibility* und *VerticalScrollBarVisibility*.

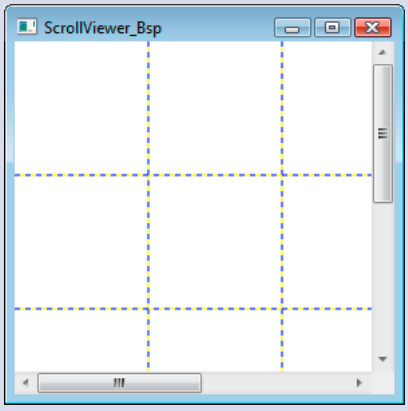
Häufiger Verwendungszweck dieses Controls ist die Darstellung von Grafiken, die zu groß für die Anzeige im Fenster sind.

Beispiel 2.31: Ein *Grid* mit fester Größe soll in einem *ScrollViewer* angezeigt werden

XAML

```
<ScrollViewer Name="scrollViewer1" HorizontalScrollBarVisibility="Visible"
              VerticalScrollBarVisibility="Visible">
  <Grid Width="500" Height="500" ShowGridLines="True">
    <Grid.RowDefinitions>
      <RowDefinition Height="*" />
      <RowDefinition Height="*" />
      <RowDefinition Height="*" />
...
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="*" />
...
    </Grid.ColumnDefinitions>
  </Grid>
</ScrollViewer>
```

Ergebnis



Entsprechend der Größe des enthaltenen Controls (*Grid*) wird der Scrollbereich angepasst.

2.12 Menu, ContextMenu

Nachdem wir uns schon eine Reihe der geläufigsten Controls angesehen haben, wollen wir uns endlich an ein wesentliches Oberflächenelement wagen, das wohl in fast keinem Programm fehlt: die Menüleiste. An dieser Stelle gleich ein wichtiger Hinweis, damit Ihre Ambitionen nicht zu weit gehen:

HINWEIS: WPF unterstützt keine MDI-Anwendungen!

Eine Alternative finden Sie jedoch unter *http://wpfmdi.codeplex.com.*

2.12.1 Menu

Ein eigenes Menü erstellen Sie recht einfach per XAML mit dem <*Menu*>-Element, in das Sie ineinander geschachtelte <*MenuItem*>-Elemente einfügen. Zusätzlich können Sie zum Gruppieren noch <*Separator*>-Elemente verwenden. Die Auswertung können Sie, wie nicht anders zu erwarten, mit dem *Click*-Ereignis vornehmen.

Beispiel 2.32: Ein einfaches Menü erstellen

XAML

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

Zunächst ein *DockPanel,* um das Menü auch oben zu verankern:

```
  <DockPanel>
```

Hier das Menü:

```
    <Menu DockPanel.Dock="Top" Height="22" Name="menu1">
```

Die erste Gruppe von *MenuItem*s:

```
      <MenuItem Header="_Datei">
        <MenuItem Header="Neu"/>
        <MenuItem Header="Öffnen"/>
        <MenuItem Header="Sichern"/>
```

Hier ein Separator (Trennstrich):

```
        <Separator/>
        <MenuItem Header="Ende"/>
      </MenuItem>
```

Noch zwei Hauptmenüpunkte:

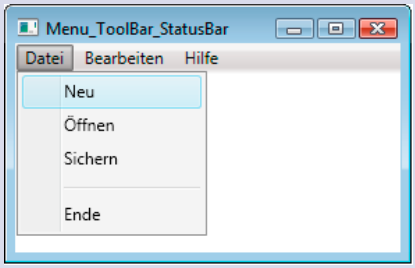
```
      <MenuItem Header="_Bearbeiten"/>
      <MenuItem Header="_Hilfe"/>
    </Menu>
```

Das folgende *StackPanel* füllt den verbliebenen Platz im *DockPanel*:

```
    <StackPanel>
    </StackPanel>
  </DockPanel>
 </Window>
```

Ergebnis

Im Endergebnis dürfte folgendes *Window* angezeigt werden:



2.12.2 Tastenkürzel

Wie Sie sicher bereits im obigen Beispiel bemerkt haben, werden die Alt-Tasten-Shortcuts durch einen Unterstrich markiert. Möchten Sie erweiterte Tastenkombinationen verwenden, können Sie diese über das *InputGestureText*-Attribut einem Menüpunkt zuweisen.

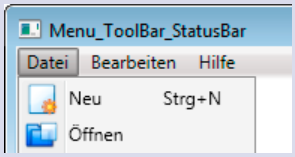
Beispiel 2.33: Verwendung von *InputGestureText*

XAML

```
<MenuItem Header="Neu" InputGestureText="Strg+N" Click="MenuItem_Click">
  <MenuItem.Icon>
    <Image Source="Images/filenew.png" Width="22"/>
  </MenuItem.Icon>
</MenuItem>
```

Ergebnis

Doch ein erster Test wird Sie auf den harten Boden der Realität zurückholen, die Tastenfolge wird zwar im Menü angezeigt



HINWEIS: Alternativ können Sie in WPF sogenannte Commands verwenden, wir gehen in Abschnitt 3.4 darauf ein.

2.12.3 Grafiken

Bisher sehen unsere Menüpunkte noch recht trist aus, es fehlen die bekannten Grafiken. Doch auch dies ist kein Problem, WPF bietet zwei Varianten für die Anzeige von Grafiken an:

- Ein zusätzliches Icon links neben dem Texteintrag
- Eine beliebige Kombination von Grafik/Elementen innerhalb des Menüeintrags

Beispiel 2.34: Einblenden eines zusätzlichen Icons links neben dem Menüeintrag

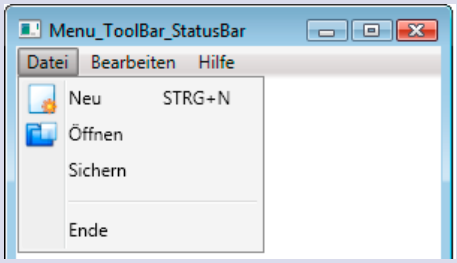
XAML

```
    <Menu DockPanel.Dock="Top" Height="22" Name="menu1">
      <MenuItem Header="_Datei">
        <MenuItem Header="Neu">
```

Hier weisen Sie die Grafik zu:

```
          <MenuItem.Icon>
            <Image Source="Images/filenew.png" Width="22"/>
          </MenuItem.Icon>
        </MenuItem>
        <MenuItem Header="Öffnen">
          <MenuItem.Icon>
            <Image Source="Images/fileopen.png" Width="22"/>
          </MenuItem.Icon>
        </MenuItem>
...
```

Ergebnis



Beispiel 2.35: Ein grafischer Menüeintrag mit freier Gestaltung (zusätzliche *ComboBox*)

XAML

```
    <Menu DockPanel.Dock="Top" Height="22" Name="menu1">
      <MenuItem Header="_Datei">
...
        <MenuItem>
```

Mit einem *MenuItem.Header*-Element bietet sich die Möglichkeit, beliebige Elemente in einem Menüeintrag unterzubringen:

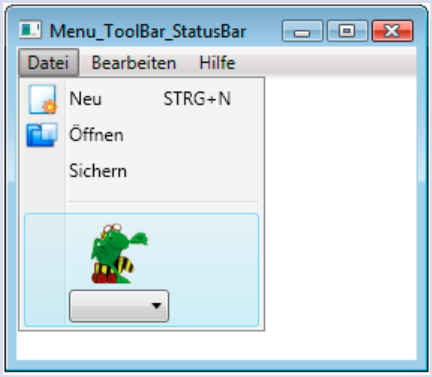
```
          <MenuItem.Header>
```

Fügen Sie zunächst ein Layout-Element ein, danach ist die Positionierung von Grafiken oder anderen Elementen ein Kinderspiel:

```
            <StackPanel>
              <Image Source="images/frosch.gif" Height="40" Margin="4"/>
              <ComboBox>
                <ComboBoxItem>Zeile 1</ComboBoxItem>
                <ComboBoxItem>Zeile 2</ComboBoxItem>
                <ComboBoxItem>Zeile 3</ComboBoxItem>
                <ComboBoxItem>Zeile 4</ComboBoxItem>
                <ComboBoxItem>Zeile 5</ComboBoxItem>
              </ComboBox>
            </StackPanel>
          </MenuItem.Header>
        </MenuItem>
  ...
```

Ergebnis

Die folgende Abbildung zeigt des Ergebnis unserer Bemühungen, ob Sie eine *ComboBox* unbedingt im Menü unterbringen müssen, ist allerdings fraglich.



2.12.4 Weitere Möglichkeiten

Neben den bereits vorgestellten Möglichkeiten bietet sich mit den Eigenschaften *IsCheckable* und *IsChecked* die Möglichkeit, Optionen innerhalb des Menüs zu aktivieren bzw. zu deaktivieren.

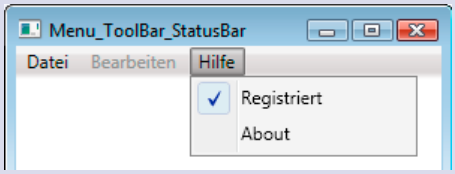
Den Menüpunkt selbst können Sie mit *IsEnabled=False* deaktivieren.

Beispiel 2.36: *IsCheckable* und *IsEnabled*

XAML

```
    <Menu DockPanel.Dock="Top" Height="22" Name="menu1">
...
      <MenuItem Header="_Bearbeiten" IsEnabled="False"/>
      <MenuItem Header="_Hilfe">
        <MenuItem Header="Registriert" IsCheckable="True" IsChecked="True"/>
        <MenuItem Header="About" IsEnabled="True"/>
      </MenuItem>
    </Menu>
```

Ergebnis



2.12.5 ContextMenu

Auf das Kontextmenü trifft weitgehend das bereits Gesagte zu, mit dem Unterschied, dass ein *ContextMenu* einem spezifischen Control zugeordnet wird.

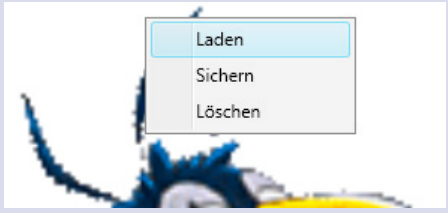
Beispiel 2.37: Erzeugen eines Kontextmenüs

XAML

```
    <Image Source="Images/rudi.gif" >
     <Image.ContextMenu>
        <ContextMenu>
          <MenuItem Header="Laden" />
          <MenuItem Header="Sichern" />
          <MenuItem Header="Löschen" />
        </ContextMenu>
      </Image.ContextMenu>
    </Image>
```

Ergebnis

Das erzeugte Kontextmenü:



2.13 ToolBar

WPF-Anwendungen können eine Werkzeugleiste mit dem *ToolBar*-Control implementieren, allerdings haben Sie es hier nicht, wie vielleicht erwartet, mit einem hoch komplexen Control, sondern „lediglich“ einem Container für die bereits aufgeführten Controls zu tun. Das beginnt bereits beim Einfügen des Controls, Sie als Programmierer müssen sich darum kümmern, dass der *ToolBar* auch dahin kommt wo er hin soll. Verwenden Sie dazu ein *DockPanel* und richten Sie das Control an der gewünschten Seite aus.

HINWEIS: Optional können Sie einen/mehrere Toolbar(s) in einem *ToolBarTray* unterbringen, hier ist dann auch ein Positionieren und Verschieben möglich.

Beispiel 2.38: Ein einfacher *ToolBar* mit drei Schaltflächen und einer *ComboBox*

XAML

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

 <DockPanel>
   <Menu DockPanel.Dock="Top" Height="22" Name="menu1">
...
   </Menu>
```

Zunächst der äußere Rahmen (ein *ToolBarTray*):

```
    <ToolBarTray DockPanel.Dock="Top">
```

Dann der eigentliche *Toolbar*:

```
      <ToolBar>
```

Und jetzt passiert das Gleiche wie in den anderen Layout-Containern (z. B. *StackPanel* mit horizontaler Ausrichtung):

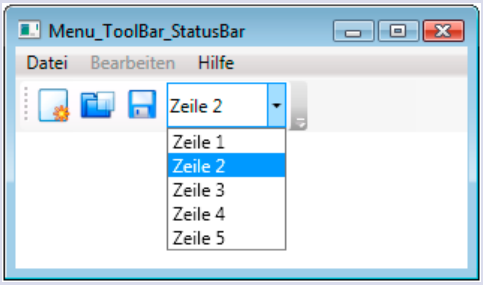
```
        <Button Width="30" Height="30">
          <Image Source="Images/filenew.png"/>
        </Button>
        <Button Width="30" Height="30">
          <Image Source="Images/fileopen.png"/>
        </Button>
        <Button Width="30" Height="30">
          <Image Source="Images/filesave.png"/>
        </Button>
```

Das funktioniert natürlich auch mit einer *ComboBox*:

```
        <ComboBox Width="80" Height="30">
          <ComboBoxItem>Zeile 1</ComboBoxItem>
```

```
          <ComboBoxItem>Zeile 2</ComboBoxItem>
          <ComboBoxItem>Zeile 3</ComboBoxItem>
          <ComboBoxItem>Zeile 4</ComboBoxItem>
          <ComboBoxItem>Zeile 5</ComboBoxItem>
        </ComboBox>
      </ToolBar>
    </ToolBarTray>
  </Window>
```

Ergebnis



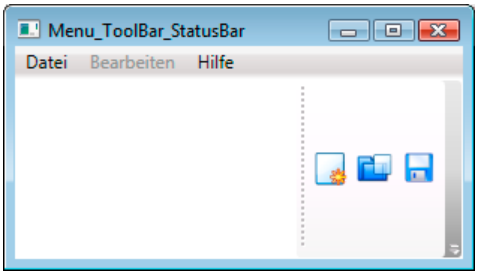
Der ToolBarTray

Warum fügen wir eigentlich den *ToolBar* in einen *ToolBarTray* ein? Die Antwort finden Sie schnell, wenn Sie beispielsweise den *ToolBar* am rechten oder linken Rand des Windows platzieren möchten.

Mit den folgenden Anweisungen

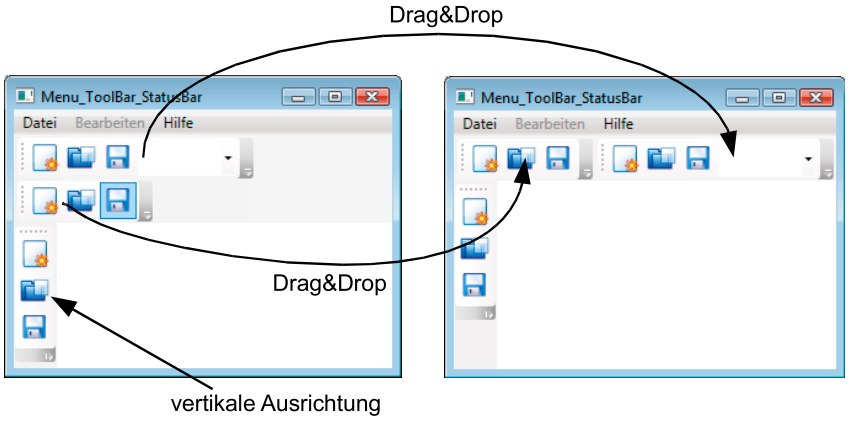
```
    <ToolBar DockPanel.Dock="Right">
      <Button Width="30" Height="30">
        <Image Source="Images/filenew.png"/>
      </Button>
...
```

... wird zwar der *ToolBar* rechts platziert, es erfolgt aber keine vertikale Ausrichtung. Eine entsprechende Eigenschaft ist nicht vorhanden.



Auch das Platzieren (per Code oder Drag&Drop) mehrerer einzelner *ToolBar*s wird so schnell zum Geduldspiel.

Fügen Sie hingegen die einzelnen *ToolBar*-Controls in einen *ToolBarTray* ein, können Sie diesen per *Orientation*-Eigenschaft problemlos vertikal ausrichten. Auch das Platzieren der einzelnen *ToolBar*s per Drag&Drop durch den Anwender ist jetzt möglich:

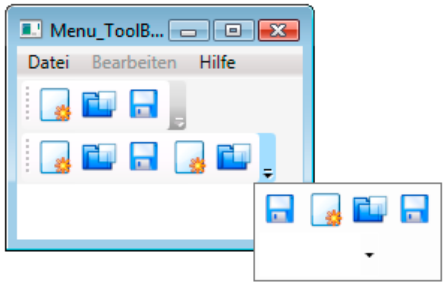


Mit der Verwendung des *ToolBarTray* ergeben sich jedoch noch weitere Möglichkeiten:

- Mit der *ToolBar*-Eigenschaft *Band* legen Sie die Zeile innerhalb des *ToolbarTray* fest (siehe obige Abbildung mit zweizeiligem *ToolBarTray.*
- Die Reihenfolge mehrerer *ToolBar*-Controls innerhalb einer Zeile legen Sie über die Eigenschaft *BandIndex* fest.
- Möchten Sie das Verschieben von *ToolBar*s innerhalb des *ToolBarTray* verhindern, setzen Sie deren *IsLocked* auf *True.*

HINWEIS: Blenden Sie nicht benötigte *ToolBar*-Controls mit *Visibility=Collapsed* aus, wenn Sie den vom *ToolBar* benötigten Platz freigeben möchten. Andernfalls genügt ein *Hidden*, der *ToolBarTray* bleibt in diesem Fall sichtbar.

Was passiert eigentlich, wenn nicht genügend Platz für die komplette Darstellung des *ToolBar*s bleibt? In diesem Fall werden normalerweise die nicht mehr darstellbaren Controls in einem extra Menü (Überlaufbereich) angezeigt:



Sie als Programmierer können darüber entscheiden, wann welche Controls im Überlaufbereich angezeigt werden sollen. Dazu steht die angehängte Eigenschaft *ToolBar.OverflowMode* zur Verfügung. Für jedes einzelnen Control können Sie festlegen, ob es niemals (*Never*) bei Bedarf (*AsNeeded*) oder immer (*Always*) im Überlaufbereich angezeigt wird.

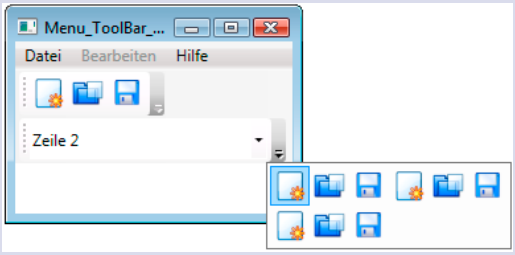
Beispiel 2.39: Eine *ComboBox* soll nie im Überlaufbereich angezeigt werden

XAML

```
<ToolBarTray DockPanel.Dock="Top">
  <ToolBar Band="1" >
    <Button Width="30" Height="30">
      <Image Source="Images/filenew.png"/>
    </Button>
    <ComboBox ToolBar.OverflowMode="Never" Width="80" Height="30">
      <ComboBoxItem>Zeile 1</ComboBoxItem>
      <ComboBoxItem>Zeile 2</ComboBoxItem>
      <ComboBoxItem>Zeile 3</ComboBoxItem>
      <ComboBoxItem>Zeile 4</ComboBoxItem>
      <ComboBoxItem>Zeile 5</ComboBoxItem>
    </ComboBox>
  </ToolBar>
```

Ergebnis

Obwohl die *ComboBox* das letzte Control innerhalb des *ToolBar*s ist, wird diese jetzt niemals im Überlaufbereich erscheinen. Stattdessen wird sie im „Notfall“ einfach abgeschnitten:



Bleibt eine letzte Frage: wie reagiere ich auf Ereignisse? Die Antwort ist ganz einfach: wie bei jedem anderen Control auch, d. h. mit *Click* oder *SelectionChanged*.

2.14 StatusBar, ProgressBar

Nach dem Menü und der Werkzeugleiste fehlt zu einer „kompletten“ Oberfläche meist noch eine Statusleiste für die Ausgabe von Programminformationen.

2.14.1 StatusBar

Wie auch beim Menü oder bei der Werkzeugleiste sind **Sie** dafür verantwortlich, den *StatusBar* an den gewünschten Platz zu bringen. Üblicherweise werden Sie diese Aufgabe mit einem *DockPanel* realisieren, dieses benötigen Sie sowieso für die beiden anderen genannten Controls.

Zwei Varianten zur Nutzung des *StatusBar* bieten sich an:

- Sie verwenden innerhalb des *StatusBar*s so genannte *StatusBarItems*, in die Sie wiederum per Layout-Control (*StackPanel, Grid*) weitere Controls (*Label*, *TextBox, Button* etc.) einfügen können, oder
- Sie verzichten auf die *StatusBarItem*s und setzen einfach die benötigen Controls in den Content des *StatusBar*.

Variante eins hat den Vorteil der Übersichtlichkeit, der einfachen Gruppierbarkeit von Controls sowie der Möglichkeit, die Controls auch recht einfach am rechten Rand auszurichten. Dazu verwenden Sie die *DockPanel.Dock*-Eigenschaft des *StatusBarItems*[5].

HINWEIS: Meist müssen Sie die Höhe/Breite von enthaltenen Controls vorgeben, andernfalls sind diese nicht sichtbar.

Beispiel 2.40: Verwendung des *StatusBar* mit rechter Ausrichtung eines *ProgressBar*

XAML

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

Zunächst Ausrichtung im Fenster:

```
    <StatusBar DockPanel.Dock="Bottom">
```

Ein paar einfach positionierte Controls:

```
      <Label Content="Suchtext"/>
      <TextBox Width="50">*</TextBox>
      <Button>Suchen</Button>
```

Sie könne auch einen Trenner einfügen:

```
      <Separator/>
```

[5] Der *StatusBar* ist ein verkapptes *DockPanel*.

Wir richten ein *StatusBarItem* am rechten Rand aus

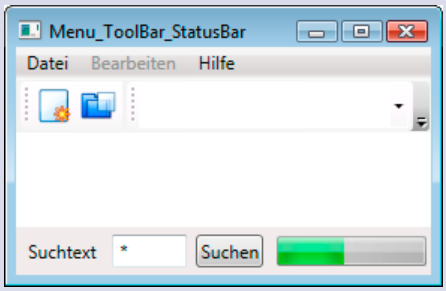
```
      <StatusBarItem DockPanel.Dock="Right">
```

... und fügen einen *ProgressBar* ein:

```
        <ProgressBar Width="100" Height="20" Value="45"/>
      </StatusBarItem>
    </StatusBar>
 ...
  </DockPanel>
</Window>
```

Ergebnis

Die Laufzeitansicht unseres *StatusBar*s:



2.14.2 ProgressBar

Nicht alles läuft so schnell wie wir es gern hätten, und so hat auch in Zeiten von Quad- oder Octa-Core-Prozessorsystemen der gute alte Fortschrittsbalken seine Daseinsberechtigung behalten.

In WPF-Anwendungen verwenden Sie dazu das *ProgressBar*-Control, das zwei Modi kennt:

- *IsIndeterminate=True*,
  der Fortschrittsbalken stellt eine endlose Animation dar, um „Bewegung“ zu zeigen[6]:

- *IsIndeterminate=False*,
  der bekannte Fortschrittsbalken, der mit *Minimum*, *Maximum* und *Value* konfiguriert wird:

[6] Das ideale Control für den Datei kopieren-Dialog unter Windows (gleich fertig, gleich fertig ...)

Das Control selbst kann horizontal oder vertikal angeordnet werden (*Orientation*), bester Aufbewahrungsort für den *ProgressBar* dürfte sicher der *StatusBar* sein.

Beispiel 2.41: Verwendung des *ProgressBar*-Controls

XAML

```
<StatusBar DockPanel.Dock="Bottom">
  <Label Content="Suchtext"/>
  <TextBox Width="50">*</TextBox>
  <StatusBarItem DockPanel.Dock="Right">
    <ProgressBar Width="100" Height="20" Value="45"/>
  </StatusBarItem>
  <Button>Suchen</Button>
  <Separator/>
</StatusBar>
```

2.15 Border, GroupBox, BulletDecorator

Nachdem wir uns bisher mit ganz praktischen Controls beschäftigt haben, wollen wir auch etwas für die Ordnung bzw. fürs Auge vorstellen. Die im Folgenden gezeigten Controls haben eigentlich nur den Zweck, die Optik des Programms zu verbessern.

2.15.1 Border

Möchten Sie einen (fast) frei definierbaren Rahmen um einzelne Controls zeichnen, bietet sich die Verwendung eines *Border*-Controls an.

Fügen Sie in den *Content* dieses Controls ein eigenes Layout-Control ein, um die darin enthaltenen Controls sinnvoll anzuordnen. Bei der Konfiguration des Borders können Sie neben Vorder- und Hintergrundfarbe/-muster auch die Rahmenbreite (*BorderThickness*) und den Eckradius (*CornerRadius*) festlegen. Letzteres kann für jede Ecke einzeln erfolgen, geben Sie in diesem Fall einfach vier statt einem Wert an (links oben, rechts oben, rechts unten, links unten). Gleiches gilt auch für die Rahmenbreite (links, oben, rechts, unten).

Die Abstände zum Rand bzw. zum Inhalt legen Sie wie gewohnt mit *Margin* bzw. *Padding* fest.

Beispiel 2.42: Verwendung *Border*

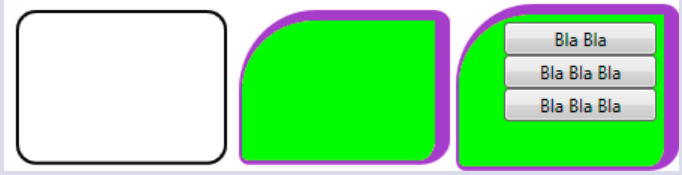
XAML

```
  <UniformGrid Margin="4" Columns="3" Rows="3">

Variante 1:

    <Border Margin="4" BorderThickness="2" CornerRadius="12" BorderBrush="Black"/>

Variante 2:

    <Border Margin="4" BorderThickness ="2,7,10,3" CornerRadius="50,5,15,5"
            Background="Chartreuse" BorderBrush="DarkOrchid"/>
```

Variante 3:

```
    <Border Padding="30,5,5,5" BorderThickness="2,7,10,3" CornerRadius="50,5,15,5"
            Background="Chartreuse" BorderBrush="DarkOrchid">
      <StackPanel>
        <Button>Bla Bla</Button>
        <Button>Bla Bla Bla</Button>
        <Button>Bla Bla Bla</Button>
      </StackPanel>
    </Border>
  </UniformGrid>
```

Ergebnis



2.15.2 GroupBox

Ein naher Verwandter des *Border*-Controls ist die *GroupBox*. Statt der Möglichkeit, den Eckradius zu beeinflussen, können Sie hier eine Kopfzeilenbeschriftung (*Header*) realisieren. Dabei müssen Sie sich nicht auf reinen Text beschränken, Sie können auch andere Controls und damit zum Beispiel auch Grafiken etc. in den Kopfbereich einfügen.

Beispiel 2.43: Verwendung der *GroupBox*

XAML

```
  <UniformGrid Margin="4" Columns="3" Rows="3">
```

Zunächst die einfachste Variante mit reinem Text:

```
    <GroupBox Header="Stammdaten" BorderThickness="2" BorderBrush="HotPink"
 Padding="5">
      <StackPanel>
        <Button>Bla Bla</Button>
        <Button>Bla Bla Bla</Button>
        <Button>Bla Bla Bla</Button>
      </StackPanel>
    </GroupBox>
```

Hier die Variante mit einer *CheckBox* im Header:

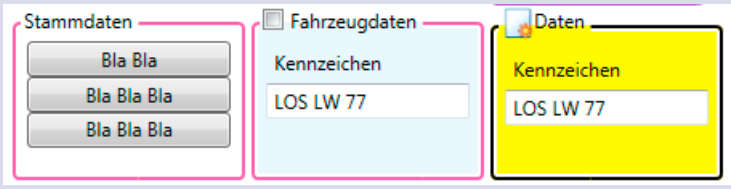
```
    <GroupBox BorderThickness="2" BorderBrush="HotPink" Padding="5"
 Background="AliceBlue">
      GroupBox.Header>
        <CheckBox>Fahrzeugdaten</CheckBox>
      </GroupBox.Header>
        <StackPanel>
        <Label>Kennzeichen</Label>
        <TextBox>LOS LW 77</TextBox>
      </StackPanel>
    /GroupBox>
```

Es geht auch mit einer Grafik und Text:

```
    <GroupBox BorderThickness="2" BorderBrush="Black" Padding="5"
 Background="Yellow">
      <GroupBox.Header>
        <StackPanel Orientation="Horizontal">
            <Image Source="Images/Filenew.png"  Width="20" />
            <TextBlock> Daten</TextBlock>
        </StackPanel>
      </GroupBox.Header>
      <StackPanel>
        <Label>Kennzeichen</Label>
        <TextBox>LOS LW 77</TextBox>
      </StackPanel>
    </GroupBox>
    <StackPanel>
  </UniformGrid>
```

Ergebnis

Die drei Varianten:



HINWEIS: Setzen Sie *RadioButtons* in eine *GroupBox,* um diese als logische Gruppe zu organisieren. So brauchen Sie nicht die *GroupName*-Eigenschaft festzulegen, und die Zusammenhänge fallen auch optisch auf.

2.15.3 BulletDecorator

Dieses Control dient der Anzeige von Aufzählungen, d.h., einem Aufzählungszeichen/-grafik (*BulletDecorator.Bullet*) und einem beschreibenden Text[7].

Beispiel 2.44: Verwendung *BulletDecorator*

XAML

```
  <UniformGrid Margin="4" Columns="3" Rows="3">

Variante 1 (Ellipse + Text):

    <StackPanel>
      <BulletDecorator Margin="5">

Das Aufzählungszeichen definieren:

        <BulletDecorator.Bullet>
          <Ellipse Width="10" Height="10" Fill="Red" />
        </BulletDecorator.Bullet>

Den Textteil definieren:

        <TextBlock Margin="5 0 0 0" TextWrapping="NoWrap">
          Zeile 1
        </TextBlock>
      </BulletDecorator>

Und jetzt die nächste Aufzählung:

      <BulletDecorator Margin="5">
        <BulletDecorator.Bullet>
          <Ellipse Width="10" Height="10" Fill="Red" />
        </BulletDecorator.Bullet>
        <TextBlock Margin="5 0 0 0" TextWrapping="NoWrap">
          Zeile 2
        </TextBlock>
      </BulletDecorator>
    </StackPanel>
```

[7] Mit einem *Grid* können Sie derartige Aufgaben sicher einfacher realisieren.

Variante 2 (mit *CheckBox):*

```
    <BulletDecorator Margin="0,5,0,0">
      <BulletDecorator.Bullet>
        <CheckBox/>
      </BulletDecorator.Bullet>
      <TextBlock Width="100" TextWrapping="Wrap" HorizontalAlignment="Left"
 Margin="5,0,0,0">
        Mit CheckBox
      </TextBlock>
    </BulletDecorator>
```

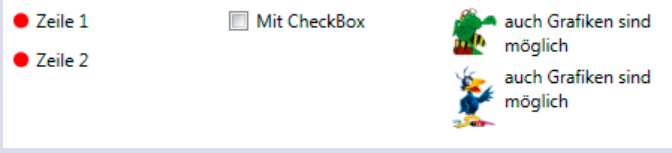
Variante 3 (mit Grafik):

```
    <StackPanel>
      <BulletDecorator Margin="0,5,0,0">
        <BulletDecorator.Bullet>
          <Image Source="Images/frosch.gif" Width="30"/>
        </BulletDecorator.Bullet>
        <TextBlock Width="100" TextWrapping="Wrap" HorizontalAlignment="Left"
                   Margin="5,0,0,0">
          auch Grafiken sind möglich
        </TextBlock>
      </BulletDecorator>
```

Und hier der nächste Eintrag:

```
      <BulletDecorator Margin="0,5,0,0">
        <BulletDecorator.Bullet>
          <Image Source="Images/rudi.gif" Width="30"/>
        </BulletDecorator.Bullet>
        <TextBlock Width="100" TextWrapping="Wrap" HorizontalAlignment="Left"
                   Margin="5,0,0,0">
           auch Grafiken sind möglich
        </TextBlock>
      </BulletDecorator>
    </StackPanel>
  </UniformGrid>
```

Ergebnis



2.16 RichTextBox

Um es gleich vorweg zu sagen, dieses Control ist nicht mit seinen doch recht rudimentär entwickelten Vorgängern bei den Windows Forms oder den Win32-Controls zu vergleichen. Aufgabe des Controls ist das komfortable Bearbeiten von formatierten Dokumenten. Basis ist dabei ein so genanntes *FlowDocument*, in dem Sie neben den bekannten Zeichenformatierungen (Fett, Kursiv, Unterstrichen, Hochstellen ...) auch wesentlich komplexere Formate, wie Grafiken, Absätze, Listen, Tabellen etc., realisieren können.

Das Control unterstützt beim Im-/Export neben den bekannten Formaten RTF, TEXT auch XAML und XAML-Packages.

HINWEIS: Statt Sie auf den folgenden Seiten mit endlosen Eigenschaftsauflistungen zu foltern (und davon hat das Control reichlich), wollen wir es uns „aufgaben-orientiert" ansehen.

2.16.1 Verwendung und Anzeige von vordefiniertem Text

Eine *RichTextBox* ist mit dem gleichnamigen Element schnell definiert, doch was ist mit dem Inhalt?

Einzig gültiges Element im Content der *RichTextBox* ist ein *FlowDocument*, auf dieses können Sie zur Laufzeit über die *Document*-Eigenschaft des Controls zugreifen. Innerhalb dieses *FlowDocuments* können Sie

- *BlockUIContainer*-
- *List*-
- *Paragraph*-
- *Section*-
- oder *Table*-Elemente

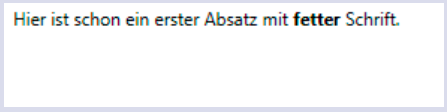
definieren. Mehr dazu ab Abschnitt 2.17 (*FlowDocument*), an dieser Stelle beschränken wir uns auf einen simplen Absatz mit einfachen Formatierungen.

Beispiel 2.45: Eine *RichTextBox* mit vordefiniertem Text anzeigen

XAML

```
<RichTextBox>
  <FlowDocument>
    <Paragraph>
        Hier ist schon ein erster Absatz mit <Bold>fetter</Bold> Schrift.
    </Paragraph>
  </FlowDocument>
</RichTextBox>
```

Ergebnis



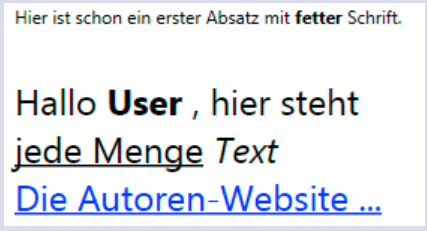
Beispiel 2.46: Eine *RichTextBox* mit vordefiniertem Text anzeigen (Fortsetzung)

XAML

Natürlich ist auch die Definition weiterer Absätze mit anderen Formatierungen möglich:

```
...
       <Paragraph FontSize="24">
         Hallo
         <Bold>User</Bold> , hier steht
         <Underline>jede Menge</Underline>
         <Italic>Text</Italic>
         <LineBreak/>
         <Hyperlink NavigateUri="http://www.doko-buch.de/">
             Die Autoren-Website ...
         </Hyperlink>
         <LineBreak />
        </Paragraph>
...
```

Ergebnis



HINWEIS: Hyperlinks sind in den hier vorgestellten WPF-Windows-Anwendungen nicht funktionsfähig, hier müssen Sie für die Logik selbst sorgen. Nutzen Sie dazu die Möglichkeit, die *Hyperlink*-Klasse abzuleiten und in der neuen Klasse das *RequestNavigate*-Ereignis für die Navigation zu verwenden.

HINWEIS: Formatierungen, wie Fett, Kursiv und Unterstrichen, können Sie in der *RichTextBox* zur Laufzeit problemlos per Tastatur-Befehl realisieren. Wählen Sie einfach *Strg+Shift+F*, *Strg+Shift+K* oder *Strg+Shift+U*[8].

[8] Die Tastenbelegung unterscheidet sich von der in der Hilfe angegebenen US-Belegung.

2.16.2 Neues Dokument zur Laufzeit erzeugen

Nachdem der Anwender mehr oder weniger ziellos mit dem Editor gearbeitet hat, kommt häufig der Wunsch auf, das Dokument komplett zu löschen.

Diese Aufgabe lösen Sie, indem Sie der *Document*-Eigenschaft einfach ein neues leeres *FlowDocument*-Objekt zuweisen.

Beispiel 2.47: Komplettes Dokument löschen bzw. erstellen

C#

```
private void newb_Click(object sender, RoutedEventArgs e)
{
    rtb1.Document = new FlowDocument();
}
```

2.16.3 Sichern von Dokumenten

Was nützt der beste Editor, wenn man den Inhalt nicht sichern kann? Eine direkte *Save*-Methode werden Sie sowohl am Control als auch beim *Document*-Objekt vergeblich suchen.

Gesichert werden kann das komplette Dokument (oder auch nur einzelne Abschnitte), indem Sie ein *TextRange*-Objekt aus dem Dokument erzeugen. Dazu müssen Sie den Beginn und das Ende des Textbereichs definieren. Der *TextRange* selbst bietet uns dann die gewünschte *Save*-Methode, bei der Sie auch das Datenformat (Text, RTF, XAML, XAML-Package) angeben können.

HINWEIS: Sichern können Sie in einen geöffneten Stream, damit stehen Ihnen alle Wege (Memory, Datenbank, Datei ...) offen.

Beispiel 2.48: Sichern des kompletten Inhalts des *RichtTextBox*

C#

```
private void Saveb_Click(object sender, RoutedEventArgs e)
{
```

Den nötigen Datei-Sichern-Dialog einblenden:

```
    SaveFileDialog dialog = new SaveFileDialog();
    dialog.Filter = "Xaml(*.xaml)|*.xaml";
```

Im Erfolgsfall:

```
    if ((bool)dialog.ShowDialog())
    {
```

Stream erzeugen:

```
        FileStream fs = (FileStream)dialog.OpenFile();
```

Das komplette Dokument als *TextRange* erfassen:

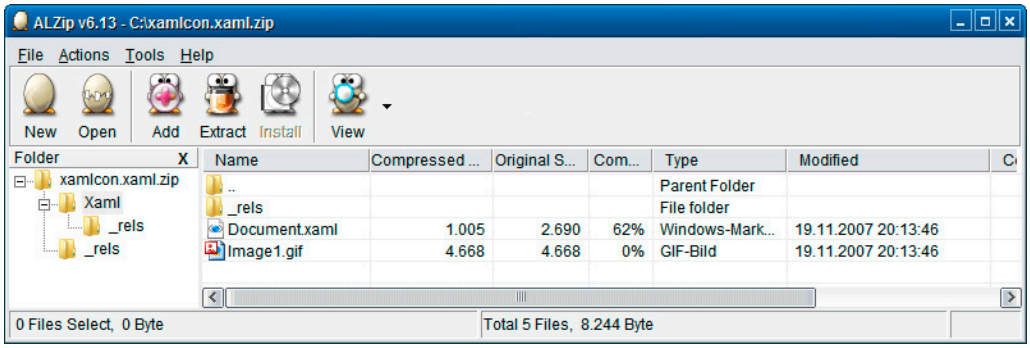
```
            TextRange tr = new TextRange(rtb1.Document.ContentStart,
                                         rtb1.Document.ContentEnd);
```

Sichern:

```
            tr.Save(fs, DataFormats.XamlPackage);
            fs.Close();
        }
    }
```

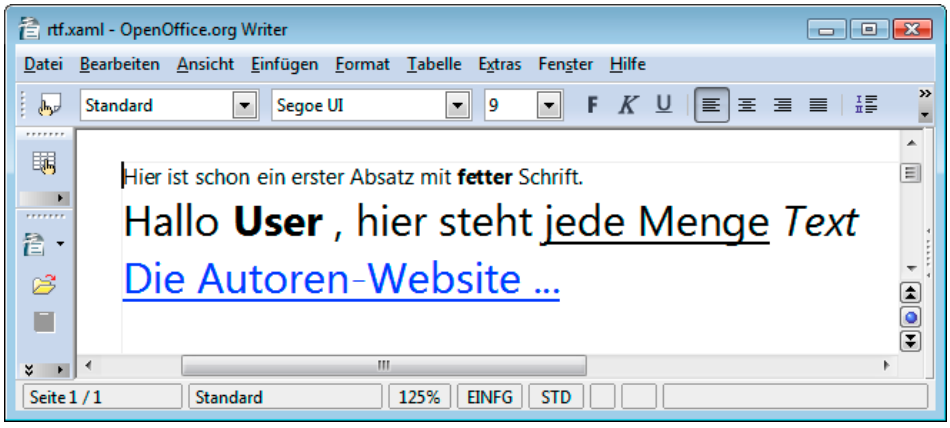
Bei der im Beispiel erzeugte XAML-Package-Datei handelt es sich um ein komprimiertes Format (ZIP), in dem sowohl die textuellen Inhalte als auch Grafiken etc. gesichert werden.

Möchten Sie den Inhalt dieser Dateien betrachten, genügt es, wenn Sie die Datei in .ZIP umbenennen und mit einem Entpacker-Programm öffnen:



Das Sichern im RTF-Format erfordert nur unwesentliche Änderungen an obigem Beispielcode. Sie müssen für den Dateidialog nur eine andere Extension vergeben und beim Aufruf der *Save*-Methode die Konstante *DataFormats.rtf* nutzen.

Dass der Export ganz gut funktioniert, sehen Sie an folgendem Beispiel, selbst der Hyperlink arbeitet wie gewünscht:



HINWEIS: Soll nur die aktuelle Auswahl gesichert werden, können Sie sich den Aufwand mit dem *TextRange*-Objekt sparen, die Eigenschaft *Selection* ist bereits ein *TextRange,* den Sie mit der Methode *Save* sichern können.

Beispiel 2.49: Aktuelle Auswahl sichern

C#

```
FileStream fs = (FileStream)dialog.OpenFile();
rtb1.Selection.Save(fs, DataFormats.XamlPackage);
fs.Close();
```

2.16.4 Laden von Dokumenten

Soll das Dokument zu einem späteren Zeitpunkt wieder geladen werden, gehen wir im Grunde den selben Weg. Auch hier wird ein *TextRange*-Objekt erzeugt, in das wir diesmal den Inhalt laden.

Beispiel 2.50: Laden eines Dokuments (XAML-Package)

C#

```
        private void openb_Click(object sender, RoutedEventArgs e)
        {
```

Dateidialog öffnen:

```
            OpenFileDialog dialog = new OpenFileDialog();
            dialog.Filter = "Xaml(*.xaml)|*.xaml";
```

Im Erfolgsfall (Datei ausgewählt):

```
            if((bool)dialog.ShowDialog())
            {
```

Stream erzeugen und Bereich festlegen:

```
                FileStream fs = (FileStream)dialog.OpenFile();
                TextRange tr = new TextRange(rtb1.Document.ContentStart,
                                             rtb1.Document.ContentEnd);
```

Laden:

```
                tr.Load(fs, DataFormats.XamlPackage);
                fs.Close();
            }
        }
```

HINWEIS: Soll das geöffnete Dokument an der aktuellen Kursorposition eingefügt werden, nutzen Sie die *Selection*-Eigenschaft (siehe vorhergehendes Beispiel).

2.16.5 Texte per Code einfügen/modifizieren

Nicht alle Texte werden ausschließlich „von Hand" bearbeitet, viele Funktionen werden in Textverarbeitungen automatisiert. Und so stellt auch die *RichTextBox* reichlich Möglichkeiten zur codegesteuerten Bearbeitung der Texte zur Verfügung.

Beispiel 2.51: Einfügen des Tagesdatums an der aktuellen Kursor-Position

C#

Die aktuelle Position des Kursors können Sie mit *Selection.End* ermitteln, diese Eigenschaft stellt ein *TextPointer*-Objekt zur Verfügung, über das sich unter anderem auch Texte einfügen lassen:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    rtb1.Selection.End.InsertTextInRun(DateTime.Now.ToLongDateString());
}
```

Beispiel 2.52: Einfügen neuer Absatz

C#

Auch hier nutzen wir wieder ein *TextPointer*-Objekt, um die Methode *InsertParagraphBreak* aufzurufen:

```
rtb1.Selection.End.InsertParagraphBreak();
```

Beispiel 2.53: Absatzfarbe ändern

C#

Über ein *TextPointer*-Objekt haben Sie auch Zugriff auf den aktuellen Absatz, d. h., Sie können dessen Eigenschaften bearbeiten:

```
rtb1.Selection.End.Paragraph.Background = Brushes.AliceBlue;
```

Beispiel 2.54: Neuen Absatz mit Fließtext am Dokumentende einfügen

C#

Alle Absätze/Objekte werden über die *Blocks*-Collection verwaltet. Hängen Sie neue Absätze einfach an diese Auflistung an. Allerdings können Sie nicht wie in XAML direkt den Text im *Paragraph*-Objekt ausgeben, sondern Sie müssen einen Fließtext (*Run*) erzeugen und diesem den Text übergeben.

```
rtb1.Document.Blocks.Add(new Paragraph(new Run("Mein neuer Absatz")));
```

Beispiel 2.55: Text vor dem aktuellen Absatz einfügen

C#

Auch hier hilft Ihnen die *Blocks*-Auflistung, allerdings müssen Sie in diesem Fall die *InsertBefore*-Methode aufrufen. Als Referenz ist die aktuelle Position des Textkursors wichtig. Die ermitteln wir, wie schon in den vorhergehenden Beispielen, über das *Selection*-Objekt.

```
rtb1.Document.Blocks.InsertBefore(rtb1.Selection.Start.Paragraph,
                                  new Paragraph(new Run("Neuer Absatz im Text")));
```

2.16.6 Texte formatieren

Sicher ist Ihnen auch schon aufgefallen, dass unsere *RichTextBox* einen entscheidenden Makel hat: es fehlt ein vernünftiger *ToolBar*, der uns zum Beispiel die bekannten Zeichenformatierungfunktionen zur Verfügung stellt.

Beispiel 2.56: *ToolBar*-Funktionalität für „Fett“ und „Kursiv“ realisieren

XAML

Zunächst die Definition des *ToolBar*s, wir verwenden *ToggleButtons,* da es sich um einen Umschaltprozess handelt*:*

```
<Window x:Class="BSP_Controls.RichTextBox_Bsp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="RichTextBox_Bsp" Height="300" Width="481">

 <DockPanel>
   <ToolBarTray DockPanel.Dock="Top">
     <ToolBar >
```

Variante 1: Wir verwenden so genannte Commands, um die entsprechende Funktion (Fett/ NichtFett) zu realisieren. Dazu muss die Schaltfläche mit dem Zielcontrol (*RichtTextBox*) über die Eigenschaft *CommandTarget* verbunden werden. Die eigentliche Funktion wird mit der Eigenschaft *Command* festgelegt:

```
       <ToggleButton Name="boldb" Height="32" Width="32"
                     Command="EditingCommands.ToggleBold"
                     CommandTarget="{Binding ElementName=rtf1}">
         <Image Source="Images/boldhs.png" />
       </ToggleButton>
```

Variante 2: Wir nutzen das *Click*-Ereignis, um die Funktion zu realisieren:

```
       <ToggleButton Name="italicb" Height="32" Width="32" Click="italicb_Click">
         <Image Source="Images/italicHS.png" />
       </ToggleButton>
     </ToolBar>
   </ToolBarTray>
```

Ergebnis

Die erzeugte Werkzeugleiste:

C#

Das *Click*-Ereignis für die zweite Variante:

```
private void italicb_Click(object sender, RoutedEventArgs e)
{
```

Mit *ApplyPropertyValue* können Sie gezielt einzelne Eigenschaften des Textes beeinflussen. Übergeben wird der Name der Eigenschaft und der neue Wert:

```
    rtb1.Selection.ApplyPropertyValue(FlowDocument.FontStyleProperty,
                                      FontStyles.Italic);
}
```

Starten Sie jetzt das Programm, führt ein Klick auf die *Kursiv*-Schaltfläche dazu, dass die entsprechende Formatierung auf den Text angewendet wird. Allerdings beschränkt sich der obige Code darauf, die Formatierung einmalig zu setzen. Ein Löschen ist so nicht möglich. Dazu müssten wir vorher den aktuellen Wert abfragen.

Alternativ bot sich die Variante mit der Verwendung von *Commands* an. Hier ist das Kommando „EditingCommands.ToggleBold“, d. h., jeder Klick führt zum Umschalten des bisherigen Zustands. Auf Quellcode können wir bei dieser Version gänzlich verzichten, damit dürfte diese Version der sinnvollste Weg sein.

Ein Problem haben wir jedoch nach wie vor, bewegen wir den Kursor durch die *RichTextBox,* werden die aktuellen Formatierungen nicht im ToolBar angezeigt. Über das *SelectionChanged*-Ereignis der *RichTextBox* können wir auf die Änderung der Kursorposition reagieren und den Status der Schaltflächen an die aktuelle Formatierung anpassen:

```
private void rtb1_SelectionChanged(object sender, RoutedEventArgs e)
{
```

Zunächst ermitteln wir den aktuellen Formatierungsstatus für Fett:

```
    object propval = rtb1.Selection.GetPropertyValue(FontWeightProperty);
```

Wenn die Eigenschaft bereits gesetzt ist,

```
    if (propval != DependencyProperty.UnsetValue)
```

können wir deren Wert auswerten:

```
        boldb.IsChecked = (FontWeight)propval == FontWeights.Bold;
```

Das Gleiche für Kursiv:

```
    propval = rtb1.Selection.GetPropertyValue(FontStyleProperty);
    if (propval != DependencyProperty.UnsetValue)
       italicb.IsChecked = (FontStyle)propval == FontStyles.Italic;
}
```

HINWEIS: Obige Auswertung ist leider recht code-intensiv, da müssen sich die Microsoft-Programmierer sicherlich noch etwas Sinnvolleres einfallen lassen.

2.16.7 EditingCommands

Wie im vorhergehenden Beispiel gezeigt, ist mit der Verwendung von Commands eine recht einfache Möglichkeit gegeben, Aktoren (Schaltflächen etc.) mit Aktionen (*Command*) zu verknüpfen, die auf ein spezifisches Control wirken (*CommandTarget*).

HINWEIS: Mehr zu Commands im Abschnitt 3.4, wir wollen hier nicht zu viel vorgreifen.

Welche Commands Ihnen für die *RichTextBox* zur Verfügung stehen, soll die folgende Auflistung zeigen (die Bezeichner dürften selbsterklärend sein):

- **Cursorbewegung**
  (*Backspace, MoveDownByLine, MoveDownByPage, MoveDownByParagraph, MoveLeftByCharacter, MoveLeftByWord, MoveRightByCharacter, MoveRightByWord, MoveToLineEnd, MoveToLineStart, MoveToDocumentEnd, MoveToDocumentStart, MoveUpByLine, MoveUpByPage, MoveUpByParagraph, TabBackward, TabForward*)
- **Markierung**
  (*SelectDownByLine, SelectDownByPage, SelectDownByParagraph, SelectLeftByCharacter, SelectLeftByWord, SelectRightByCharacter, SelectRightByWord, SelectToDocumentEnd, SelectToDocumentStart, SelectToLineEnd, SelectToLineStart, SelectUpByLine, SelectUpByPage, SelectUpByParagraph*)
- **Schrift**
  (*DecreaseFontSize, IncreaseFontSize, ToggleBold, ToggleItalic, ToggleSubscript, ToggleSuperscript, ToggleUnderline*)
- **Absatzausrichtung**
  (*AlignCenter, AlignJustify, AlignLeft, AlignRight*)
- **Absatzformat**
  (*ToggleBullets, ToggleNumbering, DecreaseIndentation, IncreaseIndentation*)
- **Rechtschreibung**
  (*CorrectSpellingError, IgnoreSpellingError*)
- **Befehle**
  (*Delete, DeleteNextWord, DeletePreviousWord, EnterLineBreak, EnterParagraphBreak*)
- **Sonstiges**
  (*ToggleInsert*)

Alternativ sollten Sie auch einen Blick auf folgende Library werfen, diese bietet einen *RichTextBoxFormatBar*, der die wichtigsten Formatierungsoptionen anbietet: *http://wpftoolkit.codeplex.com.*

2.16.8 Grafiken/Objekte einfügen

Dass neben Text auch andere Objekte in die *RichTextBox* eingefügt werden können, soll das folgende Beispiel zeigen, bei dem zur Laufzeit eine Grafik aus den Programmressourcen eingefügt und mit einem Ereignis verknüpft wird.

Beispiel 2.57: Grafik einfügen

C#

```
        private void imageb_Click(object sender, RoutedEventArgs e)
        {
```

Neues *Image*-Objekt erzeugen:

```
            Image img = new Image();
```

Die eigentliche Bitmap zuweisen:

```
            img.Source = BitmapFrame.Create(new
                                Uri("pack://application:,,,/images/frosch.gif"));
```

Größe bestimmen und Ereignis zuweisen:

```
            img.Height = 100;
            img.Width = 100;
            img.MouseDown += delegate {MessageBox.Show("Finger weg!");};
```

Und jetzt Grafik als neuen letzten Absatz einfügen:

```
            rtb1.Document.Blocks.Add(new Paragraph( new InlineUIContainer(img)));
        }
```

Controls wie *Image* etc. müssen in einem so genannten *InlineUIContainer* gekapselt werden, bevor sie im *Paragraph* eingefügt werden.

Ergebnis

Das Beispielprogramm in Aktion:

Sichern Sie jetzt das Dokument, wird die Grafik ebenfalls mit abgespeichert (z. B. in einem XAML-Package).

2.16.9 Rechtschreibkontrolle

Dass Sie die Rechtschreibkontrolle der *RichTextBox* mit

```
<RichTextBox Name="rtb1" SpellCheck.IsEnabled="True" ...
```

aktivieren können, dürfte Ihnen bereits von der einfachen *TextBox* her bekannt sein. Doch ein Blick in den Editor wird für lange Gesichert sorgen: alles Fehler und das trotz korrekter Schreibweise! Hier hilft das zusätzliche Setzen der Sprache:

```
<RichTextBox Name="rtb1" SpellCheck.IsEnabled="True" Language="de" ...
```

2.17 FlowDocumentPageViewer & Co.

Neben dem Editieren von formatierten Texten steht häufig auch deren Anzeige auf der Wunschliste des Programmierers. Neben der bereits vorgestellten *RichTextBox* bietet WPF hier mit

- *FlowDocumentPageViewer*,
- *FlowDocumentReader* und
- *FlowDocumentScrollViewer*

ein reichhaltiges Arsenal, um Flow-Dokumente anzuzeigen.

2.17.1 FlowDocumentPageViewer

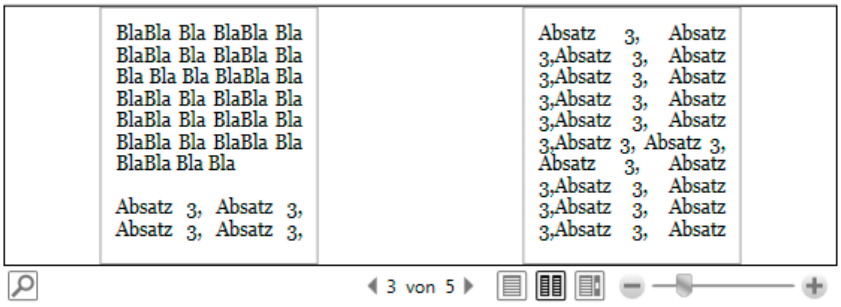
Für die seiten-orientierte Anzeige von Dokumenten bietet sich das *FlowDocumentPageViewer*-Control an. Über die Navigationstasten am unteren Rand können Sie zwischen den Seiten blättern, der Zoomfaktor lässt sich über einen Schieberegler vorgeben:

HINWEIS: Das Textfeld für die Suchfunktion blenden Sie über die Tastenkombination *Strg+F* oder über die Methode *Find* ein.

2.17.2 FlowDocumentReader

Im Gegensatz zum vorhergehenden Control können Sie beim *FlowDocumentReader* zwischen verschiedenen Anzeigemodi wechseln (ein- und mehrseitig, sowie fortlaufend), die Textsuchfunktion steht per eigener Schaltfläche zur Verfügung.

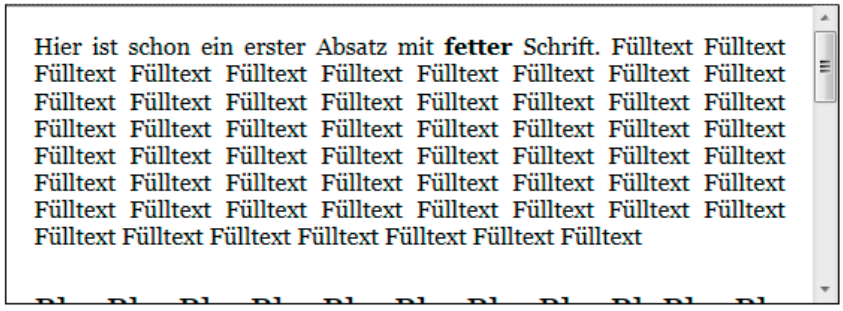
Zoom- und Blätter-Funktion entsprechen dem *FlowDocumentPageViewer*.



Durch die Unterstützung der verschiedenen Anzeigemodi ist dieser Viewer zwar leistungsfähiger, aber leider auch etwas langsamer.

2.17.3 FlowDocumentScrollViewer

Für die reine Fließtextdarstellung ohne Seiten sollten Sie den *FlowDocumentScrollViewer* einsetzen. Dieser verfügt lediglich über einen Scrollbar, um im Text zu blättern, der Zoomfaktor ist nur per Code (*Zoom*) einstellbar:



2.18 FlowDocument

Obwohl es eigentlich kein eigenes Control ist, wollen wir dennoch kurz auf das *FlowDocument* eingehen, verwenden wir dieses doch in der *RichTextBox* bzw. in den *FlowDocumentPageViewer, -Reader, -ScrollViewer*-Controls.

Ein *FlowDocument* stellt das Objektmodell eines formatierten Fließtextes dar, der in den o. g. Controls in unterschiedlichen Modi angezeigt oder auch editiert werden kann.

FlowDocument-Objekte können folgende untergeordneten Elemente enthalten:

- *Paragraph* (Absätze, die wiederum Texte, Floater, Figures etc. enthalten können)
- *BlockUIContainer* (Kapseln von Controls, z. B. *Image*)
- *List* (eine Auflistung)
- *Section* (Zusammenfassung von Abschnitten)
- *Table* (eine Tabelle)

2.18.1 FlowDocument per XAML beschreiben

Zunächst wollen wir Ihnen an einem Beispiel die Beschreibung des *FlowDocuments* per XAML-Code demonstrieren, dieses können Sie direkt als *Content* einem der obigen Reader zuweisen:

Beispiel 2.58: Ein etwas umfangreicheres *FlowDocument*, das einige Möglichkeiten aufzeigt.

XAML

Zunächst das *FlowDocument* definieren (minimale Spaltenbreite, automatische Silbentrennung)

```
<FlowDocument ColumnWidth="400" IsHyphenationEnabled="True">
```

Für die folgenden Abschnitte gilt eine Schriftgröße von 12:

```
  <Section FontSize="12">
```

Ein erster Abschnitt mit Fließtext und verschiedenen Formatierungen:

```
    <Paragraph>
      <Bold>Fette Schrift</Bold> Hier steht Fließtext. Hier steht Fließtext.
      Hier steht Fließtext. <Underline>Hier steht Fließtext.</Underline> Hier steht
      Fließtext. Hier steht Fließtext.
      Hier steht Fließtext. Hier steht Fließtext. Hier steht Fließtext. Hier
steht Fließtext.
```

Ein frei positionierbarer Rahmen, in dem wiederum andere Elemente (in diesem Fall ein Absatz) enthalten sein können:

```
          <Figure Width="150" Height="100" Background="CornflowerBlue"
              HorizontalAnchor="PageLeft" HorizontalOffset="100"
 VerticalOffset="20">
            <Paragraph FontStyle="Italic" Foreground="White">
              Ein freier Bereich, der über Koordinatenangaben positioniert wird.
            </Paragraph>
          </Figure>
```

Auch dies ein Rahmen, der jedoch nicht absolut positioniert werden kann:

```
          <Floater Background="LightYellow" Width="300" HorizontalAlignment="Right">
            <Paragraph>
              Noch ein freier Bereich, der horizontal über HorizontalAlignment
 positioniert
              werden kann.
            </Paragraph>
          </Floater>
        </Paragraph>
```

Ein normaler Absatz:

```
        <Paragraph>
          Bla Bla Bla Bla Bla Bla Bla Bla Bla Bla Bla Bla Bla Bla Bla  ...
        </Paragraph>
```

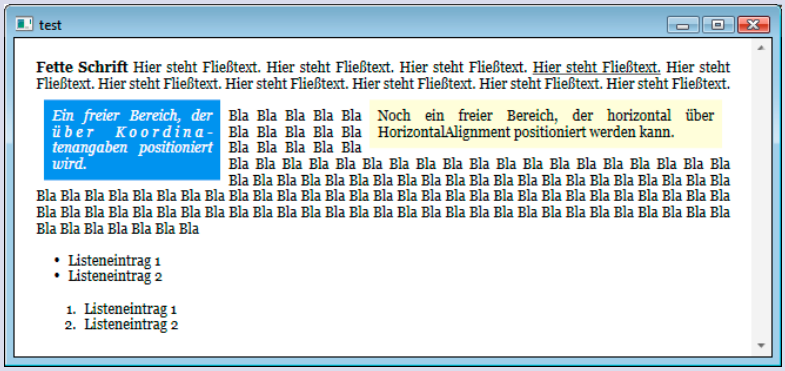
Eine Liste:

```
        <List MarkerStyle="Disc">
          <ListItem>
            <Paragraph>Listeneintrag 1</Paragraph>
          </ListItem>
          <ListItem>
            <Paragraph>Listeneintrag 2</Paragraph>
          </ListItem>
        </List>
```

Eine Liste mit Aufzählung:

```
        <List MarkerStyle="Decimal">
          <ListItem>
            <Paragraph>Listeneintrag 1</Paragraph>
          </ListItem>
          <ListItem>
            <Paragraph>Listeneintrag 2</Paragraph>
          </ListItem>
        </List>
 ...
      </Section>
    </FlowDocument>
```

Ergebnis



2.18.2 FlowDocument per Code erstellen

Neben der recht übersichtlichen Möglichkeit, Flow-Dokumente per XAML-Code zu erstellen (zum Beispiel auch direkt aus der *RichTextBox* heraus), können Sie natürlich auch Ihre Programmierfähigkeiten zum Einsatz bringen. Ein einfaches Beispiel zeigt die Vorgehensweise:

Beispiel 2.59: Ein *FlowDocument* per Code erstellen

C#

Fügen Sie in die Oberfläche zunächst einen *FlowDocumentPageViewer* ein und geben Sie diesem den Bezeichner „FlowDocumentPageViewer1“.

Mit dem Laden des Fensters schreiten wir zur Tat:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
```

Eine *FlowDocument*-Instanz erzeugen:

```
    FlowDocument flowDoc = new FlowDocument();
```

Einen ersten Absatz hinzufügen, dieser enthält zunächst einen einfachen Text der fett ausgeben wird:

```
    Paragraph para = new Paragraph(new Bold(
                     new Run("Wir schreiben etwas Text in den ersten
Absatz.")));
```

An den bestehenden Absatz hängen wir noch etwas Text an:

```
    para.Inlines.Add(new Run(" Hier kommt noch mehr Text im selben
Absatz"));
```

Den Absatz an das *FlowDocument* anhängen:

```
            flowDoc.Blocks.Add(para);
```

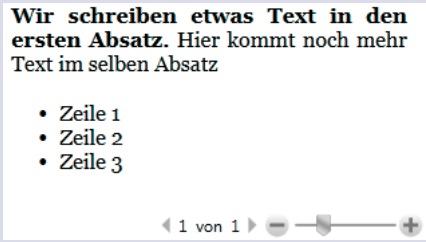
Eine Liste erzeugen:

```
            List liste = new List();
            liste.ListItems.Add(new ListItem(new Paragraph(new Run("Zeile 1"))));
            liste.ListItems.Add(new ListItem(new Paragraph(new Run("Zeile 2"))));
            liste.ListItems.Add(new ListItem(new Paragraph(new Run("Zeile 3"))));
            flowDoc.Blocks.Add(liste);
```

Das *FlowDocument* zur Anzeige bringen:

```
            FlowDocumentPageViewer1.Document = flowDoc;
        }
```

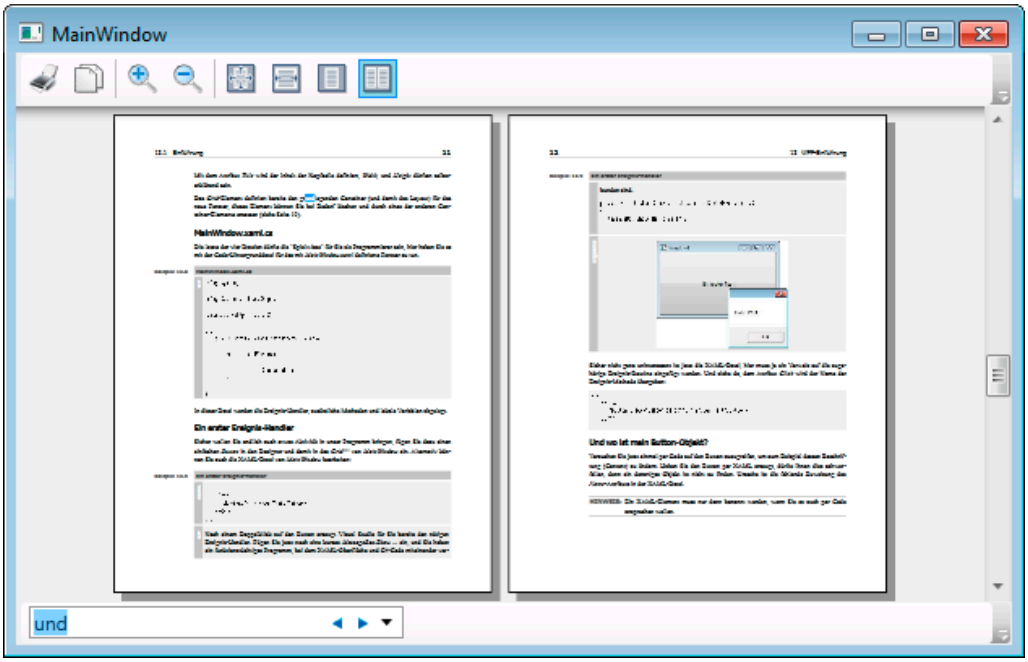
Ergebnis



HINWEIS: Auf weitere Möglichkeiten (Controls, Tabellen etc.) von Flow-Dokumenten wollen wir an dieser Stelle nicht eingehen, das überlassen wir besser der Spezialliteratur.

2.19 DocumentViewer

Neben den auf die Flow-Dokumente festgelegten Controls findet sich auch ein *Document Viewer*-Control, das ausschließlich zur Anzeige von XPS-Dokumenten verwendet werden kann. Neben einer Druckoption können Sie die Daten auch in die Zwischenablage kopieren, die Ansicht skalieren und zwischen unterschiedlichen Seitendarstellungen wechseln. Last, but not least, verfügt das Control auch über ein einfache Suchfunktion innerhalb des Textes.



Wie Sie ein externes XPS-Dokument laden, zeigt das folgende Beispiel:

Beispiel 2.60: Laden eines XPS-Dokuments in den *DocumentViewer*

XAML

```
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="MainWindow" Height="238" Width="348" Loaded="Window_Loaded"
        WindowStartupLocation="CenterScreen">
        <DocumentViewer Name="DocumentViewer1" />
</Window>
```

C#

Fügen Sie zunächst die Assembly *ReachFramework.dll* als Verweis zu Ihrem Projekt hinzu. Nachfolgend können Sie den entsprechenden Ereigniscode übernehmen.

Namespaces importieren:

```
...
using System.Windows.Xps.Packaging;
using System.IO;

namespace WpfApplication2
{
    public partial class MainWindow : Window
    {
...
```

```
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
```

Zunächst ein *XpsDocument* aus der Datei „c:\Test.xps“ erstellen, nachfolgend können wir eine *FixedDocumentSequence* für die Anzeige abrufen:

```
            XpsDocument doc = new XpsDocument("c:\\test.xps", FileAccess.Read);
            DocumentViewer1.Document = doc.GetFixedDocumentSequence();
        }
```

HINWEIS: Mehr zu diesem Thema finden Sie im Abschnitt 5.1.1, wo wir uns sowohl dem Erstellen von XPS-Dokumenten als auch der Verwendung der *DocumentViewer*-Komponente zuwenden werden.

2.20 Expander, TabControl

Gerade Dialogfenster mit vielen Eingabefeldern/Optionen leiden unter immer demselben Problem: es ist zu wenig Platz vorhanden. Diesem Missstand sollen das *Expander*- und das *TabControl* abhelfen.

2.20.1 Expander

Das *Expander*-Control ermöglicht es, zwischen einem aufgeklappten und einem geschlossenen Zustand hin- und herzuschalten. Die Höhe des Controls im aufgeklappten Zustand bestimmt sich aus der Höhe des enthaltenen Layout-Containers bzw. des enthaltenen Controls.

Sichtbar bleibt in jedem Fall die per *Header* definierte Beschriftung und eine Schaltfläche, mit der Sie das Control aufklappen können. Die Aufklapprichtung wird mit *ExpandedDirection* (*Down, Left, Right, Up*) definiert.

Beispiel 2.61: Verwendung Expander

XAML

Zunächst ordnen wir alle drei *Expander* in einem *StackPanel* an:

```
<StackPanel>
```

Die Beschriftung, Hintergrundfarbe und die Aufklapprichtung festlegen:

```
    <Expander Header="Private Daten" Background="LightGray" ExpandDirection="Right" >
```

Der Inhalt des Expanders ist z. B. ein weiteres *StackPanel*, mit dem die Eingabefelder beschriftet bzw. angeordnet werden:

```
      <StackPanel Margin="5">
        <Label Content="Vorname:" />
        <TextBox></TextBox>
        <Label Content="Nachname:"/>
        <TextBox></TextBox>
```

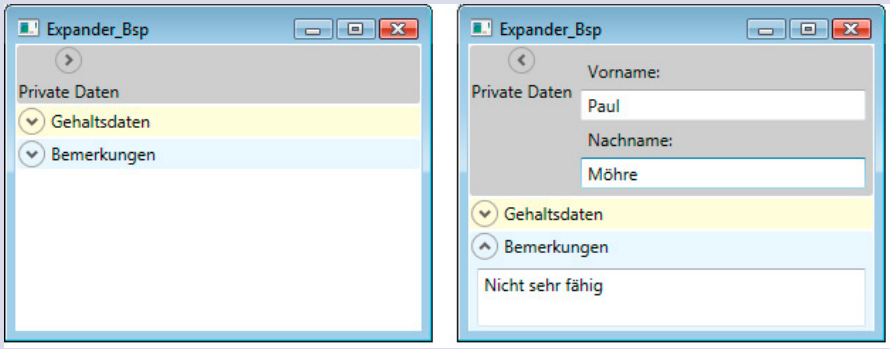
Die Höhe des Expanders wird durch die Summe der einzelnen Controls bestimmt:

```
      </StackPanel>
    </Expander>
    <Expander Header="Gehaltsdaten" Background="LightYellow">
      <StackPanel Margin="5">
        <Label Content="Gehalt:" />
        <TextBox></TextBox>
        <Label Content="Steuerklasse:"/>
        <TextBox></TextBox>
      </StackPanel>
    </Expander>
```

Hier blenden wir statt eines Layout-Controls gleich eine *TextBox* im *Expander* ein und legen die Höhe der *TextBox* entsprechend der gewünschten Höhe fest:

```
    Expander Header="Bemerkungen" Background="AliceBlue">
      <TextBox Margin="5" Height="150">
      </TextBox>
    </Expander>
  </StackPanel>
```

Ergebnis



HINWEIS: Über die *IsExpanded*-Eigenschaft können Sie per Code den aktuellen Zustand abfragen bzw. beeinflussen (z. B. Schließen bei Fokusverlust).

2.20.2 TabControl

Statt wie beim *Expander* einzelne Bereich einzublenden, werden beim *TabControl* die einzelnen Registerkarten (*TabItem*s) komplett umgeschaltet, es ist also immer nur eine Seite sichtbar. Wo das Register angezeigt wird, bestimmen Sie mit der Eigenschaft *TabStripPlacement* (*Left*, *Right*, *Top*, *Bottom*). Die Auswahl des aktiven *TabItem*s erfolgt mit der Maus oder über die Eigenschaften *SelectedIndex* bzw. *SelectedItem.* Alternativ können Sie mit *IsSelected* den Status eines *TabItem*s abfragen.

Beispiel 2.62: Ein *TabControl* mit drei *TabItems*

XAML

```
<Window x:Class="BSP_Controls.Tab_Bsp" ...>
```

Das *TabControl*:

```
  <TabControl>
```

Und hier auch schon die erste Registerkarte mit einfacher Beschriftung:

```
    <TabItem Header="Tabseite 1">
      Hier steht der Content!
    </TabItem>
```

Die zweite Registerkarte mit einem Layout-Control im Content:

```
    <TabItem Header="Tabseite 2">
      <StackPanel>
        <Label>Beliebige Controls ...</Label>
      </StackPanel>
    </TabItem>
```

Die dritte Registerkarte macht von der Möglichkeit Gebrauch, im Header auch Grafiken bzw. beliebige Controls anzuzeigen:

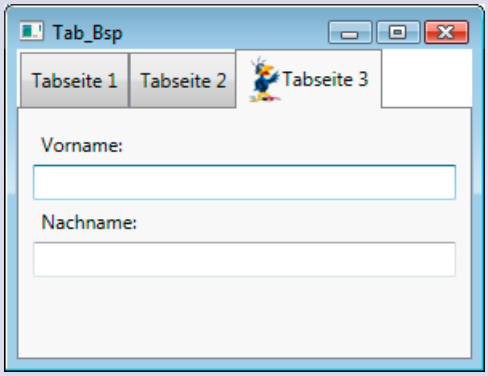
```
    <TabItem>
      <TabItem.Header>
        <StackPanel Orientation="Horizontal">
          <Image Source="Images/rudi.gif" Height="30"/>
          <TextBlock  VerticalAlignment="Center">Tabseite 3</TextBlock>
        </StackPanel>
      </TabItem.Header>
```

Der Content besteht wiederum aus einem Layout-Control und den darin enthaltenen Controls:

```
      <StackPanel Margin="5">
        <Label Content="Vorname:" />
        <TextBox></TextBox>
        <Label Content="Nachname:"/>
        <TextBox></TextBox>
      </StackPanel>
    </TabItem>
  </TabControl>
</Window>
```

Ergebnis

Das *TabControl* zur Laufzeit:



HINWEIS: Möchten Sie per Programm auf Änderungen reagieren, nutzen Sie das *SelectionChanged*-Ereignis für die Auswertung.

2.21 Popup

Für die Anzeige von Zusatzinformationen können Sie den *ToolTip* eines Controls verwenden. Dieser hat allerdings den Nachteil, dass er einem spezifischen Control zugeordnet ist. Anders das Popup-Fenster (ja es handelt sich um ein eigenes Fenster!), das zwar nicht automatisch eingeblendet wird, dafür aber an jeder beliebigen Stelle stehen kann.

HINWEIS: Einmal eingeblendet, bleibt das Fenster an der aktuellen Position stehen, auch wenn das übergeordnete Fenster verschoben wird.

Beispiel 2.63: Einfaches *Popup* einblenden

XAML

```
...
 <Canvas>
   <Button Canvas.Left="206" Canvas.Top="37" Height="30" Name="button1" Width="43"
           Click="button1_Click">...</Button>
   <Popup Name="pop2" Placement="MousePoint" VerticalOffset="25"
HorizontalOffset="25"
          Width="100">
     <TextBlock Height="50" Margin="1" TextWrapping="Wrap" Background="LightBlue" >
        Bla Bla Bla ...
     </TextBlock>
 </Popup>
```

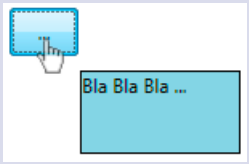
C#

Das Ein-/Ausblenden des Popup-Fensters wird mit der Eigenschaft *IsOpen* realisiert:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    pop2.IsOpen = !pop2.IsOpen;
}
```

Ergebnis

Klicken Sie zur Laufzeit auf den Button, wird das Popup-Fenster relativ zur aktuellen Cursorpostion (siehe *Placement*) eingeblendet:



Weitere Möglichkeiten zum Platzieren bieten sich mit den folgenden Eigenschaften:

- *Placement* (*Absolute, Bottom, Mouse, Relative* ...)
- *PlacementTarget* (ein Control, auf das sich relative Angaben beziehen)
- *PlacementRectangle* (optionales Rechteck, zu dem das Popup relativ angezeigt wird)
- *HorizontalOffset* (zusätzliche horizontale Verschiebung der Koodinaten)
- *VerticalOffset* (zusätzliche vertikale Verschiebung der Koordinaten)

Neben der Positionierung bietet sich auch die Möglichkeit, das Popup-Fenster animiert einzublenden. Setzen Sie dazu die Eigenschaft *PopupAnimation* auf *Fade*, *Scroll* oder *Slide.*

HINWEIS: Damit die Animation auch sichtbar ist, muss *AllowsTransparency* auf *True* gesetzt werden.

Beispiel 2.64: Weitere Möglichkeiten für die Konfiguration des *Popup*

XAML

Relative Platzierung zu einer *TextBox*, automatisches Einblenden mit Animation (wenn die *TextBox* den Fokus erhält) bzw. Ausblenden (wenn Fokusverlust).

```
...
   <TextBox Canvas.Left="28" Canvas.Top="37" Height="31" Name="textBox1"
Width="168"/>
   <Popup Name="pop1" Width="180"
```

Das Bezugselement für die Koordinatenangaben festlegen:

```
PlacementTarget="{Binding ElementName=textBox1}" Placement="Relative"
```

Die relative Platzierung:

```
VerticalOffset="35" HorizontalOffset="130"
```

Die Animation festlegen:

```
PopupAnimation="Slide" AllowsTransparency="True"
```

Mittels Datenbindung an die *IsFocused*-Eigenschaft der *TextBox* öffnen/schließen wir das Popup-Fenster:

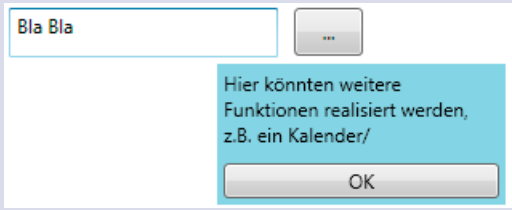
```
IsOpen="{Binding ElementName=textBox1, Path=IsFocused, Mode=OneWay}">
```

Hier folgen die Inhalte des Popup-Fensters:

```
<StackPanel Background="LightBlue">
  <TextBlock Height="50" Margin="4" TextWrapping="Wrap" Background="LightBlue" >
      Hier könnten weitere Funktionen realisiert werden, z.B. ein
      Kalender/Taschenrechner
   </TextBlock>
  <Button Margin="4">OK</Button>
</StackPanel>
</Popup>
```

Ergebnis

Nachdem die *TextBox* den Fokus erhalten hat, dürfte das Popup-Fenster eingeblendet werden:



2.22 TreeView

Sicher jedem vom Explorer her bekannt, darf auch in WPF ein *TreeView*-Control nicht fehlen. Allerdings könnte die Verwendung dieses Controls viele gestandene Windows-Forms und Win32-Programmierer schnell in den Wahnsinn treiben, unterscheidet sich doch die Programmierung in vielen Punkten vom bisherigen Vorgehen. Dies ist vor allem der freien Programmierbarkeit dieses Controls geschuldet. Was Sie in den einzelnen *TreeViewItems* anzeigen ist, wie in WPF üblich, von Ihnen frei definierbar.

Doch ganz so kompliziert wollen wir nicht anfangen, ein einfaches Beispiel zeigt zunächst, wie Sie eine rein textorientierte *TreeView* per XAML-Code definieren:

Beispiel 2.65: Grundprinzip der Schachtelung der *TreeViewItem*-Elemente

XAML

Definition der eigentlichen *TreeView*:

```
<TreeView>
```

Der erste Knoten, über die *Header*-Eigenschaft bestimmen Sie die Beschriftung:

```
  <TreeViewItem Header ="Root" IsExpanded="True">
```

Unter-Knoten schachteln Sie einfach:

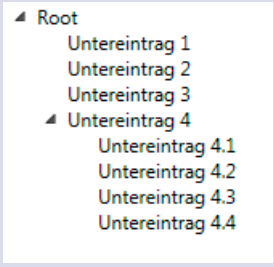
```
    <TreeViewItem Header ="Untereintrag 1" IsExpanded="True"/>
    <TreeViewItem Header ="Untereintrag 2" IsExpanded="True"/>
    <TreeViewItem Header ="Untereintrag 3" IsExpanded="True"/>
    <TreeViewItem Header ="Untereintrag 4" IsExpanded="True">
```

Eine weitere Unterebene:

```
      <TreeViewItem Header ="Untereintrag 4.1" IsExpanded="True"/>
      <TreeViewItem Header ="Untereintrag 4.2" IsExpanded="True"/>
      <TreeViewItem Header ="Untereintrag 4.3" IsExpanded="True"/>
      <TreeViewItem Header ="Untereintrag 4.4" IsExpanded="True"/>
    </TreeViewItem>
  </TreeViewItem>
</TreeView>
```

Ergebnis

Die Darstellung der so definierten *TreeView*:



Beispiel 2.66: Alternativ können Sie die *TreeView* auch per Code füllen

C#

```
        private void Btn1_Click(object sender, RoutedEventArgs e)
        {
            TreeViewItem tvi;
```

Bisherige Inhalte löschen:

```
            Tv1.Items.Clear();
```

Den ersten Eintrag erzeugen und weitere Untereinträge hinzufügen:

```
            tvi = new TreeViewItem { Header = "Root" };
            tvi.Items.Add(new TreeViewItem { Header = "Untereintrag 1" });
            tvi.Items.Add(new TreeViewItem { Header = "Untereintrag 2" });
            tvi.Items.Add(new TreeViewItem { Header = "Untereintrag 3" });
            tvi.Items.Add(new TreeViewItem { Header = "Untereintrag 4" });
            Tv1.Items.Add(tvi);
        }
```

Was ist daran so kompliziert, werden Sie sicher nach den bisherigen Beispielen fragen. Die Antwort kommt spätestens bei der Aufgabenstellung, Grafiken in die Einträge einzufügen.

Beispiel 2.67: Frei definierte *TreeViewItems* mit Grafiken bzw. mit Schaltfläche

XAML

```
    <TreeView>
```

Hier nochmal ein ganz „normaler“ Knoten:

```
      <TreeViewItem Header ="Root" IsExpanded="True" Tag="0">
```

Und jetzt wird es kompliziert:

```
        <TreeViewItem>
```

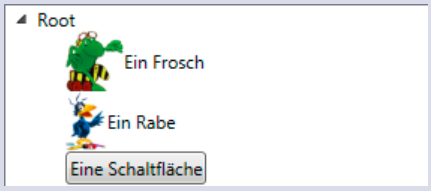
Eine Image-Eigenschafte gibt es nicht, stattdessen erzeugen wir den kompletten Eintrag per *StackPanel*, das wiederum aus einem *Image* und einem *TextBlock* besteht:

```
          <TreeViewItem.Header>
            <StackPanel Orientation="Horizontal">
              <Image Source="images\frosch.gif" Height="40"/>
              <TextBlock Text="Ein Frosch" VerticalAlignment="Center"/>
            </StackPanel>
          </TreeViewItem.Header>
        </TreeViewItem>
        <TreeViewItem>
          <TreeViewItem.Header>
            <StackPanel Orientation="Horizontal">
              <Image Source="images\rudi.gif" Height="40"/>
              <TextBlock Text="Ein Rabe" VerticalAlignment="Center"/>
            </StackPanel>
          </TreeViewItem.Header>
        </TreeViewItem>
```

Dass es auch mit ganz beliebigen Controls geht, zeigt die folgende Variante, bei der wir einen *Button* im *TreeViewItem* einblenden:

```
        <TreeViewItem>
          <TreeViewItem.Header>
            <Button>Eine Schaltfläche</Button>
          </TreeViewItem.Header>
        </TreeViewItem>
      </TreeViewItem>
    </TreeView>
```

Ergebnis



Nach all den „Oberflächlichkeiten“ wollen wir uns jetzt den „inneren Werten“ des Controls zuwenden:

- Den aktuellen Knotenzustand können Sie mit *IsExpanded* abfragen oder setzen.
- Der aktuell gewählte Eintrag lässt sich über die Eigenschaft *SelectedItem* ermitteln, alternativ können Sie auch die Liste der Einträge durchlaufen und die *IsSelected*-Eigenschaft abfragen.
- Auf Änderungen der Auswahl können Sie mit *SelectedItemChanged* bzw. *Selected* (für *TreeViewItem*) reagieren.
- Um bei Bedarf Scrollbars einzublenden, sollten Sie *Width* und *Height* der *TreeView* explizit setzen.
- Mit der Methode *BringIntoView* können Sie gezielt einen *TreeViewItem* einblenden, d. h., bei Bedarf werden Knoten geöffnet und die *TreeView* scrollt den gewünschten Eintrag in den sichtbaren Bereich.
- Nutzen Sie die *Tag*-Eigenschaft, um einzelnen *TreeViewItem*s zusätzliche Informationen zuzuordnen, oder erzeugen Sie gleich eine neue Klasse, die Sie von *TreeViewItem* ableiten und um zusätzliche Eigenschaften und Methoden bereichern.

HINWEIS: Um der Thematik „DataBinding“ in Kapitel 4 nicht vorzugreifen, verzichten wir an dieser Stelle auf weitere Erläuterungen.

2.23 ListView

Die *ListView* als Ableitung der schon vorgestellten *ListBox* zeigt zunächst das gleiche Verhalten, wenn Sie einem *ListView*-Element weitere *ListViewItems* hinzufügen.

Beispiel 2.68: *ListView* mit *ListViewItems*

XAML

```
<ListView>
  <ListViewItem>Zeile 1</ListViewItem>
  <ListViewItem>Zeile 2</ListViewItem>
  <ListViewItem>Zeile 3</ListViewItem>
</ListView>
```

Ergebnis

Die erzeugte Ansicht dürfte Sie nicht überraschen:

Doch im Gegensatz zur einfachen *ListBox* lassen sich für einen *ListView* so genannte Ansichten (Views) definieren, die auch recht einfach austauschbar sind. Eine dieser Ansichten ist die von WPF bereits vordefinierte *GridView,* die eine Collection in Tabellenform darstellen kann.

HINWEIS: Die einzelnen Elemente innerhalb diese Ansicht werden nicht per XAML/Code, sondern per Datenbindung zugeordnet.

Beispiel 2.69: Prinzip der Datenbindung in einem *ListView*-Control

XAML

Zunächst die Datenquelle zuordnen:

```
<ListView Name="lv1" ItemTemplate="{DynamicResource CustomerTemplate}"
          ItemsSource="{Binding Path=Table}">
```

Hier wird die spezielle View, in diesem Fall die *GridView*, definiert:

```
  <ListView.View>
    <GridView>
```

Die Spalten der *GridView* mit den Spalten der obigen Datenquelle verbinden:

```
      <GridViewColumn Header="Id" DisplayMemberBinding="{Binding Path=Id}"/>
      <GridViewColumn Header="Vorname" DisplayMemberBinding="{Binding
              Path=VName}"/>
      <GridViewColumn Header="Nachname" DisplayMemberBinding="{Binding
              Path=NName}"/>
    </GridView>
  </ListView.View>
</ListView>
```

C#

Per Code müssen Sie zur Laufzeit noch die *Table* eines *DataSet*s zuweisen:

```
lv1.DataContext = ds.Tables[0].DefaultView;
```

2.24 DataGrid

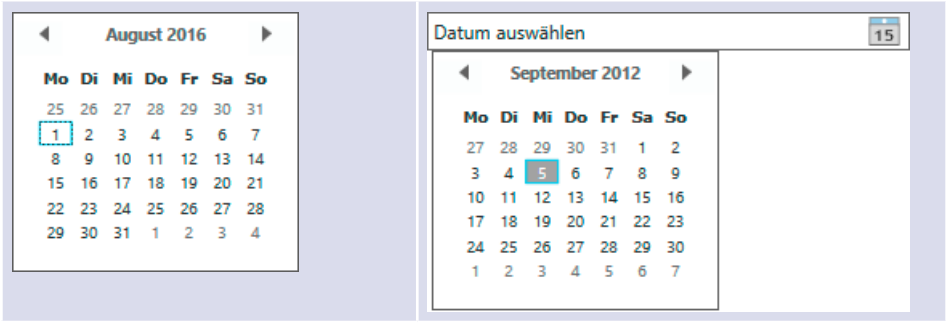
Welcher Programmierer hat es bisher nicht vermisst? Für viele war das Fehlen eines leistungsfähigen *DataGrid*s, das im Gegensatz zur *ListView* auch das Editieren der Daten beherrscht, ein entscheidender Faktor für das Festhalten an „alten" Windows Forms-Anwendungen. Doch das Warten hat sich gelohnt, das früher nur im zusätzlichen WPF-Toolkit enthaltene *DataGrid* findet sich seit der Framework-Version 4.0 auch im WPF-Werkzeugkasten.

HINWEIS: Da das Control mit einem großem Funktionsumfang aufwarten kann, beschränken wir uns im Rahmen dieses Buchs auf einige praktische Aufgabenstellungen, die wir aber aus naheliegenden Gründen in Abschnitt 5.8 untergebracht haben.

2.25 Calendar/DatePicker

Seit WPF 4 gibt es auch zwei Controls für die Anzeige/Eingabe von Kalenderdaten. Während der *Calendar* für die Anzeige von Jahres- bzw. Monatsübersichten geeignet ist, steht der *DatePicker* lediglich für die Eingabe eines Datums zur Verfügung.

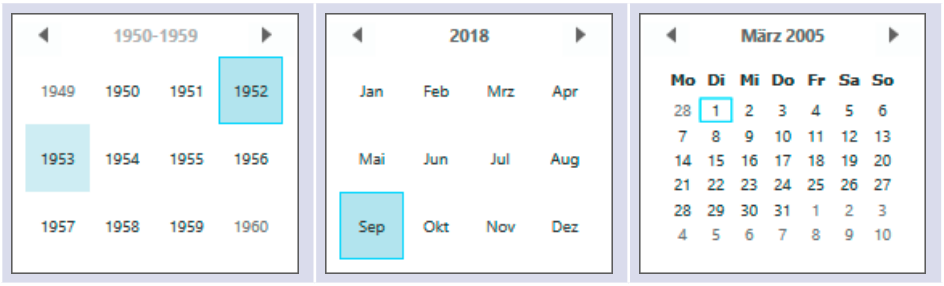
Die beiden Controls:



Einige Anwendungsbeispiele zeigen Ihnen die Verwendung.

DisplayMode

Wie schon erwähnt, lässt das *Calendar*-Control über die *DisplayMode*-Eigenschaft unterschiedliche Anzeigemodi (*Decade, Year, Month*) zu:



DisplayDate, SelectedDate und SelectedDates

Diese wohl wichtigsten Eigenschaften geben Auskunft über die aktuelle Auswahl im jeweiligen Control:

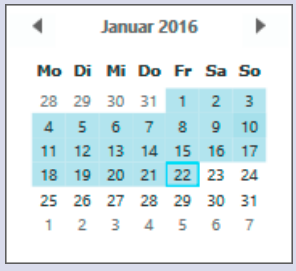
- Bei *DisplayDate* handelt es sich um den Anzeigewert des Controls nach dem Start. Dies ist meist das aktuelle Datum.
- *SelectedDate* entspricht der Benutzerauswahl, es handelt sich um **einen** Datumswert (Voraussetzung ist der *SingleDate*-Auswahlmodus).
- Die *SelectedDates*-Collection können Sie nutzen, wenn das *Calendar*-Control eine Mehrfachauswahl zulässt.

Auswahlmodi

Für den *DatePicker* steht die Frage nicht, hier können Sie immer nur ein einziges Datum wählen, wohingegen der *Calendar* mittels *SelectionMode*-Eigenschaft zwischen folgenden Modi unterscheiden kann:

- *None* (nur Anzeige von *DisplayDate*)
- *SingleDate* (Einzeldatum; per *SelectedDate* auslesen)
- *SingleRange* (Datumsbereich; per *SelectedDates* auslesen) und
- *MultipleRange* (mehrere Datumsbereiche; per *SelectedDates* auslesen)

SingleRange:



MultipleRange:

Die Auswahl der Bereiche per XAML-Code erfordert etwas Schreibarbeit:

Beispiel 2.70: Bereichsauswahl per XAML-Code

XAML

```
<Window x:Class="BSP_Controls.Calendar_Bsp"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="Calendar_Bsp" Height="300" Width="300">
    <StackPanel>
        <Calendar Name="Calendar1" DisplayMode="Month"
SelectionMode="MultipleRange">
            <Calendar.SelectedDates>
                <sys:DateTime>2/15/2020</sys:DateTime>
                <sys:DateTime>2/16/2020</sys:DateTime>
                <sys:DateTime>2/18/2020</sys:DateTime>
            </Calendar.SelectedDates>
        </Calendar>
```

Ergebnis

Der Nutzer hat jetzt die Möglichkeit, diese Bereiche zu belassen oder zu erweitern. Die Abfrage der Bereiche erfolgt per *SelectedDates*-Collection:

Beispiel 2.71: Abfrage der gewählten Datumswerte

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    foreach (var datum in Calendar1.SelectedDates)
       MessageBox.Show(datum.ToString());
}
```

Sperrtage

Zu einem guten Kalender gehört auch die Möglichkeit, einzelne Datumswerte zu sperren. Über die Collection *BlackoutDates* steht Ihnen diese Funktionalität zur Verfügung.

HINWEIS: In diesem Fall müssen Sie *DateRange*-Eigenschaften setzen (von/bis).

Beispiel 2.72: Auswahl von Sperrtagen

XAML

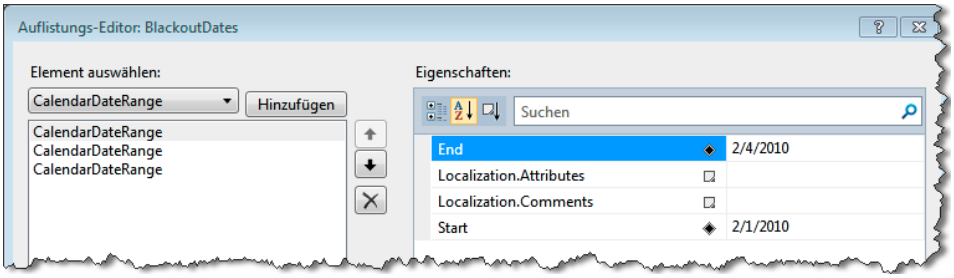
```
<Window x:Class="BSP_Controls.Calendar_Bsp"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="Calendar_Bsp" Height="300" Width="300">
    <StackPanel>
        <Calendar Name="Calendar1" DisplayMode="Month"
SelectionMode="MultipleRange">
```

Setzen der Bereiche:

```
            <Calendar.BlackoutDates>
                <CalendarDateRange Start="2/1/2010" End="2/4/2010"/>
                <CalendarDateRange Start="2/11/2010" End="2/12/2010"/>
                <CalendarDateRange Start="2/16/2010" End="2/16/2010"/>
            </Calendar.BlackoutDates>
        </Calendar>
```

Ergebnis

Alternativ steht Ihnen ein entsprechender Auflistungseditor zur Verfügung:



Calendar skalieren

Wollen Sie das *Calendar*-Control skalieren (*Width/Height*) werden Sie sicher zunächst enttäuscht sein, das Control ändert seine Größe um keinen Millimeter (bzw. Pixel)! Doch nicht verzagen, mit zwei Alternativen bekommen Sie das Control auf die gewünschte Größe:

Beispiel 2.73: Skalieren mit der *ViewBox*

XAML

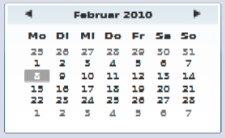
```
<Viewbox>
    <Calendar x:Name="CalendarControl"
              HorizontalAlignment="Left" VerticalAlignment="Top">
    </Calendar>
</Viewbox>
```

Beispiel 2.74: Skalieren mit Transformation

XAML

```
<Calendar  HorizontalAlignment="Left" VerticalAlignment="Top">
    <Calendar.RenderTransform>
        <ScaleTransform ScaleX="1.5" ScaleY=".5" />
    </Calendar.RenderTransform>
</Calendar>
```

Ergebnis



HINWEIS: Weitere Anpassungsmöglichkeiten für das Control bestehen durch die Verwendung von Templates, wir gehen ab Abschnitt 3.5 darauf ein.

Damit wollen wir die Thematik „Kalender“ abschließen.

2.26 InkCanvas

Mit dem *InkCanvas* möchten wir Ihnen noch ein zunächst recht unscheinbares Control vorstellen, das im Zuge der größer werdenden Verbreitung von Tablet-PCs sicher noch an Bedeutung gewinnen wird. Das Control stellt eine Zeichenfläche zur Verfügung, in der Sie mit einem Stift (oder auch der Maus) freie Zeichnungen realisieren oder Markierungen vornehmen können. Dazu kann optional in den Hintergrund des Controls eine Grafik eingeblendet werden (z. B. ein Wegeplan).

HINWEIS: Auf das Thema „Multitouch“ (ab Windows 7) gehen wir im Rahmen dieses Buchs nicht ein, es mangelt den Autoren schlicht an der nötigen Hardware.

Die von Maus, Stift oder Code erzeugten Linien/Striche werden in einer internen *Strokes*-Collection verwaltet. Einzelne *Stroke*-Objekte bestehen wiederum aus Zeichenpunkten (*StylusPoints*), haben eine Stiftform (*StylusTip*), eine Größe (*Width, Height*) und weisen eine Farbe auf.

Die Größe des Controls richtet sich beim Entwurf zunächst nach dem umgebenden Layout-Control. Handelt es sich bei diesem zum Beispiel um einen *ScrollViewer*, wird die Größe des *InkCanvas* zur Laufzeit an die maximalen Stiftpositionen angepasst. Das heißt, zeichnen Sie über den „Rand" des Controls hinweg, werden die Stiftbewegungen zunächst aufgezeichnet. Erst nach dem Zeichenende werden die Abmessungen des Controls an die nun neuen maximalen Ausdehnungen angepasst.

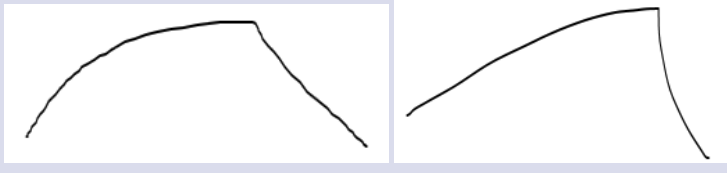
2.26.1 Stift-Parameter definieren

Wie schon erwähnt, kann der Zeichenstift verschiedene Eigenschaften aufweisen, die zusammen mit den Eingabekoordinaten in der *Strokes*-Collection abgespeichert werden.

Einstellen können Sie diese Werte über die *DefaultDrawingAttributes*-Eigenschaft des *InkCanvas*:

Eigenschaft	Beschreibung
Color	Die für den Stift verwendete Farbe.
FitToCurve	Sollen die Linien geglättet werden[9], setzen Sie den Wert auf *True*. Nach der Zeichenoperation wird in diesem Fall die Linie automatisch an einen Kurvenverlauf angepasst.
Height, Width	Bestimmt die Höhe und Breite des Zeichenstifts.
StylusTip	Bestimmt die Grundform des Zeichenstifts (Rectangle, Ellipse).

Beispiel 2.75: Linienverlauf ohne und mit *FitToCurve*



Beispiel 2.76: Linie mit rundem und eckigem Stift

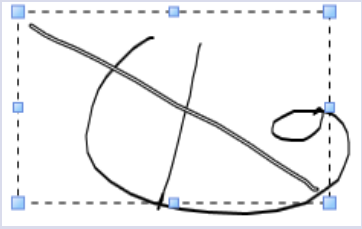
[9] z. B. um Zitterbewegungen und Ruckler auszugleichen ...

2.26.2 Die Zeichenmodi

Der *InkCanvas* kennt verschiedene Zeichenmodi (*EditingMode*), über die Sie die jeweils ausführbare Aktion vorgeben:

EditingMode	Beschreibung
None	Keine Reaktion auf Stifteingaben.
Ink	Normales Zeichnen.
GestureOnly	Während der Stiftbewegung ist die Figur sichtbar, diese wird nach dem Zeichenende gelöscht. Dieser Modus dient der Auswertung der Gestik (z. B. ein gezeichneter Kreis, eine Linie etc.).
InkAndGesture	Zeichnen und Gestikauswertung.
Select	Auswahl von *Stroke*-Objekten, diese können zum Beispiel verschoben oder gelöscht werden.
EraseByPoint	Löschen von Linienstücken (ähnlich Radiergummi in einem Malprogramm).
EraseByStroke	Löschen kompletter *Stroke*-Objekte (ähnlich einem Vektorgrafikprogramm).

Beispiel 2.77: Markierte (Select) *Stroke*-Objekte



2.26.3 Inhalte laden und sichern

Möchten Sie die Zeichnung sichern bzw. zu einem späteren Zeitpunkt erneut laden, müssen Sie sich mit der *Strokes*-Collection beschäftigen. Diese verfügt über die erforderliche *Save*-Methode bzw. einen geeigneten Konstruktor, um die Daten aus einem Stream einzulesen.

Beispiel 2.78: Sichern der Daten

C#

```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    using (FileStream fs = new FileStream(@"c:\test.ink", FileMode.Create,
                                          FileAccess.Write))
    {
        InkCanvas1.Strokes.Save(fs);
        fs.Close();
    }
}
```

Beispiel 2.79: Laden der Daten

C#

```
private void Button_Click_2(object sender, RoutedEventArgs e)
{
    InkCanvas1.Strokes.Clear();
    using (FileStream fs = new FileStream(@"c:\test.ink", FileMode.Open,
                                          FileAccess.Read))
    {
        InkCanvas1.Strokes = new StrokeCollection(fs);
        fs.Close();
    }
}
```

2.26.4 Konvertieren in eine Bitmap

Außer im *InkCanvas* können Sie mit den gesicherten Daten nichts anfangen. Da stellt sich schnell die Frage, wie Sie Ihre Kunstwerke in einem lesbaren Format sichern können. Am Beispiel des BMP-Formats wollen wir Ihnen die Vorgehensweise aufzeigen.

Beispiel 2.80: Sichern der aktuellen Zeichnung im BMP-Format

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    using (FileStream fs = new FileStream(@"c:\test.bmp",FileMode.Create,
                                          FileAccess.Write))
    {
```

Zielbitmap entsprechend der Größe der Zeichenfläche erzeugen:

```
        RenderTargetBitmap rtb = new
              RenderTargetBitmap((int)InkCanvas1.ActualWidth,
                          (int)InkCanvas1.ActualHeight, 0, 0,
                    PixelFormats.Default);
```

Daten ausgeben:

```
        rtb.Render(InkCanvas1);
```

Mit dem BMP-Encoder kodieren:

```
        BmpBitmapEncoder encoder = new BmpBitmapEncoder();
        encoder.Frames.Add(BitmapFrame.Create(rtb));
```

Speichern:

```
        encoder.Save(fs);
        fs.Close();
    }
}
```

2.26.5 Weitere Eigenschaften

Mit Hilfe der Eigenschaft *IsHighlighter* schalten Sie den Stift in einen teiltransparenten Modus, sodass darunter liegende Zeichnungen sichtbar bleiben.

Verwenden Sie Zeichentablets (z. B. von Watcom), hat der Anpressdruck des Stift einen Einfluss auf die Linienbreite. Setzen Sie *IgnorePressure* auf *True,* um dies zu verhindern.

Nicht in jedem Fall ist die aktuelle Zeichnung komplett zu sehen, bzw. Einzelheiten sind so klein, dass sie nicht sichtbar sind. Hier hilft die Verwendung einer Layout-Transformation weiter (Sie erinnern sich, dass es sich bei allen WPF-Controls um Vektorgrafiken handelt).

Beispiel 2.81: Skalieren des *InkCanvas* mit Hilfe eines *Slider*-Controls

XAML

```
      <InkCanvas Name="InkCanvas1">
        <InkCanvas.LayoutTransform>
          <ScaleTransform ScaleX="{Binding ElementName=slider2,Path=Value}"
                          ScaleY="{Binding ElementName=slider2,Path=Value}" />
        </InkCanvas.LayoutTransform>
      </InkCanvas>
...
      <Slider Height="26" Name="slider2" Maximum="5" Minimum=".1" Value="1"
Width="120" />
```

Ergebnis

Die Skalierung in Aktion:

2.27 Ellipse, Rectangle, Line und Co.

Fast hätten wir Sie vergessen, aber für die Oberflächengestaltung werden neben den komplexeren Controls auch die einfachen Elemente wie

- Ellipse/Kreis (*Ellipse*),
- Rechteck (*Rectangle*) und
- Linien (*Line*)

benötigt. Wie jedem anderen Control auch, können Sie diesen grafischen Primitiven auch Ereignisse zuordnen, um zum Beispiel auf einen Maus-Klick zu reagieren.

2.27.1 Ellipse

Für das Zeichnen von Kreisen oder Ellipsen nutzen Sie das *Ellipse*-Control. Die Eigenschaften *Height* und *Width* sind, sicher wenig überraschend, für die Abmessungen der Ellipse verantwortlich, die Linienfarbe stellen Sie über die Eigenschaft *Stroke,* die Linienstärke mit *StrokeThickness* ein. Die Füllung bestimmen Sie über *Fill*.

Beispiel 2.82: Verwendung *Ellipse*

XAML

```
<Ellipse Width="50" Height="80" Stroke="Blue" StrokeThickness="5" Fill="Aqua"/>
<Ellipse Width="80" Height="80" Stroke="Black" StrokeThickness="2" Fill="Yellow"
         MouseDown="Ellipse_MouseDown"/>
```

C#

Die Ereignismethode:

```
private void Ellipse_MouseDown(object sender, MouseButtonEventArgs e)
{
   MessageBox.Show("MouseDown");
}
```

Ergebnis

Die Ausgaben:

2.27.2 Rectangle

Auch hier bestimmen *Height* und *Width* die äußeren Abmessungen, *Stroke* und *StrokeThickness* legen Farbe und Linienstärke fest. Auch eine Eigenschaft *Fill* für die Füllung steht wieder zur Verfügung. Zusätzlich bieten die Eigenschaften *RadiusX* und *RadiusY* die Möglichkeit, einen Eckradius zu definieren.

Beispiel 2.83: Verwendung *Rectangle*

XAML

```
   <Rectangle Width="50" Height="80" Stroke="Red" StrokeThickness="5"
Fill="Green"/>
   <Rectangle Width="80" Height="80" RadiusX="30" RadiusY="20" Stroke="Red"
              StrokeThickness="3" Fill="WhiteSmoke"/>
```

Ergebnis

2.27.3 Line

Auch das Zeichnen von Linien wollen wir nicht vergessen, mit *X1*,*Y1* und *X2*,*Y2* legen Sie Start- und Endpunkt der Linie fest. *Stroke* und *StrokeThickness* stellen auch hier Farbe und Linienstärke ein.

Beispiel 2.84: Verwendung *Line*

XAML

```
<Line X1="10" Y1="10" X2="100" Y2="100" Stroke="Black" StrokeThickness="3"/>
<Line X1="10" Y1="100" X2="100" Y2="10" Stroke="Red" StrokeThickness="5"/>
```

Ergebnis

Die beiden gezeichneten Linien:

2.28 Browser

Wie kann es anders sein, auch in WPF findet der ambitionierte Programmierer ein entsprechendes Control wieder, das die Funktionalität des Microsoft Internet Explorers kapselt. Im Gegensatz zu seinem Windows Forms Pendant verfügt das still und heimlich mit der .NET-Version 3.3 SP1 eingeführte Control über einige recht interessante Methoden, die im Zusammenhang mit der Darstellung von Datenbankinhalten von Interesse sind.

Die Verwendung selbst ist recht einfach, es genügt, wenn Sie der Eigenschaft *Source* eine entsprechende Webadresse zuweisen, damit deren Inhalte dargestellt werden.

Beispiel 2.85: Adresse per XAML-Code zuweisen

XAML

```
<WebBrowser Name="webBrowser1" Source="http://www.spiegel.de" />
```

Beispiel 2.86: Einfaches Browserformular mit Eingabezeile und Adressübergabe per Code

XAML

```
<Window x:Class="BSP_Controls.Browser_Bsp"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Browser_Bsp" Height="362" Width="635">
    <DockPanel>
```

```
        <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
            <TextBox Name="txt1" FontSize="14" Margin="0,0,0,5"
                KeyDown="txt1_KeyDown">http://www.doko-buch.de</TextBox>
        </StackPanel>
        <WebBrowser Name="webBrowser1" />
    </DockPanel>
</Window>
```

C#

Die Adressübergabe zur Laufzeit:

```
        private void txt1_KeyDown(object sender, KeyEventArgs e)
        {
            if (e.Key == Key.Enter)
                webBrowser1.Source = new Uri(txt1.Text);
        }
```

Ergebnis

Alternativ können Sie die Navigation zur Zielseite auch mit den Methoden

- *Navigate* (Uri übergeben),
- *NavigateToStream* (lädt die Inhalte aus einem *Stream*) oder
- *NavigateToString* (lädt die Inhalte aus dem übergebenen *String*)

starten. Interessant ist hier vor allem *NavigateToStream,* was im Zusammenhang mit der Darstellung von Blob-Daten aus Ressourcen, Datenbanken oder Webdiensten recht interessant ist.

Beispiel 2.87: Laden von Ressourcedaten

C#

Laden einer eingebetteten HTML-Seite (*info.htm*):

```
using System.IO;
...
        private void Button2_Click(object sender, RoutedEventArgs e)
        {
            Uri uri = new Uri(@"pack://application:,,,/info.htm",
UriKind.Absolute);
            Stream src = Application.GetResourceStream(uri).Stream;
            webBrowser1.NavigateToStream(src);
        }
```

Wie Sie auch direkt Strings für das Erstellen der HTML-Seite verwenden können, zeigt folgendes einfaches Beispiel:

Beispiel 2.88: Verwendung von *NavigateToString*

C#

```
    public partial class Browser_Bsp : Window
    {
        public Browser_Bsp()
        {
            InitializeComponent();
            webBrowser1.NavigateToString("<HTML><BODY>Ein einfaches" +
             " <b>HTML-Dokument</b><br> mit wenig <u>Text</u>.</BODY></HTML>");
        }
```

Ergebnis

Ein einfaches **HTML-Dokument**
mit wenig Text.

Auch die *Refresh*-Methode für das erneute Laden einer Seite sollten wir nicht vergessen.

Über den aktuellen Stand der Dinge können Sie sich mit Hilfe der drei Ereignisse *Navigating*, *Navigated* und *LoadCompleted* informieren.

Die Navigation zwischen mehreren Seiten ist über die Methoden *GoBack* und *GoForward* möglich. Ob der Aufruf der Methoden zulässig ist, bestimmen Sie mit den Eigenschaften *CanGoBack* und *CanGoForward*.

2.29 Ribbon

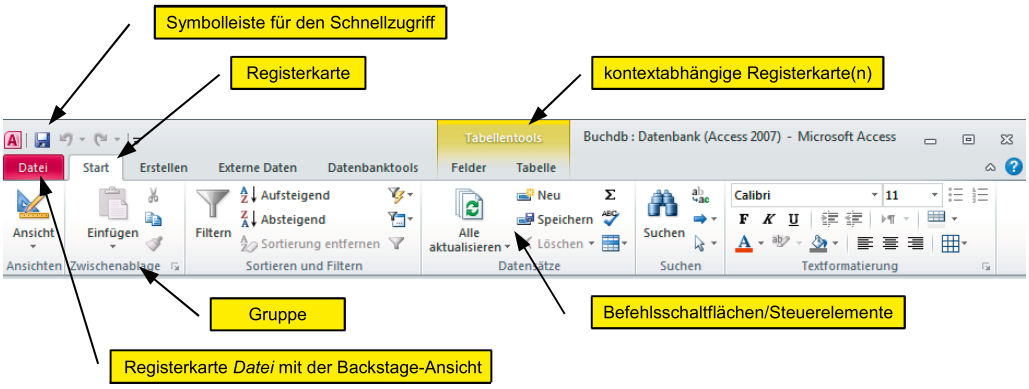
Bereits mit Office 2007 wurde von Microsoft das weit verbreitete Menü- und Toolbar-Konzept komplett über den Haufen geworfen, seitdem übernimmt der *Ribbon* (bzw. auf Neudeutsch das Menüband) diese Funktion.

Aus guten Gründen scheiden sich an diesem Konzept die Geister – der angeblich intuitiven Funktionsweise steht auf der anderen Seite ein wesentlich erhöhter Platzbedarf und eine geringe Übersichtlichkeit mit langen Mauswegen gegenüber. Aber natürlich ist der Ribbon viel bunter und moderner, ein Trend der auch bei den neuen Windows-Oberflächen zu beobachten ist ...

Auch Ihre WPF-Anwendungen können von den Vorzügen des *Ribbon* profitieren, bietet doch Microsoft eine entsprechende Library.

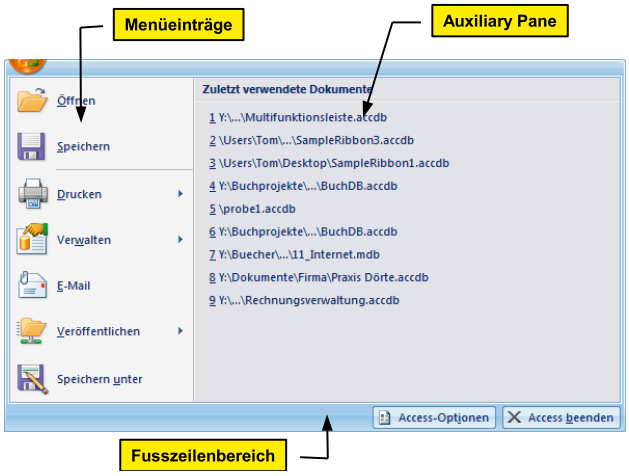
2.29.1 Allgemeine Grundlagen

Bevor wir uns der eigentlichen Programmierung widmen, wollen wir uns zunächst einmal näher mit einem *Ribbon* und den entsprechenden Begrifflichkeiten beschäftigen, andernfalls kann es schnell zu Missverständnissen kommen. Die folgende Abbildung zeigt das „Opfer“ unserer anschließenden Programmierversuche am Beispiel von Microsoft Access 2010:



- Registerkarte *Datei*
  Diese ersetzt im Wesentlichen den früheren Office-Button bzw. das noch ältere *Datei*-Menü. Hinter der Registerkarte verbirgt sich in den neueren Office-Versionen die so genannte *Backstage*-Ansicht. **Diese können Sie mit dem vorliegenden Ribbon-Control nicht realisieren.** Sie haben nur die Möglichkeit, ein Anwendungsmenü zu erstellen, dieses entspricht der Ansicht in den Office 2007-Applikationen.
- Symbolleiste für den Schnellzugriff
  Hier können Sie kleine Schaltflächen einblenden, die dem Anwender immer zur Verfügung stehen sollen.
- Registerkarten
  Diese stellen aufgabenorientierte Funktionen zur Verfügung.
- Gruppen
  Diese teilen die Funktionen innerhalb der Registerkarten in einzelne Bereiche.
- Befehlsschaltflächen und andere Steuerelemente

Das Anwendungsmenü selbst gliedert sich in eine Reihe von Menüeinträgen (*Menu Items*), den Zusatzbereich (Auxiliary Pane) sowie den Fußzeilenbereich (*Footer Pane*):



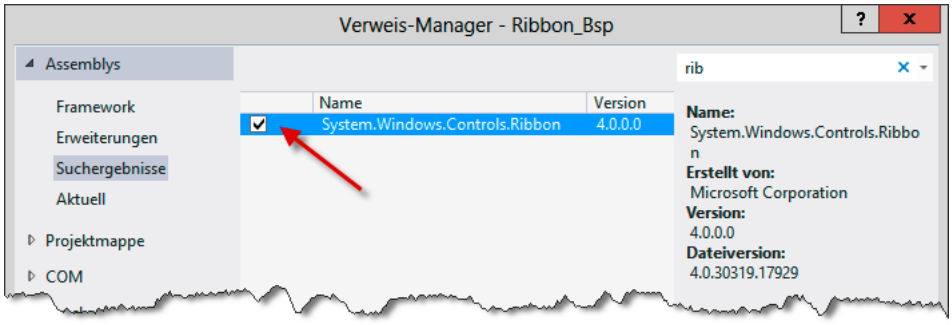
Damit schließen wir unsere kurzen theoretischen Erörterungen zum *Ribbon* ab und wenden uns der Praxis zu.

2.29.2 Download/Installation

Waren Sie in den früheren Versionen darauf angewiesen, sich die Ribbon-Library nachträglich herunterzuladen und zu installieren, ist diese seit Visual Studio 2012 bereits in der Standardinstallation enthalten.

Wer noch mit einer früheren Version von Visual Studio arbeitet, muss die Library nachinstallieren. Sie finden diese unter *http://www.microsoft.com/en-us/download/details.aspx?id=11877*.

Für das erfolgreiche Einbinden des Ribbons in Ihr WPF-Projekt müssen Sie diesem noch einen Verweis auf *System.Windows.Controls.Ribbon* hinzufügen:



Damit sind die Vorbereitungen abgeschlossen, wir können uns der Umsetzung zuwenden.

2.29.3 Erste Schritte

Haben Sie den Verweis auf die Assembly erfolgreich eingebunden, steht den ersten Gehversuchen nichts mehr im Wege. Öffnen Sie ein *Window* und positionieren Sie ein *Ribbon*-Element am oberen Fensterrand. Sie können dazu ein *DockPanel* oder ein *Grid* nutzen. Wir haben uns für die zweite Variante entschieden:

Beispiel 2.89: Einfacher *Ribbon* im Standard-Window

XAML

```
<Window x:Class="Ribbon_Bsp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
```

Gliedern der Seite:

```
    <Grid>
        <Grid.RowDefinitions>
```

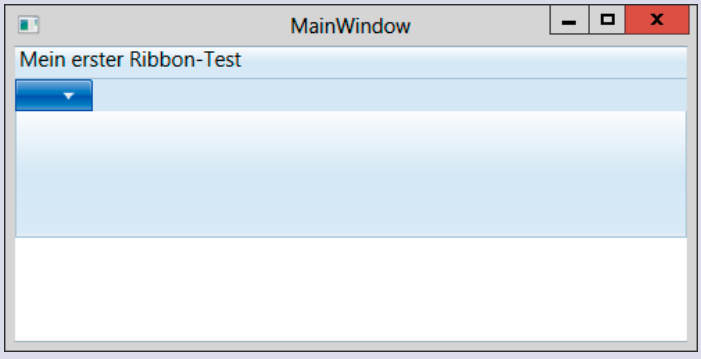
Der *Ribbon* bestimmt die Höhe der ersten Zeile im Grid:

```
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
        </Grid.RowDefinitions>
```

Den *Ribbon* der ersten Zeile zuordnen:

```
        <Ribbon Grid.Row="0" Title="Mein erster Ribbon-Test">
        </Ribbon>
    </Grid>
</Window>
```

Ergebnis



Wie Sie sehen, sehen Sie zunächst nicht viel. Lediglich ein Titel, die Schaltfläche für das Anwendungsmenü (funktionslos) und die Platzhalter für die späteren Registerkarten werden dargestellt.

Auf den Titel sollten Sie im Allgemeinen verzichten, dieser kostet nur Platz und hat keinen echten Nutzen (der Window-Titel sollte eigentlich reichen).

2.29.4 Registerkarten und Gruppen

Nächster Schritt ist das Erzeugen eigener Registerkarten, in die wiederum die Gruppen eingefügt werden. Die eigentlichen Befehlselemente sind dann den Gruppen zuzuordnen.

Beispiel 2.90: Registerkarten und Gruppen erzeugen

XAML

```
...
        <Ribbon Grid.Row="0" >
```

Erst die Registerkarte:

```
            <RibbonTab Header="Registerkarte 1" >
```

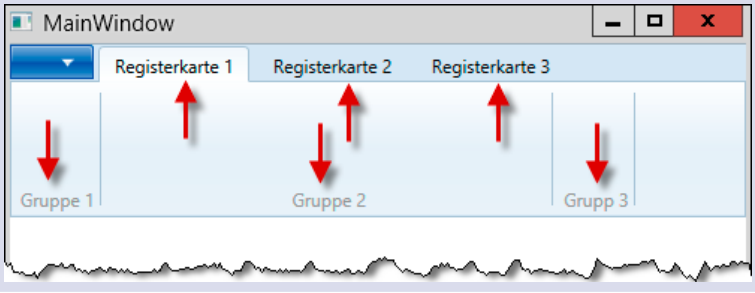
Dann die Gruppen:

```
                <RibbonGroup Header="Gruppe 1">
                </RibbonGroup>
                <RibbonGroup Header="Gruppe 2" Width="300">
                </RibbonGroup>
                <RibbonGroup Header="Grupp 3">
                </RibbonGroup>
            </RibbonTab>
            <RibbonTab Header="Registerkarte 2">
            </RibbonTab>
            <RibbonTab Header="Registerkarte 3">
            </RibbonTab>
```

Die Sichtbarkeit von Gruppen oder ganzen Registerkarten können Sie mit der Eigenschaft *Visibility* steuern, ein Ein-/Ausblenden zur Laufzeit ist jederzeit möglich:

```
            <RibbonTab Header="Registerkarte 4" Visibility="Collapsed">
            </RibbonTab>
        </Ribbon>
...
```

Ergebnis



2.29.5 Kontextabhängige Registerkarten

Neben den meist standardmäßig angezeigten Registerkarten gibt es auch kontextabhängige Registerkarten (meist farblich markiert), die im Zusammenhang mit Auswahlvorgängen oder Selektionen des Nutzers zusätzlich eingeblendet werden. Auch diese können Sie selbst realisieren:

Beispiel 2.91: Kontextabhängige Registerkarten erzeugen

XAML

```
...
        <Ribbon Grid.Row="0" >
```

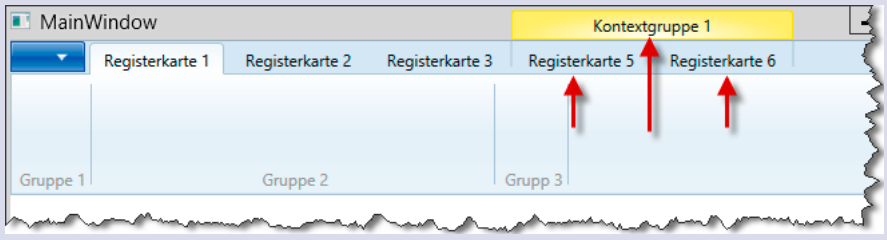
Hier definieren wir die Gruppen mit ihren Farben:

```
            <Ribbon.ContextualTabGroups>
                <RibbonContextualTabGroup Header="Kontextgruppe 1"
                                          Visibility="Visible"
Background="#FFFFE90C" >
                </RibbonContextualTabGroup>
            </Ribbon.ContextualTabGroups>
...
```

Später können wir auf die obigen Gruppen Bezug nehmen:

```
            <RibbonTab Header="Registerkarte 5"
ContextualTabGroupHeader="Kontextgruppe 1">
            </RibbonTab>
            <RibbonTab Header="Registerkarte 6"
ContextualTabGroupHeader="Kontextgruppe 1">
            </RibbonTab>
...
```

Ergebnis



2.29.6 Einfache Beschriftungen

Möchten Sie innerhalb der Gruppen kurze Texte anzeigen, können Sie dafür das *RibbonTwoLineText*-Element nutzen:

Beispiel 2.92: Beschriftungen realisieren

XAML

```
...
        <Ribbon Grid.Row="0" >
            <RibbonTab Header="Registerkarte 1" >
...
                <RibbonGroup Header="Gruppe 2" Width="300">
                    <RibbonTwoLineText Text="Textzeile 1"/>
                    <RibbonTwoLineText Text="Textzeile 2"/>
                </RibbonGroup>
...
```

Ergebnis

2.29.7 Schaltflächen

Die Ribbon-Library bietet vier verschiedene Varianten von Schaltflächen an:

- nur kleines Bild (16x16 Pixel)
- kleines Bild und Text
- nur großes Bild (32x32 Pixel)
- großes Bild und Text

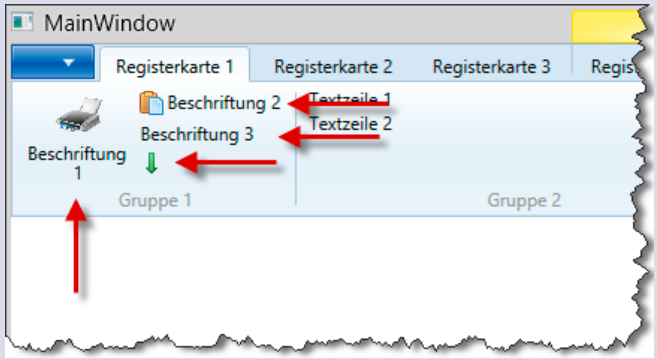
Steuern können Sie das Aussehen über die Attribute *Label* (die Beschriftung), *SmallImageSource* und/oder *LargeImageSource*.

Beispiel 2.93: Schaltflächen definieren

XAML

```
...
        <Ribbon Grid.Row="0" >
            <Ribbon.ContextualTabGroups>
...
            <RibbonTab Header="Registerkarte 1" >
                <RibbonGroup Header="Gruppe 1">
                    <RibbonButton Label="Beschriftung 1"
                                  LargeImageSource="Images/printer.png" />
                    <RibbonButton Label="Beschriftung 2"
                                  SmallImageSource="Images/page_paste.png" />
                    <RibbonButton Label="Beschriftung 3" />
                    <RibbonButton SmallImageSource="Images/arrow_down.png"/>
                </RibbonGroup>
                <RibbonGroup Header="Gruppe 2" Width="300">
...
```

Ergebnis



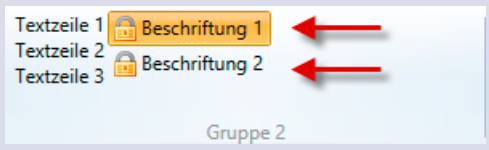
Als Alternative für den einfachen *RibbonButton* bietet sich noch der *RibbonToggleButton* an. Dieser hat prinzipiell das gleiche Aussehen und dieselben Konfigurationsmöglichkeiten, kann aber zwei Schaltzustände darstellen:

Beispiel 2.94: Verwendung *RibbonToggleButton*

C#

```
...
                <RibbonGroup Header="Gruppe 2" Width="300">
                    <RibbonTwoLineText Text="Textzeile 1"/>
                    <RibbonTwoLineText Text="Textzeile 2"/>
                    <RibbonTwoLineText Text="Textzeile 3"/>
                    <RibbonToggleButton Label="Beschriftung 1"
                           SmallImageSource="Images/lock.png"   IsChecked="True"
/>
                    <RibbonToggleButton Label="Beschriftung 2"
                           SmallImageSource="Images/lock.png"  />
                </RibbonGroup>
...
```

Ergebnis



Schaltflächen zusammenfassen

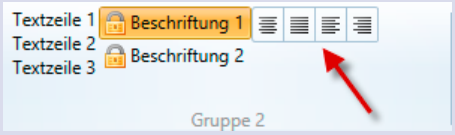
Möchten Sie mehrere Schaltflächen so anordnen, dass ein Zusammenhang zwischen diesen hergestellt wird, können Sie die *RibbonControlGroup* verwenden. Prominentes Beispiel dürfte die Zuordnung der Textausrichtung sein:

Beispiel 2.95: Verwendung *RibbonControlGroup*

C#

```
...
                    <RibbonControlGroup>
                        <RibbonButton
SmallImageSource="Images/text_align_center.png"  />
                        <RibbonButton
SmallImageSource="Images/text_align_justify.png"  />
                        <RibbonButton SmallImageSource="Images/text_align_left.png"
/>
                        <RibbonButton
SmallImageSource="Images/text_align_right.png"  />
                    </RibbonControlGroup>
...
```

Ergebnis



2.29.8 Auswahllisten

Hier bietet sich die *RibbonComboBox* an. Diese wird im *Ribbon* mit einer *RibbonGallery* kombiniert. Dies hat den Vorteil, dass die Einträge auch mehrspaltig und gruppiert dargestellt werden können.

Beispiel 2.96: Darstellen von Listen im Ribbon

C#

```
...
```

Wir kombinieren *RibbonComboBox* und *RibbonGallery*:

```
                    <RibbonComboBox SelectionBoxWidth="100" IsEditable="True"
                        Name="RibbonComboBox1">
                        <RibbonGallery SelectedValue="Eintrag 4"
SelectedValuePath="Content">
```

Eine erste Rubrik erzeugen:

```
                    <RibbonGalleryCategory Header="Wählen Sie einen Eintrag"
                                MaxColumnCount="2">
                        <RibbonGalleryItem Content="Eintrag 1" Foreground="Green" />
                        <RibbonGalleryItem Content="Eintrag 2" Foreground="Red" />
```

```
                        <RibbonGalleryItem Content="Eintrag 3" Foreground="Blue" />
                        <RibbonGalleryItem Content="Eintrag 4" Foreground="Black" />
                        <RibbonGalleryItem Content="Eintrag 5" Foreground="Black" />
                        <RibbonGalleryItem Content="Eintrag 6" Foreground="Black" />
                        <RibbonGalleryItem Content="Eintrag 7" Foreground="Black" />
                        </RibbonGalleryCategory>
```

Eine zweite Rubrik erzeugen:

```
                            <RibbonGalleryCategory  Header="Rubrik 2"
 MaxColumnCount="2">
                                <RibbonGalleryItem Content="Eintrag 1"/>
                                <RibbonGalleryItem Content="Eintrag 2" />
                                <RibbonGalleryItem Content="Eintrag 3" />
                                <RibbonGalleryItem Content="Eintrag 4" />
                                <RibbonGalleryItem Content="Eintrag 5" />
                                <RibbonGalleryItem Content="Eintrag 6" />
                                <RibbonGalleryItem Content="Eintrag 7" />
                            </RibbonGalleryCategory>
                        </RibbonGallery>
                    </RibbonComboBox>
```

Diese Liste füllen wir zur Laufzeit:

```
                    <RibbonComboBox SelectionBoxWidth="100" IsEditable="False"
                                    Name="RibbonComboBox2" >
```

Beachten Sie, dass wir auf Änderungen in der *RibbonGallery*, nicht in der *RibbonComboBox* reagieren müssen:

```
                        <RibbonGallery SelectedValue="Zeile 4"
                                       SelectedValuePath="Content"
                                       SelectionChanged="RibbonGallery_
                                       SelectionChanged_1" >
                            <RibbonGalleryCategory  Header="Kategorie 1"
                                  MaxColumnCount="1" Name="RibbonGalleryCategory1"/>
                            <RibbonGalleryCategory  Header="Kategorie 2"
                                  MaxColumnCount="1" Name="RibbonGalleryCategory2"/>
                        </RibbonGallery>
                    </RibbonComboBox>
...
...
 public partial class MainWindow : RibbonWindow
    {
        public MainWindow()
        {
            InitializeComponent();
```

Die *RibbonGallery* in zwei Kategorien füllen:

```
            for (int i = 0; i < 10; i++)
                RibbonGalleryCategory1.Items.Add("Zeile " + i.ToString());
            for (int i = 0; i < 6; i++)
                RibbonGalleryCategory2.Items.Add("Zeile " + i.ToString());
        }
```

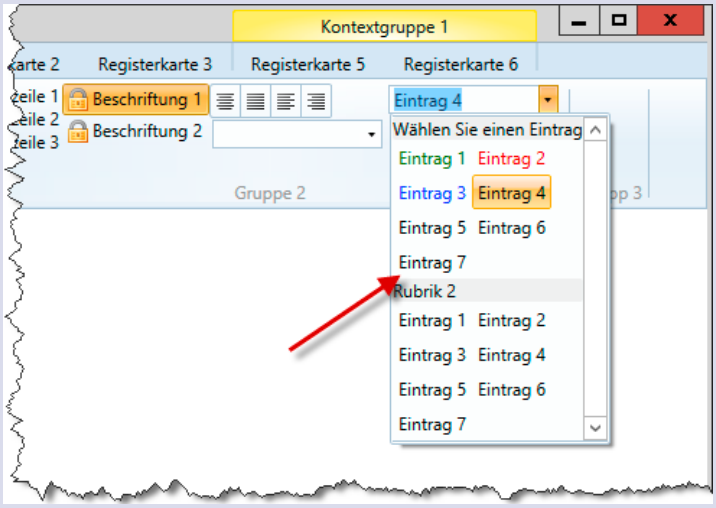
Anzeige der neuen Auswahl:

```
        private void RibbonGallery_SelectionChanged_1(object sender,
                                    RoutedPropertyChangedEventArgs<object> e)
        {
```

Bestimmen der Auswahl:

```
            MessageBox.Show(e.NewValue.ToString());
        }
    }
...
```

Ergebnis



Eine *RibbonGallery* können Sie auch in den folgenden Steuerelementen definieren:

- *RibbonSplitMenuButton*
- *RibbonApplicationSplitMenuItem*
- *RibbonSplitMenuItem*

Wir gehen nicht im Einzelnen darauf ein, dies würde den Rahmen diese Kapitels sprengen.

2.29.9 Optionsauswahl

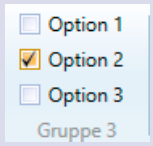
Als Alternative zum *RibbonToggleButton* bietet sich die *RibbonCheckBox* an. Diese wird wie ihr WPF-Pendant konfiguriert und ausgewertet:

Beispiel 2.97: Verwendung *RibbonCheckBox*

XAML

```
...
                <RibbonGroup Header="Gruppe 3">
                    <RibbonCheckBox Label="Option 1"/>
                    <RibbonCheckBox Label="Option 2" IsChecked="True"/>
                    <RibbonCheckBox Label="Option 3"/>
                </RibbonGroup>
...
```

Ergebnis



2.29.10 Texteingaben

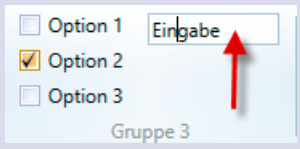
Hinter der *RibbonTextBox* verbirgt sich nichts anderes als das allseits bekannte Textfeld, das bekanntlich der Eingabe von Zeichenketten dient. Allerdings sollten Sie hier nicht allzu viel erwarten, die diversen Beschränkungen und Ereignisse wie bei normalen Textfeldern sind nicht realisierbar. Aber das ist sicher auch nicht der eigentliche Zweck dieses Controls.

Beispiel 2.98: Verwendung *RibbonTextBox*

XAML

```
...
                <RibbonGroup Header="Gruppe 3">
                    <RibbonCheckBox Label="Option 1"/>
                    <RibbonCheckBox Label="Option 2" IsChecked="True"/>
                    <RibbonCheckBox Label="Option 3"/>
                    <RibbonTextBox Text="Eingabe" MaxLength="10"  />
                </RibbonGroup>
...
```

Ergebnis



2.29.11 Screentips

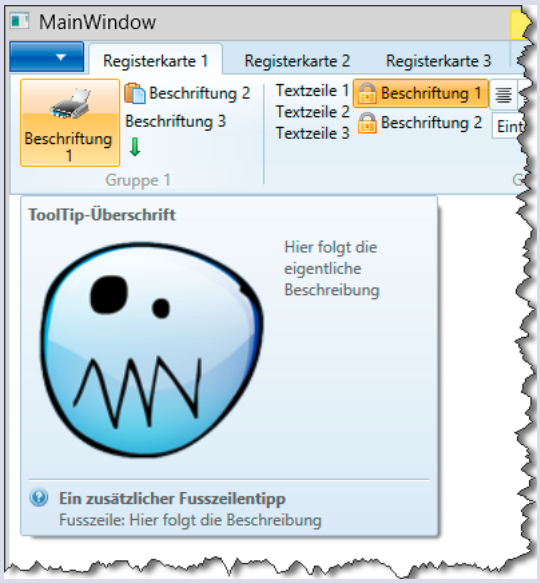
Auch wenn Sie es mitunter lästig finden, für den unerfahrenen Endanwender sind die kleinen Hilfetexte, die in den Screentips angezeigt werden, meist unentbehrlich. Steuern können Sie die Inhalte über die Attribute *ToolTipTitle*, *ToolTipImageSource*, *ToolTipDescription*, *ToolTipFooterTitle*, *ToolTipFooterImageSource* und *ToolTipFooterDescription*.

Beispiel 2.99: Einen Screentip realisieren

XAML

```
...
    <RibbonButton Label="Beschriftung 1"
                  LargeImageSource="Images/printer.png"
                  ToolTipTitle="ToolTip-Überschrift"
                  ToolTipImageSource="Images/user.png"
                  ToolTipDescription="Hier folgt die eigentliche Beschreibung"
                  ToolTipFooterTitle="Ein zusätzlicher Fusszeilentipp"
                  ToolTipFooterImageSource="Images/help.png"
                  ToolTipFooterDescription="Fusszeile: Hier folgt die
Beschreibung" />
...
```

Ergebnis



2.29.12 Symbolleiste für den Schnellzugriff

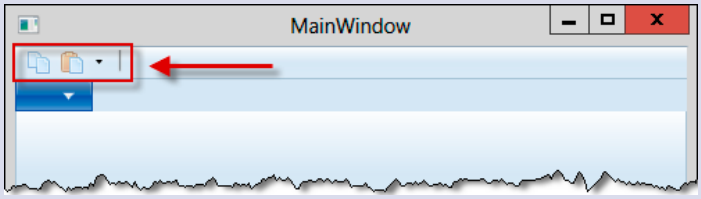
Die Symbolleiste für den Schnellzugriff stellt dem Anwender jederzeit einige wichtige Funktionen im oberen Fensterbereich zur Verfügung. Fügen Sie in den *Ribbon* ein *RibbonQuickAccessToolBar*-Element ein, dieses enthält dann die *RibbonButton*- oder *RibbonSplitButton*-Elemente.

Beispiel 2.100: Symbolleiste für den Schnellzugriff implementieren

XAML

```
...
        <Ribbon Grid.Row="0" >
            <Ribbon.QuickAccessToolBar>
                <RibbonQuickAccessToolBar>
                    <RibbonButton SmallImageSource="Images/page_copy.png"/>
                    <RibbonSplitButton SmallImageSource="Images/page_paste.png"/>
                </RibbonQuickAccessToolBar>
            </Ribbon.QuickAccessToolBar>
        </Ribbon>
...
```

Ergebnis



Achten Sie darauf, dass Sie den beiden Schaltflächen die Eigenschaft *SmallImageSource* zuweisen, nicht *LargeImageSource*, es sollen ja auch nur kleine Schaltflächen angezeigt werden. Auf die Angabe einer Beschriftung können Sie aus naheliegenden Gründen ebenfalls verzichten.

2.29.13 Das RibbonWindow

Leider entspricht das Ergebnis unserer Bemühungen nicht ganz den Erwartungen, die Schnellzugriffsleiste wird nicht, wie erwartet, in der Titelzeile des Fensters eingeblendet, sondern darunter.

Besser klappt es mit der *RibbonWindow*-Klasse, die diese Aufgabe übernimmt. *RibbonWindow* integriert die Schnellzugriffsleiste wie gewünscht in die Kopfzeile, die nötigen Anpassungen Ihres Programms zeigt das folgende Beispiel.

Beispiel 2.101: Verwenden der *RibbonWindow*-Klasse

XAML

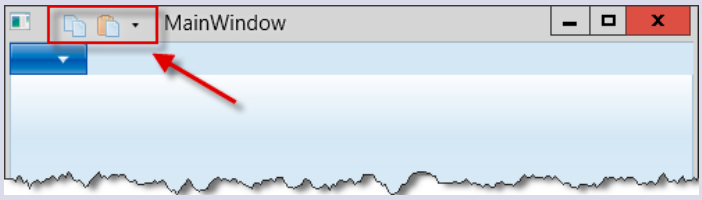
```
<RibbonWindow x:Class="Ribbon_Bsp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="250" Width="500"
WindowStartupLocation="CenterScreen">
...
</RibbonWindow>
```

C#

```
...
using System.Windows.Controls.Ribbon;
...
    public partial class MainWindow : RibbonWindow
    {
...
```

Ergebnis

Das hier sieht schon eher nach dem bekannten Ribbon aus:



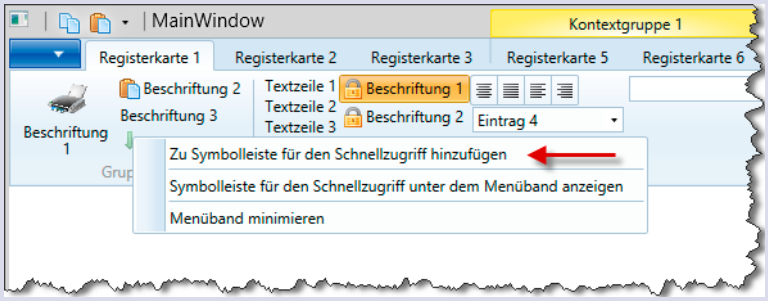
Ist einer Schaltfläche im normalen Ribbon ein *Command* zugeordnet und setzen Sie die Eigenschaft *CanAddToQuickAccessToolBarDirectly* dieser Schaltfläche auf *True*, kann der Anwender diese Schaltfläche zur Laufzeit der Schnellzugriffsleiste hinzufügen (Kontextmenü, rechte Maustaste).

Beispiel 2.102: Schnellzugriffsleiste zur Laufzeit erweitern

XAML

```
...
 <RibbonButton SmallImageSource="Images/arrow_down.png"
               CanAddToQuickAccessToolBarDirectly="True"
               Command="ApplicationCommands.Close"/>
...
```

Ergebnis



2.29.14 Menüs

Ganz nebenbei hat der *Ribbon* auch einen *RibbonMenuButton* zu bieten, was allerdings gleich die Frage nach dem Sinn auslöst, soll das Menüband doch die guten alten Menüs ersetzen.

Sie können dieses Steuerelement als Container für eingelagerte *RibbonMenuItem*-Elemente nutzen oder aber auch die schon bekannten Ribbon-Controls darin unterbringen. Zusätzlich bietet ein *RibbonSeparator* die Möglichkeit, die einzelnen Controls optisch voneinander zu trennen. Dies kann als simpler Strich oder auch als Beschriftung erfolgen, wie es das folgende Beispiel zeigt.

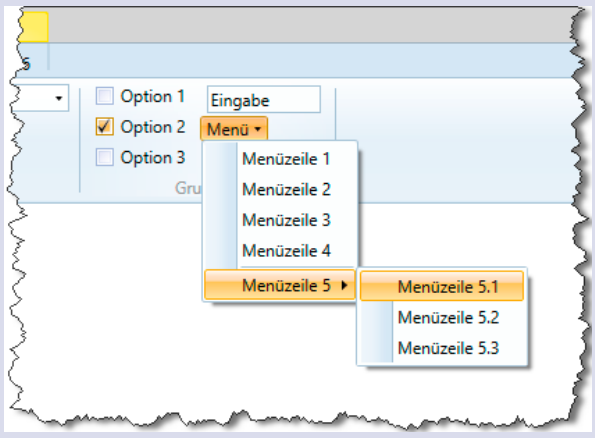
Genügt Ihnen eine Menüebene nicht, lassen sich weitere *RibbonMenuItem*-Steuerelemente hierarchisch eingliedern, Sie erhalten dann die bekannten Untermenüs.

Beispiel 2.103: Ein Menü implementieren

XAML

```
...
                <RibbonMenuButton Label="Menü">
                    <RibbonMenuButton.Items>
                        <RibbonMenuItem Header="Menüzeile 1" />
                        <RibbonMenuItem Header="Menüzeile 2" />
                        <RibbonButton Label="Ein Button"/>
                        <RibbonMenuItem Header="Menüzeile 3" />
                        <RibbonMenuItem Header="Menüzeile 4" />
                        <RibbonSeparator/>
                        <RibbonMenuItem Header="Menüzeile 5" >
                            <RibbonMenuItem Header="Menüzeile 5.1" />
                            <RibbonMenuItem Header="Menüzeile 5.2" />
                            <RibbonMenuItem Header="Menüzeile 5.3" />
                        </RibbonMenuItem>
                    </RibbonMenuButton.Items>
                </RibbonMenuButton>
...
```

Ergebnis



HINWEIS: Erliegen Sie bitte nicht der Versuchung, ein mehrfach geschachteltes Menü zu erzeugen, dies ist der Übersicht sicher nicht zuträglich und widerspricht dem Grundkonzept des Menübands!

Nach diesem „Rundflug“ über die verfügbaren Steuerelemente wollen wir uns noch mit einem Menü-Bereich beschäftigen, den wir bisher in unseren Betrachtungen sträflich vernachlässigt haben.

2.29.15 Anwendungsmenü

Leider steht in der vorliegenden Form des Ribbons nicht die vom aktuellen Microsoft Office bekannte Backstage-Ansicht zur Verfügung, Sie müssen mit dem einfacheren Anwendungsmenü Vorlieb nehmen. Dieses gliedert sich in eine Reihe von Schaltflächen auf der linken Seite, eine Detailbereich auf der rechten Seite und einem Fusszeilenbereich.

Beispiel 2.104: Ein Anwendungsmenü implementieren

XAML

```
...
        <Ribbon Grid.Row="0" >
            <Ribbon.QuickAccessToolBar>
            </Ribbon.QuickAccessToolBar>
            <Ribbon.ApplicationMenu>
```

Das Symbol für das Anwendungsmenü festlegen:

```
                <RibbonApplicationMenu SmallImageSource="Images/help.png">
```

Der erste Haupteintrag:

```
                    <RibbonApplicationMenuItem ImageSource="Images/CP45.ico"
                        Header="Eintrag 1">
```

Dieser Eintrag hat weitere Untereinträge:

```
                        <RibbonApplicationMenuItem Header="Untereintrag 1.1"  />
                        <RibbonApplicationMenuItem Header="Untereintrag 1.2"  />
                        <RibbonApplicationMenuItem Header="Untereintrag 1.3"  />
                        <RibbonApplicationMenuItem Header="Untereintrag 1.4"  />
                        <RibbonApplicationMenuItem Header="Untereintrag 1.5"  />
                    </RibbonApplicationMenuItem>
```

Zweiter Haupteintrag:

```
                    <RibbonApplicationMenuItem ImageSource="Images/CP49.ico"
                                               Header="Eintrag 2"/>
```

Dritter Haupteintrag:

```
                    <RibbonApplicationMenuItem ImageSource="Images/CP09.ico"
                                               Header="Eintrag 3"/>
```

In den Detailbereich blenden wir eine Liste der zuletzt geöffneten Dateien ein:

```
<RibbonApplicationMenu.AuxiliaryPaneContent>
```

Ein *Grid*, um Überschrift und Liste zu trennen:

```
<Grid ScrollViewer.VerticalScrollBarVisibility="Auto">
```

Verzichten Sie nicht auf die automatische Anzeige der Scrollbars, ist die Liste länger, würden sonst Einträge abgeschnitten werden.

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="*"/>
</Grid.RowDefinitions>
```

Die kleine Kopfzeile:

```
<Border Grid.Row="0" BorderBrush="DarkBlue"
        BorderThickness="0,0,0,1">
    <Label Content="Zuletzt geöffnete Dateien"  />
</Border>
```

Die Liste befüllen wir zur Laufzeit:

```
    <ListBox Grid.Row="1" Name="DateiListe1" />
</Grid>
</RibbonApplicationMenu.AuxiliaryPaneContent>
```

In der Fußzeile des Anwendungsmenüs blenden wir einen Copyright-Vermerk und eine Schaltfläche zum Beenden ein:

```
<RibbonApplicationMenu.FooterPaneContent>
    <Grid Margin="2" >
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="Auto"/>
        </Grid.ColumnDefinitions>
```

Der Text:

```
        <TextBlock Grid.Column="0" VerticalAlignment="Stretch">
                  (c) DOKO-Buch</TextBlock>
```

Die Schaltfläche:

```
        <RibbonButton Grid.Column="1" Label="Beenden"
                  SmallImageSource="Images/arrow_down.png"/>
    </Grid>
</RibbonApplicationMenu.FooterPaneContent>
</RibbonApplicationMenu>
</Ribbon.ApplicationMenu>
...
```

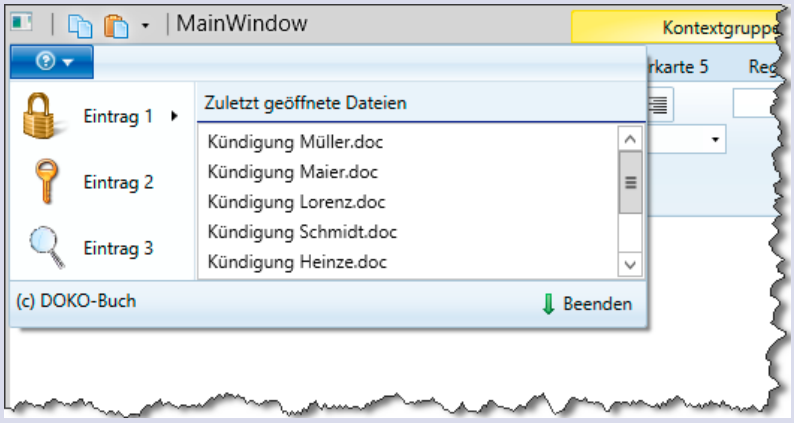
C#

Die Liste zur Laufzeit füllen:

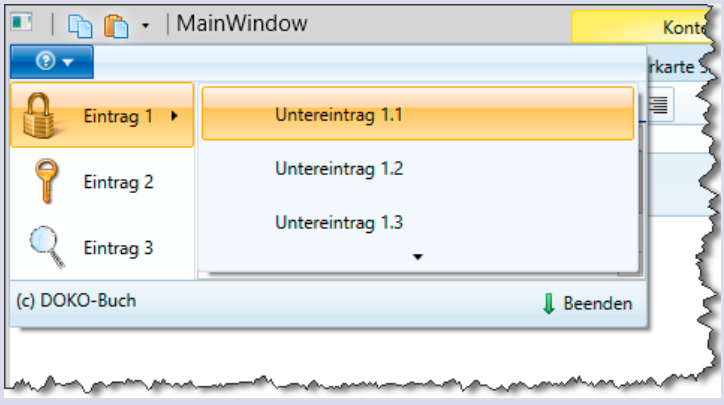
```
...
   public MainWindow()
        {
            InitializeComponent();
             DateiListe1.Items.Add("Kündigung Müller.doc");
               DateiListe1.Items.Add("Kündigung Maier.doc");
               DateiListe1.Items.Add("Kündigung Lorenz.doc");
               DateiListe1.Items.Add("Kündigung Schmidt.doc");
               DateiListe1.Items.Add("Kündigung Heinze.doc");
               DateiListe1.Items.Add("Kündigung Koch.doc");
               DateiListe1.Items.Add("Kündigung Walter.doc");
               DateiListe1.Items.Add("Kündigung Gewinnus.doc");
        }
...
```

Ergebnis

Das geöffnete Anwendungsmenü:



Die Untereinträge des ersten Menüpunktes:



Gerade beim Aufklappen der Menüeinträge wird schnell klar, dass hier die Übersichtlichkeit massiv leidet, da die Liste der letzten Dateien verdeckt wird. Verzichten Sie also besser auf derartige Verschachtelungen, zumal die Höhe des Untermenüs durch das Anwendungsmenü beschränkt ist.

2.29.16 Alternativen

Sie haben es vielleicht schon bemerkt – auch bei der aktuellen Ribbon-Library ist bei Microsoft der erste Elan frühzeitig erloschen, und so fehlen einige Funktionen, andere sind fehlerhaft etc. Eine relevante Pflege scheint auch nicht mehr stattzufinden.

Eine sinnvolle (kostenlose) Alternative finden Sie unter *http://fluent.codeplex.com*.

2.30 Chart

Auch hier scheinen die Entwickler unter akutem Zeitmangel zu leiden, denn in Visual Studio ist die Komponente immer noch nicht enthalten, stattdessen finden Sie sie unter *http://wpf.codeplex.com/releases/view/40535* bzw. per NuGet unter „WPF Toolkit“.

Nach der Installation binden Sie einen Verweis auf *System.Windows.Controls.DataVisualization.Toolkit* in Ihr Projekt ein (*Projekt|Verweis hinzufügen*).

Ein kleines Beispiel zeigt die Möglichkeiten:

Beispiel 2.105: Kuchendiagramm erzeugen

XAML

```
...
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Namespace einbinden:

```
        xmlns:tk="clr-namespace:System.Windows.Controls.DataVisualization.Charting;
                 assembly=System.Windows.Controls.DataVisualization.Toolkit"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
```

Chart erzeugen:

```
        <tk:Chart Name="Chart1" Title="Kuchendiagramm">
            <tk:PieSeries DependentValuePath="Value" IndependentValuePath="Key"
                          ItemsSource="{Binding}" IsSelectionEnabled="True" />
        </tk:Chart>
    </Grid>
</Window>
...
```

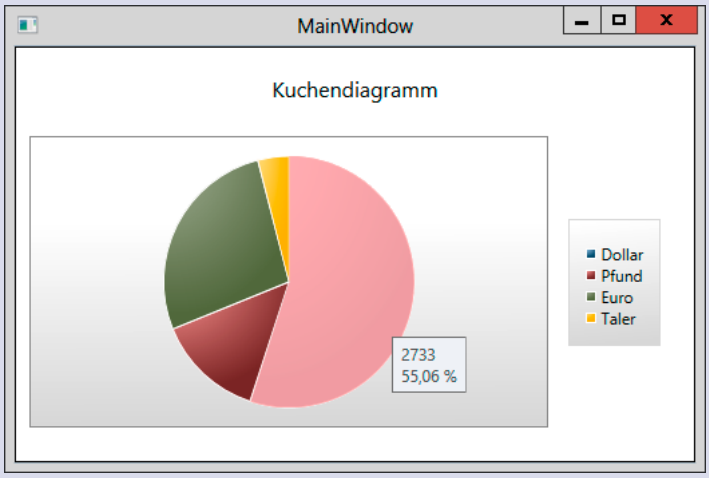
C#

```
...
  public MainWindow()
        {
            InitializeComponent();
            Dictionary<string, int> liste = new Dictionary<string, int>();
            liste.Add("Dollar", 2733);
            liste.Add("Pfund", 687);
            liste.Add("Euro", 1344);
            liste.Add("Taler", 200);
```

Daten zuweisen:

```
            this.DataContext = liste;
        }
...
```

Ergebnis

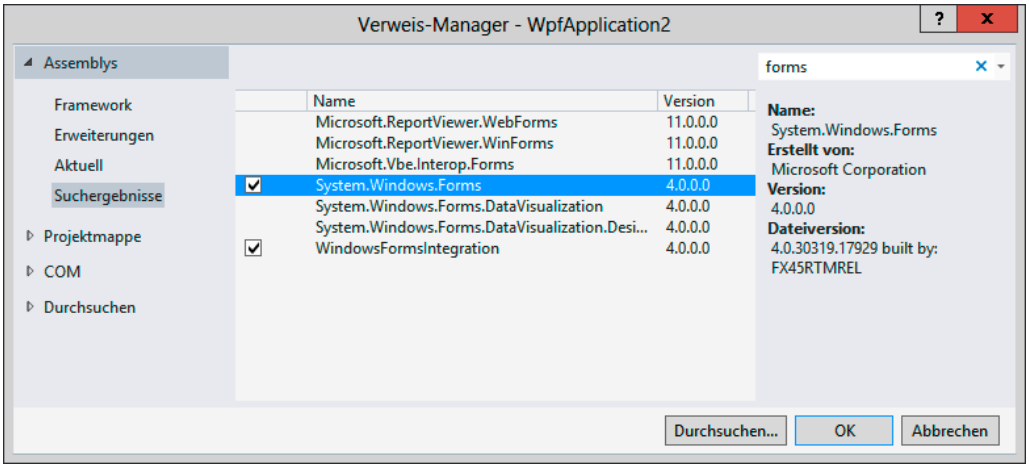


Bitte haben Sie Verständnis dafür, dass wir an dieser Stelle nicht alle Charttypen und Eigenschaften in epischer Breite vorstellen können. Im Internet finden Sie einige, mehr oder weniger aussagekräftige, Beispiele für die Arbeit mit dieser Library.

2.31 WindowsFormsHost

Wer immer noch seinen geliebten Windows Forms Controls nachtrauert, bekommt mit dem *WindowsFormsHost* die Möglichkeit, diese in eine WPF-Anwendung zu integrieren. Die Vorgehensweise entspricht etwa der beim entsprechenden Windows Forms Pendant.

Fügen Sie Ihrem Projekt zunächst einen Verweis auf die Assembly *System.Windows.Forms.dll* hinzu. Fügen Sie diesen Namespace sowohl der XAML als auch der Klassendefinition des gewünschten Fensters hinzu (siehe folgende Abbildung)



und setzen Sie das gewünschte Windows Forms-Control in ein *WindowsFormsHost*-Control (quasi als Container). Vergeben Sie einen Namen für das Windows Forms-Control, um im Quellcode auf dieses zugreifen zu können.

Beispiel 2.106: Die Windows Forms *CheckedListbox* in WPF nutzen

XAML

```
<Window x:Class="WindowsFormsHost_Bsp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Namespace einfügen für die Verwendung der Windows Forms-Controls:

```
        xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"
        Title="Verwendung Windows Forms Host" Height="207" Width="266">
    <Grid>
```

Hier kommt der Host:

```
        <WindowsFormsHost Name="windowsFormsHost1">
```

Und hier können Sie das Windows Forms-Control einfügen (vergessen Sie nicht den Verweis auf den WindowsForms-Namespace):

```
            <wf:CheckedListBox x:Name="checkedListbox1"
                Width="200" Height="150" BackColor="Red" />
```

Beachten Sie die Namensvergabe über das *x:Name*-Attribut, andernfalls wird Ihr Control später nicht gefunden!

```
        </WindowsFormsHost>
    </Grid>
</Window>
```

C#

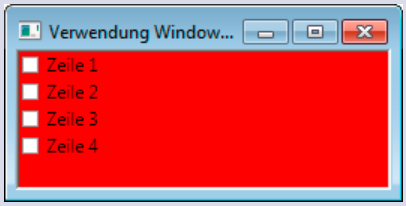
Namespace-Einbindung nicht vergessen:

```
...
using System.Windows.Forms;

namespace WindowsFormsHost_Bsp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
```

Und hier nutzen wir das Control wie gewohnt:

```
            InitializeComponent();
            checkedListbox1.Items.Add("Zeile 1");
            checkedListbox1.Items.Add("Zeile 2");
            checkedListbox1.Items.Add("Zeile 3");
            checkedListbox1.Items.Add("Zeile 4");
        }
    }
}
```

Ergebnis



3 Wichtige WPF-Techniken

Nachdem wir im vorhergehenden Kapitel einige ganz praktische Erfahrungen mit den WPF-Controls gesammelt haben, wollen wir uns jetzt mit den Besonderheiten der WPF-Programmierung im Vergleich zur Windows Forms-/Win32-Programmierung beschäftigen.

Damit Sie die WPF-Philosophie verstehen, wollen wir Ihnen zu Beginn einige wichtige Eckpfeiler dieser Technologie erklären.

3.1 Eigenschaften

Da die Definition und Verwendung von Eigenschaften in WPF etwas anders organisiert ist als in den bekannten Windows Forms-Anwendungen, sollten wir zunächst auf dieses Thema näher eingehen.

3.1.1 Abhängige Eigenschaften (Dependency Properties)

Mit den abhängigen (Dependency) und angehängten (Attached) Eigenschaften erweitert WPF das Spektrum der CLR-Eigenschaften. Abhängige Eigenschaften bieten gegenüber den „normalen“ Eigenschaften folgende erweiterte Möglichkeiten:

- eine interne Prüfung (Validierung),
- automatisches Aktualisieren von Werten,
- die Verwendung von Callback-Methoden zur Signalisierung von Wertänderungen
- sowie die Vorgabe von Defaultwerten.

Notwendig wurde diese Erweiterung des Eigenschaftensystems, um viele der WPF-Features (Animationen, Datenbindung Styles etc.) zu realisieren. So werden beispielsweise Werte von Eigenschaften überwacht, Änderungen führen automatisch zu Änderungen an den abhängigen Objekten.

Abhängige Eigenschaften werden nicht als private Felder definiert, sondern als statische, öffentliche Instanzen der Klasse *System.Windows.DependencyProperty*, die über *get-* und *set-*

Methoden angesprochen werden. Die Verwaltung der Eigenschaft wird vom WPF-Subsystem übernommen (daher auch die *Register*-Methode, siehe folgendes Beispiel), die der Eigenschaft übergeordnete Klasse stellt quasi nur noch eine Schnittstelle zu dieser Eigenschaft zur Verfügung.

Neben dem Wert können mit dem Default-Wert und der Callback-Methode auch weitere Metadaten zu einer Eigenschaft gespeichert werden.

Beispiel 3.1: Definition einer abhängigen Eigenschaft *Durchmesser* für ein Objekt *Kreis*

C#

```
using System.Windows;

namespace BSP_Controls
{
```

Wir definieren eine neue Klasse *Kreis* und leiten diese gleich von *DependencyObject* ab:

```
    class Kreis : DependencyObject
    {
```

Hier die eigentliche Definition der abhängigen Eigenschaft (übergeben werden der Name, der Datentyp, das abhängige Objekt, optional die Metadaten, d. h. der Defaultwert und eine Callback-Methode):

```
        public static readonly DependencyProperty DurchmesserProperty =
            DependencyProperty.Register("Durchmesser", typeof(double),
typeof(Kreis),
                                        new FrameworkPropertyMetadata(11.1,
                                        new
PropertyChangedCallback(OnDurchmesserChanged)));
```

Wie Sie sehen, handelt es sich nicht mehr um ein privates Feld, vielmehr wird die bisher übliche Kapselung aufgegeben, die Verwaltung des Zustands wird von WPF übernommen, die Instanz meldet seine „lokalen Speicher“ nur noch an (*Register*).

Die folgende Definition ist nur noch die allgemeine Schnittstelle nach außen, wie Sie es auch von den normalen Eigenschaften kennen:

```
        public double Durchmesser
        {
            get { return (double)(GetValue(DurchmesserProperty)); }
            set { SetValue(DurchmesserProperty, value); }
        }
```

Hier eine Callback-Methode, mit der eine Wertänderung überwacht werden kann:

```
        private static void OnDurchmesserChanged(DependencyObject obj,
            DependencyPropertyChangedEventArgs args)
        {
            MessageBox.Show( (obj as Kreis).Durchmesser.ToString());
        }
    }
}
```

3.1.2 Angehängte Eigenschaften (Attached Properties)

Mit den angehängten Eigenschaften (*Attached Properties*), einer speziellen Form der Dependency Properties, wird der Versuch unternommen, die Flut an WPF-Element-Eigenschaften etwas einzudämmen. Das Problem: Wird ein Element/Control in einem Container platziert, hängt beispielsweise dessen Position vom umgebenden Container (*Grid, Canvas, DockPanel* etc.) ab. Für jede Art von Container werden aber andere Eigenschaften zur Positionierung benötigt. Aus diesem Grund stellen die übergeordneten Elemente die zum Positionieren nötigen Eigenschaften zur Verfügung (*Canvas.Top, Canvas.Left, DockPanel.Dock, Grid.Column* ...), das eingelagerte Element nutzt diese lediglich in seinem Kontext. Der Vorteil: nur wenn sich beispielsweise ein *Button* in einem *Canvas* befindet, werden auch die Eigenschaften *Canvas.Top, Canvas.Left* bereitgestellt und verwendet.

Beispiel 3.2: Positionieren einer Schaltfläche in einem *Canvas*-Control mit Attached Properties

XAML

```
<Canvas Name="canvas1" >
  <Button Canvas.Left="74" Canvas.Top="70" Height="45" Name="button1"
          Width="89">Beschriftung
  </Button>
</Canvas>
```

3.2 Einsatz von Ressourcen

Bevor wird uns im Weiteren mit Styles etc. beschäftigen, müssen wir zunächst noch einen Blick auf die Ressourcen-Verwaltung in WPF-Anwendungen werfen, da diese die Voraussetzung für die Zuordnung darstellt.

3.2.1 Was sind eigentlich Ressourcen?

In WPF-Anwendungen zählen nicht nur Grafiken, Strings, Sprachinformationen oder beliebige binäre Informationen zu den Ressourcen, sondern auch die Beschreibung von Styles, Füllmustern oder sogar ganzer Controls.

Eine Ressource besteht immer aus einem eindeutigen Schlüssel (Key) und dem eigentlichen Content, der jederzeit austauschbar ist, da Bezüge auf die Ressource immer nur den Schlüssel verwenden.

3.2.2 Wo können Ressourcen gespeichert werden?

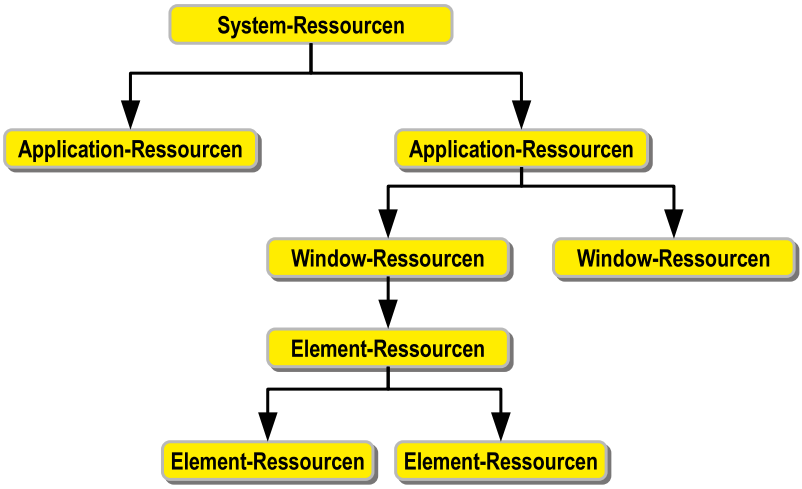
Eine Möglichkeit, Ressourcen in Ihrer Anwendung abzulegen, haben Sie sicher ganz unbewusst schon zur Kenntnis genommen. Die Rede ist von der Datei *App.xaml*, in der bereits ein *Resources*-Abschnitt vordefiniert ist:

```
<Application x:Class="BSP_Controls.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Test.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Hierbei handelt es sich um anwendungsweit verfügbare Ressourcen, die allen Windows/Pages und deren Elementen zur Verfügung stehen.

Neben diesen Ressourcen können Sie auch jedem Element und jedem Window eigene Ressourcen zuordnen, zusätzlich sind auch so genannte System-Ressourcen (z. B. Systemfarben) verfügbar.

Die folgende Abbildung zeigt die mögliche Hierarchie von Ressourcen:



HINWEIS: Wird eine Ressource referenziert und damit auch gesucht, beginnt die Suche immer beim aktuellen Element. Wird die Ressource hier nicht gefunden, wird im übergeordneten Element (Container → Window → Application → System) gesucht. Damit ist auch klar, dass untergeordnete Elemente gleichnamige Ressourcen von übergeordneten Elementen überschreiben können. Innerhalb einer Hierarchieebene ist dies nicht möglich, da es sich in diesem Fall nicht um einen eindeutigen Schlüssel handelt.

Im XAML-Quellcode stellt sich die Definition von *Resources*-Abschnitten wie folgt dar:

Die Window-Ressourcen:

```
<Window>
  <Window.Resources>
    ...
  </Window.Resources>

  <StackPanel>
```

Die Ressourcen eines Containers:

```
    <StackPanel.Resources>
      ...
    </StackPanel.Resources>
...
    <Button>
```

Die Ressourcen eines Elements:

```
      <Button.Resources>
      ...
      </Button.Resources>
    </Button>
  </StackPanel>
</Window>
```

3.2.3 Wie definiere ich eine Ressource?

Haben Sie sich dafür entschieden, auf welcher Ebene Sie die Ressource definieren, können Sie zur Tat schreiten:

- Zunächst müssen Sie sich über die Art der Ressource im Klaren sein, diese bestimmt den Elementnamen (z. B. ein *ImageBrush,* mit dem ein Hintergrund festgelegt werden kann).
- Neben dieser Information müssen Sie sich noch für einen eindeutigen Schlüssel entscheiden, über den die Ressource angesprochen werden soll.
- Last, but not least, müssen Sie auch noch die eigentlichen Informationen zur Ressource (beim *ImageBrush* ist dies die *ImageSource)* festlegen.

Beispiel 3.3: Erzeugen und Verwenden einer Ressource auf Fenster-Ebene

XAML

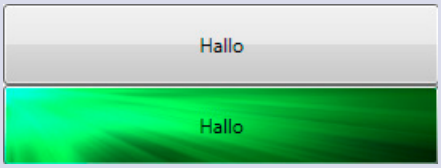
```
<Window x:Class="Konzepte.Ressourcen_Bsp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Ressourcen_Bsp" Height="329" Width="464">
 <Window.Resources>
   <ImageBrush x:Key="bck" ImageSource="back.jpg" />
 </Window.Resources>
 ...
```

Die Verwendung:

```
<StackPanel>
  Button Background="{StaticResource bck}">Hallo</Button>
</StackPanel>
...
```

Ergebnis

Die Unterschiede in der Darstellung dürften Ihnen sicher auffallen:



HINWEIS: Beim Einsatz von (statischen) Ressourcen ist es wichtig, dass diese **vor** der Verwendung definiert wurden, andernfalls kann die Ressource nicht gefunden werden. Zusätzlich müssen Sie auf die Schreibweise achten, hier wird die Groß-/Kleinschreibung berücksichtigt!

Alternativ können Sie bei der Verwendung der Ressource auch die folgende Syntax nutzen:

```
<Button Height="50">
  <Button.Background>
    <StaticResource ResourceKey="bck" />
  </Button.Background>
  Hallo
</Button>
```

3.2.4 Statische und dynamische Ressourcen

Wie Sie im vorhergehenden Beispiel bereits gesehen haben, können Ressourcen statisch, d. h. unveränderlich, über den Schlüssel zugeordnet werden:

```
StackPanel>
  <Button Background="{StaticResource bck}">Hallo</Button>
</StackPanel>
```

Voraussetzung war das vorherige Definieren dieser Ressource. Spätere Änderungen an der Ressource werden nicht registriert und haben keine Auswirkung auf das verwendende Element.

Neben dieser Variante besteht auch die Möglichkeit, Ressourcen dynamisch festzulegen. In diesem Fall können Sie die Ressourcen zur Laufzeit zuordnen bzw. bereits vorhandene Ressourcen einfach austauschen.

Beispiel 3.4: Verwendung einer noch nicht definierten dynamischen Ressource

XAML

Zunächst die Zuordnung von Eigenschaft und Ressource:

```
<Button Height="50" Background="{DynamicResource bck1}"
        Click="Button_Click">Hallo</Button>
```

C#

Mit dem Klick auf die Schaltfläche wollen wir eine Ressource zuordnen:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
```

Neuen *ImageBrush* erzeugen:

```
    ImageBrush imgbck = new ImageBrush();
```

Eine Bitmap zuweisen:

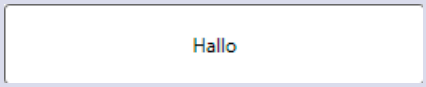
```
    imgbck.ImageSource = new BitmapImage(new
Uri("pack://application:,,,/back2.jpg"));
```

Die Ressource erzeugen (Window-Ressource):

```
    this.Resources.Add("bck1", imgbck);
}
```

Ergebnis

Vor dem Klick auf die Schaltfläche:



Nach dem Klick auf die Schaltfläche:

Und warum verwenden wir nicht einfach immer dynamische Ressourcen? Da die Verwaltung dynamischer Ressourcen deutlich aufwändiger ist, würden hier unnötigerweise Systemressourcen verschwendet werden.

3.2.5 Wie werden Ressourcen adressiert?

Möchten Sie auf eingebundene Ressourcen (z. B. Grafiken) Ihrer Anwendung zugreifen, müssen Sie beispielsweise bei der Zuordnung von Grafiken per Code eine bestimmte Syntax einhalten, andernfalls wird die Ressource an der falschen Stelle gesucht und dann wohl auch nicht gefunden.

Die dazu erforderliche URI können Sie absolut, d. h. mit voller Pfadangabe, inklusive der aktuellen Assembly angeben oder relativ zur aktuellen Assembly.

Beispiel 3.5: Absolute Pfadangabe (Bild liegt in der Projekt-Root, Buildvorgang= Resource)

C#

```
Uri uri = new Uri("pack://application:,,,/Bild.jpg");
```

Beispiel 3.6: Absolute Pfadangabe (das Bild liegt im Unterverzeichnis \\Images des aktuellen Projekts, Buildvorgang=Resource)

C#

```
Uri uri = new Uri("pack://application:,,,/Images/Bild.jpg");
```

Beispiel 3.7: Relative Pfadangabe (das Bild liegt im gleichen Verzeichnis wie die Assembly)

C#

```
Uri uri = new Uri(@".\Back3.jpg", UriKind.Relative);
```

Beispiel 3.8: Relative Pfadangabe (das Bild liegt im relativen Unterverzeichnis \\Images der Assembly)

C#

```
Uri uri = new Uri(@".\Images\Back3.jpg", UriKind.Relative)
```

Neben den gezeigten Möglichkeiten können Sie unter anderem auch auf eingebundene Assemblies zugreifen, eine (mehr oder weniger fehlerhafte) Dokumentation zu dieser Gesamtthematik finden Sie unter *http://msdn.microsoft.com/de-de/library/aa970069.aspx*.

3.2.6 System-Ressourcen einbinden

Neben den in Ihrer Anwendung definierten Ressourcen können Sie auch System-Ressourcen verwenden. Die Einbindung erfolgt entweder statisch oder dynamisch, im erstem Fall reagiert die Anwendung jedoch nicht auf aktuelle Änderungen an den Systemeinstellungen.

Beispiel 3.9: Einbinden von System-Ressourcen

XAML

Anzeige der *VirtualScreenWidth* in einem *Label* (Ressourcenabfrage):

```
<Label Content="{StaticResource {x:Static
SystemParameters.VirtualScreenWidthKey}}"/>
```

Anzeige, ob User-Interface-Effekte aktiviert sind (Wertabfrage):

```
<CheckBox IsChecked="{x:Static SystemParameters.UIEffects}" Content="UIEffects" />
```

3.3 Das WPF-Ereignis-Modell

Nachdem wir in den bisherigen Beispielen recht sparsam mit der Verwendung von Event-Handlern umgegangen sind, wollen wir uns jetzt diesem Thema etwas intensiver widmen.

3.3.1 Einführung

Sicher haben Sie auch schon mehr oder weniger unbewusst von den Ereignissen in WPF Gebrauch gemacht. Nach dem Klick, z. B. auf eine Schaltfläche, wird der entsprechende Ereignis-Handler von Visual Studio bereitgestellt.

Beispiel 3.10: *Button* mit zugehöriger Ereignisbehandlung

XAML

```
<StackPanel>
  <Button Content="Klick mich!" Click="Button_Click"/>
</StackPanel>
```

C#

Der Ereignis-Handler:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hallo");
}
```

So weit – so gut, das kennen Sie sicher auch schon von der Programmierung in Win32-/Windows Forms-Anwendungen so. Doch was passiert, wenn Sie in WPF eine Schaltfläche aus einzelnen Elementen „zusammenbasteln“?

Beispiel 3.11: Button mit Text und Grafik

XAML

```
<Button Click="Button_Click_1">
  <StackPanel Orientation="Horizontal" Margin="10">
    <Image Source="Images/Flash.png" Width="56" Height="46" />
    <TextBlock VerticalAlignment="Center">Klick mich!</TextBlock>
  </StackPanel>
</Button>
```

Ergebnis

Rein intuitiv haben Sie sicher auch das *Click*-Ereignis dem Button zugeordnet. Doch warum sollte das eigentlich funktionieren? Was passiert, wenn der Nutzer auf das *Image* oder den *TextBlock* klickt, zusätzlich befindet sich „darunter" ja noch das *StackPanel*.

Das Zauberwort heißt hier „Routed Events", ein Verfahren, um auftretende Ereignisse in der Element-Hierarchie weiterzugeben.

3.3.2 Routed Events

WPF unterscheidet bei den Routed Events zwei verschiedene Varianten, die Sie als Programmierer auch auseinander halten sollten:

- **Tunneling Events**
  Ausgehend vom Wurzelelement (*Window/Page)* werden die Ereignisse bis zum auslösenden Element weitergereicht. Diese Events werden **vor** den zugehörigen Bubbling Events ausgelöst.
- **Bubbling Events**
  Ausgehend vom aktivierten Element werden die Ereignisse zum jeweils übergeordneten Element weitergereicht, d. h. im Endeffekt bis zum *Window* oder zur *Page*.

HINWEIS: Tunneling Events sind durch den Vorsatz „Preview..." gekennzeichnet, Bubbling Events verzichten auf einen Vorsatz.

Beispiel 3.12: Ablauf der Ereigniskette für einen Mausklick mit der linken Taste

XAML

```
<Window x:Class="Konzepte.Ereignisse"
...
    Title="Ereignisse" MouseLeftButtonDown="Window_MouseLeftButtonDown"

PreviewMouseLeftButtonDown="Window_PreviewMouseLeftButtonDown">
   <Button MouseLeftButtonDown="Button_MouseLeftButtonDown"
           PreviewMouseLeftButtonDown="Button_PreviewMouseLeftButtonDown">
     <StackPanel Orientation="Horizontal" Margin="10"
                 MouseLeftButtonDown="StackPanel_MouseLeftButtonDown"

PreviewMouseLeftButtonDown="StackPanel_PreviewMouseLeftButtonDown">
       <Image Source="Images/Flash.png" Width="56" Height="46"
              MouseLeftButtonDown="Image_MouseLeftButtonDown"
              PreviewMouseLeftButtonDown="Image_PreviewMouseLeftButtonDown" />
       <TextBlock VerticalAlignment="Center"
              MouseLeftButtonDown="TextBlock_MouseLeftButtonDown"

PreviewMouseLeftButtonDown="TextBlock_PreviewMouseLeftButtonDown">Klick mich!
       </TextBlock>
     </StackPanel>
   </Button>
</Window>
```

Ergebnis

Die Ereigniskette nach einem Klick auf das *Image*:

Window: *PreviewMouseLeftButtonDown*

Button: *PreviewMouseLeftButtonDown*

StackPanel: *PreviewMouseLeftButtonDown*

Image: *PreviewMouseLeftButtonDown*

Image: *MouseLeftButtonDown*

StackPanel: *MouseLeftButtonDown*

Wie Sie sehen können, wird zunächst die komplette Tunneling-Ereigniskette durchlaufen, nachfolgend die Bubbeling-Events[1].

Im Normalfall werden Sie wohl nur Bubbling-Events verwenden, Tunneling-Events nutzen Sie beispielsweise, um Ereignisse bzw. deren Weiterleitung zu blockieren.

Beispiel 3.13: Das Weiterleiten der Ereignisse verhindern

C#

Wir werten gleich das erste Ereignis aus (das Tunneling Event beginnt an der Root, d. h. dem Window):

```
        private void Window_PreviewMouseLeftButtonDown(object sender,
 MouseButtonEventArgs e)
        {
```

Ereignis behandelt (Klappe zu, Affe tot):

```
            e.Handled = true;
            List1.Items.Add("Window_PreviewMouseLeftButtonDown");
        }
```

Beispiel 3.14: Das auslösende Element bestimmen

C#

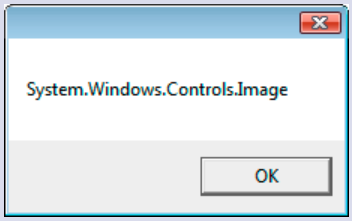
Den Ursprung für ein Ereignis können Sie mit dem *...EventArgs.Source*-Parameter bestimmen:

```
        private void Window_PreviewMouseLeftButtonDown(object sender,
 MouseButtonEventArgs e)
        {
            MessageBox.Show(e.Source.ToString());
            List1.Items.Add("Window_PreviewMouseLeftButtonDown");
        }
```

[1] Der Button löst ein *Click*-Ereignis aus, die dazu nötige Logik verhindert das Auslösen entsprechender *MouseLeftButtonDown*-Ereignisse, deshalb ist hier die Ereigniskette zu Ende.

Ergebnis

Beim Klick auf das Image wird auch dieses als *Source* übermittelt:



3.3.3 Direkte Events

Auch diese Form der Ereignisse ist nach wie vor präsent, hierbei handelt es sich um die ganz normalen .NET-Ereignisse, wie Sie auch in Windows-Forms-Anwendungen auftreten.

Direkte Events werden eingesetzt, wenn die Verwendung von Bubbling oder Tunneling keinen Sinn macht, beispielsweise beim *MouseLeave*-Ereignis, das sehr objektbezogen ausgelöst wird.

Sie erkennen diese Ereignisse an einem fehlenden *Preview...*-Pendant.

3.4 Verwendung von Commands

Im Zusammenhang mit der Entwicklung von Anwendungen ist es Ihnen sicher auch schon passiert, dass Sie eine Menüfunktion zum x-ten Male neu programmiert haben. Das Gleiche trifft sicher ebenfalls auf den Toolbar zu. Das Prozedere ist doch immer gleich:

1. Methode mit der eigentlichen Logik erstellen
2. Menüpunkt erstellen
3. Menüpunkt Tastenkürzel zuweisen, Tastaturabfrage implementieren
4. Menüpunkt Ereignismethode zuweisen und Methode von 1. aufrufen
5. Toolbar-Button bereitstellen
6. Toolbar-Button Ereignismethode zuweisen und Methode von 1. aufrufen
7. Logik für das Sperren von 2. und 5. erstellen, wenn die Funktion nicht zur Verfügung steht.

3.4.1 Einführung zu Commands

In WPF-Anwendungen können Sie diesen Aufwand wesentlich verringern, indem Sie entweder die von WPF vordefinierten Commands verwenden, oder indem Sie selbst eigene Commands implementieren.

Die neue Vorgehensweise (vordefinierte Commands, z. B. Einfügen in die Zwischenablage) im Vergleich zum bisherigen Vorgehen:

1. entfällt, da für viele Controls bereits implementiert
2. **Menüpunkt erstellen und Command zuweisen**
3. entfällt, da per Command automatisch zugewiesen
4. entfällt, da per Command automatisch zugewiesen
5. **Toolbar-Button bereitstellen und Command zuweisen**
6. entfällt, da per Command automatisch zugewiesen
7. entfällt, da Command diese Logik bereitstellt.

Das sieht doch schon wesentlich freundlicher aus als die mühsame und fehleranfällige erste Variante. Ähnlich einfach gestaltet sich die Wiederverwendung von selbst erstellten Kommandos, wenn Sie diese bereits einmal erstellt haben.

3.4.2 Verwendung vordefinierter Commands

Statt vieler Worte wollen wir Ihnen zunächst an einem Beispiel die Vorgehensweise bei der Verwendung von Commands demonstrieren.

Beispiel 3.15: Programmieren der Funktionen „Kopieren und Einfügen“ für ein Formular mit zwei TextBoxen

XAML

Der XAML-Code der Oberfläche:

```
<Window x:Class="Konzepte.Commands_Bsp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Commands_Bsp" Height="373" Width="411">
 <DockPanel>
```

Die Menüdefinition:

```
    <Menu DockPanel.Dock="Top" Height="22" Name="menu1">
...
      <MenuItem Header="_Bearbeiten">
```

Hier werden die beiden Menüpunkte „Kopieren“ und „Einfügen“ definiert:

```
        <MenuItem Command="ApplicationCommands.Copy"/>
        <MenuItem Command="ApplicationCommands.Paste"/>
```

Sie können auf die Angabe des Headers sowie der Tastenkürzel verzichten, dies stellt die obige *Command*-Definition automatisch zur Verfügung[2].

```
      </MenuItem>
    </Menu>
    <ToolBarTray DockPanel.Dock="Top">
      <ToolBar>
```

Hier findet sich bereits die Definition für die *ToolBar*-Buttons, auch in diesem Fall genügt die Zuordnung der *Command*-Eigenschaft:

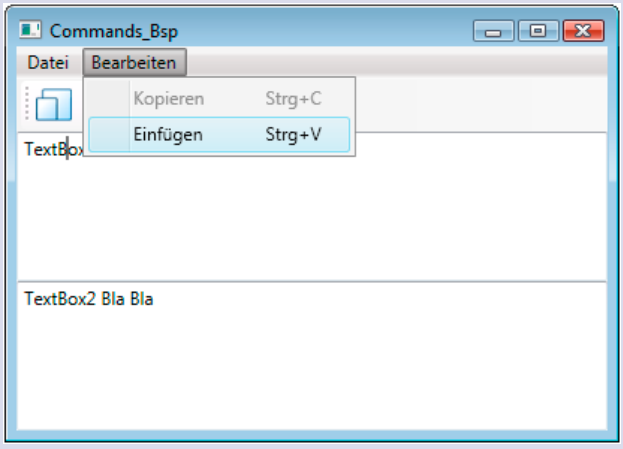
```
        <Button Width="30" Height="30" Command="ApplicationCommands.Copy">
          <Image Source="Images/editcopy.png"/>
        </Button>
        <Button Width="30" Height="30" Command="ApplicationCommands.Paste">
          <Image Source="Images/editpaste.png"/>
        </Button>
      </ToolBar>
    </ToolBarTray>
    <StackPanel>
```

Hier noch die beiden *TextBox*en, für die die Funktionen implementiert werden:

```
      <TextBox Name="txt1" Height="100">TextBox1 Bla Bla</TextBox>
      <TextBox Name="txt2" Height="100">TextBox2 Bla Bla</TextBox>
...
    </StackPanel>
  </DockPanel>
</Window>
```

Ergebnis

Und wo bleibt der Code? Kurze Antwort: Ihr Programm ist an dieser Stelle bereits fertig. Nach dem Start können Sie sich von der Funktionstüchtigkeit überzeugen:



[2] Geben Sie trotzdem einen Wert an, überschreibt dies die vom Command bereitgestellten Werte.

Beide Menüpunkte verfügen über eine Beschriftung und ein Tastenkürzel, befindet sich Text in der Zwischenablage, ist das Menü *Einfügen* aktiviert, andernfalls ist der Menüpunkt gesperrt. Der Menüpunkt *Kopieren* ist nur aktiv, wenn Sie in einer der *TextBox*en Text markieren und diese *TextBox* den Eingabefokus besitzt.

Nicht schlecht, wenn Sie dies mit der konventionellen Variante vergleichen, bei der Sie sicher wesentlich mehr Code produziert hätten.

3.4.3 Das Ziel des Commands

Doch gerade bei obigem Beispiel wird sicher bei manchem die Frage aufkommen, wohin denn eigentlich der Text eingefügt wird, haben wir doch zwei Textfelder. Die Frage ist sicher berechtigt, aber im obigen Beispiel noch recht einfach lösbar: Ziel des jeweils gewählten Commands ist standardmäßig immer das gerade aktive Control, d. h. die *TextBox*, die den Eingabefokus besitzt.

Schwieriger wird es, wenn die Zwischenablageinhalte gezielt in ein bestimmtes Control eingefügt werden sollen. In diesem Fall müssen Sie neben der *Command*- auch die *CommandTarget*-Eigenschaft definieren. Allerdings genügt in diesem Fall nicht die reine Angabe des Element-Namens, Sie müssen die Binding-Syntax verwenden.

Beispiel 3.16: Zwischenablageinhalte sollen immer in *TextBox2* eingefügt werden

XAML

Wir fügen dem betreffenden Menüpunkt ein *CommandTarget* hinzu:

```
  <MenuItem Header="_Bearbeiten">
    <MenuItem Command="ApplicationCommands.Copy"/>
    <MenuItem Command="ApplicationCommands.Paste" CommandTarget="{Binding
ElementName=txt2}"/>
  </MenuItem>
...
```

Ergebnis

Starten Sie jetzt das Programm, ist es egal, welches Control den Fokus besitzt, drücken Sie die Tastenkombination *Strg+V* oder wählen Sie den entsprechenden Menüpunkt, wird der Text immer in die *TextBox2* eingefügt.

HINWEIS: Diese Vorgehensweise müssen Sie auch wählen, wenn Sie Commands von einzelnen Schaltflächen aus starten wollen. Da diese den Fokus erhalten können, würde nie klar sein, welches Ziel die Aktion haben soll.

Beispiel 3.17: „Freistehender" *Button* für das Einfügen des Zwischenablageinhalts

XAML

```
<Button Command="ApplicationCommands.Paste" CommandTarget="{Binding
ElementName=txt2}">
    Paste (TextBox 2)
</Button>
```

3.4.4 Vordefinierte Commands

WPF bietet bereits „ab Werk" eine ganze Reihe von Commands, die in der täglichen Programmierpraxis immer wieder anfallen. Diese Commands gliedern sich in die folgenden Gruppen:

- *ApplicationCommands*
  (z. B. *Cut, Past, Help, New, Print, Save, Stop, Undo*)
- *ComponentCommands*
  (z. B. *ExtendSelectionLeft*, *MoveLeft*)
- *NavigationCommands*
  (z. B. *FirstPage, GotoPage, Refresh*, *Search*)
- *MediaCommands*
  (z. B. Play, *Pause, FastForward*, *IncreaseVolume*)
- *EditingCommands*
  (z. B. *Delete*, *MoveUpByLine, ToggleBold*[3])

HINWEIS: Doch Achtung: Nicht das Command an sich stellt die Logik, z. B. für das Einfügen von Zwischenablageinhalten, zur Verfügung, sondern das jeweilige Ziel-Objekt, d. h. ein spezifisches Control wie *TextBox* oder *Image.*

Ob, und wenn ja welche, Commands implementiert sind, müssen Sie von Fall zu Fall ausprobieren. Sehr umfassend ist z. B. die Unterstützung bei *TextBox* und *RichTextBox* sowie beim *MediaElement.*

3.4.5 Commands an Ereignismethoden binden

Wie schon erwähnt, implementiert nicht jedes Control alle verfügbaren Commands. So steht zwar ein *ApplicationCommand.Open* (d. h. Beschriftung und Tastaturkürzel) zur Verfügung, im Normalfall passiert allerdings nichts, da kein Control eine entsprechende Logik implementiert hat, was bei derart komplexen Abläufen sicher auch nicht möglich ist.

[3] siehe dazu auch ab Abschnitt 2.16 (*RichtTextBox*)

Aus diesem Grund besteht die Möglichkeit, einem Command eine entsprechenden Ereignisbehandlung per *CommandBinding* zuzuordnen. Zwei Ereignisse sind hier von zentraler Bedeutung:

- *Execute (*die eigentlich auszuführende Logik) und
- *CanExecute* (eine Abfrage, ob die Funktion überhaupt zur Verfügung steht)

Beispiel 3.18: Implementieren des *ApplicationCommand.Open*

XAML

Nutzen Sie für die Zuordnung der beiden Ereignismethoden am besten das Wurzel-Element (*Windows/Page*):

```
<Window x:Class="Konzepte.Commands_Bsp"
...
    Title="Commands_Bsp" Height="373" Width="411">
 <Window.CommandBindings>
   <CommandBinding Command="ApplicationCommands.Open"
                   Executed="OpenCmdExecuted" CanExecute="OpenCmdCanExecute"/>
 </Window.CommandBindings>
...
```

C#

Die beiden zugehörigen Ereignismethoden aus der Klassendefinition des Windows:

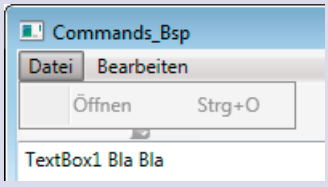
```
void OpenCmdExecuted(object target, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Was soll ich öffnen????");
    // hier steht die eigentliche Logik
}
```

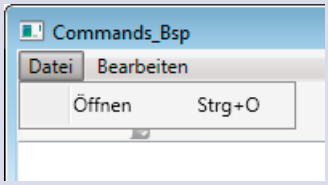
Nur wenn die erste *TextBox* auch leer ist, ist das Öffnen neuer Dateien möglich:

```
void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    if (txt1.Text.Length == 0)
       e.CanExecute = true;
    else
       e.CanExecute = false;
}
```

Ergebnis

Ein Test zur Laufzeit zeigt, dass die entsprechende Schaltfläche nur freigegebenen ist, wenn die *TextBox* leer ist:





Klicken Sie auf den obigen Menüpunkt oder verwenden Sie die Tastenkombination *Strg+O*, wird unsere MessageBox aus der Methode *OpenCmdExecuted* ausgeführt.

HINWEIS: Selbstverständlich können Sie CommandBinding auch per Code realisieren, wie das folgende Beispiel zeigt.

Beispiel 3.19: Ereignismethode per Code zuweisen

C#

```
public Commands_Bsp()
{
    InitializeComponent();
    CommandBinding cmdOpen = new CommandBinding(ApplicationCommands.Open);
    cmdOpen.Executed += new ExecutedRoutedEventHandler(cmdOpen_Executed);
}

void cmdOpen_Executed(object sender, ExecutedRoutedEventArgs e)
{ ... }
```

3.4.6 Wie kann ich ein Command per Code auslösen?

Neben der bereits beschriebenen Variante, Commands per Zuordnung im XAML-Code auszulösen, besteht auch die Möglichkeit, diese direkt mit der *Execute*-Methode aufzurufen.

Beispiel 3.20: Direkter Aufruf eines Commands in einer Ereignismethode

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    ApplicationCommands.Paste.Execute(null, txt2);
}
```

HINWEIS: Im zweiten Parameter übergeben Sie das *CommandTarget*.

3.4.7 Command-Ausführung verhindern

Nicht immer und zu jeder Zeit können Kommandos einfach ausgeführt werden. Unter bestimmten Bedingungen steht eine Funktion nicht zur Verfügung, in diesem Fall sollen die Menüpunkte/Schaltflächen abgeblendet sein, um den Anwender nicht unnötig zu verwirren.

Die Lösungsmöglichkeit haben wir Ihnen bereits beim Zuordnen von Ereignismethoden gezeigt. Im *...CanExecute*-Ereignis wird mit dem Parameter *e.CanExecute* entschieden, ob eine Aktion ausführbar ist oder nicht.

Beispiel 3.21: Command-Ausführung verhindern

C#

```
void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
      e.CanExecute = false;
}
```

3.5 Das WPF-Style-System

Mit den WPF-Styles kommen wir jetzt zu einem Thema, das sicher von ganz zentraler Bedeutung für die oberflächenorientierten WPF-Anwendungen ist und auch einen der wesentlichsten Unterschiede zu den konventionellen Windows-Anwendungen darstellt.

3.5.1 Übersicht

Doch worum geht es eigentlich? Sicher haben Sie nach der Lektüre der beiden vorhergehenden Kapitel festgestellt, dass WPF-Controls mit Bergen von Eigenschaften ausgestattet sind, die es möglich machen, fast alle Aspekte der Darstellung zu beeinflussen.

Doch gerade für aufwändige Oberflächen ergeben sich einige Fragen:

1. Wie kann ich mehr als einem Control ein spezifisches Aussehen zuweisen?
2. Wie kann ich das Aussehen unter bestimmten Bedingungen ändern?
3. Wie kann ich das grundsätzliche Aussehen eines Controls komplett ändern?
4. Wie kann ich einfache Animationen realisieren?

Für alle diese Fragen bietet WPF eine Antwort:

5. Verwendung von benannten oder Typ-Styles
6. Verwendung von Triggern
7. Verwendung von Templates
8. Verwendung von StoryBoards

HINWEIS: Im Rahmen dieses Abschnitts werden wir die o. g. Themen nur recht oberflächlich streifen, da dies eigentlich ein Hauptarbeitsgebiet für den Designer und nicht für den Programmierer ist[4]. Außerdem ist Visual Studio für einige der obigen Aufgaben das falsche Tool. Hier kommen Sie mit *Blend for Visual Studio* (siehe Abschnitt 3.9) wesentlich weiter, da der Designer teilweise komfortabler ist. Wir weisen Sie an den entsprechenden Stellen darauf hin.

3.5.2 Benannte Styles

In den bisherigen Beispielen haben wir die Eigenschaften jedes Controls einzeln gesetzt, was bei größeren Ansammlungen recht schnell zum Geduldsspiel ausarten kann. Denken Sie nur an unseren Taschenrechner aus dem WPF-Einführungskapitel, wo ca. zwanzig einzelne Tasten zu konfigurieren sind (Schrift, Randabstände, Farben etc.). Viel schöner wäre doch hier eine zentrale Vorschrift, wie ein derartiger Button auszusehen hat.

Genau diese Aufgabe übernimmt ein *benannter Style*. Dieser wird einmal definiert und kann dann per Key den einzelnen Elementen zugewiesen werden.

Beispiel 3.22: Alle Schaltflächen für den Taschenrechner sollen einen Randabstand von 2, eine fette Schrift und eine gelbe Hintergrundfarbe bekommen.

XAML

```
<Window x:Class="Konzepte.Taschenrechner_1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window2" Height="199" Width="388">
```

Oh, jetzt wird auch klar, warum wir uns in diesem Kapitel bereits mit Ressourcen beschäftigt haben. Hoffentlich haben Sie diesen Abschnitt nicht gelangweilt überblättert!

```
    <Window.Resources>
```

Wir definieren einen neuen Style und legen dessen *Key* fest (über diesen könne wir später auf den Style zugreifen bzw. auf diesen verweisen):

```
    <Style x:Key="myBtnStyle">
```

Innerhalb des *Style*-Elements können Sie per *Setter* die gewünschten Eigenschaften beeinflussen:

```
      <Setter Property="Control.Margin" Value="2" />
      <Setter Property="Control.Background" Value="Yellow" />
      <Setter Property="Control.FontWeight" Value="UltraBold" />
```

[4] Hier ist die Borderline zwischen Designer und Programmierer, mit teilweise diffusem Verlauf ...

Legen Sie jeweils *Property* und *Value* fest.

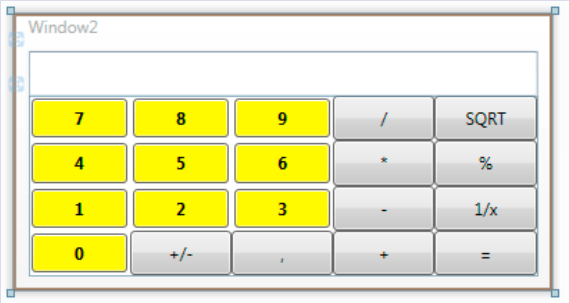
```
    </Style>
  </Window.Resources>
```

Und jetzt kommt unser neuer Style zum Einsatz (setzen Sie diesen nur bei den Zifferntasten):

```
    <UniformGrid Name="uniformGrid1" Columns="5" Rows="4" Grid.Row="1">
      <Button  Name="button1"  Style="{StaticResource myBtnStyle}">7</Button>
      <Button  Name="button2"  Style="{StaticResource myBtnStyle}">8</Button>
      <Button  Name="button3"  Style="{StaticResource myBtnStyle}">9</Button>
  ...
```

Ergebnis

Schon im Designer dürfte sich jetzt etwas getan haben:



Was passiert eigentlich, wenn wir den Style einer *TextBox* zuweisen? Probieren Sie es ruhig aus, es kann nichts passieren. Vielleicht sind Sie überrascht, aber auch die *TextBox* wird nach dieser Aktion im grässlichen gelben Outfit erscheinen und fette Schrift anzeigen.

Warum dies so ist? Ganz einfach, auch die *TextBox* verfügt über die aufgeführten Eigenschaften und übernimmt diese automatisch vom Style.

3.5.3 Typ-Styles

Ein „fauler“ Programmierer wird immer einen einfacheren Weg suchen, und so ist es sicher mühsam, den Style jedem einzelnen Button einzeln zuzuweisen, zumal die Syntax auch recht umfangreich ist. Aus diesem Grund gibt es noch eine zweite Art von Styles, die nicht über einen Key, sondern indirekt über den Klassennamen zugeordnet werden.

Der Vorteil: Alle WPF-Elemente, die Instanzen dieser Klasse sind, übernehmen automatisch diesen Style, ohne dass dies explizit angegeben werden muss.

Beispiel 3.23: (Fortsetzung) Mit Ausnahme der Zifferntaste sollen alle Tasten einen grünen Hintergrund und weiße Schrift bekommen.

XAML

```
  <Window.Resources>
```

Hier der benannte Style für die Zifferntasten:

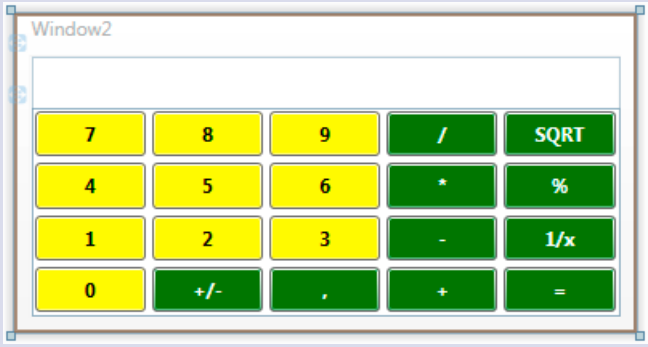
```
    <Style x:Key="myBtnStyle">
...
    </Style>
```

Und hier der Default-Style für alle Elemente vom Typ *Button*:

```
    <Style TargetType="{x:Type Button}">
      <Setter Property="Margin" Value="2" />
      <Setter Property="Background" Value="Green" />
     <Setter Property="FontWeight" Value="UltraBold" />
      <Setter Property="Foreground" Value="White" />
    </Style>
```

Ergebnis

Ohne weitere Änderungen im XAML-Code dürfte sich jetzt bereits folgender Anblick bieten:



HINWEIS: Hätten Sie die Style-Definition in der Datei *App.xaml* eingefügt, würde es sich um einen anwendungsweiten Style handeln, alle Windows bzw. die enthaltenen Schaltflächen würden dieses Outfit bekommen.

Doch was, wenn sich auf Ihrem Formular zum Beispiel ein paar *ToggleButton*-Controls befinden? Diese sind von obiger Styledefinition nicht betroffen, handelt es sich doch um eine andere Klasse.

3.5.4 Styles anpassen und vererben

Es gibt immer wieder Ausnahmen von der Regel, und so kommen Sie meist nicht darum herum, den Style einzelner Controls speziell anzupassen. Drei Varianten bieten sich dazu an:

1. Sie überschreiben einzelne Attribute direkt im Element.
2. Sie ersetzen den Style auf einer niedrigeren Ebene (statt *Application* z. B. in einem *StackPanel*).
3. Sie vererben den Style und passen ihn unter neuem Namen an.

Styles anpassen (überschreiben)

Das Überschreiben von Styles ist eigentlich ganz intuitiv möglich, geben Sie den Key des Styles an oder nutzen Sie einen Typ-Style wie bisher. Gleichzeitig erweitern Sie die Attribut-Liste des betreffenden Elements, um die gewünschten Änderungen vorzunehmen.

Beispiel 3.24: (Fortsetzung) Die Taste „0“ soll rot hinterlegt werden (der Style gibt gelb vor).

XAML

```
<Button  Name="button16" Style="{StaticResource myBtnStyle}" Background="Red"
>0</Button>
```

Style ersetzen

Fällt ein komplettes Formular aus dem Rahmen, oder möchten Sie einzelne Elemente einer Gruppe (*StackPanel*, *Grid* etc.) mit einem angepassten Style versehen, können Sie auch den zentral gültigen Style ersetzen. Definieren Sie dazu einen neuen Ressource-Abschnitt und fügen Sie die neue Style-Definition in diesen ein.

Beispiel 3.25: (Fortsetzung) Ersetzen des zentralen Button-Styles

XAML

```
<Window x:Class="Konzepte.Taschenrechner_1"
...
```

Hier die übergreifende Definition:

```
 <Window.Resources>
   <Style TargetType="{x:Type Button}">
...
   </Style>
 </Window.Resources>
...
   <UniformGrid Name="uniformGrid1" Columns="5" Rows="4" Grid.Row="1">
```

Hier wir der Style **ersetzt**, d. h., alle obigen Einstellungen gehen verloren:

```
      <UniformGrid.Resources>
        <Style TargetType="{x:Type Button}">
          <Setter Property="Margin" Value="2" />
          <Setter Property="Background" Value="Blue" />
          <Setter Property="FontWeight" Value="UltraBold" />
          <Setter Property="Foreground" Value="White" />
        </Style>
      </UniformGrid.Resources>
      ...
```

HINWEIS: Elemente, die sich in der Hierarchie oberhalb des *UniformGrids* befinden sind nicht von dieser Anpassung betroffen, hier gilt wieder die zentrale Version.

Styles vererben

Wer schreibfaul ist und beispielsweise nicht den kompletten Style austauschen will, kann diesen auch einfach vererben. Dazu wird in die Definition des Styles das Attribut *BasedOn* aufgenommen, das per Binding auf den Basisstyle verweist.

Ob Sie in diesem neuen Style bestehende Eigenschaften überschreiben (einfach erneut definieren) oder neue Eigenschaften hinzufügen, bleibt Ihnen überlassen.

Beispiel 3.26: Vererben eines Styles

XAML

```
    <Window.Resources>
```

Das Original:

```
    <Style x:Key="myBtnStyle">
      <Setter Property="Control.Margin" Value="2" />
      <Setter Property="Control.Background" Value="Yellow" />
      <Setter Property="Control.FontWeight" Value="UltraBold" />
    </Style>
```

Der „Erbe“ mit geringfügiger Änderung:

```
    <Style x:Key="myBtnStyle2" BasedOn="{StaticResource myBtnStyle}">
```

Hier wird eine Eigenschaft überschrieben:

```
      <Setter Property="Control.Margin" Value="5" />
```

Hier wird eine neue Eigenschaft gesetzt:

```
      <Setter Property="Control.FontStyle" Value="Italic" />
    </Style>
  </Window.Resources>
```

Ergebnis

Links der neue Style, rechts der Basis-Style:

Styleänderung per Code

Ihr Programm ist nicht darauf angewiesen, immer den gleichen Style zu verwenden. So ist es problemlos möglich, auch zur Laufzeit einen Style per Code neu zu setzen (z. B. unter bestimmten Bedingungen).

Beispiel 3.27: Der Style von *button10* wird geändert

C#

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{ button10.Style = (Style)FindResource("myBtnStyle2");  }
```

HINWEIS: Den Style bzw. dessen Instanz finden Sie mit der per *FindResource* über dessen Key.

HINWEIS: Bevor Sie sich weiter in diese Thematik vertiefen, sollten Sie zunächst einen Blick auf den folgenden Abschnitt werfen, wahrscheinlich löst das Ihre Aufgabenstellung wesentlich eleganter.

3.6 Verwenden von Triggern

In unseren bisherigen Experimenten waren die vom Style vorgenommenen Änderungen immer statischer Natur, d. h., einmal gesetzt, blieb die Optik immer gleich. Doch dies kann sicher nicht der Weisheit letzter Schluss sein.

Wenn jetzt in Ihnen der Programmierer wieder durchkommt und Sie an C# und Ereignisprozeduren denken, vergessen Sie es gleich wieder. Für (fast) alle Aufgabenstellungen ist auch hier XAML die beste Lösung.

Für die Reaktion auf Eigenschaftsänderungen, Ereignisse, Datenänderungen etc. können Sie in WPF-Anwendungen so genannte Trigger verwenden und damit zum Beispiel den Style ändern. Dabei sind Sie nicht auf einen einzelnen Trigger angewiesen, sondern Sie können der Trigger-Collection auch mehrere Ereignisse mit unterschiedlichen Bedingungen zuweisen.

Im Folgenden wollen wir uns die verschiedenen Triggerarten näher ansehen.

3.6.1 Eigenschaften-Trigger (Property Triggers)

Sicher kennen Sie auch das eine oder andere Programm, das exzessiv Gebrauch von diversen optischen Spielereien macht. Wird beispielsweise mit dem Mauskursor auf ein Control gezeigt, ändert sich dessen Rahmen oder die Hintergrundfarbe. Gleiches gilt für Eingabefelder die den Focus erhalten etc. In all diesen Fällen ändern sich Eigenschaften (*IsMouseOver*, *IsFocused*), auf die Sie bei der konventionellen Programmierung mit Ereignismethoden reagieren können. Mit Eigenschaften-Triggern können Sie Ihren C#-Quellcode von derartigem Ballast befreien und direkt per XAML-Code Änderungen am Control vornehmen.

Beispiel 3.28: Eine *TextBox* soll auf Änderungen von *IsMouseOver* und *IsFocused* mit Farbänderungen reagieren.

XAML

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

Einen Style für die *TextBox* erzeugen:

```
  <Window.Resources>
    <Style x:Key="myStyle" TargetType="{x:Type TextBox}">
```

Der Außenabstand soll immer 2 betragen:

```
      <Setter Property="Margin" Value="2" />
```

Hier werden die Trigger definiert:

```
      <Style.Triggers>
```

Unter der Bedingung ...

```
        <Trigger Property="IsMouseOver" Value="True">
```

... wird die folgende Eigenschaft gesetzt:

```
          <Setter Property="Background" Value="Yellow" />
        </Trigger>
```

Unter der Bedingung ...

```
        <Trigger Property="IsFocused" Value="True">
```

... werden die folgenden Eigenschaften gesetzt:

```
          <Setter Property="Background" Value="Blue" />
          <Setter Property="Foreground" Value="White" />
        </Trigger>
      </Style.Triggers>
    </Style>
  </Window.Resources>
  <Grid>
    <StackPanel>
```

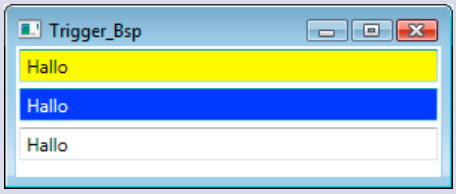
Hier verwenden wir den Style:

```
      <TextBox Style="{StaticResource myStyle}" >Hallo</TextBox>
      <TextBox Style="{StaticResource myStyle}" >Hallo</TextBox>
    </StackPanel>
  </Grid>
 </Window>
```

Änderungen, die durch einen Trigger vorgenommen wurden, werden automatisch wieder rückgängig gemacht, wenn die Bedingung nicht mehr eingehalten wird (automatisches Wiederherstellen des Standardwertes).

Ergebnis

Die folgende Abbildung zeigt die Laufzeitansicht, die erste *TextBox* erfüllt die Bedingung *IsMouseOver=True*, die zweite *TextBox* hat den Eingabefokus und die dritte *TextBox* ist im Standardzustand:



Doch was, wenn Sie mehr als eine Bedingung benötigen? Auch das ist kein Problem, in diesem Fall erzeugen Sie einfach einen „multi-condition property trigger“.

Beispiel 3.29: Der *TextBox*-Hintergrund soll rot werden, wenn die *TextBox* den Eingabefokus besitzt und wenn kein Text enthalten ist.

XAML

```
  <Window.Resources>
    <Style x:Key="myStyle" TargetType="{x:Type TextBox}">
...
```

Einen Multi-Condition-Trigger definieren:

```
        <MultiTrigger>
          <MultiTrigger.Conditions>
```

Die beiden folgenden Bedingungen müssen zutreffen:

```
            <Condition Property="IsFocused" Value="True"/>
            <Condition Property="Text" Value="" />
          </MultiTrigger.Conditions>
```

Hier die Aktion:

```
          <Setter Property="Background" Value="Red" />
        </MultiTrigger>
      </Style.Triggers>
    </Style>
  </Window.Resources>
```

3.6.2 Ereignis-Trigger

Ereignis-Trigger werden durch bestimmte Ereignisse (vom Typ *RoutedEvent*) ausgelöst. Im Gegensatz zu den Eigenschaften-Triggern können Sie über derartige Trigger jedoch direkt keine Eigenschaften ändern, Sie können „lediglich“ Animationen starten, die sich wiederum auf Eigenschaften auswirken.

HINWEIS: Ereignis-Trigger setzen geänderte Eigenschaften nicht wieder zurück, dafür sind **Sie** als Programmierer verantwortlich (z. B. im Pendant des betreffenden Ereignisses).

Beispiel 3.30: Wird der Mauskursor über einen Button bewegt, wird dieser transparent.

XAML

Zunächst die Style-Definition:

```
<Style TargetType="{x:Type Button}">
  <Style.Triggers>
```

Hier kommt unser Ereignis-Trigger:

```
    <EventTrigger RoutedEvent="Button.MouseEnter">
```

Wir reagieren mit einer Aktion über die Hintergründe:

```
      <EventTrigger.Actions>
       <BeginStoryboard>
```

Hier die eigentliche Aktion, die Eigenschaft *Opacity* soll in 4 Sekunden von 1 auf 0.25 verringert werden:

```
          <Storyboard>
            <DoubleAnimation From="1" To="0.25" Duration="0:0:4"
                             Storyboard.TargetProperty="(Opacity)"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
```

Beim *MouseLeave*-Ereignis gehen wir den umgekehrten Weg und stellen die ursprüngliche Transparenz wieder her:

```
    <EventTrigger RoutedEvent="Button.MouseLeave">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation From="0.25" To="1" Duration="0:0:4"
                             Storyboard.TargetProperty="(Opacity)"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Style.Triggers>
</Style>
```

3.6.3 Daten-Trigger

Mit diesen Triggern können Sie auf das Ändern beliebiger Eigenschaften reagieren, die Verbindung zu den entsprechenden Eigenschaften stellen Sie per Bindung her. Als Reaktion auf eine Eigenschaftsänderung können Sie, wie auch bei den Eigenschaften-Triggern, mit einem *Setter*-Element bestimmte Eigenschaften ändern.

HINWEIS: Die durch den Trigger geänderten Eigenschaften werden automatisch zurückgesetzt, wenn die Bedingung nicht mehr übereinstimmt.

Beispiel 3.31: Enthält die *TextBox* mehr als zehn Zeichen, wird sie grün eingefärbt.

XAML

```
<Window.Resources>
  <Style x:Key="myStyle" TargetType="{x:Type TextBox}">
...
      <DataTrigger Binding="{Binding RelativeSource={RelativeSource Self},
                    Path=Text.Length}" Value="10">
          <Setter Property="Background" Value="Green" />
      </DataTrigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

3.7 Einsatz von Templates

Im Folgenden möchten wir Sie zunächst mit ein paar Grundaussagen konfrontieren, bevor wir uns der Thematik „Templates“ bzw. „Vorlagen“ widmen:

- WPF-Controls haben prinzipiell keine Zeichenlogik, es handelt sich um Lookless Controls.
- WPF-Controls stellen lediglich eine Sammlung von Verhalten dar.

Spinnen die Autoren? Wo kommen denn sonst die ganzen optischen Spielereien her? Die Antwort auf dieses Paradoxon: Die Zeichenlogik eines Controls wird nur vom Layout/Styling bestimmt, jedes Control besitzt ein Standardaussehen (Default-Template), das komplett ersetzt werden kann.

Hier haben Sie es mit der Spielwiese der Designer zu tun, aus dem guten alten viereckigen Button kann ein gänzlich anderes Objekt werden, das jedoch nach wie vor das wesentliche Verhalten eines Buttons (Klick) besitzt. Aus Sicht des Programmierers kann dieser neue Button wie der Standard-Button verwendet werden. Damit ersparen Templates uns vielfach die Mühe, Controls umständlich abzuleiten und deren Zeichenlogik per C#-Code komplett neu zu implementieren.

HINWEIS: Im Gegensatz zu den Styles können Sie bei den Templates nicht nur die vorhandenen Eigenschaften beeinflussen, sondern das Control auch von Grund auf neu zusammenbauen.

3.7.1 Neues Template erstellen

Ein etwas komplexeres Beispiel soll die prinzipielle Vorgehensweise verdeutlichen, eine Komplettübersicht dieses Themas können wir Ihnen an dieser Stelle leider nicht geben.

Beispiel 3.32: Erzeugen und Verwenden eines Templates

XAML

Unsere Schaltflächen sollen ellipsenförmig sein und einen Farbverlauf aufweisen. Ist die Maus über dem Control, soll sich die Schriftstärke ändern. Ein Niederdrücken der Schaltfläche führt zu einem umgekehrten Farbverlauf im Control.

```
<Window.Resources>
```

Zunächst erstellen wir einen neuen Type-Style für *Button*-Elemente (Sie können auch einen benannten Style verwenden):

```
    <Style TargetType="{x:Type Button}">
```

Hier greifen wir erstmals auf das Template zu, die Definition ist etwas verschachtelt, da es sich um eine recht komplexe Zuweisung handelt:

```
      <Setter Property="Template">
        <Setter.Value>
```

Der Eigenschaft *Template* wird ein *ControlTemplate* zugewiesen:

```
          <ControlTemplate TargetType="{x:Type Button}">
```

Für die innere Ausrichtung nutzen wir zunächst ein *Grid*:

```
            <Grid HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
 ClipToBounds="False">
```

Und hier haben wir es schon mit der Optik zu tun, wir erzeugen eine Ellipse mit einem Farbverlauf (dies ist der Defaultzustand):

```
            <Ellipse Name="elli">
              <Ellipse.Fill>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1" >
                  <LinearGradientBrush.GradientStops>
                    <GradientStop Color="#fff399" Offset="0.1"/>
                    <GradientStop Color="#ffe100" Offset="0.5"/>
                    <GradientStop Color="#feca00" Offset="0.9"/>
```

```
            </LinearGradientBrush.GradientStops>
          </LinearGradientBrush>
        </Ellipse.Fill>
      </Ellipse>
```

In unserem Button soll auch etwas angezeigt werden, dafür ist das *ContentPresenter*-Element verantwortlich:

```
<ContentPresenter x:Name="PrimaryContent" HorizontalAlignment="Center"
                  VerticalAlignment="Center"
```

Den eigentlichen Inhalt holen wir uns wiederum vom ursprünglichen Element (dem *Button*), deshalb auch die etwas umständliche Bindung:

```
      Content="{Binding Path=Content, RelativeSource={RelativeSource
TemplatedParent}}"
        />
      </Grid>
```

Nicht nur die Standardanzeige unseres Buttons wollen wir beeinflussen, sondern auch die Reaktion auf Maus und Klicken:

```
      <ControlTemplate.Triggers>
```

Es wird geklickt, d. h., wir zeichnen einen neuen Hintergrund. Dazu benötigen wir allerdings den Namen der Ellipse:

```
          <Trigger Property="Button.IsPressed" Value="True">
           <Setter Property="Fill" TargetName="elli" >
```

Achtung: Diese Namen sind nur innerhalb des Templates verwendbar!

```
           <Setter.Value>
             <LinearGradientBrush StartPoint="0,1" EndPoint="0,0" >
               <LinearGradientBrush.GradientStops>
                 <GradientStop Color="#fff399" Offset="0.1"/>
                 <GradientStop Color="#ffe100" Offset="0.5"/>
                 <GradientStop Color="#feca00" Offset="0.9"/>
               </LinearGradientBrush.GradientStops>
             </LinearGradientBrush>
           </Setter.Value>
         </Setter>
       </Trigger>
```

Die Maus wird über das Control bewegt, in diesem Fall wird lediglich die Schriftstärke geändert:

```
      <Trigger Property="Button.IsMouseOver" Value="True">
        <Setter Property="FontWeight" Value="Bold" />
      </Trigger>
```

Hier könnten Sie noch auf weitere Eigenschaftsänderungen reagieren ...

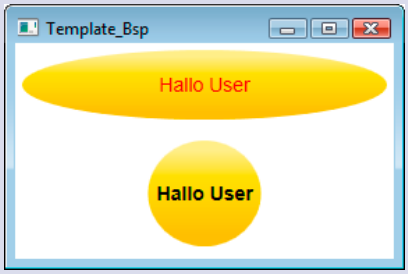
```
            </ControlTemplate.Triggers>
          </ControlTemplate>
        </Setter.Value>
      </Setter>
```

Last but not least, setzen wir noch ein paar Eigenschaften:

```
      <Setter Property="Foreground" Value="Black" />
      <Setter Property="FontFamily" Value="Arial" />
      <Setter Property="FontSize" Value="14" />
    </Style>
  </Window.Resources>
```

Ergebnis

Und wie sieht nun unsere neue Schaltfläche aus? Hier die Antwort:



Doch wie können Sie innerhalb des Templates auf die ursprüngliche Definition von Eigenschaften zugreifen? Hier hilft Ihnen ebenfalls Binding weiter.

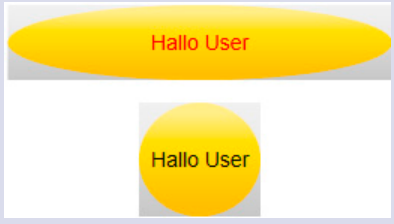
Beispiel 3.33: Verwendung der *Button.Background*-Eigenschaft

XAML

```
          <ControlTemplate TargetType="{x:Type Button}">
            <Grid HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
                  ClipToBounds="False">
              <Rectangle Fill="{TemplateBinding Property=Background}"/>
              <Ellipse Name="elli">
```

Ergebnis

Starten Sie das Programm mit dieser Änderung, taucht im Hintergrund zunächst der Default-Farbverlauf eines Buttons auf (so ist *Background* für einen *Button* auch gesetzt).



Sie können aber über den Style gleich noch eine andere Hintergrundfarbe auswählen:

```
    <Setter Property="FontSize" Value="14" />
    <Setter Property="Background" Value="Blue" />
  </Style>
```

Werfen wir noch einen Blick auf das *ContentPresenter*-Element in unserem obigen Template. Dessen Content stammt vom ursprünglichen Element ab und bietet damit ebenfalls die Möglichkeit, komplexe Controls „zusammenzubasteln".

Beispiel 3.34: Wir definieren einen zusätzlichen Button mit einer Grafik, der Button übernimmt den vorliegenden Style.

XAML

```
<Button Height="76" Margin="10" Width="113">
 <StackPanel Orientation="Horizontal">
    <Image Source="Images/flash.png" Width="26" Height="26" Margin="0,0,10,0"/>
    <TextBlock VerticalAlignment="Center">Action</TextBlock>
  </StackPanel>
</Button>
```

Ergebnis

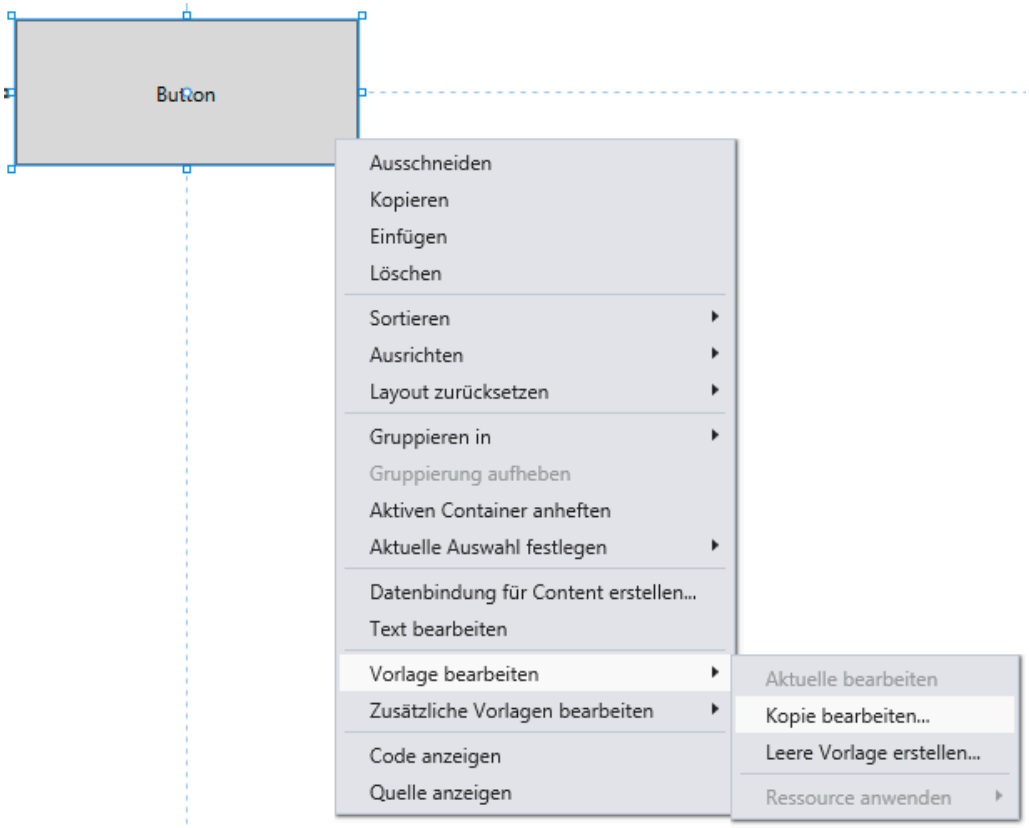
Das zusammengewürfelte Endergebnis (Grafik und Text per *Content,* Grundlayout per *Template)* präsentiert sich nach wie vor als vollwertiger Button:

3.7.2 Template abrufen und verändern

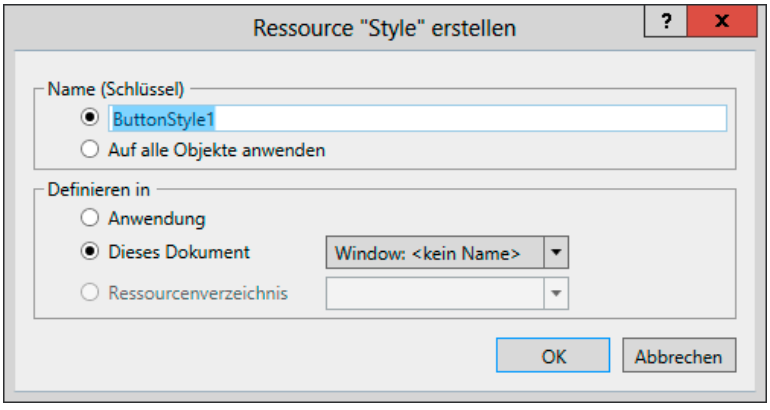
Möchten Sie von einem vorhanden Control das Template abrufen und eventuell verändern, ist dies seit Visual Studio 2012 kein Problem mehr[5].

Sie können von jedem Control direkt aus dem Designer heraus eine Kopie des Default-Templates editieren. Markieren Sie das betreffende Control im WPF-Editor-Fenster und klicken Sie auf die rechte Maustaste. Über das Kontextmenü rufen Sie *Vorlage bearbeiten* | *Kopie* bearbeiten auf:

[5] In früheren Versionen waren Sie auf die Anwendung *Expression Blend* angewiesen, mittlerweile ist der WPF-Editor in Visual Studio massiv aufgerüstet worden.



Im folgenden Dialog bestimmen Sie, ob der Style per Key (benannt) oder per Typ zugeordnet werden soll und wo der Style erstellt wird (siehe folgende Abbildung). Auf das Speichern im aktuellen Window sollten Sie aus Gründen der Übersichtlichkeit verzichten.



Beispiel 3.35: Die derart erstellte Kopie sieht wie folgt aus (Beispiel *Button*):

XAML

```
<Application x:Class="WpfApplication2.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
```

Hier geht es mit der eigentlichen Definition des Buttons los, bei der zunächst einige Einstellungen zugewiesen werden:

```
        <Style TargetType="{x:Type Button}">
            <Setter Property="FocusVisualStyle">
                <Setter.Value>
                    <Style>
                        <Setter Property="Control.Template">
                            <Setter.Value>
                                <ControlTemplate>
                                    <Rectangle Margin="2" SnapsToDevicePixels="True"
                                        Stroke="{DynamicResource {x:Static
                                        SystemColors.ControlTextBrushKey}}"
                                        StrokeThickness="1" StrokeDashArray="1 2"/>
                                </ControlTemplate>
                            </Setter.Value>
                        </Setter>
                    </Style>
                </Setter.Value>
            </Setter>
            <Setter Property="Background" Value="#FFDDDDDD"/>
            <Setter Property="BorderBrush" Value="#FF707070"/>
```

Hier wird direkt eine Systemressource genutzt, die Anpassung erfolgt dynamisch, d. h. bei Änderungen in der Systemsteuerung:

```
            <Setter Property="Foreground"
                    Value="{DynamicResource {x:Static
                            SystemColors.ControlTextBrushKey}}"/>
            <Setter Property="BorderThickness" Value="1"/>
            <Setter Property="HorizontalContentAlignment" Value="Center"/>
            <Setter Property="VerticalContentAlignment" Value="Center"/>
            <Setter Property="Padding" Value="1"/>
            <Setter Property="Template">
                <Setter.Value>
```

Das Default-Template als Kopie:

```
                    <ControlTemplate TargetType="{x:Type Button}">
                        <Border x:Name="border" BorderBrush="{TemplateBinding
BorderBrush}"
                                BorderThickness="{TemplateBinding BorderThickness}"
                                Background="{TemplateBinding Background}"
                                SnapsToDevicePixels="True">
                                <ContentPresenter x:Name="contentPresenter"
                                  ContentTemplate="{TemplateBinding ContentTemplate}"
```

```
                            Content="{TemplateBinding Content}"
                            ContentStringFormat="{TemplateBinding
ContentStringFormat}"
                            Focusable="False" HorizontalAlignment="{TemplateBinding
                            HorizontalContentAlignment}" Margin="{TemplateBinding
Padding}"
                            RecognizesAccessKey="True"
                            SnapsToDevicePixels="{TemplateBinding
SnapsToDevicePixels}"
                            VerticalAlignment="{TemplateBinding
                        VerticalContentAlignment}"/>
                    </Border>
```

Die Reaktion auf Eigenschaften-Trigger:

```
                    <ControlTemplate.Triggers>
                        <Trigger Property="IsDefaulted" Value="True">
                            <Setter Property="BorderBrush" TargetName="border"
                    Value="{DynamicResource {x:Static
SystemColors.HighlightBrushKey}}"/>
                        </Trigger>
```

Reaktion auf Mausbewegung:

```
                        <Trigger Property="IsMouseOver" Value="True">
                            <Setter Property="Background" TargetName="border"
                                    Value="#FFBEE6FD"/>
                            <Setter Property="BorderBrush" TargetName="border"
                                    Value="#FF3C7FB1"/>
                        </Trigger>
```

Reaktion auf Klicken:

```
                        <Trigger Property="IsPressed" Value="True">
                            <Setter Property="Background" TargetName="border"
                                    Value="#FFC4E5F6"/>
                            <Setter Property="BorderBrush" TargetName="border"
                                    Value="#FF2C628B"/>
                        </Trigger>
                        <Trigger Property="IsEnabled" Value="False">
                            <Setter Property="Background" TargetName="border"
                                    Value="#FFF4F4F4"/>
                            <Setter Property="BorderBrush" TargetName="border"
                                    Value="#FFADB2B5"/>
                            <Setter Property="TextElement.Foreground"
                                    TargetName="contentPresenter"
Value="#FF838383"/>
                        </Trigger>
                    </ControlTemplate.Triggers>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>

</Application.Resources>
</Application>
```

HINWEIS: Ändern Sie in diesem Template beispielsweise die Farben im *IsMouseOver*-Trigger (siehe fett hervorgehobene Zeilen), wird sich das Aussehen alle Schaltflächen in Ihrer Anwendung ändern.

Hauptzielgruppe dieses Features dürfte jedoch nicht der Programmierer sondern der Designer der Anwendung sein. Gleiches trifft ebenfalls auf das im folgenden Abschnitt behandelte Storyboard zu.

3.8 Transformationen, Animationen, StoryBoards

Im Folgenden wollen wir in einem „Schnelldurchlauf für Programmierer" noch einen Blick auf WPF-typische Features werfen, auch wenn diese im Allgemeinen nicht zum Hautarbeitsgebiet des Programmierers gehören.

3.8.1 Transformationen

Mit Hilfe von Transformationen können Sie in WPF das optische Standardverhalten problemlos verändern, ohne sich um Templates oder Styles kümmern zu müssen. Sie können die Größe, Position, Drehung und Verzerrung der betroffenen Controls über die einfache Zuweisung einer entsprechenden Transformation ändern (statische Änderung).

HINWEIS: Damit sind diese Operationen auch die Vorstufe für einfache Animationen (dynamische Änderungen in Abhängigkeit von der Zeit).

Folgende Möglichkeiten stehen Ihnen zur Verfügung:

Transformation	Beschreibung
RotateTransform	Element um einen bestimmten Winkel drehen
ScaleTransform	Element vergrößern/verkleinern
SkewTransform	Element verformen
TranslateTransform	Element verschieben
MatrixTransform	Zusammenfassen der obigen Transformationen per 3x3 Transformationsmatrix

Lassen Sie uns nun an einfachen Beispielen die Wirkung der jeweiligen Transformation demonstrieren.

Drehen mit RotateTransform

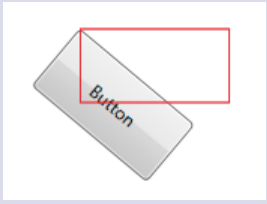
Mit *RotateTransform* realisieren Sie eine Drehung des Elements im Uhrzeigersinn. Den Drehpunkt können Sie optional festlegen, per Default ist dies die linke obere Ecke.

Beispiel 3.36: Button um 40° drehen

XAML

```
<Canvas>
    <Button Canvas.Left="100" Canvas.Top="100" Content="Button" Height="50"
            Name="button1" Width="100" >
        <Button.RenderTransform>
            <RotateTransform Angle="40"/>
        </Button.RenderTransform>
    </Button>
    <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50" Name="rectangle1"
               Stroke="Red" Width="100" />
</Canvas>
```

Ergebnis



Das *Angle*-Attribut bestimmt den Drehwinkel, um den per *CenterX*- und *CenterY*-Attribut festgelegten Drehpunkt. Natürlich können Sie für eine andere Drehrichtung auch negative Werte übergeben.

HINWEIS: Durch die Rotation wird das Koordinatensystem des gedrehten Elements verändert!

Skalieren mit ScaleTransform

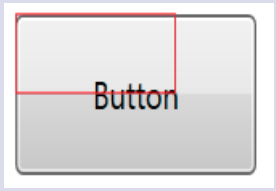
Soll ein Element skaliert werden, nutzen Sie eine *ScaleTransform*.

Beispiel 3.37: Button skalieren

XAML

```
<Canvas>
    <Button Canvas.Left="100" Canvas.Top="100" Content="Button" Height="50"
            Name="button1" Width="100" >
        <Button.RenderTransform>
            <ScaleTransform ScaleX="1.5" ScaleY="2"/>
        </Button.RenderTransform>
    </Button>
    <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50" Name="rectangle1"
               Stroke="Red" Width="100" />
</Canvas>
```

Ergebnis



Mit den *ScaleX*- und *ScaleY*-Attributen bestimmen Sie den Skalierungsfaktor für die X- bzw. Y-Achse. Mit *CenterX*- und *CenterY* bestimmen Sie den Fixpunkt von dem aus die Skalierung gestartet wird, dies ist per Default die linke obere Ecke.

Verformen mit SkewTransform

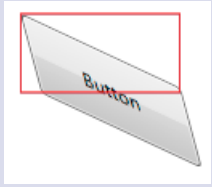
Mit *SkewTransform* verformen Sie das Koordinatensystem um bestimmte Winkel.

Beispiel 3.38: Verformen des Koordinatensystems

XAML

```
<Canvas>
    <Button Canvas.Left="100" Canvas.Top="100" Content="Button" Height="50"
            Name="button1" Width="100" >
        <Button.RenderTransform>
            <SkewTransform AngleY="25" AngleX="15" />
        </Button.RenderTransform>
    </Button>
    <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50" Name="rectangle1"
               Stroke="Red" Width="100" />
</Canvas>
```

Ergebnis



Den Verformungsgrad bestimmen Sie mit den Attributen *AngleX*- und *AngleY*. *CenterX* und *CenterY* legen den Ursprungspunkt fest.

Verschieben mit TranslateTransform

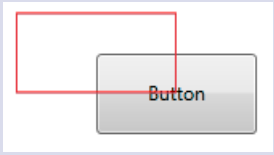
Auch die Verschiebung eines Elements ist mit *TranslateTransform* kein Problem. Sie können mit den X- und Y-Attributen die horizontale bzw. vertikale Verschiebung bestimmen.

Beispiel 3.39: Button verschieben

XAML

```
<Canvas>
    <Button Canvas.Left="100" Canvas.Top="100" Content="Button" Height="50"
            Name="button1" Width="100" >
        <Button.RenderTransform>
            <TranslateTransform X="50" Y="25"/>
        </Button.RenderTransform>
    </Button>
    <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50" Name="rectangle1"
               Stroke="Red" Width="100" />
</Canvas>
```

Ergebnis



Und alles zusammen mit TransformGroup

Sicher sind Sie auch schon versucht gewesen, mehrere der obigen Effekte gleichzeitig zu realisieren. Allerdings dürfte Ihnen die Syntax-Prüfung des XAML-Editors hier einen Strich durch die Rechnung gemacht haben.

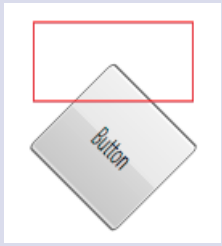
HINWEIS: Um mehrere Transformationen gleichzeitig zuzuweisen, müssen Sie eine *TransformGroup* verwenden.

Beispiel 3.40: Mehrere Transformationen gleichzeitig anwenden

XAML

```
<Canvas>
    <Button Canvas.Left="100" Canvas.Top="100" Content="Button" Height="50"
            Name="button1" Width="100" >
        <Button.RenderTransform>
            <TransformGroup>
                <ScaleTransform ScaleX=".75" ScaleY="1.5"/>
                <RotateTransform Angle="45"></RotateTransform>
                <TranslateTransform X="50" Y="25"/>
            </TransformGroup>
        </Button.RenderTransform>
    </Button>
    <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50" Name="rectangle1"
               Stroke="Red" Width="100" />
</Canvas>
```

Ergebnis



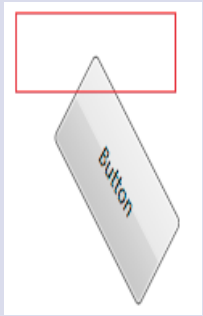
Doch Achtung: Hier spielt die Reihenfolge in der Gruppe eine bedeutende Rolle, wie folgende kleine Änderung (erst Drehung, dann Skalierung) zeigt. Ursache ist die Veränderung des Koordinatensystems des betroffenen Controls.

Beispiel 3.41: Veränderung der Transformationsreihenfolge

XAML

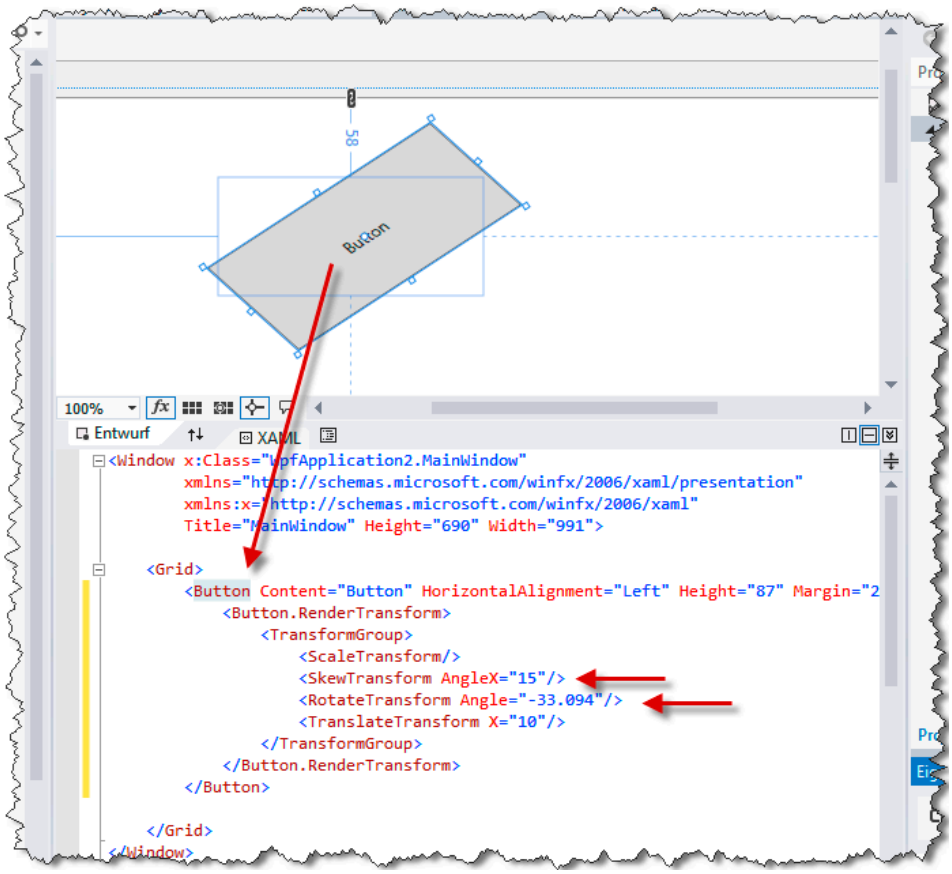
```
<TransformGroup>
    <RotateTransform Angle="45"></RotateTransform>
    <ScaleTransform ScaleX=".75" ScaleY="1.5"/>
    <TranslateTransform X="50" Y="25"/>
</TransformGroup>
```

Ergebnis

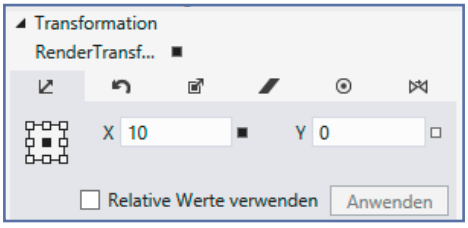


Hilfe durch den WPF-Editor

Wem die obigen Ausführungen zu kompliziert und wenig intuitiv waren, der kann es auch einfacher haben. Nutzen Sie einfach den in Visual Studio integrierten WPF-Editor, dieser ist mittlerweile sogar ganz brauchbar und bietet unter anderem auch die Möglichkeit, die RotateTransformation per Maus umzusetzen.



Im Eigenschaftenfenster können Sie zusätzlich die anderen Transformationsarten getrennt parametrieren:



3.8.2 Animationen mit dem StoryBoard realisieren

Für die Realisierung von Animationen werden in WPF so genannte Storyboards verwendet, die wiederum einzelne oder mehrere Animationen (zeitliche Veränderungen von Eigenschaften) enthalten können. Storyboards können wiederum über bestimmte Ereignis-Trig-

ger ausgelöst, angehalten, fortgesetzt oder auch beendet werden (alternativ natürlich auch per Code).

Ein erstes einfaches Beispiel soll die prinzipielle Vorgehensweise beim Animieren einer Eigenschaft (in diesem Fall der Transparenz) demonstrieren.

Beispiel 3.42: Ausblenden eines Buttons, wenn die Maus darüber bewegt wird

XAML

```
    <Canvas>
```

Zunächst den Button definieren:

```
        <Button Canvas.Left="100" Canvas.Top="100" Content="Button" Height="50"
                Name="button1" Width="100" >
```

Wir reagieren mit einem *Trigger* auf das Hineinbewegen der Maus:

```
            <Button.Triggers>
                <EventTrigger RoutedEvent="Button.MouseEnter">
```

Auf Grund des ausgelösten Trigger-Ereignisses wird das folgende *StoryBoard* ausgeführt:

```
                    <BeginStoryboard>
                        <Storyboard x:Name="Storyboard1">
```

Das *StoryBoard* enthält eine *DoubleAnimation* (Verändern einer *Double*-Eigenschaft) mit einer Zeitdauer (*Duration*) von 4 Sekunden:

```
                            <DoubleAnimation Duration="0:0:4"
```

Ziel der Eigenschaftsänderung ist *button1*:

```
                             Storyboard.TargetName="button1"
```

Die zu ändernde Eigenschaft ist *Opacity*:

```
                             Storyboard.TargetProperty="Opacity"
```

Die Eigenschaft wird in der o.g. Zeitdauer von 1 auf 0 geändert:

```
                             From="1" To="0" />
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger>
            </Button.Triggers>
        </Button>
        <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50" Name="rectangle1"
                   Stroke="Red" Width="100" />
    </Canvas>
```

Das war hoffentlich nicht allzu abschreckend, es geht teilweise auch einfacher und alternativ könnten Sie auch mit sinnvollen Werkzeugen den obigen XAML-Code erstellen.

Animation per C#-Code realisieren

Im obigen Fall müssen Sie immer ein StoryBoard einsetzen, um die Animation(en) zu kapseln. Etwas einfacher geht es, wenn Sie lediglich eine Animation per C#-Code realisieren wollen. In diesem Fall erstellen Sie einfach eine Instanz der gewünschten Animation (davon gibt es je nach Zieleigenschaft unterschiedliche), parametrieren diese und starten die Animation, indem Sie diese an die *BeginAnimation*-Methode des gewünschten Controls übergeben.

Beispiel 3.43: Eine einfache Animation per Code definieren und ausführen

C#

```
...
using System.Windows.Media.Animation;
...
        private void button2_Click(object sender, RoutedEventArgs e)
        {
```

Instanz erstellen:

```
            DoubleAnimation ani = new DoubleAnimation();
```

Parametrieren:

```
            ani.From = 1;
            ani.To = 0;
```

Starten (Transparenz ändern):

```
            button2.BeginAnimation(Button.OpacityProperty, ani);
        }
```

Das war doch gar nicht so schwierig, oder?

Animation per Code steuern

Vielleicht dämmert es Ihnen schon, komplexe Animationen bzw. die Zusammenfassung mehrerer Animationen als Storyboard sind kaum für die tägliche Praxis des C#-Programmierers geeignet. Abgesehen davon, dass Sie Unmengen von C#-Code erzeugen, fehlt bei vielen Animationen einfach die Vorstellungskraft. Dauernde Programmstarts zum Ausprobieren der Effekte zehren auch an den Nerven und kosten Zeit. Ganz nebenbei ist auch die Parametrierung vieler Eigenschaften mit C# eine Pein, hier kann XAML seine Vorteile deutlich ausspielen.

Viel besser ist es, die Storyboards mit einem Programm wie Microsoft Blend zu erstellen und nachträglich in Ihre Anwendung einzufügen. Zum Starten der Animation können Sie entweder, wie bereits gezeigt, einen Trigger verwenden, oder Sie nutzen das *Storyboard* per Code. Dazu stellt die *Storyboard*-Klasse mehrere Methoden bereit, mit denen Sie die Animation gezielt kontrollieren können:

Methode	Beschreibung
Begin	Animationen des *Storyboards* starten.
Pause	Wiedergabe anhalten.
Resume	Wiedergabe fortsetzen.
Seek	Bei der Wiedergabe zu einer Position im *Storyboard* springen. Verwenden Sie eine *TimeSpan-Wert*.
Stop	Wiedergabe anhalten und Wiedergabeposition zurück setzen.

HINWEIS: Auf das Ende der Animationen im *Storyboard* können Sie mit dessen *Completed*-Ereignis reagieren.

Beispiel 3.44: Als Ressource definierte Animation per Code starten

XAML

```
<Window x:Class="Animation_Bsp.MainWindow"
...
        Title="MainWindow" Height="350" Width="525">
```

Als Window-Ressource definieren wir ein *Storyboard*:

```
    <Window.Resources>
```

Achten Sie darauf, einen *Key* zu vergeben:

```
        <Storyboard x:Key="storyboard2">
            <DoubleAnimation Duration="0:0:4" Storyboard.TargetName="button1"
                             Storyboard.TargetProperty="Width"  To="300" />
        </Storyboard>
    </Window.Resources>
        <Canvas>
        <Button Canvas.Left="100" Canvas.Top="100" Content="Button" Height="50"
                Name="button1" Width="100" Click="button1_Click">
        </Button>
...
```

C#

Zunächst den Namespace importieren:

```
...
using System.Windows.Media.Animation;
...
```

Zur Laufzeit können wir unser *Storyboard* suchen und mit der *Begin*-Methode starten:

```
        private void button1_Click(object sender, RoutedEventArgs e)
        {
            Storyboard sb = FindResource("storyboard2") as Storyboard;
            if (sb != null) sb.Begin();
```

Ist für das *Storyboard* kein *TargetName* vorgegeben, können Sie das *Storyboard* auch auf jedes andere Control anwenden, wenn Sie dieses an die *Begin*-Methode übergeben:

```
            if (sb != null) sb.Begin(button2);
        }
```

HINWEIS: Ein Klick auf eine andere Taste könnte beispielsweise mit *Storyboard.Stop* die Animation anhalten.

Selbstverständlich können Sie ein in den Ressourcen abgelegtes Storyboard auch per XAML einbinden bzw. starten. In diesem Fall benötigen Sie nicht eine Zeile C#-Code, können aber die Animationen (bzw. die übergeordneten Storyboards) zentral verwalten.

Beispiel 3.45: Alternative zum vorhergehenden Beispiel

XAML

```
<Window x:Class="Animation_Bsp.MainWindow"
...
    <Window.Resources>
        <Storyboard x:Key="storyboard2">
            <DoubleAnimation Duration="0:0:4" Storyboard.TargetName="button1"
                             Storyboard.TargetProperty="Width" To="300" />
        </Storyboard>
    </Window.Resources>
...
        <Button Canvas.Left="50" Canvas.Top="206" Content="Button" Height="23"
                Name="button4" Width="75" >
```

Per Trigger starten wir ein *Storyboard*:

```
            <Button.Triggers>
                <EventTrigger RoutedEvent="Button.MouseEnter">
```

Hier weisen wir die Ressource zu:

```
                    <BeginStoryboard Storyboard="{StaticResource storyboard2}" />
                </EventTrigger>
            </Button.Triggers>
        </Button>
    </Canvas>
</Window>
```

Haben Sie ein *Storyboard* ohne *TargetName* (universelle Verwendung), müssen Sie diesen beim Einbinden der Ressource angeben, um auch das Ziel der Animation zu bestimmen:

Beispiel 3.46: Ziel bestimmen

XAML

```
...
    <EventTrigger RoutedEvent="Button.MouseEnter">
        <BeginStoryboard Storyboard="{StaticResource storyboard4}"
                         Storyboard.TargetName="button4" />
    </EventTrigger>
...
```

Mehrere Animationen zusammenfassen

Dass eine Animation nur auf der linearen Änderung einer Eigenschaft basiert, dürfte wohl selten der Fall sein. Meist werden mehrere Eigenschaften gleichzeitig geändert. Auch das ist mit dem *Storyboard* kein Problem, wie es das folgende Beispiel zeigt:

Beispiel 3.47: *Storyboard* mit drei Animationen

XAML

```
<Window x:Class="Animation_Bsp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <Storyboard x:Key="storyboard3">
            <DoubleAnimation Duration="0:0:4" Storyboard.TargetProperty="Width"
                   To="300" />
            <DoubleAnimation Duration="0:0:4"
                     Storyboard.TargetProperty="RenderTransform.Angle" To="360" />
            <ColorAnimation  Duration="0:0:3"
                     Storyboard.TargetProperty="Foreground.Color" From="Red"
                               To="Blue" />
        </Storyboard>
    </Window.Resources>
```

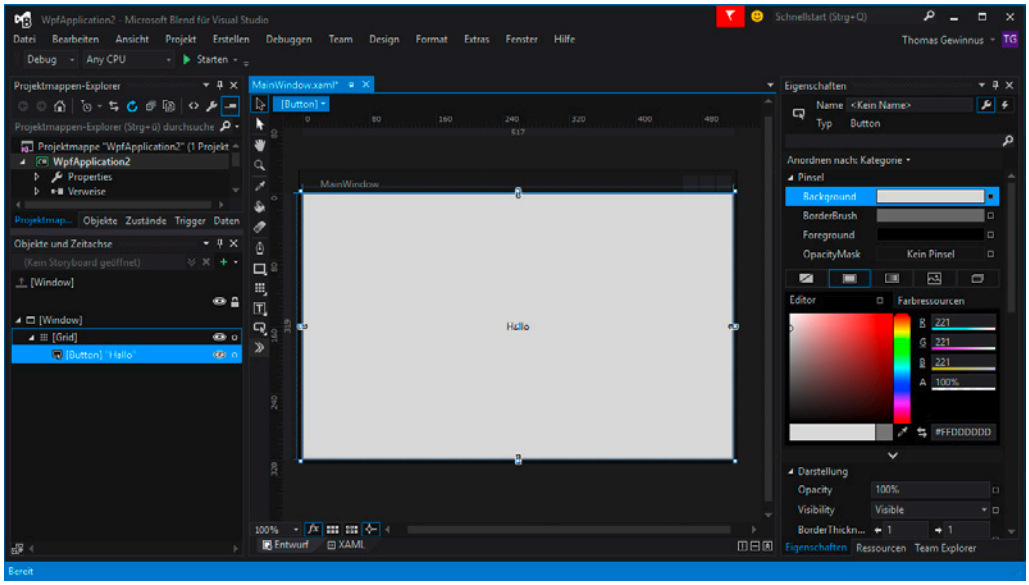
Das soll zu diesem Thema genügen. Auch wenn WPF im Bereich „Animationen“ fast unbegrenzte Möglichkeiten bietet, so sprechen diese doch kaum den Programmierer sondern eher den Designer der Anwendung an. Machen Sie sich also nicht die Mühe, komplexe Storyboards per XAML oder gar C#-Quellcode zu erstellen, sondern nutzen Sie hier die Vorteile von Microsoft Blend, erstellen Sie damit interaktiv den entsprechende XAML-Code und fügen Sie diesen in die Ressourcen Ihrer Anwendung ein. Damit ersparen Sie sich viele graue Haare und haben mehr Zeit für die eigentliche Anwendungsentwicklung.

3.9 Praxisbeispiel

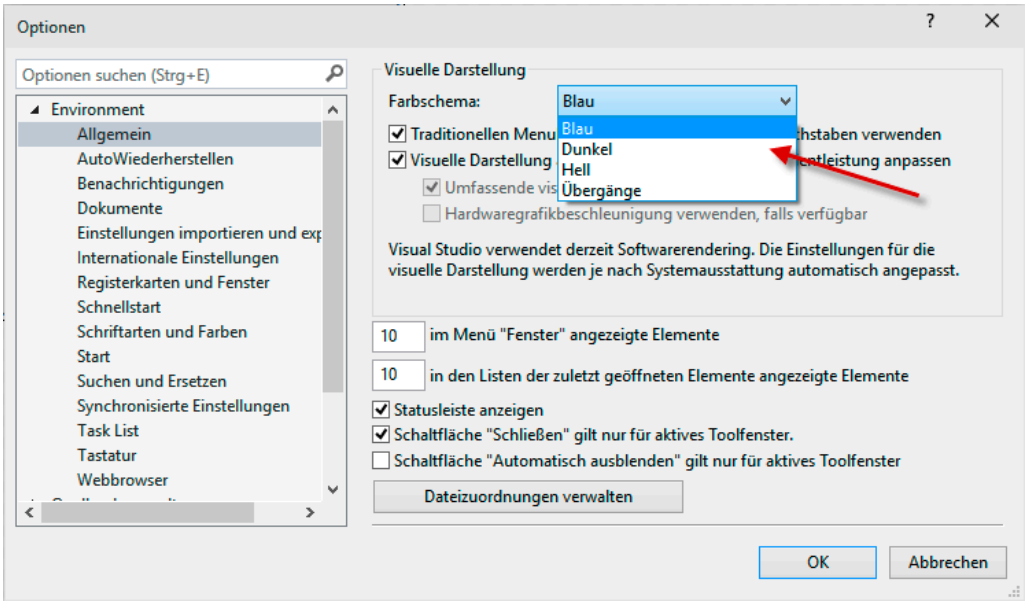
3.9.1 Arbeiten mit Microsoft Blend für Visual Studio

Seit der Einführung von WPF bietet Microsoft neben Visual Studio auch eine zweite Entwicklungsumgebung an, die sich jedoch gezielt an den Designer wendet. Die Rede ist von *Microsoft Blend für Visual Studio*.

Für den Visual Studio-Entwickler dürfte die Oberfläche von Microsoft Blend auf den ersten Blick etwas gewöhnungsbedürftig sein, was sicher auch an der „eigenartigen" Farbgebung liegt:



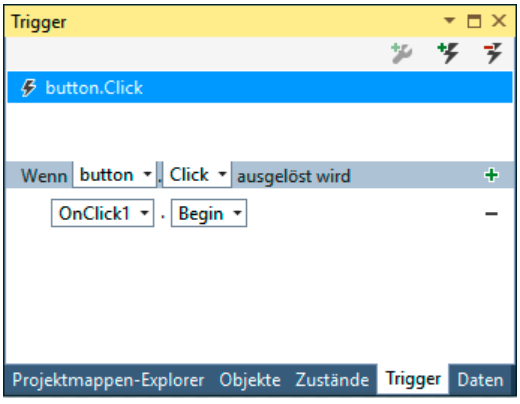
Wer es gern etwas freundlicher mag, kann das Farbschema wechseln (*Extras*|*Optionen*):



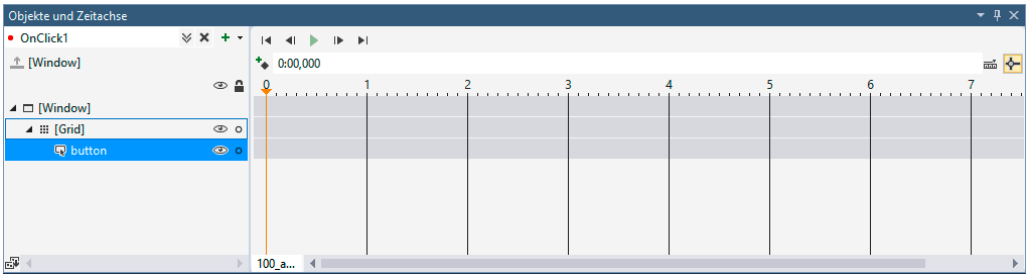
Der wesentlichste Unterschied zu Visual Studio zeigt sich zunächst im Grundansatz, dass die Entwicklung von Aktionen (Trigger) quasi per Assistent und nicht per Quellcode abläuft. C#-Quelltext wird stiefmütterlich behandelt, der Designer soll mit den vorhandenen Triggern etc. arbeiten.

Eine Animation realisieren

Mit dem Klick auf eine Schaltfläche soll eine Animation gestartet werden. Dazu wählen Sie zunächst im „Triggers“-Bereich die Taste „+Event“ und bestimmen nachfolgend über die beiden Auswahllisten welches Objekt (Button) und welche Aktion (Click) zugeordnet werden:



Nachfolgend steht die Frage, ob Sie eine neue Timeline (Storyboard Ressource) erstellen wollen. Diese Timeline dürfte Hobbyfilmern sicher bekannt vorkommen, handelt es sich doch um eine Zeitachse, bei der Sie Objekt-Eigenschaften zu bestimmten Zeitpunkten definiert setzen können:



Die Verwendung ist recht simpel, wählen Sie beispielsweise in der Timeline den Zeitpunkt 5 (Sekunden) und verschieben Sie jetzt ein Control und ändern Sie dessen Größe[6].

Der XAML-Code

Nach dem Umschalten in die XAML-Ansicht können Sie bereits den erzeugten Code (Storyboard) einsehen:

```
...
 <Window.Resources>
   <Storyboard x:Key="Storyboard1">
```

Eine Transformation (Verschiebung X-Achse):

```
     <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
Storyboard.TargetName="button"
      Storyboard.TargetProperty="(UIElement.RenderTransform).(TransformGroup.
Children)[3].
      (TranslateTransform.X)">
       <SplineDoubleKeyFrame KeyTime="00:00:05" Value="96.5"/>
     </DoubleAnimationUsingKeyFrames>
```

Eine Transformation (Verschiebung Y-Achse):

```
     <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
Storyboard.TargetName="button"
       Storyboard.TargetProperty="(UIElement.RenderTransform).(TransformGroup.
Children)[3].
        (TranslateTransform.Y)">
       <SplineDoubleKeyFrame KeyTime="00:00:05" Value="97.5"/>
     </DoubleAnimationUsingKeyFrames>
...
   </Storyboard>
 </Window.Resources>
```

[6] Im Kopf der Design-Ansicht sollte „Zeitachse-Aufzeichnung ist an“ angezeigt werden.

Und hier werden Storyboard und Ereignis miteinander verknüpft:

```
<Window.Triggers>
  <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="button">
    <BeginStoryboard x:Name="OnClick2_BeginStoryboard"
                     Storyboard="{StaticResource OnClick1}"/>
  </EventTrigger>
</Window.Triggers>
```

C# oder VB-Code für die Ereignisbehandlung sind an dieser Stelle nicht erzeugt worden und auch nicht notwendig.

Test

Starten Sie das Projekt mit F5 und klicken Sie auf die Schaltfläche. Diese sollte sich jetzt mit einer Animation bewegen und gleichzeitig drehen. Sollen diese Aktionen getrennt ablaufen, müssen Sie in der Timeline mehrere Zwischenpunkte setzen und die jeweiligen Aktionen ausführen.

Doch was nützen Ihnen als C#-Programmierer all diese Möglichkeiten?

Zwei Szenarien sind denkbar:

- Sie erstellen die WPF-Anwendung komplett selbst und nutzen die erweiterten Möglichkeiten von Blend, um Storyboards und komplexe Oberflächendefinitionen zu erstellen. Die erzeugten XAML-Daten können Sie in Ihr Visual Studio-Projekt (z. B. per Zwischenablage) importieren, oder Sie öffnen das betreffende Projekt gleich mit Visual Studio.
- Sie arbeiten mit einem Designer zusammen, der das Projekt in Blend optisch aufbereitet und Sie können den erforderlichen Quellcode beitragen. Ein Doppelklick auf die dem Window zugeordnete C#-Datei bewirkt, dass diese gleich in Visual Studio geöffnet wird, alternativ öffnen Sie das Projekt komplett mit Visual Studio.

4 WPF-Datenbindung

Nachdem wir schon an der einen oder anderen Stelle auf Datenbindung zurückgegriffen haben, wollen wir uns jetzt direkt mit dieser Thematik beschäftigen.

Im Unterschied zu den Windows Forms-Anwendungen sind Sie bei der Datenbindung nicht auf spezielle Controls angewiesen, in einer WPF-Anwendung kann fast jede Eigenschaft (Abhängigkeitseigenschaft) an andere Eigenschaften gebunden werden.

Als Datenquelle können Sie beispielsweise:

- Eigenschaften anderer WPF-Controls (Elemente),
- Ressourcen,
- XML-Elemente oder
- beliebige Objekte (auch ADO.NET-Objekte, z. B. *DataTable)*

verwenden.

4.1 Grundprinzip

Zunächst wollen wir Ihnen das Grundprinzip der Datenbindung in WPF an einem recht einfachen Beispiel demonstrieren.

Beispiel 4.1: Datenbindung zwischen *Slider* und *ProgressBar*

XAML

Fügen Sie in ein *Window* einen *ProgressBar* und einen *Slider* ein. Mit dem *Slider* soll der aktuelle Wert des *ProgressBar* direkt und ohne zusätzlichen Quellcode verändert werden.

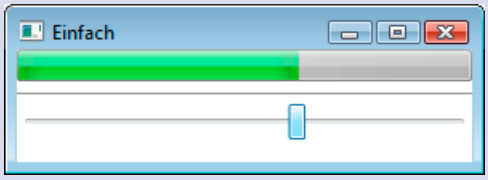
```
<StackPanel>
```

Hier sehen Sie auch schon den Ablauf: Das Ziel (*ProgressBar*) bindet seine Eigenschaft *Value* an die Quelle (*Slider*) mit deren Eigenschaft *Value*.

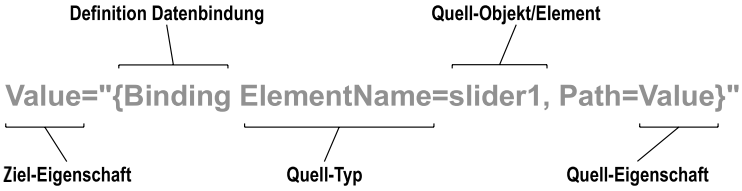
```
    <ProgressBar Height="20" Name="progressBar1" Maximum="100"
                 Value="{Binding ElementName=slider1, Path=Value}"/>
    Separator Height="10"/>
    <Slider Name="slider1" Maximum="100" />
  </StackPanel>
```

Ergebnis

Zur Laufzeit können Sie den *Slider* beliebig verändern, der *ProgressBar* passt sofort seinen Wert an:



Sehen wir uns noch einmal die Syntax im Detail an:



HINWEIS: Kann die Quelleigenschaft nicht automatisch in den Datentyp der Zieleigenschaft konvertiert werden, können Sie zusätzlich einen Typkonverter angeben (siehe dazu Abschnitt 4.7.1).

4.1.1 Bindungsarten

Das vorhergehende Beispiel zeigte bereits recht eindrucksvoll, wie einfach sich Eigenschaften verschiedener Objekte miteinander verknüpfen lassen. Doch das ist noch nicht alles. Über ein zusätzliches Attribut *Mode* lässt sich auch bestimmen, in welche Richtungen die Bindung aktiv ist, d.h., ob die Werte nur von der Quelle zum Ziel oder auch umgekehrt übertragen werden. Die folgende Tabelle zeigt die möglichen Varianten:

Typ	Beschreibung
OneTime	Mit der Initialisierung wird der Wert einmalig von der Quelle zum Ziel kopiert. Danach wird die Bindung aufgehoben.
OneWay	Der Wert wird nur von der Quelle zum Ziel übertragen (readonly). Ändert sich der Wert des Ziels, wird die Bindung aufgehoben.

Typ	Beschreibung
OneWayToSource	Der Wert wird vom Ziel zur Quelle übertragen (writeonly). Ändert sich der Wert der Quelle, bleibt die Bindung erhalten, eine Wertübertragung findet jedoch nicht statt.
TwoWay	(meist Defaultwert[1]) Werte werden zwischen Quelle und Ziel in beiden Richtungen übertragen.

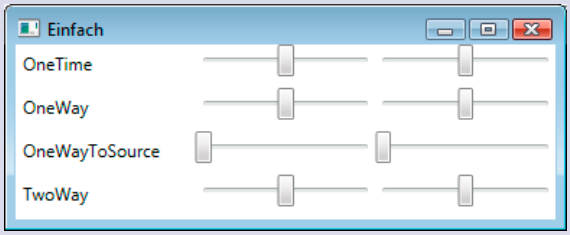
Beispiel 4.2: Testen der verschiedenen Bindungsarten

XAML

```
...
    <StackPanel Grid.Column="2">
      <Slider Name="sl2" Maximum="100" Height="30"
              Value="{Binding ElementName=sl1, Path=Value, Mode=OneTime}"/>
      <Slider Name="sl4" Maximum="100" Height="30"
              Value="{Binding ElementName=sl3, Path=Value, Mode=OneWay}"/>
      <Slider Name="sl6" Maximum="100" Height="30"
              Value="{Binding ElementName=sl5, Path=Value, Mode=OneWayToSource}"/>
      <Slider Name="sl8" Maximum="100" Height="30"
              Value="{Binding ElementName=sl7, Path=Value, Mode=TwoWay}"/>
    </StackPanel>
...
```

Ergebnis

Verschieben Sie ruhig einmal die *Slider* im Testprogramm. Jeweils der linke und der rechte *Slider* bilden eine Datenbindung und sollten auch das entsprechende Verhalten zeigen:



4.1.2 Wann eigentlich wird die Quelle aktualisiert?

Im obigen Beispiel scheint alles ganz einfach zu sein, Sie ziehen an einem Schieberegler und der andere bewegt sich mit. Doch was ist, wenn Sie beispielsweise eine *TextBox* in einer Datenbindung verwenden? Hier steht die Frage, **wann** der „gewünschte" Wert wirklich in der *TextBox* steht.

Eine eingegebene Ziffer ist vielleicht nicht der richtige Wert, kann aber schon als gültiger Inhalt interpretiert werden. Nicht in jedem Fall möchte man deshalb sofort einen Datenaustausch zwischen Ziel und Quelle zulassen (bei *TwoWay* oder *OneWayToSource*).

[1] Bei Bindung an eine *ItemsSource* wird per Default OneWay -Binding verwendet.

Über das optionale Attribut *UpdateSourceTrigger* haben Sie direkten Einfluss darauf, wann die Aktualisierung **der Quelle** durchgeführt wird. Vier Varianten bieten sich dabei an:

- *Default*
  Meist wird das *PropertyChanged*-Ereignis für die Datenübernahme genutzt, bei einigen Controls kann es auch *LostFocus* sein.
- *Explicit*
  Die Datenübernahme muss „manuell" per *UpdateSource*-Methode ausgelöst werden.
- *LostFocus*
  Die Datenübernahme erfolgt bei Fokusverlust des Ziels.
- *PropertyChanged*
  Die Datenübernahme erfolgt mit jeder Werteänderung. Dies kann bei komplexeren Abläufen zu Problemen führen, da der Abgleich, z. B. bei einem Schieberegler/Scrollbar, recht häufig vorgenommen wird.

Beispiel 4.3: Explizite Datenübernahme nur per Enter-Taste

XAML

```
<StackPanel>
  <TextBox Name="txt1">Hallo</TextBox>
  <TextBox Name="txt2"
      Text="{Binding ElementName=txt1, Path=Text, UpdateSourceTrigger=Explicit}"
      KeyDown="TextBox_KeyDown"/>
</StackPanel>
```

C#

```
private void TextBox_KeyDown(object sender, KeyEventArgs e)
 {
     if (e.Key== Key.Enter)
         txt2.GetBindingExpression(TextBox.TextProperty).UpdateSource();
 }
```

HINWEIS: Da die Bindung im XAML-Code vorgenommen wurde, müssen wir im C#-Code erst mit *GetBindingExpression* das *BindingExpression*-Objekt abrufen, um die *UpdateSource*-Methode aufzurufen.

4.1.3 Geht es auch etwas langsamer?

Am obigen Beispiel konnten Sie es ja schon beobachten: Sie verschieben den *Slider* und der zweite *Slider* reagiert sofort. Soweit so schön. Was aber, wenn Sie erst nach einiger Zeit auf die Veränderung reagieren wollen?

Hier hilft die mit WPF 4.5 eingeführte Eigenschaft *Delay* weiter. Diese verzögert die Datenübergabe um den angegebenen Wert (Millisekunden). Das heißt, erst wenn die Zeit nach einer Änderung verstrichen ist, wird der Wert weitergegeben, jede Änderung in dieser Zeitspanne setzt den internen Timer zurück und lässt die Zeit erneut laufen. Bewegen Sie also

den *Silder* dauernd hin und her passiert nichts, erst nach der letzten Bewegung und dem Ablaufen der Zeit wird auch die Änderung berücksichtigt.

Beispiel 4.4: Zeitverzögerung bei Datenbindung

XAML

```
...
    <Slider Name="sl0" Maximum="100" Height="30" Value="{Binding ElementName=sl9,
            Path=Value, Mode=TwoWay, Delay=500 }"/>
...
```

Was dem Einen oder Anderen als Spielerei vorkommen mag, ist ein fast unverzichtbares Feature im Zusammenhang mit größeren Datenmengen oder langsamen Datenverbindungen. Folgende Szenarien sind denkbar:

- 1:n-Beziehung;
  Änderungen in einer *ListBox* sollen sich nicht sofort auf die Detaildaten auswirken, sondern erst nachdem sich der Anwender für einen Datensatz final entschieden hat (Scrollen per Tastatur durch die Liste). Andernfalls kann es schnell zum Ruckeln oder Springen zwischen den Datensätzen kommen.
- Texteingaben;
  nutzen Sie laufende Eingaben als Filter oder Suchwert, kann gerade bei großen Ergebnismengen eine Verzögerung bei der Eingabe auftreten (ein kurzer Filter mit Platzhalter hat meist große Ergebnismengen zur Folge).
- Anzeige großer Datenmengen;
  mit der Auswahl in einer Liste soll eine größere Grafik angezeigt werden. Jede Änderung, z. B. beim Scrollen, führt im Normalfall zum Laden der Grafik. Hier ist eine entsprechende Verzögerung sinnvoll.

Alle obigen Fälle lassen sich natürlich auch mit einem eigenen *Timer* realisieren, aber warum kompliziert, wenn es jetzt auch wesentlich einfacher geht?

Eine Einschränkung sollten Sie allerdings beachten, auch wenn sie meist nicht von Bedeutung ist:

HINWEIS: Die Verzögerung gilt nur für eine Richtung der Datenbindung, d. h. nur für das Control, dem auch die Verzögerung zugeordnet ist.

4.1.4 Bindung zur Laufzeit realisieren

Nicht immer werden Sie mit den schon zur Entwurfszeit definierten Datenbindungen auskommen. Es ist aber auch kein Problem, die Datenbindung erst zur Laufzeit per C#-Code zu realisieren. Alles was Sie dazu benötigen ist ein *Binding*-Objekt, dessen Konstruktor Sie bereits den *BindingPath* zuweisen können. Legen Sie anschließend noch die *BindingSource* sowie gegebenenfalls den *Mode* (z. B. *OneWay*) fest. Letzter Schritt ist das eigentliche Binden mit der *SetBinding*-Methode des jeweiligen Controls.

Beispiel 4.5: Bindung zur Laufzeit realisieren

XAML

Unsere Testoberfläche:

```
<Window x:Class="Datenbindung.Bindung_Laufzeit"
...
        Title="Bindung_Laufzeit" Height="300" Width="300" Loaded="Window_Loaded">
    <StackPanel>
        <Label Name="Label1"></Label>
        <Button Name="Button1" Click="Button_Click">Test</Button>
    </StackPanel>
</Window>
```

C#

Mit dem Laden des Fensters erzeugen wir die Bindung wie oben beschrieben:

```
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
```

Wir binden an einen Button:

```
            Binding binding = new Binding("Content");
            binding.Source = Button1;
            binding.Mode = BindingMode.OneWay;
```

Binden an den Content:

```
            Label1.SetBinding(Label.ContentProperty, binding);
        }
```

Und hier verändern wir die Beschriftung des Buttons:

```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            Button1.Content = "Ein neuer Text";
        }
    }
```

Test

Nach dem Start dürfte im Label zunächst „Test“ stehen, die ursprüngliche Button-Beschriftung. Nach einem Klick auf die Schaltfläche ändern sich sowohl die Button-Beschriftung als auch die Label-Beschriftung.

Die Bindung selbst können Sie recht einfach wieder aufheben, indem Sie der Ziel-Eigenschaft der Bindung einen neuen Wert zuweisen.

Beispiel 4.6: Bindung zur Laufzeit aufheben

C#

Entweder so:

```
Label1.Content = "Bindung beendet";
```

Oder so:

```
Label1.ClearValue(Label.ContentProperty);
```

4.2 Binden an Objekte

Nachdem wir uns bereits mit dem Binden an Oberflächen-Elemente vertraut gemacht haben, wollen wir jetzt den Schritt hin zu selbstdefinierten Objekten gehen.

Prinzipiell bieten sich zwei Varianten der Instanziierung von Objekten an:

- Sie instanziieren die Objekte in XAML (in einem Ressource-Abschnitt).
- Sie instanziieren wie bisher die Objekte im Quellcode.

HINWEIS: Von der Möglichkeit, Objekte im XAML-Code zu instanziieren, halten die Autoren nicht allzu viel. Einerseits wird mit Klassen gearbeitet, die per Code definiert und verarbeitet werden, andererseits wird die Instanz in der Oberfläche, d. h. im XAML-Code, erzeugt. Das ist sicher nicht der Weisheit letzter Schluss. Gerade die üble Vermischung von Code und Oberfläche sollte eigentlich vermieden werden.

Fragwürdig werden Beispielprogramme dann, wenn im C#-Quellcode das zunächst in XAML erzeugte Objekt per *FindResource* gesucht wird (siehe folgender Abschnitt). Da pervertiert doch jede Form der sauberen Programmierung.

Wohlgemerkt wollen wir nicht die komplette Datenbindung im Code realisieren. Das ist sicher zu aufwändig und auch nicht notwendig. Doch aus Sicht des Programmierers sollte nicht die Oberfläche (XAML) sondern der Code im Mittelpunkt des Programms stehen.

4.2.1 Objekte im XAML-Code instanziieren

Erster Schritt, nach der Definition der Klasse, ist das Importieren des entsprechenden Namespaces in die XAML-Datei, andernfalls können Sie auch nicht darauf Bezug nehmen.

Beispiel 4.7: Import des aktuelle Namespace *Datenbindung* in die XAML-Datei

XAML

```
<Window x:Class="Datenbindung.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:Datenbindung"
...
```

Nachdem in XAML die entsprechende Klasse bekannt ist, kann diese auch verwendet werden, um eine eigene Instanz zu erzeugen.

Beispiel 4.8: Erzeugen der Instanz im XAML-Code (wir nutzen eine fiktive Klasse *Schüler*)

XAML

```
...
 <Window.Resources>
   <local:Schüler x:Key="sch1" Nachname="Gurkenkopf" Vorname="Siegfried" />
 </Window.Resources>
```

Die Werte im Einzelnen:

- *LOCAL*: Der Bezug auf den Namespace-Alias
- *SCHÜLER*: Der Klassenname
- *X:Key*: Der Schlüssel unter dem die Instanz verwendet werden kann
- **NACHNAME,** Vorname: Das Setzen einzelner Eigenschaften für die Instanz von *Schüler*

Letzter Schritt: wir nutzen die Möglichkeiten der Datenbindung und binden zwei *TextBox*en an die Eigenschaften *Nachname* und *Vorname*.

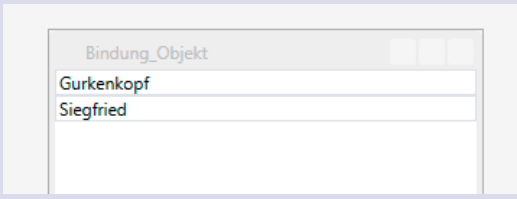
Beispiel 4.9: Bindung an das neue Objekt erzeugen

XAML

```
<StackPanel Name="StackPanel1">
  <TextBox Text="{Binding Source={StaticResource sch1}, Path=Nachname}" />
  <TextBox Text="{Binding Source={StaticResource sch1}, Path=Vorname}" />
</StackPanel>
```

Ergebnis

Schon zur Entwurfszeit dürfte in den beiden *TextBox*en der gewünschte Inhalt auftauchen:



Wem das zu viel Schreibarbeit ist, der kann mit dem *DataContext* auch eine alternative Variante der Zuweisung nutzen. Diese Eigenschaft bietet zunächst eine Alternativ zur Zuweisung von *Source*, hat jedoch zusätzlich die Fähigkeit, von übergeordneten auf unterge-

ordnete Elemente vererbt zu werden. Damit können Sie beispielsweise einem *Panel* oder sogar dem gesamten *Window* einen *DataContext* zuweisen und diesen in allen enthaltenen Elementen nutzen.

Beispiel 4.10: Vereinfachung durch Verwendung eines *DataContext*

XAML

```
  <StackPanel Name="StackPanel1" DataContext="{StaticResource sch1}">
    <TextBox Text="{Binding Path=Nachname}" />
    <TextBox Text="{Binding Path=Vorname}" />
...
```

Sie sparen sich so die Angabe von *Source* bei jedem einzelnen Element.

4.2.2 Verwenden der Instanz im C#-Quellcode

Sicher nicht ganz abwegig ist der Wunsch, zur Laufzeit per C#-Code auch mit dem Objekt zu arbeiten, um z. B. die Werte mit einer *MessageBox* anzuzeigen.

Hier wird die Programmierung dann schon recht windig, müssen Sie doch zunächst die entsprechende Ressource des *Window* suchen und typisieren, möglichen Fehlern ist Tür und Tor geöffnet.

Beispiel 4.11: Anzeige der Werte eines per XAML instanziierten Objekts

C#

```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            Schüler mySch = (Schüler)FindResource("sch1");
            MessageBox.Show(mySch.Nachname + ", " + mySch.Vorname);
        }
```

Aus Sicht eines Programmierers sieht das doch ziemlich merkwürdig aus, auch wenn sich hier der XAML-Profi freut, dass er sogar eine Instanz plus Wertzuweisung per XAML-Code realisiert hat.

Doch was passiert eigentlich mit der Datenbindung, wenn wir der Instanz ein paar neue Werte zuweisen? Ein Test ist schnell realisiert:

```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            Schüler mySch = (Schüler)FindResource("sch1");
            mySch.Nachname = "Strohkopf";
        }
```

Der nachfolgende Blick auf die Oberfläche dürfte in den meisten Fällen für Ernüchterung sorgen, haben Sie Ihre .NET-Klasse (in diesem Fall *Schüler*) nicht entsprechend angepasst, passiert überhaupt nichts und in den *TextBox*en stehen nach wie vor die alten Werte.

4.2.3 Anforderungen an die Quell-Klasse

Was ist hier schief gelaufen? Eigentlich nichts, die neuen Werte stehen wirklich im Objekt, werden aber nicht angezeigt, weil die darstellenden Elemente von einer Wertänderung nichts mitbekommen haben. Wir müssen diese quasi „wecken“, und was eignet sich dafür besser als ein Ereignis?

Auch hier gibt es bereits eine fertige Lösung:

HINWEIS: Implementieren Sie in Ihrer Klasse das Interface *INotifyPropertyChanged* (Namespace *System.ComponentModel*).

Beispiel 4.12: Unsere Klasse *Schüler* mit implementiertem *NotifyPropertyChanged*-Ereignis

C#

```
using System.ComponentModel;

namespace Datenbindung
{
    public class Schüler : INotifyPropertyChanged
    {
        string _Nachname;
        string _Vorname;
        DateTime _Geburtstag;

        public event PropertyChangedEventHandler PropertyChanged;

        public string Vorname
        {
            get { return _Vorname; }
            set
            {
                _Vorname = value;
                NotifyPropertyChanged("Vorname");
            }
        }

        public string Nachname
        {
            get { return _Nachname; }
            set
            {
                _Nachname = value;
                NotifyPropertyChanged("Nachname");
            }
        }
...

        public override string ToString()
        {
            return this._Nachname + ", " + this._Vorname;
        }
```

```
        private void NotifyPropertyChanged(String info)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(info));
        }
    }
}
```

HINWEIS: Alternativ können Sie natürlich auch Abhängigkeitseigenschaften definieren, diese verfügen „ab Werk“ über die erforderliche Benachrichtigung an die gebundenen Elemente, erfordern aber einen höheren Programmieraufwand.

HINWEIS: Damit die Klasse auch im XAML-Code instanziiert werden kann, muss diese über einen parameterlosen Konstruktor verfügen.

Einzige sinnvolle Ausnahme: Sie erzeugen per XAML Objekte und nutzen diese auch nur dort (z. B. Zugriff auf XML-Ressourcen per URL).

4.2.4 Instanziieren von Objekten per C#-Code

Eigentlich könnten wir Ihnen an dieser Stelle noch weitere Möglichkeiten zeigen, wie Sie in XAML Objekte erzeugen bzw. zuweisen können, aber dies ist weder sinnvoll noch besonders übersichtlich. Wir wollen uns stattdessen mit der Vorgehensweise bei vorhandenen, d. h. per Code erzeugten, .NET-Objekten beschäftigen.

Zunächst bleiben wir bei unserem einfachen Beispiel mit der Instanz der Klasse *Schüler*.

Beispiel 4.13: Verwendung von instanziierten Objekten in XAML

C#

Zunächst die Instanziierung:

```
..
    public partial class Window1 : Window
    {
        public Schüler Schueler;

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

Instanz erzeugen und Werte zuweisen:

```
            Schueler = new Schüler { Nachname = "Mayer", Vorname = "Alexander",
                                     Geburtstag = new DateTime(2001, 11, 7) };
```

Hier legen wir per C#-Code den *DataContext* fest:

```
            StackPanel1.DataContext = Schueler;
        }
```

Die spätere Abfrage des Objekts stellt jetzt überhaupt kein Problem dar, die Instanz liegt ja bereits vor:

```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show(Schueler.Nachname + ", " + Schueler.Vorname);
        }
...
```

XAML

Der vollständige XAML-Code:

```
<Window x:Class="Datenbindung.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">
 <StackPanel Name="StackPanel1">
   <TextBox Text="{Binding Path=Nachname}" />
   <TextBox Text="{Binding Path=Vorname}" />
   <Button Click="Button_Click">Prüfen</Button>
 </StackPanel>
</Window>
```

Der Vorteil dieser Vorgehensweise: Sie entscheiden, wie und wann die Instanz erzeugt wird, können vorher noch diverse Methoden aufrufen, profitieren von der Syntaxprüfung und haben einen lesbaren Code.

Der einzige Nachteil: Sie haben keine Wertanzeige zur Entwurfszeit, im XAML-Code ist es nicht sofort erkennbar, welches Objekt zugeordnet wird. Dies ist allerdings auch gleich wieder der Vorteil, mit einem Klick können Sie einen neuen *DataContext* zuweisen und eine andere Instanz bearbeiten.

4.3 Binden von Collections

Die bisherigen Ausführungen dürften zwar schon das Potenzial der Datenbindung demonstriert haben, doch nach der Pflicht kommt jetzt die Kür, d. h. die Arbeit mit einer Reihe von Objekten (Collections). Diese sind vor allem dann interessant, wenn Sie Objekte von Datenbanken abrufen, um diese in Eingabedialogen oder gleich in Listenfeldern darzustellen. Ausgangspunkt können hier Geschäftsobjekte, LINQ-Abfragen etc. sein.

HINWEIS: Im vorliegenden Abschnitt werden wir uns zunächst auf eine „selbstgestrickte“ Collection beziehen (wir verwenden das *Schüler*-Objekt aus dem vorhergehenden Abschnitt). Ab Abschnitt 4.5 geht es dann mit Datenbindung in Verbindung mit LINQ to SQL-Abfragen weiter.

4.3.1 Anforderung an die Collection

Wie auch bei der Klassendefinition für das einzelne Objekt,werden auch an die Collection einige Anforderungen gestellt. Zwar können die WPF-Elemente durch die Verwendung der *INotifyPropertyChanged*-Schnittstelle auf Änderungen einzelner Objekteigenschaften reagieren, das Hinzufügen oder Löschen von ganzen Objekten ist davon aber nicht betroffen. Aus diesem Grund bietet WPF auch hier ein genormtes Interface für die Rückmeldung an: *INotifyCollectionChanged*.

HINWEIS: Grundvoraussetzung für die Anzeige von Listen ist die Verwendung des *IEnumerable*-Interfaces.

Wollen Sie es sich leicht machen, können Sie direkt Objekte der Klasse *ObservableCollection* (Namespace *System.Collections.ObjectModel*) erzeugen.

Beispiel 4.14: (Fortsetzung) Erzeugen und Verwenden einer geeigneten Klasse für die Datenbindung von Collections

C#

```
using System.Collections.ObjectModel;
...
    public partial class Objects_Collections : Window
    {
```

Eine Collection von Schülern:

```
        public ObservableCollection<Schüler> Klasse;
```

Im Konstruktor des *Window* erzeugen wir eine Instanz und füllen diese mit einigen Datensätzen:

```
        public Objects_Collections()
        {
            Klasse = new ObservableCollection<Schüler>();
            Klasse.Add(new Schüler { Nachname = "Mayer", Vorname = "Alexander",
                                     Geburtstag = new DateTime(2001, 11, 7) });
            Klasse.Add(new Schüler { Nachname = "Müller", Vorname = "Thomas",
                                     Geburtstag = new DateTime(2001, 10, 18) });
            Klasse.Add(new Schüler { Nachname = "Lehmann", Vorname = "Walter",
                                     Geburtstag = new DateTime(2001, 1, 21) });
            InitializeComponent();
```

Hier dürften Sie die Verbindung zur bisherigen Vorgehensweise sehen, die Collection wird als *DataContext* für das Fenster und damit für alle untergeordneten Elemente ausgewählt:

```
            this.DataContext = Klasse;
        }
```

4.3.2 Einfache Anzeige

Damit können wir uns zunächst der einfachen Anzeige, z. B. in *TextBox*en, widmen.

Beispiel 4.15: (Fortsetzung) Binden von TextBoxen an die Collection

XAML

```
    <StackPanel Grid.Column="1" Background="Aqua">
      <Label Content="Nachname:" />
```

Hier werden die *TextBox*en an die Eigenschaften der Collection bzw. an das aktive Objekt der Collection gebunden (dies ist durch eine View bestimmt, siehe ab Abschnitt 4.4):

```
      <TextBox Name="txt1" Text="{Binding Path=Nachname}" />
      <Label Content="Vorname:" />
```

Beachten Sie auch diese mögliche Kurzsyntax, die auf die Angabe von *Path* verzichtet:

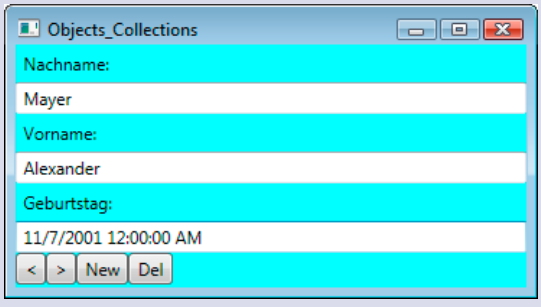
```
      <TextBox Name="txt2" Text="{Binding Vorname}" />
      <Label Content="Geburtstag:" />
      <TextBox Name="txt3" Text="{Binding Geburtstag}" />
```

Einige Schaltflächen definieren:

```
      <StackPanel Orientation="Horizontal">
        <Button Content=" &lt; " Click="Button_Click_1" />
        <Button Content=" &gt; " Click="Button_Click"/>
        <Button Content=" New " Click="Button_Click_2"/>
        <Button Content=" Del " Click="Button_Click_3"/>
      </StackPanel>
    </StackPanel>
```

Ergebnis

Das erzeugte Formular:



Nach dem Start dürfte schon etwas in den Textfeldern angezeigt werden, ein Navigieren zwischen den einzelnen Datensätzen (Objekten) ist allerdings noch nicht möglich.

4.3.3 Navigieren zwischen den Objekten

HINWEIS: An dieser Stelle müssen wir etwas vorgreifen, Abschnitt 4.4 geht auf dieses Thema im Detail ein.

Navigation zwischen Datensätzen bedeutet, dass auch irgendwo ein aktueller Datensatz gespeichert wird und entsprechende Navigationsmethoden zur Verfügung stehen. Auch bei intensiver Suche werden Sie aber derartige Eigenschaften zunächst nicht finden.

WPF erzeugt beim Binden von Collections automatisch eine Sicht auf die eigentliche Collection. Diese Sicht verwaltet den aktuellen Datensatz, bietet Navigationsmethoden an und ermöglicht das Filtern und Sortieren der Daten[2].

Diese automatisch erzeugte Sicht können Sie mit der Methode *CollectionViewSource.GetDefaultView* für eine spezifische Collection abrufen.

Beispiel 4.16: (Fortsetzung) Abrufen und Verwenden der *DefaultView* für unsere Collection

C#

Wir erweitern die Liste der lokalen Variablen, um die Sicht zu speichern:

```
        private ICollectionView view;
...
        public Objects_Collections()
        {
            Klasse = new ObservableCollection<Schüler>();
...
```

Im Konstruktor rufen wir die Sicht ab:

```
            this.view = CollectionViewSource.GetDefaultView(Klasse);
        }
```

Jetzt können wir mit dieser Sicht auch die Navigation zwischen den einzelnen Elementen der Collection realisieren.

Nächstes Objekt:

```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            view.MoveCurrentToNext();
            if (view.IsCurrentAfterLast) view.MoveCurrentToLast();
        }
```

[2] Derartige Sichten können Sie auch selbst erstellen und quasi als Schicht zwischen Daten und DataContext schieben.

Vorhergehendes Objekt:

```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    view.MoveCurrentToPrevious();
    if (view.IsCurrentBeforeFirst) view.MoveCurrentToFirst();
}
```

Wir fügen zum Testen ein neues Objekt zur Laufzeit in die Collection ein:

```
private void Button_Click_2(object sender, RoutedEventArgs e)
{
    Klasse.Add(new Schüler { Nachname = "Möhre", Vorname = "Willi",
                             Geburtstag = new DateTime(1919, 1, 1) });
}
```

Auch das Löschen von Objekten ist auf diesem Wege möglich:

```
private void Button_Click_3(object sender, RoutedEventArgs e)
{
    Klasse.Remove(view.CurrentItem as Schüler);
}
```

Nach dem Start des Beispiels können Sie zwischen den Objekten „navigieren“, Objekte hinzufügen und diese auch wieder löschen. Das Ganze kommt Ihnen sicherlich unter dem Stichwort „Datenbanknavigator“ bekannt vor.

4.3.4 Einfache Anzeige in einer ListBox

Das Anzeigen von Einzeldatensätzen ist ja schon ganz gut, wie aber steht es mit dem Füllen von ganzen Listenfeldern?

Auch hier können Sie, dank Datenbindung, schnell zu brauchbaren Ergebnissen kommen.

Beispiel 4.17: (Fortsetzung) Anbinden einer *ListBox* an unsere Collection

C#

Es genügt zunächst die einfache Zuweisung von „{Binding}“ an die *ItemsSource:*

```
<ListBox Height="100"  IsSynchronizedWithCurrentItem="True" Name="listBox1"
         ItemsSource="{Binding}"/>
```

Der Hintergrund: Da die Collection bereits direkt an das Formular gebunden ist, brauchen wir hier nicht weitere Eigenschaften zu spezifizieren. Alternativ könnten Sie hier auch die Collection per *DataContext* zuweisen.

Und wofür ist das Attribut *IsSynchronizedWithCurrentItem* verantwortlich? Hier sollten Sie sich an unsere Sicht erinnern, die auch den aktuellen „Satzzeiger“ verwaltet. Nur wenn Sie das Attribut auf *True* setzen, wird das aktuelle Item mit dem „Satzzeiger“ synchronisiert (dies gilt für beide Richtungen).

Ergebnis

Die angezeigte *ListBox* zur Laufzeit:

HINWEIS: Die *ItemsSource*-Eigenschaft kann nur verwendet werden, wenn die *Items*-Collection eines *ItemsControl* leer ist. Falls nicht, wird Ihre Anwendung eine *InvalidOperationException* auslösen.

Doch woher „weiß“ die *ListBox* eigentlich, welche Eigenschaften des *Schüler*-Objekts in der Liste darzustellen sind? Antwort: Sie weiß es nicht und verwendet in diesem Fall einfach die *ToString*-Methode des betreffenden Objekts. Wenn Sie jetzt mal kurz in Abschnitt 4.2.3 nachschlagen, werden Sie feststellen, dass wir in weiser Vorahnung bereits die *ToString*-Methode überschrieben haben und damit eine Kombination aus *Nachname* und *Vorname* zurückgeben (siehe oben).

Verwendung von DisplayMemberPath

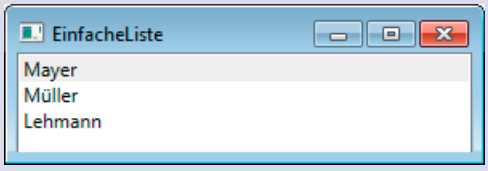
Natürlich ist das Überschreiben der *ToString*-Methode nicht der Weisheit letzter Schluss und so ist es sicher sinnvoll, noch einen anderen Weg zur Auswahl des anzuzeigenden Members zu unterstützen. Genau für diesen Zweck wird die *DisplayMemberPath*-Eigenschaft angeboten, diese bestimmt, welcher Member für den Text des Listeneintrags verwendet wird.

Beispiel 4.18: Verwendung von *DisplayMemberPath* für die Auswahl der anzuzeigenden Eigenschaft

XAML

```
<ListBox Height="100"  IsSynchronizedWithCurrentItem="True" Name="listBox1"
          ItemsSource="{Binding}" DisplayMemberPath="Nachname"/>
```

Ergebnis



Leider genügt jedoch auch diese Version der Anzeigeformatierung nicht immer und so landen wir unweigerlich bei den *DataTemplates*.

4.3.5 DataTemplates zur Anzeigeformatierung

Obige Art der Datenbindung dürfte in vielen Fällen wohl kaum genügen. Die WPF-Entwickler haben aber auch für für diesen Fall vorgesorgt und mit dem *DataTemplate* ein mächtiges Werkzeug geschaffen.

Das Prinzip: Jeder *ListBox/ComboBox* können Sie ein *DataTemplate* zuweisen, das dafür verantwortlich ist, wie das einzelne Item aufgebaut ist (quasi eine Schablone in die die Daten eingefügt werden). Und da WPF im Content eines Items fast jede Zusammenstellung von Elementen akzeptiert, können Sie hier Formatierungen beliebiger Art erzeugen (natürlich im Rahmen der XAML-Vorgaben).

Beispiel 4.19: (Fortsetzung) Wir wollen in der *ListBox* eine zweispaltige Anzeige realisieren (links der Nachname, rechts der Nachname und der Vorname).

XAML

In den Ressourcen (z. B. Window) erzeugen Sie das erforderliche *DataTemplate*:

```
<Window.Resources>
  <DataTemplate x:Key="SchülerListTemplate">
```

Das Layout bestimmen Sie:

```
    <StackPanel Orientation="Horizontal">
```

Bei der Zuweisung von Inhalten können Sie jetzt direkt auf die Eigenschaften zugreifen:

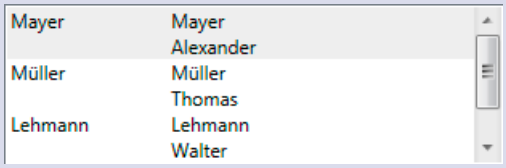
```
      <TextBlock VerticalAlignment="Top" Width="100" Text="{Binding
Path=Nachname}" />
      <StackPanel>
        <TextBlock Text="{Binding Path=Nachname}" />
        <TextBlock Text="{Binding Path=Vorname}" />
      </StackPanel>
    </StackPanel>
  </DataTemplate>
</Window.Resources>
...
```

Last but not least, müssen Sie der *ListBox* auch noch das Template zuweisen:

```
    <ListBox Height="100"  IsSynchronizedWithCurrentItem="True" Name="listBox2"
             ItemsSource="{Binding}" ItemTemplate="{StaticResource
SchülerListTemplate}"/>
```

Ergebnis

Die erzeugte *ListBox:*



Dass Sie hier auch mit Grafiken, optischen Effekten, KontextMenüs etc. arbeiten können, sollte nach den Darstellungen der vorhergehenden Kapitel klar sein.

4.3.6 Mehr zu List- und ComboBox

An dieser Stelle wollen wir uns noch einige spezielle Eigenschaften von *List-* und *ComboBox* ansehen, die in der täglichen Programmierpraxis von Bedeutung sind.

SelectedIndex

Möchten Sie Einträge in der *ListBox* auswählen bzw. bestimmen, der wievielte Eintrag (Index) in der Liste markiert ist, können Sie die *SelectedIndex*-Eigenschaft verwenden.

Beispiel 4.20: Auswahl des zweiten Eintrags

C#

```
private void Button1_Click(object sender, RoutedEventArgs e)
{
    listBox1.SelectedIndex = 1;
}
```

SelectedItem/SelectedItems

Möchten Sie das markierte Listenelement selbst abrufen bzw. das damit verbundene Objekt, verwenden Sie die *SelectedItem*-Eigenschaft. Alternativ können Sie auch eine Liste der markierten Einträge mit *SelectedItems* abrufen.

HINWEIS: Die Collection *SelectedItems* steht Ihnen nur zur Verfügung, wenn Sie *SelectionMode* auf *Multiple* festgelegt haben.

Beispiel 4.21: Verwendung *SelectedItem/SelectedItems*

C#

Wir nutzen unsere überschriebene *ToString*-Methode:

```
MessageBox.Show(listBox1.SelectedItem.ToString());
```

Wir greifen direkt auf einen Member (typisieren nicht vergessen) zu:

```
MessageBox.Show((listBox1.SelectedItem as Schüler).Nachname);
```

Wir zeigen alle markierten Einträge an:

```
foreach (Schüler s in listBox1.SelectedItems)
    MessageBox.Show(s.Nachname);
```

SelectedValuePath und SelectedValue

Mit *SelectedValuePath* können Sie festlegen, welcher Member von der Eigenschaft *SelectedValue* zurückgegeben wird. Dies ist im Zusammenhang mit Datenbanken meist der Primärindex der Tabelle, mit dem Sie einen Datensatz eindeutig identifizieren können.

HINWEIS: Ist *SelectedValuePath* nicht festgelegt, gibt *SelectedValue* das komplette Objekt zurück (entspricht *SelectedItem*).

Beispiel 4.22: Verwendung *SelectedValuePath* und *SelectedValue*

XAML

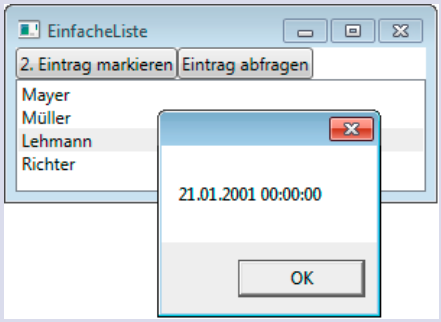
```
<ListBox  IsSynchronizedWithCurrentItem="True" Name="listBox1"
          ItemsSource="{Binding}" DisplayMemberPath="Nachname"
          SelectedValuePath="Geburtstag"/>
```

C#

```
private void Button2_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show(listBox1.SelectedValue.ToString());
}
```

Ergebnis

Das Ergebnis zur Laufzeit:



4.3.7 Verwendung der ListView

Im vorhergehenden Kapitel hatten wir die *ListView* ja bereits kurz gestreift (Trockenschwimmen), an dieser Stelle zeigen wir Ihnen die *ListView* „in Action“.

Einfache Bindung

Prinzipiell ist die *ListView* der *ListBox* recht ähnlich, die Anbindung der Einträge erfolgt ebenfalls per *ItemsSource,* die Auswahl bzw. Bestimmung (*SelectedItem, SelectedValue* etc.) der markierten Einträge ist analog realisiert.

Neu ist, das die *ListView* über Spaltenköpfe verfügt, die Sie getrennt konfigurieren können (*GridViewColumnHeader*) und gegebenenfalls auch für das Sortieren (siehe 2. Beispiel) verwenden können.

Ein weiterer Unterschied ist die Unterstützung von verschiedenen Ansichten, von denen jedoch nur die *GridView* vordefiniert ist. Im weiteren werden wir uns auch nur auf diese Ansicht beschränken.

Beispiel 4.23: (Fortsetzung) Anzeige der Collection-Daten in einer *ListView*

XAML

Zuweisen der Datenquelle (Übernahme von *Window.DataContext*):

```
      <ListView Height="100" IsSynchronizedWithCurrentItem="True"
 ItemsSource="{Binding}">
        <ListView.View>
```

Hier wird die *GridView* definiert:

```
          <GridView>
```

Die einzelnen Spalten definieren:

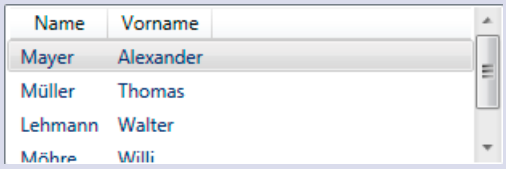
```
            <GridView.Columns>
```

Und jetzt wird es einfach, binden Sie lediglich die gewünschten Eigenschaften an die einzelnen Spalten der *GridView:*

```
              <GridViewColumn Header="Name" DisplayMemberBinding="{Binding
                      Path=Nachname}" />
              <GridViewColumn Header="Vorname"
                              DisplayMemberBinding="{Binding Path=Vorname}" />
            </GridView.Columns>
          </GridView>
        </ListView.View>
      </ListView>
```

Ergebnis

Das Endergebnis zur Laufzeit:



Sortieren der Einträge

Wie schon erwähnt, können Sie die Spaltenköpfe auch für das Sortieren der Einträge nutzen. Ein einfaches Beispiel zeigt die Vorgehensweise:

Beispiel 4.24: Sortieren nach Klick auf den jeweiligen Spaltenkopf

XAML

Unsere Änderung in der Seitenbeschreibung:

```
<ListView Name="ListView1" IsSynchronizedWithCurrentItem="True"
          ItemsSource="{Binding}">
    <ListView.View>
        <GridView>
            <GridView.Columns>
                <GridViewColumn DisplayMemberBinding="{Binding Path=Nachname}" >
                    <GridViewColumnHeader Click="SortClick" Content="Nachname" />
                </GridViewColumn>
                <GridViewColumn DisplayMemberBinding="{Binding Path=Vorname}" >
                    <GridViewColumnHeader Click="SortClick" Content="Vorname" />
                </GridViewColumn>
            </GridView.Columns>
        </GridView>
    </ListView.View>
</ListView>
```

C#

Der Quellcode fällt recht kurz aus:

```
using System.ComponentModel;
using System.Collections.ObjectModel;
...
       private void SortClick(object sender, RoutedEventArgs e)
        {
```

Zunächst die betreffende Spalte bestimmen:

```
            GridViewColumnHeader spalte = sender as GridViewColumnHeader;
```

Die Defaultview bestimmen:

```
            ICollectionView view =

CollectionViewSource.GetDefaultView(ListView1.ItemsSource);
```

Eine neue Sortierfolge festlegen:

```
            view.SortDescriptions.Clear();
            view.SortDescriptions.Add(new
SortDescription(spalte.Content.ToString(),
                                      ListSortDirection.Ascending));
```

Und aktualisieren:

```
            view.Refresh();
        }
```

Auf weitere Experimente mit der *ListView* verzichten wir an dieser Stelle, mit dem *DataGrid* steht uns ein wesentlich mächtigeres Control zur Verfügung. Mehr dazu in Abschnitt 4.8.

Wie Sie auch größere Collections bändigen, zeigt das Praxisbeispiel in Abschnitt 4.9.1.

4.4 Noch einmal zurück zu den Details

Nachdem wir in den bisherigen Abschnitten schon mehrfach vorgreifen mussten, wollen wir an dieser Stelle noch einmal kurz auf einige Details der Datenbindung eingehen.

Interessant für den Datenbankprogrammierer ist vor allem eine Zwischenschicht, die vom WPF quasi zwischen die Daten (Collections) und die reinen Anzeige-Controls (z. B. *ListView*) geschoben wird, um einige datenbanktypische Operationen zu ermöglichen:

- Verwaltung des aktuellen Satzzeigers
- Navigation zwischen den Datensätzen
- Sortierfunktion
- Filterfunktion

Die Rede ist von der Klasse *CollectionView*, um deren Erzeugung Sie sich nicht selbst kümmern müssen, da Sie diese automatisch erstellte View recht einfach abrufen können.

Beispiel 4.25: Abrufen der *CollectionView*

C#

```
...
        NWDataContext db = new NWDataContext();
        ICollectionView viewO;
...
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            lvOrder.DataContext = db.Orders;
            viewO = CollectionViewSource.GetDefaultView(lvOrder.DataContext);
        }
```

Mit dieser *CollectionView* stellt es jetzt kein Problem mehr dar, die oben gewünschten Datenbankfunktionen zu implementieren.

4.4.1 Navigieren in den Daten

Wie schon in den vorhergehenden Abschnitten gezeigt, ist eine der Hauptaufgaben der *CollectionView* die Verwaltung des „Satzzeigers“. Dazu steht Ihnen zunächst die Eigenschaft *CurrentItem* zur Verfügung, die das aktuell ausgewählte Element der gebundenen Collection zurückgibt.

Weitere interessante Eigenschaften:

Eigenschaften	Beschreibung
CurrentItem	Aktuelles Element der Auflistung.
CurrentPosition	Ordinalposition des aktuellen Elements in der Auflistung.
IsCurrentAfterLast	Befindet sich der „Satzzeiger“ hinter dem Ende der Auflistung?
IsCurrentBeforeFirst	Befindet sich der „Satzzeiger“ vor dem Beginn der Auflistung?

Die eigentliche Navigation realisieren Sie mit den folgenden Methoden:

Methoden	Beschreibung
MoveCurrentTo	Das übergebene Element wird als *CurrentItem* festgelegt.
MoveCurrentToFirst	„Satzzeiger“ auf das erste Element verschieben.
MoveCurrentToLast	„Satzzeiger“ auf das letzte Element verschieben.
MoveCurrentToNext	„Satzzeiger“ auf das folgende Element verschieben.
MoveCurrentToPosition	„Satzzeiger“ auf den angegebenen Index verschieben.
MoveCurrentToPrevious	„Satzzeiger“ auf das vorhergehende Element verschieben.

Beispiel 4.26: Navigationstasten für „Vor“ und „Zurück“

C#

Der folgende Aufwand ist nötig, um nicht hinter bzw. vor dem letzten bzw. ersten Datensatz zu landen:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    view.MoveCurrentToNext();
    if (view.IsCurrentAfterLast) view.MoveCurrentToLast();
}

private void Button_Click_1(object sender, RoutedEventArgs e)
{
    view.MoveCurrentToPrevious();
    if (view.IsCurrentBeforeFirst) view.MoveCurrentToFirst();
}
```

Beispiel 4.27: Verwendung von *CurrentItem*

C#

Löschen eines Listeneintrags per *CurrentItem* und Typisierung:

```
private void Button_Click_3(object sender, RoutedEventArgs e)
{
    Klasse.Remove(view.CurrentItem as Schüler);
}
```

4.4.2 Sortieren

Dass sich die *CollectionView* auch zum Sortieren eignet, haben wir ja bereits am Beispiel der *ListView* gezeigt, wo durch Klicken auf den Spaltenkopf die Collection nach der jeweiligen Spalte sortiert wurde.

Zum Einsatz kommt die Collection *SortDescriptions,* die neben den Membernamen auch die Sortierfolge enthält. Da es sich um eine Collection handelt, können Sie auch mehrere Elemente angeben:

Beispiel 4.28: Sortieren einer Collection

C#

```
    private void SortClick(object sender, RoutedEventArgs e)
    {
       GridViewColumnHeader spalte = sender as GridViewColumnHeader;

CollectionView abrufen:

       ICollectionView view =
 CollectionViewSource.GetDefaultView(ListView1.ItemsSource);

Bisherige Sortiervorgaben löschen:

       view.SortDescriptions.Clear();

Eine neue Sortierfolge (Spaltenname, Aufsteigend) festlegen:

       view.SortDescriptions.Add(new SortDescription(spalte.Content.ToString(),
                                 ListSortDirection.Ascending));

Ansicht aktualisieren:

       view.Refresh();
   }
```

4.4.3 Filtern

Auch wenn Sie mit dieser Variante vorsichtig sein sollten (Daten werden eigentlich vor der Anzeige gefiltert, um unnötigen Traffic zu vermeiden), so besteht doch die Möglichkeit, zur Laufzeit gezielt Daten aus der gebundenen Collection herauszufiltern. Nutzen Sie dazu die *Filter*-Eigenschaft, der Sie eine selbst zu definierende Methode zuweisen.

Beispiel 4.29: Filter festlegen

C#

Zunächst unsere Filterfunktion (alle Einträge die mit „T“ beginnen):

```
        protected bool MeinFilter(object value)
        {
            Schüler s = value as Schüler;
            return s.Vorname.StartsWith("T");
        }
```

Und hier wird der Filter zugewiesen (ein Aktualisieren ist nicht nötig):

```
        private void Button3_Click(object sender, RoutedEventArgs e)
        {
            ICollectionView view =
CollectionViewSource.GetDefaultView(listBox1.ItemsSource);
            view.Filter += MeinFilter;
        }
```

HINWEIS: Möchten Sie den Filter wieder löschen, weisen Sie der Eigenschaft einfach *null* zu.

4.4.4 Live Shaping

Die vorhergehenden Funktionen zum Sortieren, Filtern und auch Gruppieren funktionieren recht gut, auch wenn Sie zum Beispiel neue Einträge zur Collection hinzufügen. Alternativ können Sie auch die *Refresh*-Methode der *CollectionView* aufrufen.

Doch was, wenn Sie lediglich ein Objekt der Collection bearbeiten und sich so z. B. die Filterbedingung für dieses Objekt ändert? In diesem Fall werden Sie schnell feststellen, das Anzeige und Inhalt der Collection nicht mehr übereinstimmt.

Beispiel 4.30: (Fortsetzung) Fehlende Aktualisierung

C#

```
...
```

Filtern Sie die Daten und rufen Sie folgende Methode auf, passiert nichts:

```
        private void Button5_Click(object sender, RoutedEventArgs e)
        {
            Schüler schueler = Klasse.Where(sch => sch.Vorname ==
"Thomas").FirstOrDefault();
            schueler.Vorname = "aaaa";
```

Erst nach einem Refresh sind die gefilterten Daten auch aktuell:

```
            //ICollectionView view = CollectionViewSource.GetDefaultView(
                                                        listBox1.ItemsSource);
```

```
            //view.Refresh();
        }
...
```

Hier hilft Ihnen Live Shaping weiter. Über das Interface *ICollectionViewLiveShaping* können Sie bestimmte Spalten zur Überwachung anmelden.

Jeweils drei neue Member sind für die weitere Arbeit interessant:

- Die Eigenschaft *CanChangeLiveFiltering* (... *Sorting*, ...*Grouping*) bestimmt, ob die Überwachung möglich ist.
- Die Eigenschaft *IsLiveFiltering* (... *Sorting*, ...*Grouping*) bestimmt, ob die Überwachung eingeschaltet ist
- Die Collection *LiveFilteringProperties* (... *Sorting...*, ...*Grouping...*) enthält die Namen der zu überwachenden Eigenschaften.

Beispiel 4.31: Filtern mit Live Shaping

C#

```
...
        private void Button6_Click(object sender, RoutedEventArgs e)
        {
```

Filter wie bekannt festlegen:

```
            ICollectionView view =
CollectionViewSource.GetDefaultView(listBox1.ItemsSource);
            view.Filter += MeinFilter;
```

Wir rufen das neue Interface ab:

```
            ICollectionViewLiveShaping viewls = view as ICollectionViewLiveShaping;
```

Test auf Interface:

```
            if (viewls == null)
            {
                MessageBox.Show("Nicht unterstützt!");
            }
```

Ist die Überwachung möglich:

```
            if (viewls.CanChangeLiveFiltering)
            {
```

Feld *Vorname* soll überwacht werden:

```
                viewls.LiveFilteringProperties.Add("Vorname");
                viewls.IsLiveFiltering = true;
            }
        }
...
```

HINWEIS: Da diese Form der Überwachung recht ressourcenintensiv ist, sollten Sie davon nur Gebrauch machen, wenn es unbedingt nötig ist.

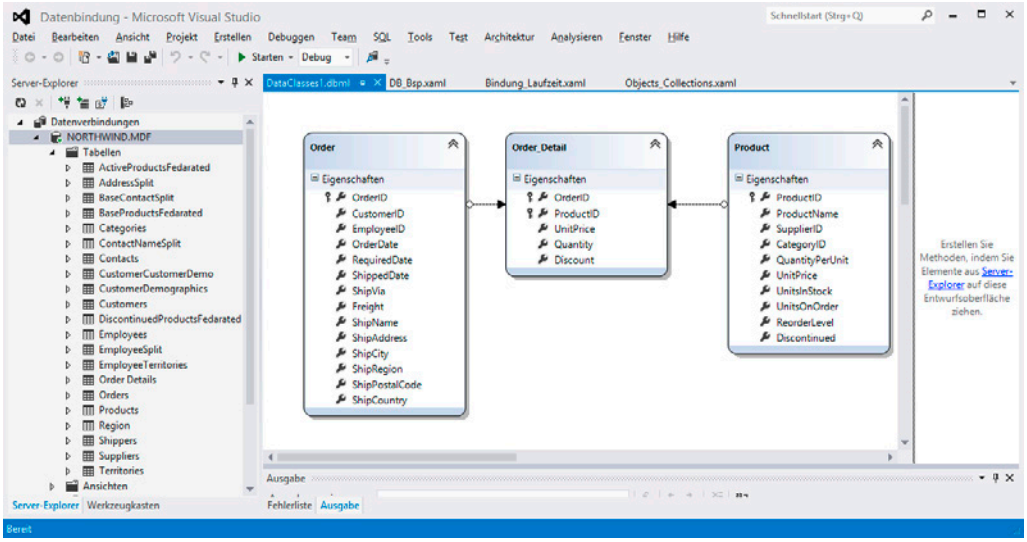
4.5 Anzeige von Datenbankinhalten

Anzeige eigener Collections gut und schön, aber wir wollen auch noch kurz einen Blick aufs große Ganze werfen und damit sind wir schon bei der „Königsdisziplin“, den Datenbanken, angelangt.

HINWEIS: Wer jetzt Berge von ADO.NET-Quellcode erwartet, den werden wir enttäuschen. Für den Zugriff auf unsere *Northwind*-Beispieldatenbank werden wir zunächst LINQ to SQL verwenden, den kleinen Bruder des Entity Frameworks.

4.5.1 Datenmodell per LINQ to SQL-Designer erzeugen

Fügen Sie Ihrem WPF-Projekt eine neue „LINQ to SQL Klasse“ hinzu, um den LINQ to SQL-Designer zu öffnen (*Projekt|Neues Element hinzufügen*). Damit haben Sie bereits die zentrale *DataContext*-Klasse[3] erstellt. Den Namen dieser Klasse können Sie jetzt gegebenenfalls über das Eigenschaftenfenster anpassen (wir wählen *NWDataContext*).



[3] Ja das ist wieder eine der „glücklich“ gewählten Namensübereinstimmungen. Dieser *DataContext* hat nichts mit dem WPF-DataContext zu tun!

In die noch leere Arbeitsfläche (diese ähnelt dem Klassendesigner) fügen Sie die benötigten SQL-Server-Tabellen ein. Nutzen Sie dazu den Server-Explorer (siehe linke Seite).

HINWEIS: Für unser Beispiel fügen Sie die Tabellen *Order, Order_Detail* und *Product* ein.

Der Designer erstellt nachfolgend automatisch die erforderlichen C#-Mapperklassen für die einzelnen Tabellen sowie deren Associations. Damit sind wir aber auch schon wieder bei den schon bekannten Collections angekommen, die weitere Vorgehensweise dürfte Ihnen also bekannt vorkommen.

HINWEIS: Sie können neben reinen Tabellen auch Views bzw. Gespeicherte Prozeduren in den Designer einfügen. Views werden wie Tabellen behandelt, Gespeicherte Prozeduren werden als Methoden der *DataContext*-Klasse mit typisierten Rückgabewerten gemappt.

4.5.2 Die Programm-Oberfläche

Nach Schließen des Designers wollen wir uns mit einem einfachen WPF-Projekt von der Funktionsfähigkeit überzeugen.

```
<Window x:Class="Datenbindung.DB_Bsp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Eine Ereignisprozedur beim Öffnen des Window:

```
    Title="DB_Bsp" Height="300" Width="476" Loaded="Window_Loaded">
```

Zweispaltiges Layout per *Grid*:

```
 <Grid>
   <Grid.ColumnDefinitions>
     <ColumnDefinition Width="75" />
     <ColumnDefinition Width="*" />
   </Grid.ColumnDefinitions>
```

Hier die *ListView* mit den vorhandenen Bestellungen (Tabelle *Order*):

```
   <ListView Grid.Column="0" Name="lvOrder" IsSynchronizedWithCurrentItem="True"
             ItemsSource="{Binding}" SelectionChanged="lvOrder_SelectionChanged"
             HorizontalAlignment="Left" >
```

Die eigentlich Bindung erfolgt per *DataContext*-Zuweisung im C#-Code. Über das *SelectionChanged*-Ereignis werden wir die Detaildaten in die zweite *ListView* „zaubern“.

```
    <ListView.View>
      <GridView>
        <GridView.Columns>
```

Wir zeigen nur die Spalte mit der Bestellnummer an:

```
          <GridViewColumn Header="OrderID" DisplayMemberBinding="{Binding OrderID}" />
        </GridView.Columns>
      </GridView>
    </ListView.View>
  </ListView>
```

Die *ListView* für die Detaildaten (die *DataContext*-Eigenschaft setzen wir im *SelectionChanged*-Ereignis der obigen *ListView*):

```
  <ListView Grid.Column="1" Name="lvOrderDetails" IsSynchronizedWithCurrentItem="True"
            ItemsSource="{Binding}" >
    <ListView.View>
      <GridView>
        <GridView.Columns>
          <GridViewColumn Header="OrderId" DisplayMemberBinding="{Binding OrderID}" />
          <GridViewColumn Header="ID" DisplayMemberBinding="{Binding ProductID}" />
```

Wer jetzt erwartet, dass die Autoren sich die Mühe machen und noch eine dritte *GridView* für die Artikelnamen einbinden, hat nicht mit der Leistungsfähigkeit von LINQ to SQL gerechnet. Es genügt die Abfrage der untergeordneten Collection *Product*:

```
          <GridViewColumn Header="Artikelname"
                          DisplayMemberBinding="{Binding Product.ProductName}" />
```

Ein sinnvoller Vorteil von objekt-relationalen Mapper-Klassen!

```
        </GridView.Columns>
      </GridView>
    </ListView.View>
  </ListView>
 </Grid>
</Window>
```

4.5.3 Der Zugriff auf die Daten

Jetzt müssen wir noch den erforderlichen C#-Code erstellen, um die Daten auch aus der Datenbank abzurufen.

```
...
    public partial class DB_Bsp : Window
    {
```

Die meiste Arbeit nimmt uns der LINQ to SQL-DataContext ab, den wir gleich zu Beginn instanziieren:

```
        NWDataContext db = new NWDataContext();
```

Wir wollen auch die Default-View zwischenspeichern, da wir diese für das Abrufen der Detaildatensätze benötigen:

```
        ICollectionView viewO;
...
```

Beim Laden des Fensters:

```
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
```

Den *DataContext* der linken *ListView* wir die Tabelle *Orders* zugewiesen:

```
            lvOrder.DataContext = db.Orders;
```

Die Default-View abrufen:

```
            viewO = CollectionViewSource.GetDefaultView(lvOrder.DataContext);
        }
```

So, das wäre schon alles, wenn da nicht noch die Detaildatenanzeige fehlen würde. Im *SelectionChanged*-Ereignis der linken *ListView* kümmern wir uns zunächst um das Abrufen des aktuellen Datensatzes und leiten aus diesem Objekt die OrderDetails ab (als Collection enthalten):

```
        private void lvOrder_SelectionChanged(object sender, SelectionChangedEventArgs e)
        {
            lvOrderDetails.DataContext = (viewO.CurrentItem as Order).Order_Details;
        }
    }
```

Wer jetzt beim Zugriff auf den Microsoft SQL Server noch Wert auf ADO.NET-Objekte legt, dem ist nicht zu helfen. Kürzer kann das Beispiel kaum ausfallen.

Ach ja, wie kommen eigentlich neue Datensätze in die Tabellen? Hier genügt es, wenn Sie z. B. ein neues *Product*-Objekt erstellen und es an die *Products*-Collection anhängen. Mit einen *SubmitChanges* des *DataContext*-Objekts (der LINQ to SQL *DataContext*) ist die Änderung dann auch schon zum Server übertragen.

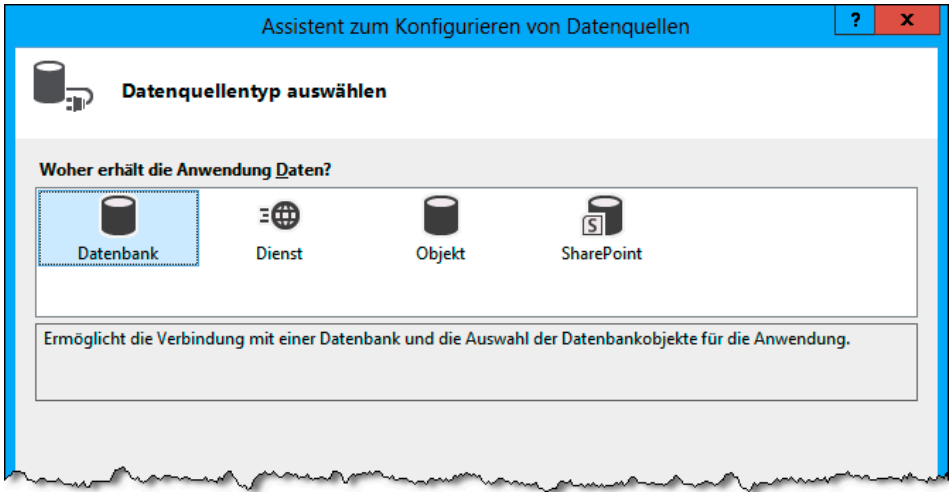
4.6 Drag & Drop-Datenbindung

Die in den vorhergehenden Abschnitten gezeigte Vorgehensweise war recht einfach und mit wenig Schreibaufwand verbunden. Wenn Sie hier weiter lesen, sind Sie vermutlich daran interessiert, noch weniger Code zu produzieren und stattdessen die Assistenten für sich arbeiten zu lassen. Mal sehen, ob Sie mit dem Ergebnis und der Vorgehensweise zufrieden sind!

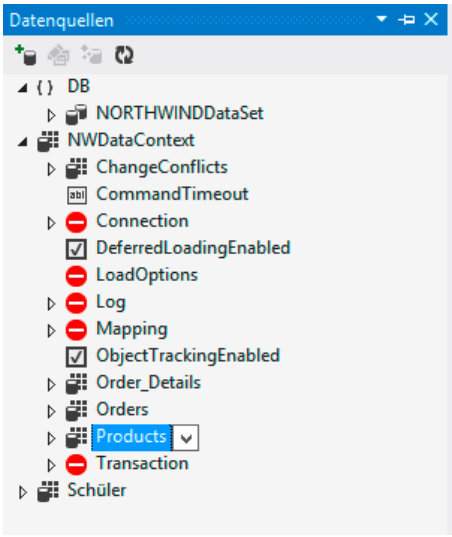
4.6.1 Vorgehensweise

Die Vorgehensweise orientiert sich an der Arbeitsweise bei den Windows Forms:

- Öffnen Sie das Datenquellen-Fenster.
- Ist die gewünschte Datenquelle noch nicht vorhanden, erzeugen Sie diese über den Klick auf den „Hinzufügen“-Button. In diesem Fall sollte jetzt der Datenquellen-Assistent erscheinen:

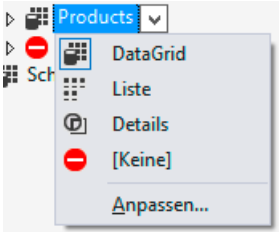


- Wählen Sie den Eintrag „Datenbank“, wenn Sie eine Datenbank gänzlich neu einfügen wollen, oder „Objekt“, wenn Sie bereits über ein Datenmodell (z. B. LINQ to SQL) verfügen.
- Haben Sie die Datenquelle bzw. die Datenobjekte erfolgreich eingebunden, dürften diese im Datenquellen-Fenster angezeigt werden:

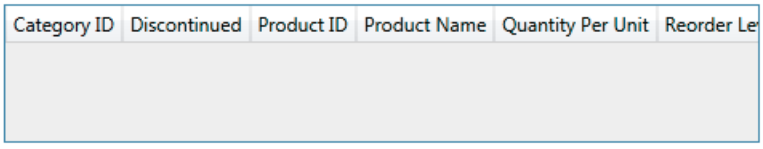


Obige Abbildung zeigt ein eingebundenes DataSet, einen LINQ to SQL-DataContext und unsere *Schüler*-Klasse.

- Wählen Sie jetzt beispielsweise eine Collection „Products“ im Datenquellen-Fenster aus, wird Ihnen folgende Auswahlliste angezeigt:

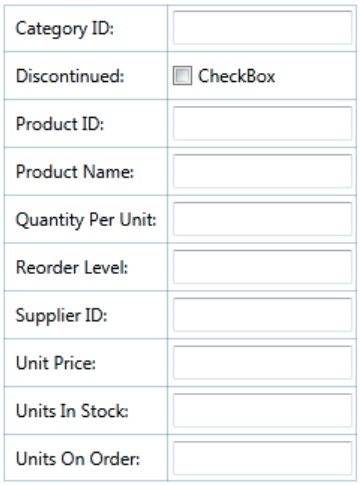


Die Auswahl bestimmt, welche Controls nach einer Drag&Drop-Operation mit dieser Collection in das Fenster eingefügt werden. Wählen Sie die erste Option wird automatisch ein komplett fertig konfiguriertes *DataGrid* in Ihr WPF-Formular eingefügt:



Analog gilt dieses auch für die Auswahl „List“, hier kommt eine *ListView* zum Einsatz, die jedoch, auf Templates aufbauend, statt statischem Text Textfelder verwendet.

Mit der Auswahl „Details“ wird Ihnen ein *Grid* mit den erforderlichen *Label*-, *TextBox*- und *CheckBox*-Controls generiert:

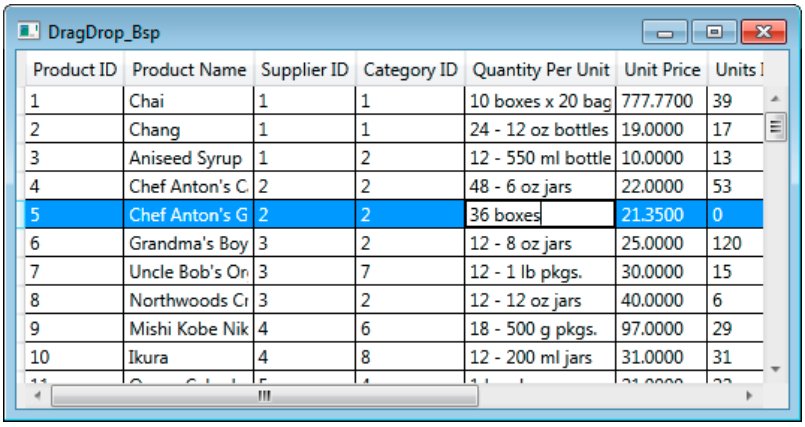


- Abschließend sollten Sie ein Blick auf den C#-Quellcode des Formulars werfen, hier ist teilweise noch mit etwas Arbeit zu rechnen:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    Datenbindung.DB.NORTHWINDDataSet NORTHWINDDataSet =
   ((Datenbindung.DB.NORTHWINDDataSet)(this.FindResource("NORTHWINDDataSet")));
    // Lädt Daten in Tabelle "Products". Sie können diesen Code nach Bedarf ändern.
    Datenbindung.DB.NORTHWINDDataSetTableAdapters.ProductsTableAdapter
    nORTHWINDDataSetProductsTableAdapter = new
      Datenbindung.DB.NORTHWINDDataSetTableAdapters.ProductsTableAdapter();
    nORTHWINDDataSetProductsTableAdapter.Fill(NORTHWINDDataSet.Products);
    System.Windows.Data.CollectionViewSource productsViewSource =
           ((System.Windows.Data.CollectionViewSource)
          (this.FindResource("productsViewSource")));
    productsViewSource.View.MoveCurrentToFirst();
}
```

Im obigen Beispiel (Anbindung eines DataSets) brauchen Sie keine Änderung vorzunehmen, bei der Anbindung von LINQ to SQL-DataContext-Objekten müssen Sie sich jedoch selbst um das Erstellen des *DataContext* kümmern.

- Einem Probelauf des Programm steht jetzt nichts mehr im Wege:



4.6.2 Weitere Möglichkeiten

Selbstverständlich können Sie jetzt noch den XAML-Code nachbearbeiten und Spalten ein-/ausblenden bzw. umformatieren. Dazu genügt es meist, wenn Sie ein neues *DataTemplate* mit dem gewünschten Eingabe-Control zuweisen:

Statt einer einfachen *TextBox*:

```
<DataGridTextColumn Binding="{Binding Path=OrderDate}" Header="Order-Date"
                    Width="SizeToHeader" />
```

können Sie zum Beispiel auch einen Kalender einblenden:

```
<DataGridTemplateColumn Header="Order Date" Width="SizeToHeader">
    <DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
            <DatePicker SelectedDate="{Binding Path=OrderDate}" />
        </DataTemplate>
    </DataGridTemplateColumn.CellTemplate>
</DataGridTemplateColumn>
```

Dass Sie auch alle anderen Formatierungsmöglichkeiten des *DataGrid* nutzen können, brauchen wir an dieser Stelle sicher nicht zu erwähnen.

Interessant für den Programmierer ist noch der automatisch erstellte <*Window.Resources*>-Abschnitt, in dem sowohl das nötige *DataSet* als auch eine *CollectionViewSource* erzeugt wird:

```
<Window x:Class="Datenbindung.DragDrop_Bsp"
...
    xmlns:my1="clr-namespace:Datenbindung">
    <Window.Resources>
        <my:NORTHWINDDataSet x:Key="NORTHWINDDataSet" />
        <CollectionViewSource x:Key="productsViewSource"
                Source="{Binding Path=Products, Source={StaticResource
NORTHWINDDataSet}}" />
    </Window.Resources>
```

Letztere können Sie dazu nutzen, um zum Beispiel eine Navigation zwischen den Datensätzen zu realisieren:

Instanz ermitteln:

```
...
     System.Windows.Data.CollectionViewSource productsViewSource =
       ((System.Windows.Data.CollectionViewSource)(this.FindResource
("productsViewSource")));
```

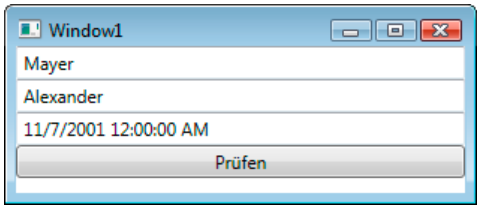
Über die *View* stehen Ihnen alle Navigationsmöglichkeiten zur Verfügung:

```
     productsViewSource.View.MoveCurrentToFirst();
```

4.7 Formatieren von Werten

In unseren Beispielen haben wir uns bislang erfolgreich davor gedrückt, Datumswerte, Währungen etc. in einem sinnvollen Format anzuzeigen bzw. zu formatieren.

Binden Sie beispielsweise einen Datumswert an eine *TextBox*, wird zunächst das Standardformat angezeigt:



Das sieht aus deutscher Sicht zunächst wenig erfreulich aus, aber mit dem *Language*-Attribut können Sie hier etwas nachhelfen.

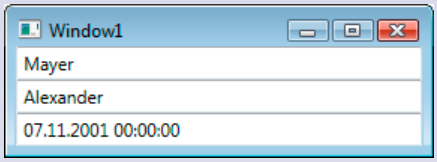
Beispiel 4.32: Verwendung *Language*-Attribut

XAML

```
    <TextBox Text="{Binding Path=Geburtstag}" Language="de"/>
```

Ergebnis

Nachfolgend sollte zumindest ein deutscher Datumswert angezeigt werden:



Doch auch dies ist noch nicht der Weisheit letzter Schluss.

4.7.1 IValueConverter

Mit Hilfe der WPF-Wertkonvertierer können Sie jede beliebige Konvertierung zwischen Quelle und Ziel einer Datenbindung realisieren. Dazu erstellen Sie eine Klasse, die das *IValueConverter*-Interface unterstützt. Diese Klasse muss zwei Methoden implementieren:

- *Convert* (von der Quelle zum Ziel)
- *ConvertBack* (vom Ziel zur Quelle)

Sicher können Sie sich denken, dass die *ConvertBack*-Methode den höheren Programmieraufwand erfordert, hat doch hier der User die Möglichkeit, zunächst beliebige Werte in die Textfelder einzugeben, die Sie dann mühsam in den geforderten Datentyp umwandeln müssen.

Beispiel 4.33: Implementieren und Verwenden eines Wert-Konvertierers

C#

An dieser Stelle wollen wir allerdings nicht das Rad neu erfinden, sondern ein Beispiel aus dem Microsoft MSDN darstellen.

Zunächst der eigentliche Wert-Konvertierer:

```
using System.Globalization;

namespace Datenbindung
{
    public partial class Window1 : Window
    {
```

Hier die neue Klasse *DateConverter,* die Sie mit entsprechenden Attributen versehen sollten:

```
        [ValueConversion(typeof(DateTime), typeof(String))]
        public class DateConverter : IValueConverter
        {
```

Konvertieren von der Quelle zum Ziel (übergeben werden die Quelleigenschaft, der Zieleigenschaft-Typ, ein Konverter-Parameter sowie die aktuellen Landeseinstellungen):

```
            public object Convert(object value, Type targetType, object parameter,
                                  CultureInfo culture)
            {
                DateTime date = (DateTime)value;
                return date.ToShortDateString();
            }
```

Konvertieren vom Ziel (z. B. *TextBox*) zur Quelle (z. B. Objekt):

```
            public object ConvertBack(object value, Type targetType, object
parameter,
                                      CultureInfo culture)
            {
                string strValue = value.ToString();
```

```
                DateTime resultDateTime;
                if (DateTime.TryParse(strValue, out resultDateTime))
                {
                    return resultDateTime;
                }
                return value;
            }
        }
```

XAML

Die Verwendung im XAML-Code:

```
<Window x:Class="Datenbindung.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Zunächst den lokalen Namespace einbinden:

```
    xmlns:local="clr-namespace:Datenbindung"
    Title="Window1" Height="300" Width="300" >
```

Die Einbindung der Klasse erfolgt per Ressource:

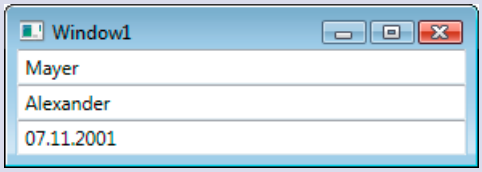
```
  <Window.Resources>
    <local:DateConverter x:Key="dateConverter"/>
  </Window.Resources>
  <StackPanel Name="StackPanel1">
    <TextBox Text="{Binding Path=Nachname}" Name="txt1" />
    <TextBox Text="{Binding Path=Vorname}" />
```

Und hier verwenden wir den Konverter bei der Bindung:

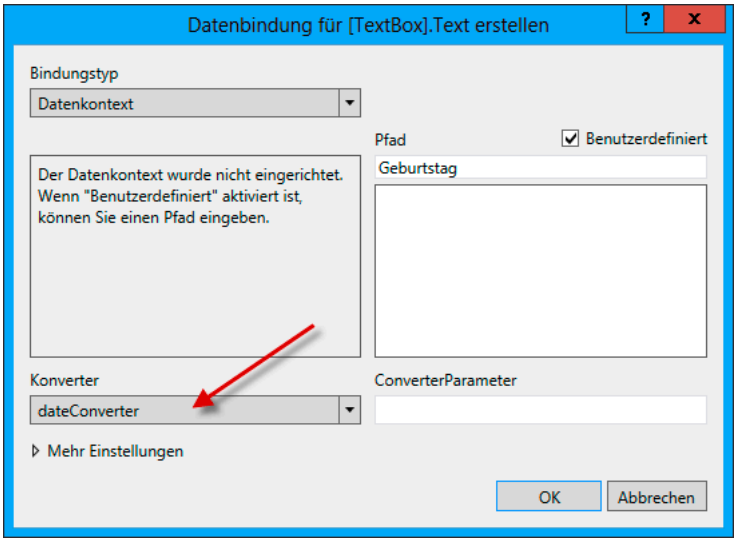
```
    <TextBox Text="{Binding Path=Geburtstag, Converter={StaticResource
 dateConverter}}" />
    <Button Click="Button_Click">Prüfen</Button>
  </StackPanel>
 </Window>
```

Ergebnis

Das neue Ergebnis sieht schon viel ansprechender aus:



HINWEIS: Zusätzlich besteht die Möglichkeit, vorhandene Wertkonvertierer per Eigenschafteneditor (siehe folgende Abbildung) zuzuweisen. Der Eigenschaften-editor erstellt, falls erforderlich, die entsprechenden Einträge im <*Window.Resources*>-Abschnitt des Formulars und weist das Attribut „Converter“ zu.



4.7.2 BindingBase.StringFormat-Eigenschaft

Nachdem Sie sich durch unser obiges Beispiel gequält haben, wollen wir Ihnen auch nicht die dritte Variante zur Formatierung von Werten vorenthalten.

Werfen Sie doch einmal einen Blick auf die *BindingBase.StringFormat*-Eigenschaft, welche die Verwendung eines *IValueConverters* in vielen Standardfällen überflüssig macht.

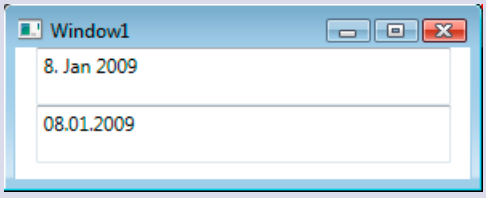
Beispiel 4.34: Zwei verschiedene Datumsformate zuweisen

XAML

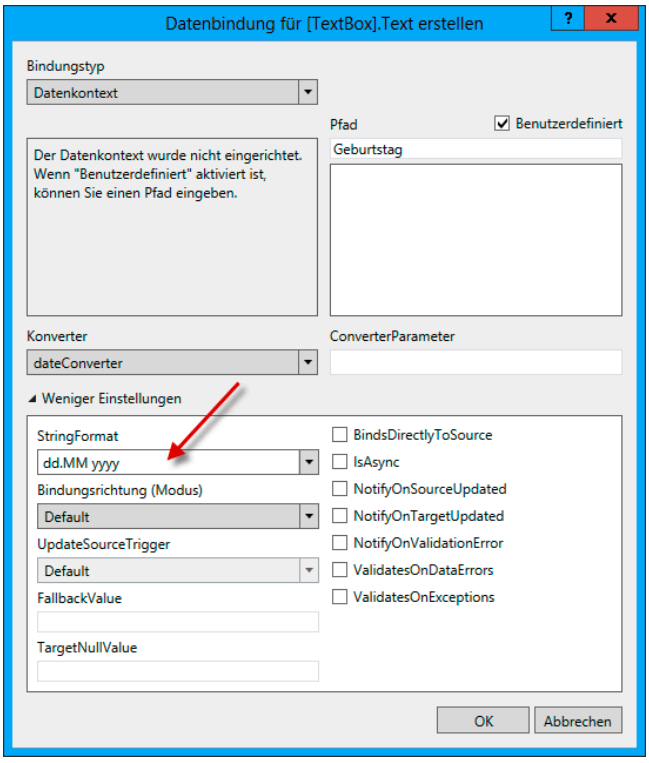
```
...
<TextBox Text="{Binding Path=Geburtstag, StringFormat= d. MMM yyyy}" />
...
<TextBox Text="{Binding Path=Geburtstag, StringFormat= dd.MM.yyyy }" />
...
```

Ergebnis

Das Ergebnis:



HINWEIS: Alternativ können Sie auch hier den Eigenschafteneditor nutzen (siehe folgende Abbildung), der eine einfache Zuweisung des Formatierungsstrings erlaubt.



4.8 Das DataGrid als Universalwerkzeug

Seit der WPF-Version 4 wird auch ein *DataGrid* regulär unterstützt, ohne zusätzliche Toolkits etc. laden zu müssen. Wie die schon besprochene *ListView* erlaubt auch das *DataGrid* die Anzeige von Collections im Tabellenformat. Zusätzlich werden Funktionen zum Editeren, Löschen, Auswählen und Sortieren angeboten.

HINWEIS: Anhand einiger Fallbeispiele wollen wir Ihnen ein Übersicht des Funktionsumfangs geben, für eine komplette Beschreibung aller Eigenschaften bzw. Möglichkeiten fehlt hier jedoch der Platz und wir verweisen auf die recht umfangreiche Hilfe zum *DataGrid*-Control.

4.8.1 Grundlagen der Anzeige

Wie fast nicht anders zu erwarten, erfolgt die Anbindung an die Datenquelle mittels *ItemsSource*-Eigenschaft, wir erzählen Ihnen an dieser Stelle also nichts Neues und verweisen auf die vorhergehenden Abschnitte.

Im Unterschied zu den bereits beschriebenen Controls bietet uns das *DataGrid* einen wesentlichen Vorteil, Sie brauchen sich nicht um das Erstellen der einzelnen Spalten zu kümmern, dank *AutoGenerateColumns*-Eigenschaft werden automatisch die erforderlichen Spalten erzeugt.

Beispiel 4.35: Anbinden des *DataGrid* an LINQ to SQL-Daten

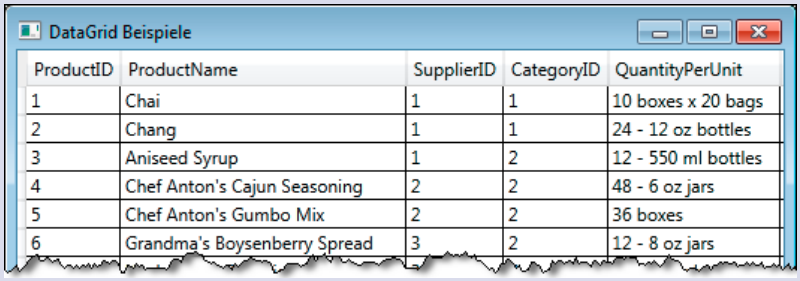
XAML

```
<DataGrid Name="DataGrid1" />
```

C#

```
public partial class DataGrid_Bsp : Window
{
    NWDataContext db = new NWDataContext();

    public DataGrid_Bsp()
    {
        InitializeComponent();
        DataGrid1.ItemsSource = db.Products;
    }
```

Ergebnis



ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit
1	Chai	1	1	10 boxes x 20 bags
2	Chang	1	1	24 - 12 oz bottles
3	Aniseed Syrup	1	2	12 - 550 ml bottles
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars
5	Chef Anton's Gumbo Mix	2	2	36 boxes
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars

Die Verwendung der *AutoGenerateColumns*-Eigenschaft ist sicher recht praktisch, doch haben Sie in diesem Fall keinen Einfluss auf Anzahl, Reihenfolge und Aussehen der Spalten.

Das *DataGrid* selbst unterscheidet in diesem Fall lediglich zwischen Spalten der Typen *DataGridTextColumn* und *DataGridCheckBoxColumn*, deren Bedeutung sich bereits durch den Namen erklärt.

4.8.2 UI-Virtualisierung

Sicher interessiert es Sie auch, wie leistungsfähig das *DataGrid* ist. Erstellen Sie ruhig einmal eine Collection mit 1.000.000 Datensätzen und weisen Sie diese als *ItemsSource* zu. Sie werden feststellen, dass das Erzeugen der Collection wesentlich länger dauert als die

Anzeige der Daten. Der Grund für dieses Verhalten basiert auf der UI-Virtualisierung, die mit Hilfe eines *VirtualizingStackPanel* als Layoutpanel innerhalb des *DataGrid* (auch *ListView*, *ListBox* etc.) verwendet wird.

Das *VirtualizingStackPanel* sorgt dafür, dass nur die gerade sichtbaren Einträge (bzw. die dazu notwendigen Controls) erzeugt werden. Was passiert, wenn dies nicht so ist, können Sie ganz einfach ausprobieren. Es genügt, wenn Sie das folgende Attribut in die Elementedefinition einfügen:

```
<DataGrid VirtualizingStackPanel.IsVirtualizing="False" Name="DataGrid1" />
```

Bitte besorgen Sie sich rechtzeitig eine Zeitung und eine Kanne Kaffee wenn Sie versuchen wollen, eine große Collection an das *DataGrid* zu binden. Im extremsten Fall kommt es zur Meldung, dass der verfügbare Arbeitsspeicher nicht ausreicht. Die Ursache dürfte schnell klar werden, wenn Sie sich vorstellen, dass für jede erforderliche Zeile und alle angezeigten Spalten die entsprechenden Anzeige-Controls generiert werden müssen.

4.8.3 Spalten selbst definieren

Gehen Ihnen die Möglichkeiten von *AutoGenerateColumns* nicht weit genug, können Sie alternativ auch selbst Hand anlegen und die einzelnen Spalten frei definieren. Setzen Sie in diesem Fall das Attribut *AutoGenerateColumns* auf *false* und fügen Sie die Spaltendefinitionen der *Columns*-Eigenschaft hinzu (die Reihenfolge der Definition entscheidet über die Anzeigereihenfolge).

Wir machen es uns in diesem Fall zunächst etwas einfacher und generieren das *DataGrid* mit allen Spaltendefinitionen per Drag&Drop-Datenbindung (siehe Abschnitt 4.6).

Beispiel 4.36: *DataGrid* mit einzeln definierten Spalten

XAML

Zunächst das Erzeugen der *CollectionViewSource*:

```
...
        <CollectionViewSource x:Key="schülerViewSource" d:DesignSource=
                              "{d:DesignInstance my:Schüler, CreateList=True}" />
...
```

Hier das *DataGrid*:

```
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
                  ItemsSource="{Binding}" Name="DataGrid1"
                  RowDetailsVisibilityMode="VisibleWhenSelected" >
```

Und hier folgen die Definitionen der einzelnen Spalten:

```
            <DataGrid.Columns>
```

Eine Textspalte erzeugen, Bindung an den Member *Nachname* herstellen, die Kopfzeile mit „Nachname“ beschriften und eine Größenanpassung vornehmen:

```
                <DataGridTextColumn x:Name="nachnameColumn"
                  Binding="{Binding Path=Nachname}" Header="Nachname"
Width="SizeToHeader" />
```

Gleiches für den Vornamen:

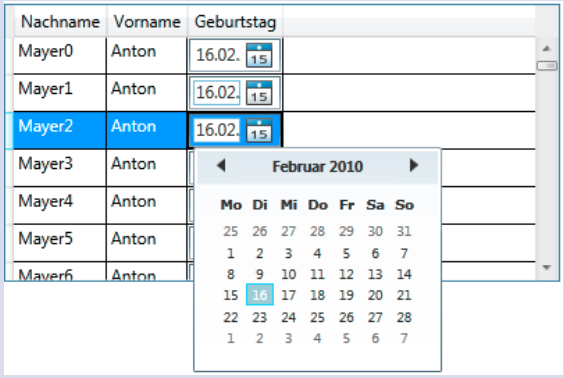
```
                <DataGridTextColumn x:Name="vornameColumn" Binding="{Binding
Path=Vorname}"
                                    Header="Vorname" Width="SizeToHeader" />
```

An dieser Stelle war der Assistent schon ganz schön „pfiffig“, statt einer einfachen Textspalte hat er bereits ein *DataTemplate* mit einem *DatePicker* für die Datumsanzeige erzeugt:

```
                <DataGridTemplateColumn x:Name="geburtstagColumn"
                                        Header="Geburtstag" Width="SizeToHeader">
                    <DataGridTemplateColumn.CellTemplate>
                        <DataTemplate>
                            <DatePicker SelectedDate="{Binding Path=Geburtstag}" />
                        </DataTemplate>
                    </DataGridTemplateColumn.CellTemplate>
                </DataGridTemplateColumn>
            </DataGrid.Columns>
        </DataGrid>
    </DockPanel>
```

Ergebnis

Das erzeugte *DataGrid*:



Anmerkung

Im obigen Beispiel ist die Datumsspalte noch zu schmal, weisen Sie einfach der Eigenschaft *Width* einen größeren Wert zu:

```
  <DataGridTemplateColumn x:Name="geburtstagColumn" Header="Geburtstag"
Width="100">
```

HINWEIS: Sie können die Spaltenbreite auch mit „*“ angeben, in diesem Fall verwendet die Spalte den restlichen verfügbaren Platz.

Wie Sie gesehen haben, steht Ihnen neben den Standard-Spaltentypen

- *DataGridTextColumn,*
- *DataGridCheckBoxColumn,*
- *DataGridComboBoxColumn,*
- *DataGridHyperlinkColumn*

auch die recht flexible *DataGridTemplateColumn* zur Verfügung. Welche Controls Sie hier einbinden (*Image*, *Chart*, *RichtextBox* etc.) bleibt Ihrer Phantasie überlassen.

Weitere Gestaltungsmöglichkeiten bieten sich mit dem Ein- und Ausblenden der Trennlinien, der Konfiguration der Spaltenköpfe per Template usw.

4.8.4 Zusatzinformationen in den Zeilen anzeigen

Nicht alle Informationen sollen immer gleich in einem Grid sichtbar sein, vielfach werden Detailfenster etc. eingeblendet, um nach der Auswahl eines Datensatzes weitere Informationen anzuzeigen. An dieser Stelle bietet das *DataGrid* mit dem *RowDetailsTemplate* ein recht interessantes Feature, versetzt Sie dieses Template doch in die Lage, unter bestimmten Umständen (*RowDetailsVisibilityMode*-Eigenschaft) zusätzliche Inhalte einzublenden.

Beispiel 4.37: Verwendung von *RowDetailsTemplate*

XAML

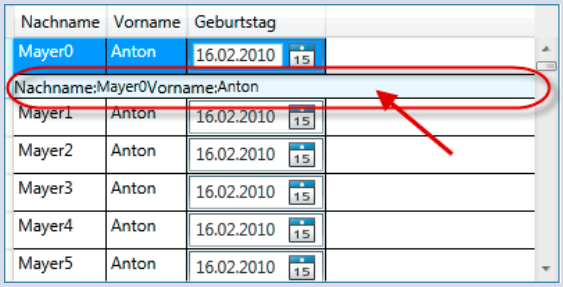
Zunächst müssen Sie bestimmen, wann die Details eingeblendet werden sollen:

```
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
            ItemsSource="{Binding}" RowDetailsVisibilityMode="VisibleWhenSelected" >
            <DataGrid.Columns>
...
            </DataGrid.Columns>
```

Nach der Spaltendefinition können Sie dann das *RowDetailsTemplate* einfügen und mit den gewünschten Informationen füllen:

```
            <DataGrid.RowDetailsTemplate>
                <DataTemplate>
                    <StackPanel Orientation="Horizontal" Background="AliceBlue">
                        <TextBlock>Nachname: </TextBlock>
                        <TextBlock Text="{Binding Path=Nachname}" FontSize="11" />
                        <TextBlock>  Vorname: </TextBlock>
                        <TextBlock Text="{Binding Path=Vorname}" FontSize="11" />
                    </StackPanel>
                </DataTemplate>
            </DataGrid.RowDetailsTemplate>
        </DataGrid>
```

Ergebnis



Mit *RowDetailsVisibilityMode* bestimmen Sie, wie die Zeilendetails angezeigt werden. Standardwert ist *Collapsed* (nicht sichtbar) alternativ steht *Visible* (immer sichtbar) oder *VisibleWhenSelected* zur Verfügung (nur die aktuelle Zeile).

4.8.5 Vom Betrachten zum Editieren

Auch wenn die umfangreichen Anzeigeoptionen das *DataGrid* für diverse Aufgaben prädestinieren, eine Hauptaufgabe dürfte in den meisten Fällen auch das Editieren der Inhalte sein.

Grundsätzlich entscheidet zunächst die übergreifende Eigenschaft *IsReadOnly* über die Fähigkeit, Inhalte des *DataGrid*s zu editieren oder nur zu betrachten. Gleiches gilt auch auf Spaltenebene, auch hier können Sie mit *IsReadOnly* darüber entscheiden, welche Spalten editierbar sind und welche nicht. Zusätzlich unterstützen Sie diverse Ereignisse vor, während und nach dem Editiervorgang (*BeginningEdit*, *PreparingCellForEdit*, *CellEditEnding* ...).

4.9 Praxisbeispiele

4.9.1 Collections in Hintergrundthreads füllen

In den vorhergehenden Beispielen haben wir es uns recht einfach gemacht. Eine Collection wurde erzeugt, gefüllt und angezeigt. Soweit so gut, aber was, wenn das Erzeugen der Collection etwas länger dauert? Ein kleines Beispielprogramm zeigt das Problem und natürlich auch die Lösung dafür. Dabei trennen wir aber zwischen der bisherigen Lösung und einer mit .NET 4.5 eingeführten Neuerung.

Oberfläche

Ein einfaches *Window* mit einigen Schaltflächen und einer *ListView* zur Anzeige der Daten:

```
<Window x:Class="Datenbindung.Laden_im_Hintergrund"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Laden_im_Hintergrund" Height="300" Width="530">
    <DockPanel>
        <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
            <Button Click="Button_Click_2">Laden Vordergrund-Thread</Button>
            <Button Click="Button_Click_3">Laden Hintergrund-Thread</Button>
            <Button Click="Button_Click_4">Laden Hintergrund Lösung</Button>
            <Button Click="Button_Click_1">Laden neu</Button>
        </StackPanel>
        <ListView Name="ListView1" IsSynchronizedWithCurrentItem="True"
                  ItemsSource="{Binding}" VirtualizingPanel.IsVirtualizing="True" />
    </DockPanel>
</Window>
```

Das Problem

Stellen Sie sich folgendes Szenario vor: Sie füllen eine Liste von *Schüler*-Objekten[4], leider dauert der Abruf jedes einzelnen Objekts etwas länger:

```
    public partial class Laden_im_Hintergrund : Window
    {
        public ObservableCollection<Schüler> _klasse;

        public Laden_im_Hintergrund()
        {
            InitializeComponent();
            _klasse = new ObservableCollection<Schüler>();
            this.DataContext = _klasse;
        }

        private void Datenabrufen()
        {
            _klasse.Clear();
            for (int i = 0; i < 100; i++)
            {
```

Hier simulieren wir eine Zeitverzögerung, z. B. eine langsame Datenverbindung:

```
        Thread.Sleep(100);  _klasse.Add(new Schüler { Nachname = "Mayer" + i.ToString(),
                    Vorname = "Alexander", Geburtstag = new DateTime(2001, 11, 7) });
    }
}

private void Button_Click_2(object sender, RoutedEventArgs e)
{
    Datenabrufen();
}
```

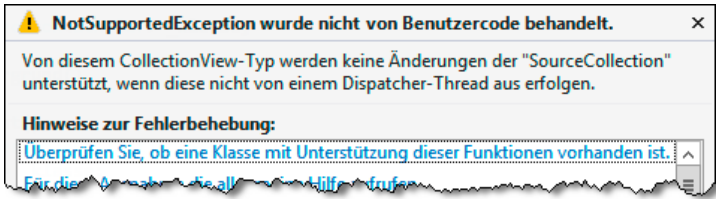
Starten Sie die Anwendung, werden Sie nach einem Klick auf die Schaltfläche feststellen, dass Ihre Anwendung „einfriert“. Diese Lösung wollen Sie dem Endanwender sicher nicht zumuten. Was liegt also näher, als diese Aufgabe in einen Hintergrundthread zu verlagern.

[4] Definition siehe Abschnitt 4.2.3.

Gesagt, getan, wir kapseln obigen Methodenaufruf in einem extra Thread:

```
        private void Button_Click_3(object sender, RoutedEventArgs e)
        {
            Task.Factory.StartNew(Datenabrufen);
        }
```

Doch nach einem Start der Anwendung werden Sie schnell wieder auf den Boden zurückgeholt:



Sie können auf die Collection nicht per Hintergrundthread zugreifen. Das ist erstmal ein Show-Stopper. Doch es gibt zwei Lösungen:

- Laden einer extra Collection im Hintergrund und kopieren dieser Collection in den Vordergrund. Nachfolgend Abgleich mit der gebundenen Collection.
- Laden der Daten im Hintergrund, Einfügen der einzelnen Einträge durch jeweiligen Wechsel in den Vordergrundthread.

Die zweite Lösung ist mit häufigen Threadwechseln verbunden, wir sehen uns also die erste Lösung näher an.

Lösung (bis .NET 4.0)

Wir lagern das Laden der Daten in eine Funktion aus, die eine komplette Liste zurückgibt:

```
        private ObservableCollection<Schüler> Datenabrufen_alteLoesung()
        {
            ObservableCollection<Schüler> _threadklasse = new
ObservableCollection<Schüler>();
            for (int i = 0; i < 100; i++)
            {
                Thread.Sleep(100);  // simuliert Laden aus der Quelle
                _threadklasse.Add(new Schüler { Nachname = "Mayer" + i.ToString(),
                 Vorname = "Alexander", Geburtstag = new DateTime(2001, 11, 7) });
            }
            return _threadklasse;
        }
```

Unser Aufruf:

```
        private void Button_Click_4(object sender, RoutedEventArgs e)
        {
```

Anzeige, dass der Nutzer warten soll:

```
            this.Cursor = Cursors.Wait;
```

In einem extra Thread werden die Daten geladen:

```
            Task.Factory.StartNew <ObservableCollection<Schüler>>(
                                    Datenabrufen_alteLoesung).ContinueWith(t =>
            {
```

Ist dies erfolgt, kopieren wir diese in die gebundene Liste:

```
                _klasse.Clear();
                foreach (var s in t.Result)
                    _klasse.Add(s);
```

Und blenden die Sanduhr aus:

```
                this.Cursor = null;
            }, TaskScheduler.FromCurrentSynchronizationContext());
        }
```

Test

Nach dem Start wird die „Sanduhr“ angezeigt, nach einigen Sekunden ist die Liste gefüllt. Wie Sie sehen, muss sich der Nutzer auch hier gedulden, die Oberfläche bleibt in dieser Zeit aber voll bedienbar.

Lösung (ab .NET 4.5)

.NET 4.5 bietet die Möglichkeit, Collections für die gleichzeitige Bearbeitung in Threads quasi anzumelden. Nutzen Sie dazu einen Aufruf der Methode *EnableCollectionSynchronization*.

```
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
```

Anmelden der Collection, wir übergeben noch das aktuelle Window-Objekt als Sperrobjekt[5]:

```
            BindingOperations.EnableCollectionSynchronization(_klasse, this);
```

Wir rufen die Daten per extra Thread ab:

```
            Task.Factory.StartNew(Datenabrufen);
        }

        private void Datenabrufen()
        {
            _klasse.Clear();
            for (int i = 0; i < 100; i++)
            {
                Thread.Sleep(100);  // simuliert Laden aus der Quelle
                _klasse.Add(new Schüler { Nachname = "Mayer" + i.ToString(),
                 Vorname = "Alexander", Geburtstag = new DateTime(2001, 11, 7) });
            }
        }
```

[5] Es tut auch jede andere Objekt-Instanz.

Test

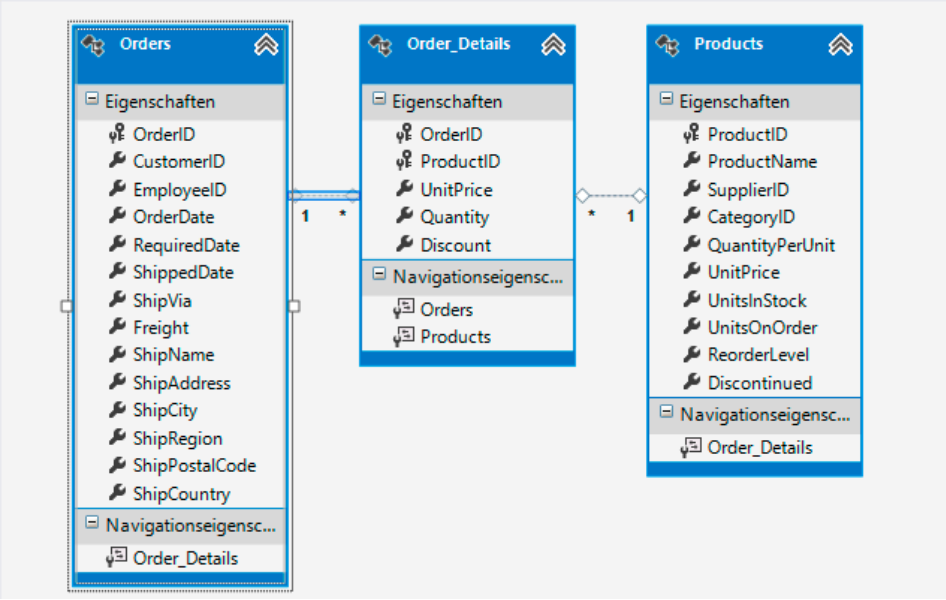
Nach dem Start werden Sie feststellen, dass die Oberfläche beweglich bleibt, und dass die Daten „tröpfchenweise“ in die Liste geladen werden, Sie können beim Füllen quasi zusehen. Eine einfache und recht elegante Lösung für das Einlesen größerer Datenmengen.

4.9.2 Drag & Drop-Bindung bei 1:n-Beziehungen

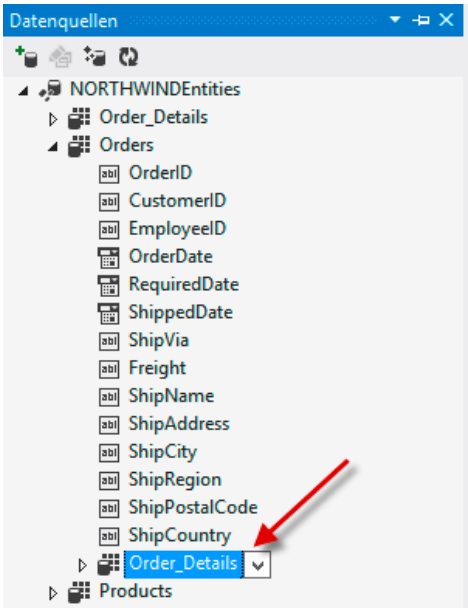
Grundlage unseres kleinen Beispiels soll in diesem Fall ein Entity Data Model sein, bei dem wir die Beziehung zwischen den Tabellen *Orders* und *Order_Details* mit zwei *DataGrids* visualisieren wollen. Wir machen es uns jetzt jedoch einfach und verwenden für den Entwurf das Datenquellen-Fenster, so kommen wir ohne eine einzige Zeile eigenen Quellcodes aus.

Oberfläche

Erstellen Sie zunächst ein neues WPF-Projekt, in das Sie die Datenbank *Northwind.mdf* einfügen. Bei der Frage nach dem zu erzeugenden Datenbankmodell wählen Sie *Entity Data Model*, um ein entsprechendes Modell zu erzeugen. Bei der Auswahl der Tabellen können Sie sich auf die Tabellen *Orders*, *Order_Details* und *Products* beschränken.



Nachfolgend wenden Sie sich dem Entwurf der Oberfläche zu. Ziehen Sie aus dem Datenquellen-Fenster den Eintrag *Orders* direkt in die Freifläche des WPF-Formulars. Automatisch wird jetzt ein *DataGrid* für die Anzeige der Bestellungen erzeugt (siehe Laufzeitansicht). Für die Anzeige der Bestellungsdetails ziehen Sie einfach den untergeordneten Knoten *Order_Details* ebenfalls in das WPF-Formular.



HINWEIS: Verwechseln Sie diesen Knoten nicht mit dem Knoten *Order_Details*, der unabhängig im DataContext definiert ist!

Damit ist unser Programm bereits „komplett“, wir wollen jedoch noch kurz einen Blick auf den generierten Quellcode werfen, um die Funktionsweise besser zu verstehen.

Quellcode (XAML)

Bei der Oberflächendefinition interessieren wir uns nur für die beiden *DataGrid*s:

```
<Window x:Class="EDM_WPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Master/Detail-Beziehung" Height="456" Width="630" mc:Ignorable="d"
        xmlns:d=http://schemas.microsoft.com/expression/blend/2008
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

Einbinden des aktuellen Namespace und Zuweisen einer Ereignismethode in der die Daten geladen werden:

```
      xmlns:my="clr-namespace:EDM_WPF" Loaded="Window_Loaded">
    <Window.Resources>
```

Für jedes der beiden DataGrids wird eine *CollectionViewSource* erzeugt:

```
        <CollectionViewSource x:Key="ordersViewSource"
                              d:DesignSource="{d:DesignInstance my:Orders,
CreateList=True}" />
        <CollectionViewSource x:Key="ordersOrder_DetailsViewSource"
```

```
                            Source="{Binding Path=Order_Details,
                            Source={StaticResource ordersViewSource}}" />
    </Window.Resources>
```

Den Datenkontext zentral zuweisen, beide *DataGrid*-Objekte befinden sich im *DockPanel* und erben deshalb diese Eigenschaft (Achtung: hier werden die Details zugewiesen):

```
    <DockPanel DataContext="{StaticResource ordersOrder_DetailsViewSource}">
```

Das Master-*DataGrid* an die entsprechende *CollectionViewSource* binden:

```
        <DataGrid DockPanel.Dock="Top" AutoGenerateColumns="False"
            EnableRowVirtualization="True"
            Height="188" ItemsSource="{Binding Source={StaticResource ordersViewSource}}"
            Name="ordersDataGrid" RowDetailsVisibilityMode="VisibleWhenSelected">
...
        </DataGrid>
```

Das Detail-*DataGrid* nutzt die im *DockPanel* definierte Bindung:

```
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
                ItemsSource="{Binding}"
        Name="order_DetailsDataGrid" RowDetailsVisibilityMode="VisibleWhenSelected" >
...
        </DataGrid>
    </DockPanel>
</Window>
```

HINWEIS: Obiger Code ist sicher nicht der Weisheit letzter Schluss, Sie räumen hier besser etwas auf und löschen die *DataContext*-Zuweisung beim *DockPanel* und ändern die Definition des Detail-DataGrids wie folgt:

```
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
                ItemsSource="{Binding Source={StaticResource
ordersOrder_DetailsViewSource}}"
                Name="order_DetailsDataGrid"
RowDetailsVisibilityMode="VisibleWhenSelected" >
```

Quellcode (C#)

Auch der automatisch erzeugte Quellcode scheint noch verbesserungswürdig:

```
    public partial class MainWindow : Window
    {
...
```

Mit dem Laden des Formulars werden die Daten geladen:

```
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            EDM_WPF.NORTHWINDEntities nORTHWINDEntities = new EDM_WPF.NORTHWINDEntities();
```

Die *CollectionViewSource*-Instanz (Master) aus der Oberfläche abrufen:

```
            System.Windows.Data.CollectionViewSource ordersViewSource =
              ((System.Windows.Data.CollectionViewSource)(this.FindResource
("ordersViewSource")));
```

Hier wird zunächst die Datenabfrage definiert (siehe Hilfsmethode):

```
            System.Data.Objects.ObjectQuery<EDM_WPF.Orders> ordersQuery =
               this.GetOrdersQuery(nORTHWINDEntities);
```

Abrufen der Daten und zuweisen an die *CollectionViewSource*:

```
            ordersViewSource.Source =
                  ordersQuery.Execute(System.Data.Objects.MergeOption.AppendOnly);
        }
```

Folgende Hilfsmethode definiert die Abfrage für die Master- **und** die Detail-Tabelle:

```
        private System.Data.Objects.ObjectQuery<Orders> GetOrdersQuery(
                                                          NORTHWINDEntities
nORTHWINDEntities)
        {
            System.Data.Objects.ObjectQuery<EDM_WPF.Orders> ordersQuery =
                     nORTHWINDEntities.Orders;
```

Hier schließen wir die Detaildaten in die Abfrage ein:

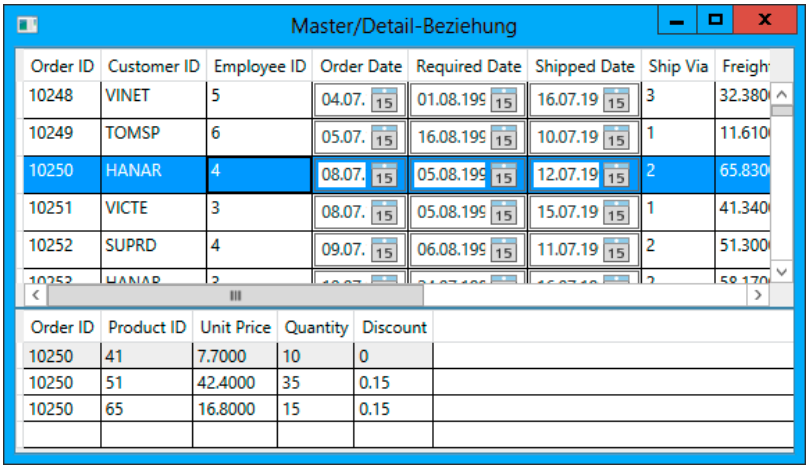
```
            ordersQuery = ordersQuery.Include("Order_Details");
            return ordersQuery;
        }
    }
```

Sie können den Quellcode durchaus noch etwas vereinfachen: Löschen Sie die Methode *GetOrdersQuery* und ändern Sie das *Window_Loaded*-Ereignis wie folgt:

```
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            EDM_WPF.NORTHWINDEntities nORTHWINDEntities = new
EDM_WPF.NORTHWINDEntities();
            System.Windows.Data.CollectionViewSource ordersViewSource =
              ((System.Windows.Data.CollectionViewSource)
                     (this.FindResource("ordersViewSource")));
            ordersViewSource.Source = nORTHWINDEntities.Orders.Include("Order_Details");
        }
```

Test

Ein Test wird Sie von der Funktionsweise überzeugen:



Wie Sie sehen, können Sie bereits mit dem per Drag & Drop erzeugten Programm gut leben, einige Änderungen machen jedoch den Quellcode übersichtlicher und besser verständlich.

5 Druckausgabe mit WPF

Mit der Einführung von WPF wurde auch eine grundsätzlich andere Vorgehensweise bei der Druckausgabe von Dokumenten gewählt, die bei mehr als einer Druckausgabeseite schnell an Komplexität gewinnt. Erschwerend kommt hinzu, dass für Druckausgabe und Druckvorschau[1] keine einheitliche Lösung gewählt wurde, was es dem Programmierer auch nicht einfacher macht.

Aus diesem Grund wollen wir Sie zunächst mit der „Trivial-Lösung" für die Ausgabe einzelner Seiten sowie die Auswahl des Zieldruckers vertraut machen, bevor wir uns den komplexeren Themen wie

- Druckvorschau,
- mehrseitige Dokumente
- und Druckerauswahl/-konfiguration

zuwenden wollen. Doch bevor es so weit ist, wollen wir Sie in einer Kurzübersicht mit den Konzepten der WPF-Druckausgabe vertraut machen.

5.1 Grundlagen

Eine Einführung zum Drucken in WPF wäre nicht komplett, wenn wir nicht kurz auf das Thema XPS-Dokumente eingehen würden.

5.1.1 XPS-Dokumente

Mit XPS (*XML Paper Specification*) versucht Microsoft einen Pendant zum derzeit weit verbreiteten Adobe PDF-Format zu etablieren. Im Grunde handelt es sich um eine geräteunabhängige vektororientierte Seitenbeschreibungssprache, die einen ungehinderten Informationsfluss aus der Anwendung bis zum finalen Ausgabegerät sicherstellen soll. Dafür findet

[1] Eine direkte Druckvorschau-Komponente werden Sie auch nicht finden, wir nehmen dafür den *DocumentViewer*.

sich nicht zuletzt auch ab Windows Vista ein entsprechender Druckertreiber, dessen Ausgaben in einer Datei landen, die Sie wiederum mit dem entsprechenden Betrachter anzeigen können.

Über die Vor- und Nachteile zum bereits etablierten PDF lässt sicher sicher streiten, was aber nichts an der Tatsache ändert, dass XPS nun mal der zentrale Weg für die Druckausgaben unserer WPF-Anwendungen ist. Allerdings hat auch Microsoft schon richtig eingeschätzt, dass eine relevante XPS-Unterstützung für das Format von kaum einem Druckerhersteller zu erwarten ist, und so wird innerhalb der Druckausgabe eine Umwandlung der XPS-Daten in die bekannten GDI- bzw. PDL-Daten vorgenommen, um auch mit den derzeit am Markt befindlichen Geräten sinnvoll arbeiten zu können.

Um all diese Hintergründe müssen Sie sich als Programmierer jedoch nicht selbst kümmern, das für Sie interessante *XpsDocumentWriter*-Objekt nimmt, je nach Endgerät, die erforderliche XPS zu GDI-Umwandlung automatisch vor.

Für die Interaktion mit dem XPS-Ausgabesystem stehen Ihnen in C# die beiden Namespaces

- *System.Printing*
- und *System.Windows.Xps*

mit den entsprechenden Klassen zur Verfügung. Zusätzlich findet sich ganz unscheinbar im Namespace *System.Windows.Controls* auch eine *PrintDialog*-Komponente, die Sie in keinem Fall mit der entsprechenden Windows Forms-Komponente verwechseln sollten. Die *PrintDialog*-Komponente bietet neben der vermuteten Dialog-Funktionalität vor allem einen einfachen Zugriff auf die installierten Drucker, um zum Beispiel Controls (und das kann auch, wie von WPF gewohnt, eine geschachtelte Anordnung sein) oder auch ganze XPS-Dokumente zu drucken. Mehr dazu finden Sie im Abschnitt 5.2.

5.1.2 System.Printing

Dieser Namespace bietet mit den enthaltenen Klassen (Auszug)

- *PrintServer*
  (repräsentiert den aktuellen Druckserver, d. h. den Computer)
- *PrintQueue*
  (repräsentiert den aktuellen Drucker und dessen Druckerwarteschlange)
- *PrintSystemJobInfo*
  (ein Druckjob und dessen Status)
- *PrintTicket*
  (die Konfiguration des Druckauftrags, wie Seitenformat, Seitendrehung etc.)

die Grundlage für die Arbeit mit der Druckausgabe in WPF.

HINWEIS: Bevor Sie auf diesen Namespace zugreifen können, müssen Sie unter Verweise noch die Assembly *System.Printing.dll* nachträglich einbinden. Je nach Funktionsumfang kann es auch nötig sein, zusätzlich die Assembly *ReachFramework.dll* einzubinden.

HINWEIS: Weitere Informationen zu den o.g. Klassen finden Sie im Abschnitt 5.4.

5.1.3 System.Windows.Xps

Wie schon erwähnt, ist XPS der Dreh- und Angelpunkt der WPF-Druckausgabe. Über den Namespace *System.Windows.Xps* stehen Ihnen die nötigen Schnittstellen-Klassen *VisualsToXpsDocument* und *XpsDocumentWriter* zur Verfügung.

Zusätzlich finden sich weitere Namespaces wie *System.IO.Packaging* und *System.Windows.Xps.Packaging*, die im Zusammenhang mit der Ausgabe von XPS-Dateien eine Rollen spielen. Hier auf alle Einzelheiten einzugehen, würde den Rahmen des Kapitels sprengen.

HINWEIS: Bevor Sie auf diesen Namespace komplett zugreifen können, müssen Sie Verweise auf die Assemblies *System.Printing.dll* und *ReachFramework.dll* einbinden.

5.2 Einfache Druckausgaben mit dem PrintDialog

Hoffentlich haben wir Sie mit den Kurzausführungen zu XPS nicht ganz verschreckt, nicht in jedem Fall müssen Sie sich mit der Gesamtkomplexität der WPF-Druckausgabe herumschlagen, es geht teilweise auch ganz einfach, wie es auch das folgende Beispiel zeigt:

Beispiel 5.1: Verwendung von *PrintDialog*

C#

```
        private void Button2_Click(object sender, RoutedEventArgs e)
        {
```

Instanz erstellen:

```
            PrintDialog dlg = new PrintDialog();
```

Dialog anzeigen:

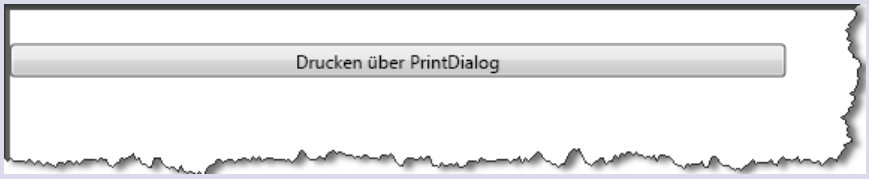
```
            if (dlg.ShowDialog() == true)
            {
```

Wird auf die Ok-Taste geklickt, drucken wir ein Control aus:

```
                dlg.PrintVisual(Button2, "Erster Test");
            }
        }
```

Ergebnis

Das Ergebnis dürfte etwa so aussehen:



Der Druckdialog selbst wird Ihnen sicher von den Windows Forms her noch bekannt sein, die Funktionalität ist zunächst gleich. Doch nach der Auswahl von Drucker, Seitenbereich etc. geht es hier erst richtig los. Über die Methode *PrintVisual* besteht die Möglichkeit, XAML-Elemente Ihres Formulars (oder auch gleich das ganze Formular) direkt an den Drucker zu senden. Da diese ohnehin vektorbasiert sind, ist auch die Druckqualität im Vergleich zu Windows Forms wesentlich besser.

HINWEIS: Wer jetzt befürchtet, immer den Dialog anzeigen zu müssen, liegt falsch, lassen Sie den Aufruf der Methode *ShowDialog* weg, wird automatisch der Standard-Drucker verwendet.

Doch Vorsicht – der oben gezeigte Aufruf von *PrintVisual* hat einige Einschränkungen, die Sie kennen sollten, bevor Sie darauf basierend Ihre Programme umschreiben:

1. Es wird immer nur **eine** Seite bedruckt, ist der zu druckende Content größer, wird er abgeschnitten.
2. Der Ausdruck wird immer an der linken oberen Blattecke ausgerichtet. Da viele Drucker jedoch einen Seitenoffset besitzen, wird in diesem Fall meist etwas am oberen und linken Rand abgeschnitten.

Als Lösung für 1. bietet sich die komplexere *PrintDocument*-Methode an, auf die wir noch zurückkommen werden, Problem 2 können Sie umgehen, wenn Sie dem zu druckenden Control einen entsprechenden *Margin* verpassen, der automatisch mit den Seitenrändern verrechnet wird.

Wo wir schon bei Seitenrändern sind: Über den *PrintDialog* können Sie auch einige wichtige Informationen zum aktuellen Ausgabemedium[2] in Erfahrung bringen:

[2] tolle Umschreibung für „Blatt“

- Seitenhöhe und Breite (PrintTicket.*PageMediaSize.Height*, *PrintTicket.PageMediaSize.Width*)
- Druckbereichshöhe und -breite (*PrintableAreaHeight*, *PrintableAreaWidth*)
- Randloser Druck möglich (*PrintTicket.PageBorderless*)
- Seitenausrichtung (*PrintTicket.PageOrientation*)

HINWEIS: Achtung: Ist das Blatt gedreht, hat dies keinen Einfluss auf die Seiten- bzw. Druckbereichsabmessungen, d. h., die Papierbreite ist beim Querformat mit *PrintableAreaHeight* zu bestimmen.

Natürlich gibt es noch dutzende weitere Eigenschaften, in den folgenden Abschnitten kommen wir noch darauf zurück.

Doch was ist eigentlich mit reiner Textausgabe? Hier sollten Sie den Namen der *PrintVisual*-Methode nicht zu genau nehmen, mit dieser Methode können Sie natürlich auch Text bzw. ein *TextBlock*-Objekt ausgeben und das auf wesentlich komfortablere Art als bei den Windows Forms. Der *TextBlock* darf natürlich auch über Formatierungen etc. verfügen (siehe Abschnitt 5.2).

Beispiel 5.2: Textausgabe auf dem Drucker

C#

```
        private void Button2_Click(object sender, RoutedEventArgs e)
        {
            PrintDialog dlg = new PrintDialog();
            if (dlg.ShowDialog() == true)
            {
```

Neue *TextBlock*-Instanz unabhängig vom Formular erzeugen:

```
                TextBlock txt = new TextBlock();
```

Ränder festlegen, so umgehen wir das Problem mit dem Seitenoffset:

```
                txt.Margin = new Thickness(15);
```

Hier der eigentliche Ausgabetext:

```
                txt.Text = "Hier folgt ein selten belangloser Text, den Sie besser
 nicht lesen sollten. Aber so wird Ihnen schnell klar, wie einfach die Textausgabe
 ist.";
```

Schriftgröße und Schriftart bestimmen:

```
                txt.FontSize = 25;
                txt.FontFamily = new FontFamily("Arial");
```

Textumbruch aktivieren:

```
                txt.TextWrapping = TextWrapping.Wrap;
```

Und jetzt müssen wir uns um die Größe des Controls kümmern (Layout-Aktualisierung), sonst ist es nicht zu sehen:

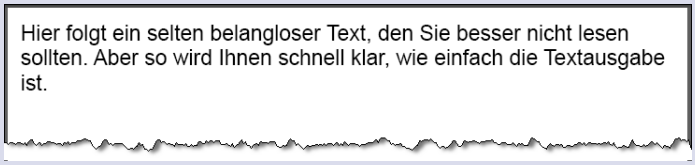
```
                txt.Measure(new Size(dlg.PrintableAreaWidth,
dlg.PrintableAreaHeight));
                txt.Arrange(new Rect(0, 0, txt.DesiredSize.Width,
txt.DesiredSize.Height));
```

Last, but no least, die Druckausgabe:

```
                dlg.PrintVisual(txt, "Erster Test");
            }
        }
```

Ergebnis

Damit kann man doch schon ganz zufrieden sein:



Damit wollen wir uns aus der „Programmierer-Kuschelecke“ verabschieden und uns in die „Wildnis“ von Druckvorschau und mehrseitiger Ausgabe wagen.

5.3 Mehrseitige Druckvorschau-Funktion

Puh..., ganz schön viele Wünsche auf einmal und WPF lässt uns an dieser Stelle doch etwas im Regen stehen. Ein Blick in den Werkzeugkasten verheißt nichts Gutes, weder für das eine noch für das andere findet sich **die** Standardlösung.

Wer den Blick in die Hilfe bzw. ins Internet wagt, findet sicher auch recht schnell die Standardlösung mit einer abgeleitetem *DocumentPaginator*-Klasse, deren Methoden Sie implementieren müssen. Doch so ganz befriedigend ist diese Lösung nicht, von der Übersicht ganz zu schweigen.

Aus diesem Grund haben sich die Autoren diverse Einzellösungen im Internet angesehen und aus all diesen Hinweisen/Lösungsvorschlägen etc. zwei halbwegs nutzbare Lösungen realisiert, die sowohl unsere Wünsche nach mehrseitigen Dokumenten erfüllen als auch eine Druckvorschau bieten.

Doch wie immer findet sich auch hier ein Haar in der Suppe: Sie müssen sich entscheiden, ob Sie mit einem flexiblen Dokument arbeiten, das über Fließtext mit eingebetteten Grafiken etc. verfügt (ein typisches Flow-Dokument) oder ob Sie die einzelnen Elemente Layout-orientiert bzw. absolut anordnen wollen (Fix-Dokument). Für beides finden Sie im Folgenden die Lösung.

5.3.1 Fix-Dokumente

Sehen wir uns zunächst die layout- und seitenorientierte Ausgabe von Dokumenten an. Wie auch bei den noch zu betrachtenden Flow-Dokumenten wollen wir die erforderliche Funktionalität in einer eigenen Klasse kapseln, um uns ganz auf die reine Ausgabe konzentrieren zu können. Ziel war es, mit so wenig Code wie möglich eine Druckausgabe und eine Druckvorschaufunktion zu implementieren.

Einsatzbeispiel

Bevor wir uns in die Details vertiefen, wollen wir uns ansehen, wie wir die spätere Klasse *FixedPrintManager* einsetzen können und was wir alles zu Papier bringen können. Sie werden sehen, dass viele Konzepte bereits in den vorhergehenden Kapiteln erläutert wurden.

Beispiel 5.3: Verwendung der Klasse *FixedPrintManager*

C#

Binden Sie zunächst die beiden Assemblies *System.Printing.dll* und *ReachFramework.dll* ein. Erstellen Sie nachfolgend ein Formular, in das Sie eine Schaltfläche und einen *DocumentViewer*[3] einbinden.

Zunächst die nötigen Namespaces:

```
...
using System.IO;
using System.Windows.Xps;
using System.Windows.Xps.Packaging;
using System.Windows.Markup;
using System.IO.Packaging;
...
```

Nach dem Klick auf die Schaltfläche setzen die hektischen Aktivitäten ein:

```
        private void Button1_Click(object sender, RoutedEventArgs e)
        {
```

Als Erstes definieren wir einige Objekte, die wir später zu Papier bringen wollen (nähere Informationen dazu finden Sie in den vorhergehenden Kapiteln.):

```
            TextBlock txt;
            Image img;
            StackPanel panel;
```

Wir erstellen eine Instanz unserer *FixedPrintManager*-Klasse:

```
            FixedPrintManager pm = new FixedPrintManager(
                                   borders: new Thickness(15), ShowPrintDialog:
true);
```

Parameter sind die Breite der Seitenränder und die Option, ob ein Druckerauswahldialog angezeigt werden soll.

[3] Sie ahnen es sicher schon, der *DocumentViewer* wird unsere Druckvorschau.

Und damit sind wir auch schon beim Darstellen der ersten Seite:

Zentrales Element ist ein *StackPanel*, wie Sie es auch von den WPF-Formularen kennen:

```
panel = new StackPanel();
```

In das *StackPanel* fügen wir einen *TextBlock* ein. Die entsprechende Instanz erzeugen wir allerdings per *NewTextBlock*-Methode unserer *FixedPrintManager*-Instanz:

```
txt = pm.NewTextBlock("Times New Roman", 40);
```

Der Vorteil: Wir müssen uns nicht um das recht umständliche Konfigurieren des *TextBlock*s kümmern, Defaultparameter ermöglichen das einfache und komfortable Setzen der Eigenschaften.

Noch etwas Text definieren und den *TextBlock* dem *StackPanel* hinzufügen:

```
          txt.Text =
  "Hier steht zum Beispiel jede Menge Text. Hier steht zum Beispiel
               jede Menge Text.Hier steht zum Beispiel jede Menge Text. .......... ";
          panel.Children.Add(txt);
```

Ein weiterer *TextBlock:*

```
          txt = pm.NewTextBlock(text: "Hier steht zum Beispiel jede Menge Text.
                Hier steht zum Beispiel jede Menge Text.Hier steht zum ..........");
          txt.Te xtAlignment = TextAlignment.Justify;
          txt.Margin = new Thickness(0, 10, 0, 20);
          panel.Children.Add(txt);
```

Ein *Image* in das *StackPanel* einfügen:

```
          img = new Image();
```

Im Gegensatz zu den beiden vorhergehenden Beispielen müssen wir im Fall des *Image*-Controls selbst für die Größenanpassung des *Image* sorgen:

```
          img.Width = pm.PageSize.Width - pm.Borders.Left - pm.Borders.Right;
          img.Stretch = Stretch.Uniform;
          img.Source = new BitmapImage(new Uri("pack://application:,,,/Desert.jpg"));
          panel.Children.Add(img);
```

Wir schließen die erste „Druck“-Seite ab, indem wir das Stackpanel an die *NewPage*-Methode übergeben:

```
          pm.NewPage(panel);
```

Hier erzeugen wir eine reine Textseite:

```
          txt = pm.NewTextBlock(Fontsize: 22);
          txt.Text = "Hier ist die zweite Seite mit Fließtext. Hier ist die zweite
                     Seite mit Fließtext. Hier ist die zweite Seite ....... ";
```

Abschließen der zweiten Seite:

```
          pm.NewPage(txt);
```

Ein ganz triviales Beispiel für die Ausgabe einzelner Controls:

```
pm.NewPage(new Calendar());
```

Natürlich können Sie bei der Druckausgabe auch alle Bildtransformationen nutzen, die Sie bereits kennengelernt haben.

Als Grundlage dient uns in diesem Fall ein *Canvas,* der auch das absolute Positionieren des Controls zulässt:

```
Canvas cv = new Canvas();
img = new Image();
img.Width = pm.PageSize.Width - pm.Borders.Left - pm.Borders.Right;
img.Stretch = Stretch.Uniform;
img.Source = new BitmapImage(new Uri("pack://application:,,,/Desert.jpg"));
img.SetValue(Canvas.TopProperty, 100.0);
img.SetValue(Canvas.LeftProperty, 1.0);
```

Die Transformation anwenden:

```
img.LayoutTransform = new RotateTransform(45);
cv.Children.Add(img);
pm.NewPage(cv);
```

Abschließend möchten wir Ihnen an einer weiteren „Druckseite“ demonstrieren, wie Sie auch XAML-Code aus den Ressourcen des aktuellen Formulars für die Druckausgabe verwenden können:

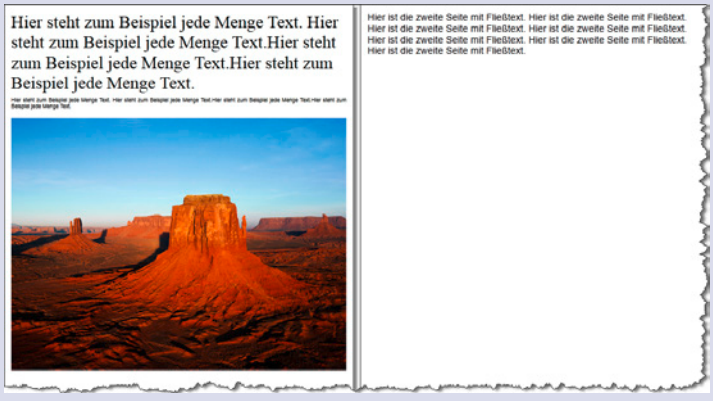
```
cv = (Canvas)Resources["ResourceData"];
cv.Measure(pm.PageSize);
cv.Arrange(new Rect(0, 0, cv.DesiredSize.Width, cv.DesiredSize.Height));
pm.NewPage(cv);
```

So, damit ist das Dokument erstellt, Sie haben jetzt zwei Möglichkeiten: Entweder Sie zeigen das Dokument in einem *DocumentViewer* an oder Sie gehen sofort zum Druck über:

```
documentViewer1.Document = pm.Document;
pm.Print();
}
```

Ergebnis

Das Ergebnis unserer Bemühungen zeigen die folgenden Abbildungen:



Vielleicht sind Sie auch der Meinung, dass der Aufwand für das Erstellen des Dokuments nicht allzu hoch war. Die Verwendung der Textblöcke hatte sogar den Vorteil, dass wir uns um Zeilenumbrüche keinen Kopf machen mussten. Die restlichen Ausgaben waren weitgehend mit der Formulardarstellung identisch, neue Konzepte müssen Sie bei dieser Form der Druckausgabe nicht erlernen.

Die Klasse FixedPrintManager

Doch nun wollen wir einen Blick auf die verwendete Klasse *FixedPrintManager* werfen:

```
public class FixedPrintManager
{
```

Einige interne Variablen:

```
    private FixedDocument _PrintDocument;
    private Size _PageSize;
    private Thickness _Borders;
    private PrintDialog _dlg;
```

Die *Borders*-Eigenschaft (Abfrage der Seitenränder):

```
    public Thickness Borders
    { get { return _Borders; }  }
```

Sie *PageSize*-Eigenschaft (Abfrage der Seitengröße, diese wird im Konstruktor gesetzt):

```
    public Size PageSize
    { get { return _PageSize; } }
```

Die Rückgabe des Dokuments für die Druckvorschau:

```
    public FixedDocument Document
    { get { return _PrintDocument; } }
```

Der Konstruktor unserer Klasse:

```
        public FixedPrintManager(Thickness borders, bool ShowPrintDialog = false)
        {
```

Ein neues *FixedDocument* erstellen:

```
            _PrintDocument = new FixedDocument();
```

Intern nutzen wir den *PrintDialog* für die Druckausgabe und die Bestimmung von Drucker und Seitenrändern:

```
            _dlg = new PrintDialog();
            if (ShowPrintDialog) _dlg.ShowDialog();
            _PageSize = new Size(_dlg.PrintableAreaWidth, _dlg.PrintableAreaHeight);
            _PrintDocument.DocumentPaginator.PageSize = _PageSize;
            _Borders = borders;
        }
```

Das Starten der Druckausgabe erfolgt mit der folgenden Methode:

```
        public void Print(string title = "Mein Druckauftrag")
        {
            _dlg.PrintDocument(_PrintDocument.DocumentPaginator, title);
        }
```

Zum Erstellen einer neuen Seite nutzen Sie folgende Methode:

```
        public void NewPage(UIElement content)
        {
```

Erzeugen einer neuen *FixedPage*, diese wird im Folgenden in das interne *FixedDocument* eingefügt:

```
            FixedPage _page = new FixedPage();
```

Seitengröße und -ränder bestimmen:

```
            _page.Width = _PrintDocument.DocumentPaginator.PageSize.Width;
            _page.Height = _PrintDocument.DocumentPaginator.PageSize.Height;
            _page.Margin = _Borders;
```

Den übergeben Seiteninhalt einfügen:

```
            _page.Children.Add(content);
```

Hier wir es etwas komplizierter, da ein direkter Zugriff auf die *AddChild*-Methode nicht möglich ist:

```
            PageContent _pageContent = new PageContent();
            ((IAddChild)_pageContent).AddChild(_page);
```

Anhängen an das *FixedDocument*:

```
            _PrintDocument.Pages.Add(_pageContent);
        }
```

Last but not least, noch unsere Methode zum Erstellen von Textblöcken. Diese macht ausgiebig von der Verwendung optionaler Parameter Gebrauch und nimmt uns das lästige Parametrieren des *TextBlock*s ab:

```
        public TextBlock NewTextBlock(string Fontname = "Arial", int Fontsize = 12,
                                      string text = "")
        {
            TextBlock txt = new TextBlock();
            txt.Width = _PageSize.Width - _Borders.Left - _Borders.Right;
            txt.FontFamily = new FontFamily(Fontname);
            txt.FontSize = Fontsize;
            txt.Text = text;
            txt.TextWrapping = TextWrapping.WrapWithOverflow;
            return txt;
        }
    }
```

Damit haben Sie bereits ein recht umfangreiches Grundgerüst für eigene Erweiterungen. Insbesondere die Ausgabe von Linien-Zeichnungen etc. ist sicher noch verbesserungswürdig, aber Sie wollen ja auch noch etwas zu tun haben.

5.3.2 Flow-Dokumente

Nachdem wir uns mit der seitenorientierten Methode, d. h. den *FixedDocuments* beschäftigt haben, wollen wir und mit den flexibleren Flow-Dokumenten befassen (siehe auch Abschnitt 2.18). Deren Vorteil liegt in der Beschreibung von formatierten Fließtexten, die wiederum aus Absätzen (*Paragraph*), eingefügten Controls (*BlockUIContainer*), Auflistungen (*List*), *Section*s und Tabellen (*Table*) bestehen können. Um Seitenränder, Seitengrößen und -ausrichtungen müssen Sie sich zum Zeitpunkt der Dokumenterstellung keinen Kopf machen, dies erfolgt automatisch bei der finalen Ausgabe auf dem jeweiligen Ausgabegerät.

Einführungsbeispiel

Ein kleines Beispiel soll Sie von den Vorzügen unserer FlowPrintManager-Klasse überzeugen, die wie auch die *FixedDocument*-Klasse über eine *Document*-Eigenschaft (für die Druckvorschau) und eine *Print*-Methode verfügt.

Beispiel 5.4: Verwendung der FlowPrintManager-Klasse

C#

Auch hier binden Sie zunächst die beiden Assemblies *System.Printing.dll* und *ReachFramework.dll* ein. Erstellen Sie nachfolgend ein Formular, in das Sie eine Schaltfläche und einen *DocumentViewer* einbinden.

```
        private void Button2_Click(object sender, RoutedEventArgs e)
        {
```

Eine Instanz unserer Klasse erstellen:

```
            FlowPrintManager fpm = new FlowPrintManager(new Thickness(75,
25,25,25), true);
```

Wir erzeugen einen neuen Abschnitt:

```
            fpm.FlowDoc.Blocks.Add(new Paragraph(new Run("Hier steht zum Beispiel
 jede Menge Text. Hier steht zum Beispiel jede Menge Text.Hier steht zum Beispiel
 jede Menge Text.Hier steht zum Beispiel jede Menge Text.")));
```

Einen leeren Absatz:

```
            fpm.FlowDoc.Blocks.Add(new Paragraph(new LineBreak()));
                  for (int i=0; i<40; i++)
```

Und hier noch ein paar weitere Absätze:

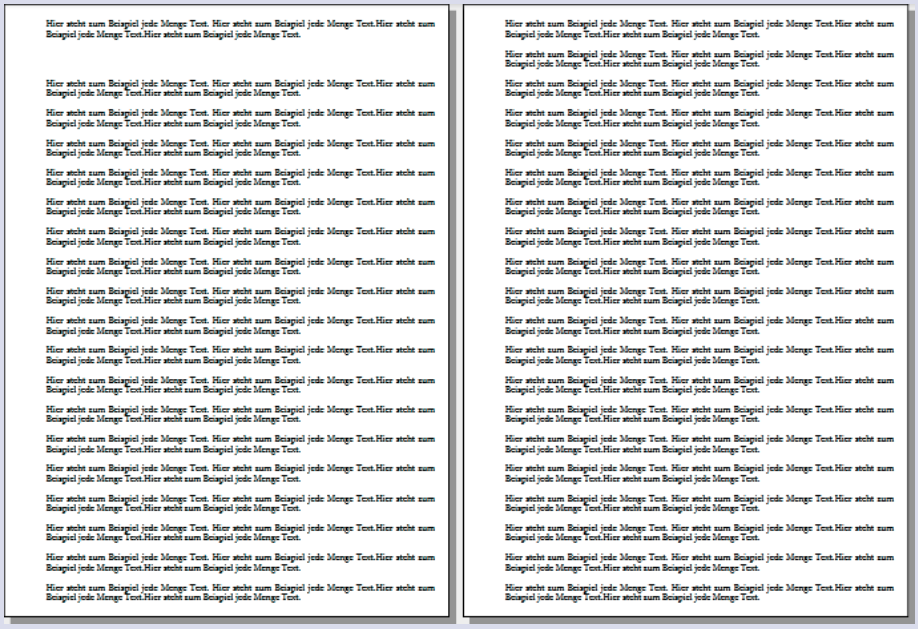
```
            for (int i=0; i<40; i++)
               fpm.FlowDoc.Blocks.Add(new Paragraph(new Run("Hier steht zum
 Beispiel jede Menge Text. Hier steht zum Beispiel jede Menge Text.Hier steht zum
 Beispiel jede Menge Text.Hier steht zum Beispiel jede Menge Text.")));
```

Und schon können Sie dieses Dokument in der Druckvorschau anzeigen:

```
            documentViewer1.Document = fpm.Document;
        }
```

Ergebnis

Die Druckvorschau bringt es an den Tag, durch unsere kleine Schleife bei der Ausgabe des letzten Absatzes wird der Text so lang, dass er nicht mehr auf eine Seite passt:



Doch wie funktioniert die Umwandlung des oben erzeugten *FlowDocument* in ein *FixedDocument*, das wir für die Druckvorschau (*DocumentViewer*) benötigen?

Die Klasse FlowPrintManager

Hier hilft uns die Möglichkeit weiter, mittels *XpsDocumentWriter* das *FlowDocument* in ein XPS-Dokument zu schreiben und aus diesem wiederum ein *FixedDocument* zu erzeugen. Doch leider ist diese Lösung im Normalfall mit einer physischen Datei verbunden, was nicht nur unschön aussieht, sondern teilweise auch Probleme nach sich ziehen kann. Aus diesem Grund haben wir uns für den Weg über einen *MemoryStream* entschieden, das komplette Handling erfolgt also im Speicher.

Doch nun zu den Details unserer *FlowPrintManager*-Klasse:

```
...
    public class FlowPrintManager
    {
```

Die internen Variablen:

```
        private FlowDocument _flowdocument;
        private FixedDocumentSequence _document;
        private MemoryStream ms;
        private Package package;
        private PrintDialog _dlg;
```

Die Eigenschaft *FlowDoc*, über die wir unsere *FlowDocument* zusammenbasteln können:

```
        public FlowDocument FlowDoc
        {
            get {return _flowdocument; }
        }
```

Die Eigenschaft *Document* liefert uns die gewünschte *FixedDocumentSequence* für die Druckvorschau:

```
        public FixedDocumentSequence Document
        {
            get
            {
```

Und hier wird es schnell etwas unübersichtlich. Grundlage des Schreibens von XPS-Dokumenten ist zunächst ein Package (ein Container mit Kompression), in welches das eigentliche XPS-Dokument eingefügt wird.

Am Anfang löschen wir zunächst einmal pauschal ein möglicherweise vom letzten Durchlauf noch vorhandenes Package:

```
                PackageStore.RemovePackage(new Uri("memorystream://data.xps"));
```

Neues Package erzeugen:

```
                PackageStore.AddPackage(new Uri("memorystream://data.xps"), package);
```

Neues *XpsDocument* im Package erzeugen:

```
                XpsDocument xpsDocument = new XpsDocument(package, CompressionOption.Fast,
                                          "memorystream://data.xps");
```

XpsDocumentWriter für das *XpsDocument* erstellen:

```
            XpsDocumentWriter writer = XpsDocument.CreateXpsDocumentWriter(xpsDocument);
```

Schreiben der Daten per *XpsDocumentWriter*:

```
                writer.Write(((IDocumentPaginatorSource)_flowdocument).DocumentPaginator);
```

Für das neue XPS-Dokument rufen wir eine *FixedDocumentSequence* ab und geben diese zurück:

```
                _document = xpsDocument.GetFixedDocumentSequence();
                xpsDocument.Close();
                return _document;
            }
        }
```

Unser Konstruktor kümmert sich um das Initialisieren der internen Variablen:

```
        public FlowPrintManager(Thickness borders, bool ShowPrintDialog = false)
        {
```

MemoryStream und *Package* erzeugen:

```
            ms = new MemoryStream();
            package = Package.Open(ms, FileMode.Create, FileAccess.ReadWrite);
```

Eventuell den *PrintDialog* anzeigen:

```
            _dlg = new PrintDialog();
            if (ShowPrintDialog) _dlg.ShowDialog();
```

Standardeinstellungen vornehmen:

```
            _flowdocument = new FlowDocument();
            _flowdocument.ColumnWidth = _dlg.PrintableAreaWidth;
            _flowdocument.PageHeight = _dlg.PrintableAreaHeight;
            _flowdocument.PagePadding = borders;
        }
```

Für die direkte Druckausgabe nutzen wir wieder den *PrintDialog*:

```
        public void Print(string title = "Mein Druckauftrag")
        {
            _dlg.PrintDocument(this.Document.DocumentPaginator, title);
        }
    }
```

Damit haben Sie auch eine recht einfache Lösung für die Ausgabe von Flow-Dokumenten.

Im folgenden Abschnitt steht nach der bisherigen Arbeit mit dem *PrintDialog*-Control ein intensiverer Blick auf die Druckerdetails und die Druckerkonfiguration im Mittelpunkt.

5.4 Druckerinfos, -auswahl, -konfiguration

Haben Sie bereits mit dem *PrintDialog*-Control gearbeitet, hat Sie sicher auch gestört, dass zur Druckerauswahl ein Dialog angezeigt wird, der gleichzeitig die Anzahl der zu druckenden Seiten etc. bestimmt. Das ist nicht in jedem Fall gewünscht, viel besser ist der direkte Zugriff auf die installierten Drucker und deren Eigenschaften.

An dieser Stelle müssen Sie sich von Ihren bisherigen Kenntnissen der Druckausgabe (Windows Forms) verabschieden, in WPF haben Sie es mit gänzlich anderen Klassen und Eigenschaften zu tun.

HINWEIS: Für die weitere Arbeit mit den entsprechenden Klassen müssen Sie die Assembly System.Printing.dll in ihr WPF-Projekt einbinden.

5.4.1 Die installierten Drucker bestimmen

Ausgangspunkt für die Bestimmung der installierten Drucker ist die *LocalPrintServer*-Klasse, die den aktuellen Computer mit seinen angeschlossenen Druckern (inklusive PDF-, XPS-Drucker etc.) kapselt.

HINWEIS: Mit PrintServer-Klasse und den entsprechenden Konstruktoren können Sie auch auf externe Printserver zugreifen. Wir beschränken uns im Weiteren jedoch auf den aktuellen PC.

Nach dem Erstellen einer Instanz können Sie über die *GetPrintQueues*-Methode eine Liste der installierten Drucker (hier per *PrintQueue* gekapselt) abrufen.

Beispiel 5.5: Anzeige einer Liste aller Drucker

C#

```
...
using System.Printing;
using System.Collections;
...
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
```

Den lokalen *PrintServer* abrufen:

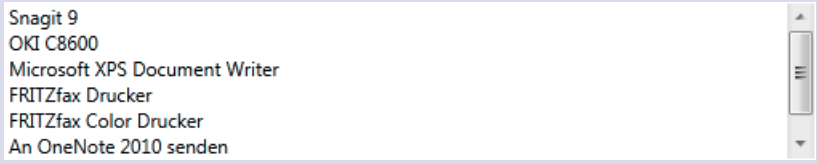
```
            var server = new LocalPrintServer();
```

Alle Drucker bestimmen und in einer *ListBox* anzeigen:

```
            var queues = server.GetPrintQueues();
            ListBox1.ItemsSource = queues;
            ListBox1.DisplayMemberPath = "Name";
        }
```

Ergebnis

Die angezeigte Liste:



Da der *ListBox* die *PrintQueue*-Objekte zugewiesen wurden, ist es nachfolgend auch recht einfach, auf die Details des jeweiligen Druckers zuzugreifen (siehe Abschnitt 5.4.3).

5.4.2 Den Standarddrucker bestimmen

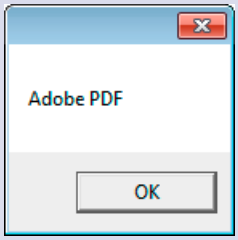
Erstellen Sie eine Instanz der *LocalPrintServer*-Klasse, können Sie über die Eigenschaft *DefaultPrintQueue* den Standarddrucker und damit dessen Eigenschaften bestimmen.

Beispiel 5.6: Standarddrucker bestimmen

C#

```
...
            var server = new LocalPrintServer();
            var queue = server.DefaultPrintQueue;
            MessageBox.Show(queue.FullName);
...
```

Ergebnis



5.4.3 Mehr über einzelne Drucker erfahren

Damit sind wir bereits bei der *PrintQueue*-Klasse angekommen. Diese ist recht auskunftsfreudig und bietet eine reiche Palette an Eigenschaften, mit denen Sie mehr über einen spezifischen Drucker erfahren können. Wie Sie den aktuellen Standarddrucker bzw. dessen zugehöriges *PrintQueue*-Objekt bestimmen haben Sie bereits im vorhergehenden Abschnitt erfahren. Um an ein *PrintQueue*-Objekt aus der *ListBox* (siehe Abschnitt 5.4.1) zu gelangen, können Sie beispielsweise das *SelectionChanged*-Ereignis nutzen:

```
        private void ListBox1_SelectionChanged(object sender, SelectionChangedEventArgs e)
        {
            var queue = ((PrintQueue)ListBox1.SelectedItem);
```

Die auf den ersten Blick unscheinbare *PrintQueue*-Klasse entpuppt sich bei genauerem Hinsehen jedoch als ergiebige Informationsquelle. Die folgende Tabelle zeigt eine Übersicht der wichtigsten Eigenschaften:

Eigenschaft	Beschreibung
AveragePagesPerMinute	Druckgeschwindigkeit in Seiten/Minute
Comment	Kommentar zum Drucker (lesen/schreiben)
CurrentJobSettings	Druckeinstellungen für den aktuellen Druckjob
DefaultPrintTicket	Die Standarddruckeinstellungen des Druckers
DefaultPriority	Die Priorität für neue Druckjobs
Description	Beschreibung für den Drucker
FullName	Name des Druckers
HasPaperProblem	Gibt es Papierprobleme?
HasToner	Gibt es Tonerprobleme?
HostingPrintServer	Printserver, der den Druckjob kontrolliert
IsBidiEnabled	Ist bidirektionale Kommunikation mit dem Drucker möglich?
IsBusy	Ist der Drucker beschäftigt?
IsDirect	Wird direkt gedruckt oder gespoolt?
IsManualFeedRequired	Manueller Papiervorschub nötig?
IsNotAvailable	Ist der Drucker verfügbar?
IsOffline	Ist der Drucker offline?
IsOutOfMemory	Ist zu wenig Speicher verfügbar?
IsOutOfPaper	Ist das Papier alle?
IsOutputBinFull	Ist das Ausgabefach voll?
IsPaperJammed	Hat sich Papier im Drucker verfangen?
IsPaused	Pausiert die Druckerwarteschlange?
IsPendingDeletion	Wird ein Druckjob gerade gelöscht?
IsPowerSaveOn	Ist der Drucker im Energiesparmodus?
IsPrinting	Druckt der Drucker gerade?
IsProcessing	Wird ein Druckjob abgearbeitet?
IsPublished	Ist der Drucker für andere Nutzer sichtbar (Netzwerk)?
IsQueued	Wird eine Druckerwarteschlange angeboten?
IsRawOnlyEnabled	Können EMF (Enhanced Meta File) Daten zum Drucken verwendet werden?
IsServerUnknown	Ist der Drucker in einem Fehlerzustand?
IsShared	Ist der Drucker für andere Nutzer freigegeben (Netzwerk)?
IsTonerLow	Ist der Toner alle?

Eigenschaft	Beschreibung
IsWaiting	Wartet der Drucker auf einen Druckjob?
IsWarmingUp	Ist der Drucker in der Warmlaufphase?
IsXpsDevice	Unterstützt der Drucker XML Paper Specification (XPS) als Seitenbeschreibungssprache?
KeepPrintedJobs	Wird der Druckjob abgespeichert?
Location	Wo steht der Drucker?
Name	Der Druckername
NeedUserIntervention	Muss ein Anwender den Drucker bedienen?
NumberOfJobs	Die Anzahl der Druckjobs
PagePunt	Warum kann die aktuelle Seite nicht gedruckt werden?
PrintingIsCancelled	Wurde der Druckjob abgebrochen?
PropertiesCollection	Eine Collection von Attribut/Wert-Paaren
QueueAttributes	Druckerqueue-Eigenschaften
QueueDriver	Der verwendete Druckertreiber
QueuePort	Der Druckerport
QueuePrintProcessor	Der verwendete Druckprozessor
QueueStatus	Queue-Status (z. B. „warming up“ „initializing“ „printing“)
SeparatorFile	Datei mit dem Deckblatt für einzelne Druckjobs
ShareName	Freigabename
StartTimeOfDay	Wann wurde der erste Druckjob gestartet?
UntilTimeOfDay	Wann wurde der letzte Druckjob gestartet?
UserPrintTicket	Die gewünschten Druckeinstellungen des Anwenders

Ups, so viele Informationen über den Drucker und den Druckspooler hatten Sie bisher sicher nur mit intensiver Hilfe der API in Erfahrung bringen können. Da hat sich seit den Windows Forms doch einiges gebessert. Ganz nebenbei können Sie mit den Methoden *Pause*, *Resume* und *Purge* der *PrintQueue*-Klasse auch den Druckspooler steuern bzw. Druckaufträge anhalten, fortsetzen und löschen. Doch an dieser Stelle verweisen wir Sie dann besser an die Hilfe zu *PrintQueue-Klasse.*

HINWEIS: Wichtigste Methode der *PrintQueue*-Klasse ist die *CreateXpsDocumentWriter*-Methode, mit der Sie den für die Druckausgabe erforderlichen *XpsDocumentWriter* erhalten (siehe Abschnitt 5.4.5).

5.4.4 Spezifische Druckeinstellungen vornehmen

Beim Betrachten der vorhergehenden Tabelle ist Ihnen sicher auch aufgefallen, dass Druckjob-spezifische Eigenschaften bisher nicht aufgetaucht sind. Einstellungen zum Druckjob

selbst nehmen Sie über die *UserPrintTicket*-Eigenschaft vor, welche die per *DefaultPrintTicket* vorgegebenen Standard-Einstellungen überschreibt.

Eigenschaften von *UserPrintTicket*:

Name	Beschreibung
Collation	Sortierfolge bei Druck von Kopien
CopyCount	Anzahl der Kopien
DeviceFontSubstitution	Verwendung von Druckerschriftarten
Duplexing	Doppelseitiger Druck
InputBin	Welcher Einzug soll verwendet werden?
OutputColor	Wie werden Farbe/Graustufen behandelt?
OutputQuality	Druckqualität
PageBorderless	Kann randlos gedruckt werden?
PageMediaSize	Papiergröße
PageMediaType	Papierart
PageOrder	Druckreihenfolge (1...n oder n ... 1)
PageOrientation	Seitenausrichtung
PageResolution	Seitenauflösung
PageScalingFactor	Skalierfaktor
PagesPerSheet	Druckseiten pro Blatt
PagesPerSheetDirection	Wie werden mehrere Seiten pro Blatt angeordnet?
PhotoPrintingIntent	Bei Unterstützung für Photodruck können hier verschiedene Qualitäten ausgewählt werden
Stapling	Optionen für das automatische Heften
TrueTypeFontMode	Optionen für die Verwendung von True Type Fonts (TTF).

Ein kleines Beispiel für die Konfiguration gefällig:

Beispiel 5.7: Druckausgabe umstellen auf Querformat mit zwei Kopien

C#

```
            var server = new LocalPrintServer();
            var queue = server.DefaultPrintQueue;
            queue.UserPrintTicket.PageOrientation = PageOrientation.Landscape;
            queue.UserPrintTicket.CopyCount = 2;
...
```

Auch hier finden Sie mittlerweile Einstellungen, für die Sie vor nicht allzu langer Zeit noch mit dem Druckertreiber intensiven „Gedankenaustausch“ betreiben konnten.

Doch sicher stellt sich manchem auch die Frage, welche Werte muss/kann man überhaupt an die einzelnen Eigenschaften übergeben, bzw. was bedeuten die Rückgabewerte dieser Eigenschaften (was bedeutet beispielsweise *OutputColor = 2*).

Hier hilft Ihnen die *GetPrintCapabilities*-Methode der *PrintQueue*-Klasse weiter. Diese gibt über Ihre Eigenschaften (diese entsprechen weitgehend der vorhergehenden Tabelle) jeweils Collections mit der Bedeutung der Einträge zurück.

Statt vieler Worte ist sicher ein Beispiel recht hilfreich:

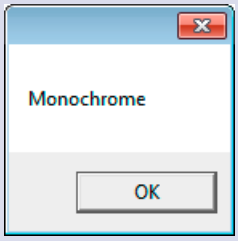
Beispiel 5.8: Welche Farbausgabeoptionen gibt es?

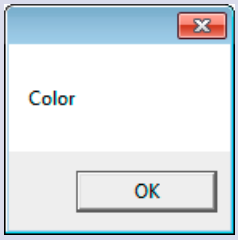
C#

```
var server = new LocalPrintServer();
var queue = server.DefaultPrintQueue;
var caps = queue.GetPrintCapabilities();
foreach (var cc in caps.OutputColorCapability)
    MessageBox.Show(cc.ToString());
```

Ergebnis

Die Anzeige:





Da es sich um eine normale Collection handelt, wissen Sie jetzt, dass 0 der „Monochromen"- und 1 der „Color"-Wiedergabe entspricht. Sie können an die *UserPrintTicket.OutputColor*-Eigenschaft also 0 oder 1 mit obiger Bedeutung übergeben. Alternativ lässt sich so auch ein entsprechender Wert richtig deuten.

HINWEIS: Obige Methode dürfte also für die Gestaltung von eigenen Druckdialogen von existenzieller Wichtigkeit sein, können Sie doch hier gleich die Werte für mögliche Auswahllisten abfragen.

Doch last, but not least, wollen wir natürlich etwas zu Papier bringen, womit wir auch schon beim letzten Absatz zu diesem Thema angekommen sind.

5.4.5 Direkte Druckausgabe

Nachdem Sie den Drucker ausreichend abgefragt und konfiguriert haben, soll natürlich auch ein Ergebnis auf dem Papier erscheinen. Hier schließt sich dann der Kreis zu den in den vorherigen Abschnitten vorgestellten Druckmöglichkeiten. Über ein per *CreateXpsDocumentWriter*-Methode erzeugtes *XpsDocumentWriter*-Objekt gelangen Ihre Ausgaben direkt an den gewünschten Drucker bzw. dessen Spooler.

Beispiel 5.9: Druckausgabe einer Textseite

C#

Standarddrucker abrufen:

```
var server = new LocalPrintServer();
var queue = server.DefaultPrintQueue;
```

Seite drehen:

```
queue.UserPrintTicket.PageOrientation = PageOrientation.Landscape;
```

Eine reine Text-Seite erstellen:

```
TextBlock txt = new TextBlock();
txt.Text = "Hier ist eine Seite mit Fließtext. Hier ist eine Seite mit Fließtext.
Hier ist eine Seite mit Fließtext. Hier ist eine Seite mit Fließtext. Hier ist eine
Seite mit Fließtext. ";
```

Den *TextBlock* müssen Sie noch skalieren, sonst ist nichts zu sehen:

```
txt.Measure(new Size((double) queue.UserPrintTicket.PageMediaSize.Height, 200));
txt.Arrange(new Rect(0, 0, txt.DesiredSize.Width, txt.DesiredSize.Height));
```

Wer sich jetzt wundert, warum wir die Seitenhöhe statt der -breite angeben, sollte sich daran erinnern, dass Drehungen keinen Einfluss auf die *Height*- bzw. *Width*-Eigenschaft haben.

Hier erstellen wir den *XpsDocumentWriter*:

```
var writer = PrintQueue.CreateXpsDocumentWriter(queue);
```

Und damit landen unsere Ausgaben bereits auf dem Drucker:

```
writer.Write(txt);
```

Natürlich steht es Ihnen frei, auch mehrseitige Dokumente zu erstellen, die Sie an die *Write*-Methode übergeben (siehe dazu ab Abschnitt 5.3). Beachten Sie allerdings, dass Sie nur einen Aufruf der *Write*-Methode realisieren können, Sie müssen als bereits ein komplettes mehrseitiges Dokument übergeben.

Damit wollen wir uns aus der Thematik „Druckausgabe“ verabschieden und uns neuen Aufgabengebieten zuwenden, auch wenn sicher noch einiges zu diesem Thema zu sagen wäre.

Teil II:
Windows Forms

- Windows Forms-Anwendungen
- Windows-Formulare verwenden
- Komponentenübersicht
- Einführung Grafikausgabe
- Druckausgabe
- Windows Forms-Datenbindung
- Erweiterte Grafikausgabe
- Ressourcen/Lokalisierung
- Komponentenentwicklung

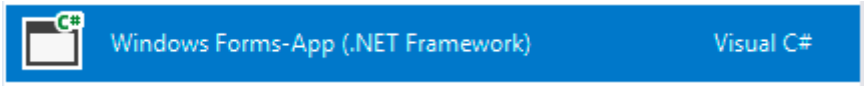
6 Windows Forms-Anwendungen

Nicht mehr ganz frisch, aber dennoch aktuell: die Windows Forms-Anwendungen. Diese entsprechen, im Gegensatz zu den WPF-Anwendungen, in der optischen Aufmachung weitgehend ihren Win32-Pendants. Der große Vorteil dieses Anwendungstyps ist die fast schon unüberschaubare Vielfalt an Komponenten, die es für fast jede Aufgabe gibt.

Ausgerüstet mit grundlegenden C#- und OOP-Kenntnissen werden Sie die folgenden Kapitel in die Lage versetzen, umfangreichere Benutzerschnittstellen eigenständig zu programmieren. Der Schwerpunkt des aktuellen Kapitels liegt zunächst auf dem Anwendungstyp selbst, nachfolgend werden wir uns mit dem Formular als der „Mutter aller Windows Forms-Komponenten“ und seiner grundlegenden Interaktion mit den Steuerelementen beschäftigen. Abschließend werfen wir einen Blick auf die einzelnen visuellen Komponenten bzw. Steuerelemente.

6.1 Grundaufbau/Konzepte

Die „gewöhnliche“ Windows Forms-Anwendung erstellen Sie in Visual Studio über die gleichnamige Vorlage:

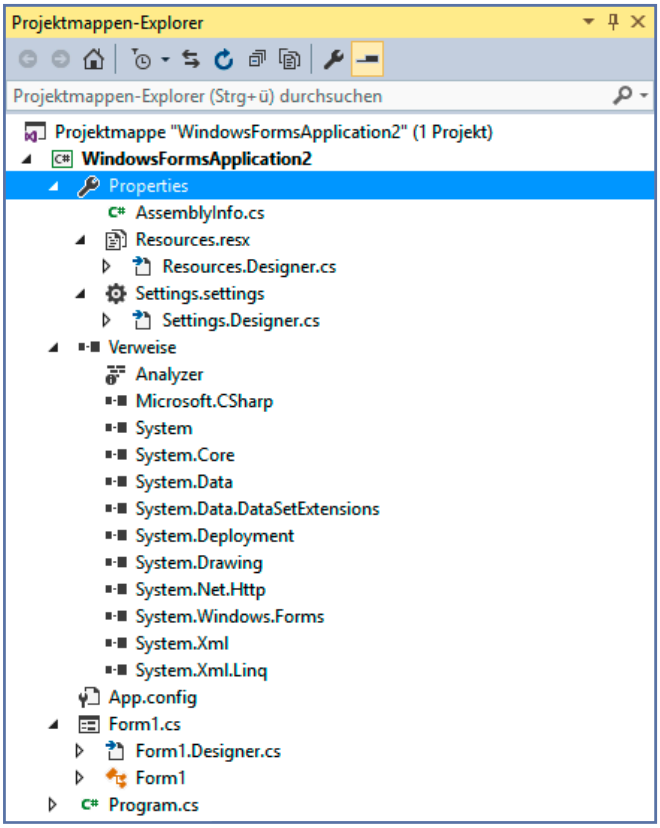


Beachten Sie in diesem Zusammenhang auch die unscheinbare kleine Combobox am oberen Dialogfensterrand, über die Sie vorgeben, für welches .NET-Zielframework Sie die Anwendung entwickeln wollen.

Bestimmte Anwendungs- bzw. Entwicklungsfeatures, wie z. B. das Entity-Framework, LINQ etc., sind nur verfügbar, wenn Sie eine Framework-Version wählen, die diese Features auch unterstützt. Informieren Sie sich also vorher, welche Features Sie unterstützen müssen.

Nachdem Sie diesen Schritt erfolgreich hinter sich gebracht haben, finden Sie im Designer bereits ein einfaches Windows Form vor. Ein Blick in den Projektmappen-Explorer (siehe

folgende Abbildung) zeigt jedoch noch eine Reihe weiterer Dateien, mit denen wir uns im Folgenden intensiver beschäftigen wollen.



Die Anwendung selbst ist zu diesem Zeitpunkt bereits ausführbar, ein Klick auf *F5* und Sie können sich davon überzeugen. Doch nach diesem Blick auf die Bühne wollen wir nun zunächst in den „Technik-Keller“ hinabsteigen, bevor wir uns in den weiteren Kapiteln mit der „Requisite“, das heißt der optischen Gestaltung der Anwendung, abmühen wollen.

6.1.1 Das Hauptprogramm – Program.cs

Der Einstiegspunkt für unsere Anwendung findet sich, wie Sie es sicher schon vermutet haben, in der Datei *Program.cs*. Der bereits vorliegende Code (Beispiel 28.1) enthält neben diversen Namespace-Importen eine Klasse *Program* mit der statischen Eintrittsmethode *Main*.

Beispiel 6.1: *Program.cs*

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    static class Program
    {
        /// <summary>
        /// Der Haupteinstiegspunkt für die Anwendung.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

static void Main()

Im einfachsten Fall enthält die *Main*-Methode lediglich drei Methodenaufrufe (auf die Details kommen wir in den folgenden Abschnitten zu sprechen). Alternativ können Sie jedoch auch eine *Main*-Methode mit der Rückgabe eines *int*-Werts und/oder der Übergabe eines *string*-Arrays realisieren (siehe dazu auch im Buch Kapitel 17, Konsolenanwendungen). Über dieses String-Array können Sie die Kommandozeilenparameter des Programms auswerten.

Beispiel 6.2: Auswerten von Kommandozeilenparametern

C#

```
...
        static void Main(string[] args)
        {
            foreach (string arg in args)
                Console.WriteLine(arg);
...
```

HINWEIS: Günstiger ist die Auswertung der Kommandozeilenparameter jedoch über die *GetCommandLineArgs*-Collection. Diese steht jederzeit zur Verfügung und erfordert nicht schon beim Programmstart eine entsprechende Auswertung.

Beispiel 6.3: Verwendung von *GetCommandLineArgs*

C#

```
...
      foreach (string arg in Environment.GetCommandLineArgs())
      {
          Console.WriteLine(arg);
      }
...
```

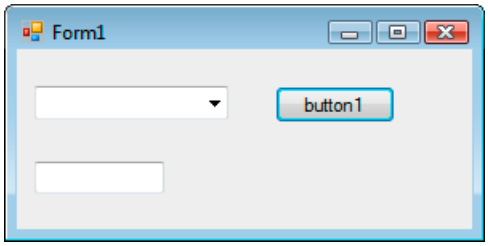
Kommen wir jetzt zum Inhalt der `Main`-Methode.

Application.EnableVisualStyles

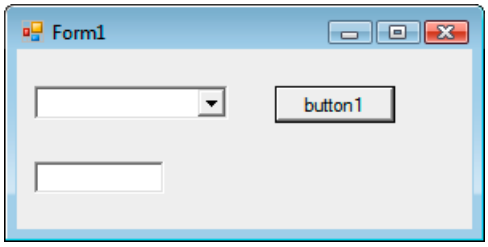
Sicher ist auch Ihnen nicht verborgen geblieben, dass sich seit Windows XP ein neues Layout für die einzelnen Windows Forms-Steuerelemente eingebürgert hat. Voraussetzung für deren Nutzung ist der entsprechende Aufruf der *Application.EnableVisualStyles*-Methode.

Wie sich der Aufruf auf ein fiktives Formular auswirkt, zeigen die beiden folgenden Abbildungen.

Mit *EnableVisualStyles*-Aufruf:



Ohne *EnableVisualStyles*-Aufruf:



Application.SetCompatibleTextRenderingDefault(false)

Mit Einführung von Windows Forms 2.0 wurde die Ausgabe von Text teilweise umgestellt. Da diese Variante nicht hundertprozentig kompatibel mit den bisherigen Ausgabeergebnissen ist, wurde für einzelne Controls eine Eigenschaft *UseCompatibleTextRendering* definiert, mit der bestimmt wird, ob die alte Variante (1.0 bzw. 1.1) oder die neuere Variante (ab 2.0) verwendet werden soll. Über die Methode *Application.SetCompatibleTextRenderingDefault* wird dieser Wert für alle Controls der Anwendung voreingestellt.

Lange Rede kurzer Sinn: Ist der übergebene Wert *true*, verwenden die Steuerelemente GDI für die Textausgabe, andernfalls GDI+.

Application.Run(new Form1())

Und damit sind wir auch schon beim eigentlichen Mittelpunkt eines jeden Windows Forms-Programms angekommen. *Application.Run* erstellt die für jede Windows-Anwendung lebensnotwendige Nachrichtenschleife, die beispielsweise Maus- und Tastaturereignisse, aber auch Botschaften des Systems oder anderer Anwendungen, entgegennimmt.

Die zweite Aufgabe dieser statischen Methode ist das Instanziieren des Hauptformulars und dessen Anzeige. Wird dieses Fenster geschlossen, ist automatisch auch die Nachrichtenschleife der Anwendung beendet, das Programm terminiert. Dies gilt auch, wenn Sie aus dem Hauptformular heraus (dies ist zunächst *Form1*) weitere Formulare öffnen.

Besteht Ihre Anwendung aus mehr als nur einem Formular, stehen Sie sicher auch vor der Frage, welches Formular denn das Hauptformular bzw. das Startobjekt sein soll. Standardmäßig ist nach dem Erstellen eines Windows Forms-Projekts *Form1* als Startobjekt definiert. Nachträglich können Sie jederzeit auch ein anderes Fenster oder auch eine spezielle Prozedur als Startobjekt festlegen. Durch Editieren des von Visual Studio automatisch generierten Konstruktoraufrufs zum Erzeugen des Startformulars *Form1* kann ein beliebiges anderes Formular der Anwendung zum Startformular gekürt werden.

Nicht in jedem Fall ist es jedoch erwünscht, dass gleich ein Formular beim Start der Anwendung angezeigt wird. Sollen beispielsweise Übergabeparameter ausgewertet werden, ist es häufig günstiger, zunächst diese zu bearbeiten und dann erst das geeignete Formular anzuzeigen. In diesem Fall ändern Sie einfach obigen Abschnitt entsprechend Ihren Wünschen.

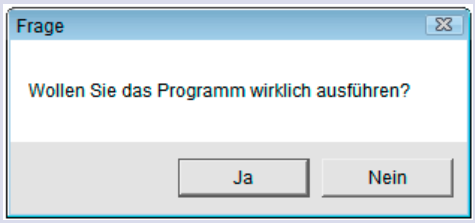
Beispiel 6.4: Eine *Main*-Methode, die zunächst eine *MessageBox*-Abfrage startet

C#

```
...
[STAThread]
static void Main()
{
    if (MessageBox.Show("Wollen Sie das Programm wirklich ausführen?", "Frage",
                        MessageBoxButtons.YesNo) == DialogResult.Yes)
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form2());
    }
}
```

Ergebnis

Ein Start der Anwendung hat jetzt zur Folge, dass zunächst die *MessageBox*-Abfrage aufgerufen wird. Mit dem Druck auf den *Ja*-Button wird der Anwendung *Form2* als Hauptformular zugewiesen und ausgeführt. Andernfalls endet die Programmausführung an dieser Stelle.



6.1.2 Die Oberflächendefinition – Form1.Designer.cs

Eigentlich nicht für Ihre Blicke bestimmt, fristet die Datei *Form1.Designer.cs* ein relativ unbeachtetes Dasein. Doch dieser Eindruck täuscht, enthält doch diese Datei die eigentliche Klassendefinition von *Form1* inklusive Initialisierung der Oberfläche. Gerade der letzte Punkt ist besonders wichtig: Jede Komponente, die Sie im Formulardesigner in das Formular einfügen und konfigurieren, wird in dieser Klassendefinition (Methode *InitializeComponent*) instanziiert und konfiguriert.

Beispiel 6.5: *Form1.Designer.cs*

C#

```
namespace WindowsFormsApplication1
{
    partial class Form1
    {
        private System.ComponentModel.IContainer components = null;

        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
                components.Dispose();
            base.Dispose(disposing);
        }

        #region Vom Windows Form-Designer generierter Code
```

Die folgende Methode sollten Sie keinesfalls selbst editieren[1]:

[1] Mit einer Ausnahme: Haben Sie später einmal einen oder mehrere Eventhandler in der Datei *Form1.cs* gelöscht, können Sie deren Zuweisung zu den einzelnen Ereignissen am bequemsten in dieser Datei aufheben. Die betreffenden Zeilen sind durch die „beliebte“ rote Wellenlinie des Editors gekennzeichnet, es genügt, wenn Sie diese Zeilen einfach löschen.

```
        private void InitializeComponent()
        {
            this.SuspendLayout();
            //
            // Form1
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(284, 112);
            this.Name = "Form1";
            this.Text = "Form1";
            this.ResumeLayout(false);
```

Ab hier folgen später die Initialisierungen der einzelnen Formularkomponenten:

```
            ...
        }
        #endregion
    }
}
```

Haben Sie sich obiges Listing genauer angesehen, wird Ihnen aufgefallen sein, dass die Klasse *Form1* als *partial* deklariert ist. Dies ermöglicht es, den Code der Klassendefinition auf mehrere Dateien aufzuteilen, was in unserem Fall bedeutet, dass wir in *Form1.cs* den Rest der Klasse *Form1* vorfinden.

6.1.3 Die Spielwiese des Programmierers – Form1.cs

Jetzt kommen wir zum eigentlichen Tummelplatz für Ihre Aktivitäten als Programmierer. Visual Studio hat bereits ein kleines Codegerüst vorbereitet, das aus dem Import diverser Namespaces sowie der restlichen Klassendefinition von *Form1* inklusive Konstruktor besteht:

Beispiel 6.6: *Form1.cs*

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
```

Der Konstruktor ist dafür verantwortlich, die Methode *InitializeComponent* aufzurufen, diese initialisiert und parametriert die einzelnen Windows Forms-Komponenten:

```
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Fügen Sie später in *Form1* eine Schaltfläche (*Button*) ein, wird die Oberflächendefinition in der Datei *Form1.Designer.cs* abgelegt. Den Code für die Ereignisbehandlung definieren Sie in *Form1.cs*.

Beispiel 6.7: Code nach Einfügen eines Buttons

C#

In *Form1.Designer.cs*:

```
        private void InitializeComponent()
        {
            this.button1 = new System.Windows.Forms.Button();
            this.SuspendLayout();
            //
            // button1
            //
            this.button1.Location = new System.Drawing.Point(55, 35);
            this.button1.Name = "button1";
            this.button1.Size = new System.Drawing.Size(75, 23);
            this.button1.TabIndex = 0;
            this.button1.Text = "button1";
            this.button1.UseVisualStyleBackColor = true;
            this.button1.Click += new System.EventHandler(this.button1_Click);
...
```

In *Form1.cs*:

```
        private void button1_Click(object sender, EventArgs e)
        {
            this.Close();
        }
```

6.1.4 Die Datei AssemblyInfo.cs

Verschiedene allgemeine Informationen, wie z. B. der Titel der Anwendung oder das Copyright, können als Attribute in die Datei *AssemblyInfo.cs* eingetragen werden. Zugriff auf diese Datei erhalten Sie über den Projektmappen-Explorer (unter *Properties*).

Beispiel 6.8: Auszug aus *AssemblyInfo.cs*

C#

```
...
[assembly: AssemblyTitle("WindowsFormsApplication1")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("WindowsFormsApplication1")]
[assembly: AssemblyCopyright("Copyright ©  2009")]
[assembly: AssemblyCulture("")]
...
```

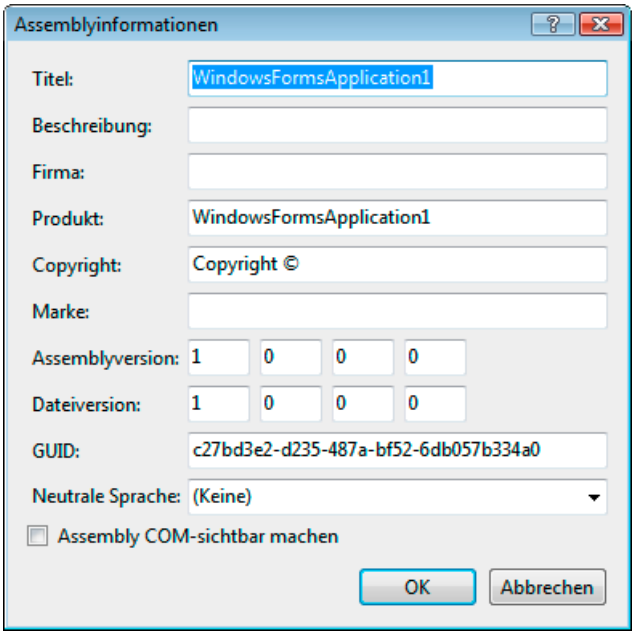
Zur Laufzeit können Sie auf die Einträge über bestimmte Eigenschaften des *Application*-Objekts zugreifen.

Beispiel 6.9: Anzeige des Eintrags unter *AssemblyProduct* in einem Meldungsfenster

C#

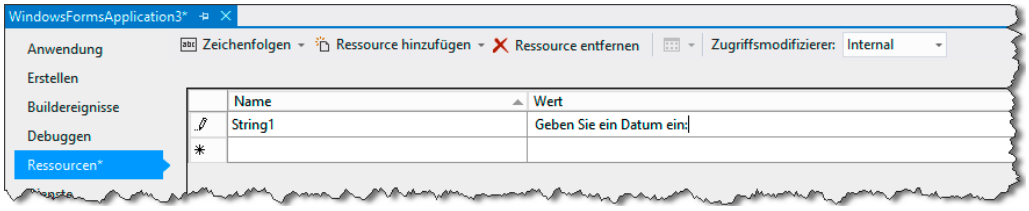
```
MessageBox.Show(Application.ProductName);
```

Editieren lassen sich die Einträge am einfachsten über den folgenden Dialog, den Sie über *Projekt|Eigenschaften|Anwendung|Assemblyinformationen* erreichen:



6.1.5 Resources.resx/Resources.Designer.cs

Beide Dateien stellen das Abbild der unter *Projekt|Eigenschaften|Ressourcen* definierten Ressourcen dar. Die XML-Datei *Resources.resx* enthält die eigentlichen Definitionen, die Sie mit dem in der folgenden Abbildung gezeigten Editor erstellen können.



Hier können Sie sehr komfortabel nahezu beliebige Ressourcen (Bilder, Zeichenketten, Audiodateien, ...) zu Ihrem Projekt hinzufügen.

Im Gegensatz dazu stellt die Datei *Resources.Designer.cs* für die Ressourcen entsprechende Mapperklassen bereit, die den späteren Zugriff auf die einzelnen Einträge vereinfachen und typisieren. Verantwortlich dafür ist die *Properties.Resources*-Klasse:

Syntax:

```
<Projekt Namespace>.Properties.Resources.<ResourceName>
```

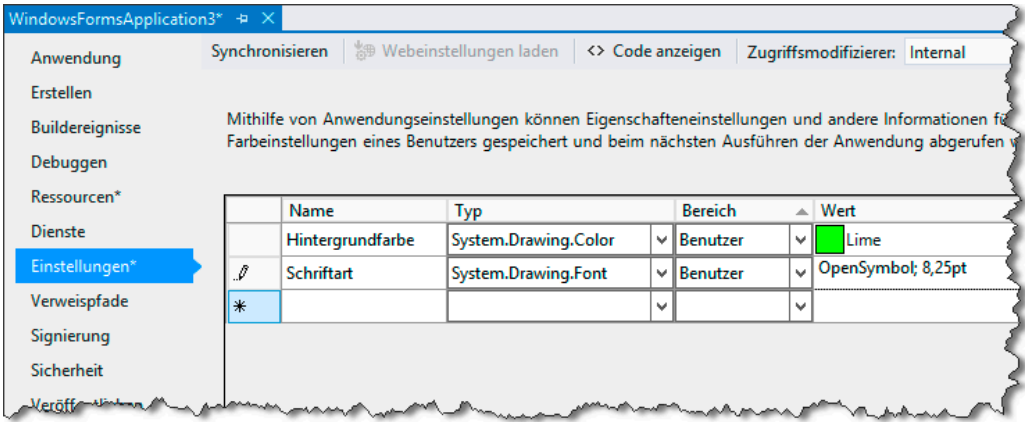
Beispiel 6.10: Die Bildressource *BeimChef.bmp* wird in einer *PictureBox* angezeigt.

C#

```
ictureBox1.Image = Properties.Resources.BeimChef;
```

6.1.6 Settings.settings/Settings.Designer.cs

Diese beiden Dateien sind zur Entwurfszeit für die Verwaltung der Programmeinstellungen verantwortlich. Die XML-Datei Settings.Settings enthält die eigentlichen Werte, die Sie jedoch nicht direkt zu bearbeiten brauchen, sondern unter Verwendung eines komfortablen Editors:



Für jede Einstellung können Sie Name, Typ, Bereich und *Wert* festlegen. Wenn Sie den Bereich als *Anwendung* spezifizieren, wird diese Einstellung später (Laufzeit) in der Konfigurationsdatei *<Anwendungsname.exe>.config* unter dem *<applicationSettings>*-Knoten gespeichert. Falls Sie *Benutzer* wählen, erfolgt die Ablage unterhalb des *<userSettings>*-Knotens.

Die Datei *Settings.Designer.cs* stellt für die oben definierten Einträge eine Mapperklasse bereit, die den späteren Zugriff auf die einzelnen Einträge vereinfacht und vor allem typisiert.

Beispiel 6.11: *Settings.Designer.cs*

C#

```
namespace WindowsFormsApplication1.Properties {
...
internal sealed partial class Settings :
global::System.Configuration.ApplicationSettingsBase {
...
        [global::System.Configuration.UserScopedSettingAttribute()]
        [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [global::System.Configuration.DefaultSettingValueAttribute("Lime")]
        public global::System.Drawing.Color Hintergrundfarbe {
            get {
                return ((global::System.Drawing.Color)(this["Hintergrundfarbe"]));
            }
            set {
                this["Hintergrundfarbe"] = value;
            }
        }
...
```

Der Zugriff auf die eingetragenen Einstellungen ist – ähnlich dem Zugriff auf Ressourcen – einfach über die *Properties.Settings*-Klasse möglich.

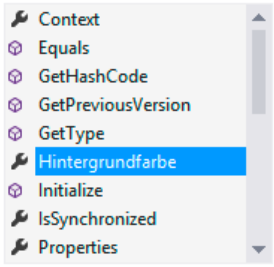
Syntax:

```
<Projekt Namespace>.Properties.Settings.Default.<SettingsName>
```

Beispiel 6.12: Verwendung der Einstellungen

C#

```
private void Form1_Load(object sender, EventArgs e)
{
    this.BackColor = Properties.Settings.Default.
}
```



Doch auch das Zurückschreiben von Nutzereinstellungen in die Datei ist über obige Klasse problemlos möglich.

Beispiel 6.13: Einstellungsänderungen in die Config-Datei zurückschreiben

C#

```
Properties.Settings.Default.myColor = Color.AliceBlue;
Properties.Settings.Default.Save();
```

HINWEIS: Da auch der spätere Programmbenutzer die zur Assembly mitgegebene XML-Konfigurationsdatei (z. B. *<Anwendungsname>.exe.config*) editieren kann, ergeben sich einfache Möglichkeiten für nachträgliche benutzerspezifische Anpassungen, ohne dazu das Programm erneut kompilieren zu müssen.

6.1.7 Settings.cs

Nanu, hatten wir diese Datei nicht gerade? Nein, hierbei handelt es sich quasi um den zweiten Teil der Klassendefinition von *Settings* aus *Settings.Designer.cs*, was dank partieller Klassendefinition problemlos möglich ist. Im Normalfall dürften Sie diese Datei nicht zu sehen bekommen, aber wenn Sie sich die Mühe machen, über den Designer den Button *Code anzeigen* anzuklicken, findet sich kurz darauf auch diese Datei in Ihrem Projekt.

Beispiel 6.14: *Setting.cs*

C#

```
namespace WindowsFormsApplication1.Properties {
    internal sealed partial class Settings {
        public Settings() {
            // Heben Sie die  Auskommentierung der unten angezeigten Zeilen auf,
            // um Ereignishandler zum Speichern und Ändern von Einstellungen
hinzuzufügen:
            // this.SettingChanging += this.SettingChangingEventHandler;
            //
            // this.SettingsSaving += this.SettingsSavingEventHandler;
            //
        }

        private void SettingChangingEventHandler(object sender,
                    System.Configuration.SettingChangingEventArgs e) {
          // Fügen Sie hier Code zum Behandeln des SettingChangingEvent-Ereignisses
hinzu.
        }

        private void SettingsSavingEventHandler(object sender,
                    System.ComponentModel.CancelEventArgs e) {
          // Fügen Sie hier Code zum Behandeln des SettingsSaving-Ereignisses
hinzu.
        }
    }
}
```

Mit den in dieser Datei bereitgestellten Eventhandler-Rümpfen bietet sich die Möglichkeit, auf das Laden, Ändern und Sichern von Einstellungen per Ereignis zu reagieren. Es genügt, wenn Sie die entsprechenden Kommentare im Konstruktor entfernen und den gewünschten Code in die Eventhandler eintragen.

6.2 Ein Blick auf die Application-Klasse

Die *Application*-Klasse aus dem Namespace *System.Windows.Forms* ist für den Programmierer von zentraler Bedeutung und unser Kapitel über die Grundlagen von Windows Forms-Anwendungen wäre höchst unvollständig, würden wir auf diese wichtige Klasse nur ganz am Rand eingehen. Den ersten Kontakt mit ihr hatten Sie bereits in der *Main*-Methode.

Die *Application*-Klasse bietet einige interessante statische Eigenschaften und Methoden, von denen wir hier nur die wichtigsten kurz vorstellen wollen:

6.2.1 Eigenschaften

Einige wichtige Eigenschaften von *Application*:

Eigenschaft	Beschreibung
AllowQuit	... darf die Anwendung beendet werden (readonly)?
CommonAppDataPath	... ein gemeinsamer Anwendungspfad
CommonAppDataRegistry	... ein gemeinsamer Registry-Eintrag
CompanyName	... der in den Anwendungsressourcen angegebene Firmenname
CurrentCulture	... ein Objekt mit den aktuellen Kulturinformation (Kalender, Währung)
ExecutablePath	... der Name der Anwendung inklusive Pfadangabe
LocalUserAppDataPath	... Anwendungspfad für die Daten des lokalen Benutzers
OpenForms	... Collection der geöffneten Formulare
ProductName	... der in den Ressourcen angegebene Produktname
ProductVersion	... die in den Ressourcen angegebene Produktversion
StartupPath	... der Anwendungspfad
UserAppDataPath	... Pfad für die Anwendungsdaten
UserAppDataRegistry	... Registry-Schlüssel für die Anwendungsdaten

HINWEIS: Einige der obigen Eigenschaften lassen sich den Einträgen der Datei *AssemblyInfo.cs* zuordnen.

Beispiel 6.15: Abrufen aller geöffneten Formulare

C#

```
foreach (Form f in Application.OpenForms)
{
   listBox1.Items.Add(f.Name);
}
```

Beispiel 6.16: Abfrage von *UserAppDataPath* und *UserAppDataRegistry*

C#

```
textBox1.Text = Application.UserAppDataRegistry.ToString();
textBox2.Text = Application.UserAppDataPath.ToString();
```

Der Registry-Schlüssel:

```
HKEY_CURRENT_USER\Software\WindowsFormsApplication1\WindowsFormsApplication1\
1.0.0.0
```

Das Verzeichnis:

```
C:\Users\Tom\AppData\Roaming\WindowsFormsApplication1\WindowsFormsApplication1\
1.0.0.0
```

6.2.2 Methoden

Wichtige Methoden von *Application*:

Methode	Beschreibung
DoEvents	... ermöglicht Verarbeitung ausstehender Windows-Botschaften
EnableVisualStyles	... ermöglicht die Darstellung mit Styles
Exit	... Anwendung beenden
ExitThread	... schließt aktuellen Thread
Run	... startet das Hauptfenster der Anwendung (Beginn einer Nachrichtenschleife für den aktuellen Thread)
Restart	... die Anwendung wird beendet und sofort neu gestartet
SetSuspendState	... versucht das System in den Standbymodus bzw. den Ruhezustand zu versetzen

Den Einsatz der *Run*-Methode haben wir im Abschnitt „Application.Run(new Form1())“ weiter vorne in diesem Kapitel besprochen. Wie Sie *DoEvents* richtig verwenden, zeigen wir Ihnen im folgenden Beispiel.

Beispiel 6.17: Eine zeitaufwendige Berechnungsschleife blockiert die Anzeige der Uhrzeit. Durch Aufrufen von *DoEvents* läuft die Uhr auch während der Berechnung weiter.

C#

Zunächst wird die „Uhr“ programmiert (vorher haben Sie eine *Timer*-Komponente in das Komponentenfach gezogen: `Interval = 1000`, `Enabled = true`):

```
private void timer1_Tick(object sender, EventArgs e)
{
    label1.Text = DateTime.Now.ToLongTimeString();
}
```

Die Berechnungsschleife wird gestartet:

```
private void button1_Click(object sender, EventArgs e)
{
    double a = 0;
    for (int i = 0; i < 100000000; i++)
    {
        a = Math.Sin(i) + a;
```

Nach jedem 1000ten Schleifendurchlauf wird CPU-Zeit freigegeben, sodass auf andere Ereignisse reagiert werden kann:

```
        if (i % 1000 ==0)  Application.DoEvents();
```

Wenn Sie obige Anweisung auskommentieren, „steht“ die Uhr für die Zeit der Berechnung.

```
    }
```

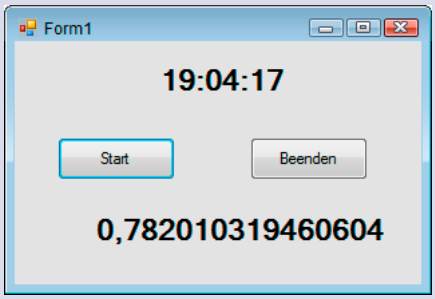
Die Ergebnisanzeige:

```
            label2.Text = a.ToString();
        }
```

Anwendung beenden:

```
        private void button2_Click(object sender, EventArgs e)
        {
            Application.Exit();
        }
    }
```

Ergebnis



Eine recht effiziente Art, für Ruhe unter dem Schreibtisch zu sorgen, zeigt das folgende kurze Beispiel.

Beispiel 6.18: Den PC in den Ruhezustand versetzen

C#

```
...
    Application.SetSuspendState(PowerState.Suspend,false,true);
...
```

6.2.3 Ereignisse

Last but not least verfügt die *Application*-Klasse auch über diverse Ereignisse:

Ereignis	Beschreibung
ApplicationExit	... wenn die Anwendung beendet werden soll
EnterThreadModal	... vor dem Eintritt der Anwendung in den modalen Zustand
Idle	... die Applikation hat gerade nichts zu tun
LeaveThreadModal	... der modale Zustand wird verlassen
ThreadException	... eine nicht abgefangene Threadausnahme ist aufgetreten
ThreadExit	... ein Thread wird beendet. Handelt es sich um den Hauptthread der Anwendung, wird zunächst dieses Ereignis und anschließend *ApplicationExit* ausgelöst.

6.3 Allgemeine Eigenschaften von Komponenten

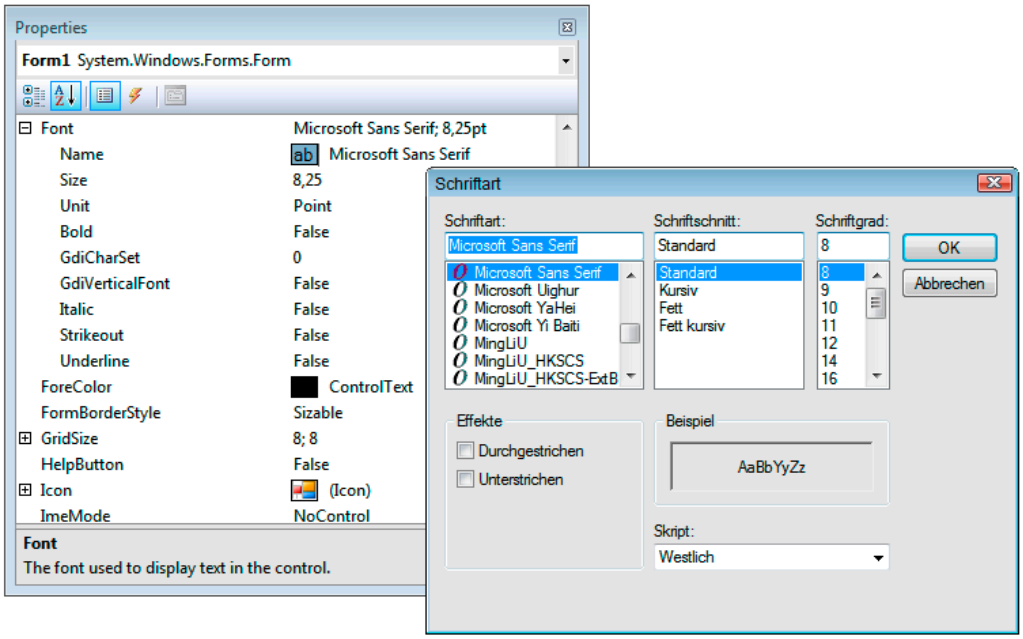
Es gibt eine Vielzahl von Standard-Properties über die (fast) alle Komponenten verfügen. Auf viele Eigenschaften kann nur zur Entwurfszeit, auf andere erst zur Laufzeit zugegriffen werden. Das ist auch der Grund, warum Letztere nicht im Eigenschaften-Fenster der Entwicklungsumgebung zu finden sind. Die Eigenschaften können zur Entwurfszeit (E) und/oder zur Laufzeit (L) verfügbar sein (r = nur lesbar).

Eigenschaft	Erläuterung	E	L
Anchor	Ausrichtung bezüglich des umgebenden Objekts	x	x
BackColor	Hintergrundfarbe	x	x
BorderStyle	Art des Rahmens	x	r
Cursor	Art des Cursors über dem Steuerelement	x	x
CausesValidation	siehe *Validate*-Ereignis		
ContextMenu	Auswahl eines Kontextmenüs	x	x
Dock	Andockstellen bezüglich des umgebenden Objekts	x	x
Enabled	aktiv/nicht aktiv	x	x
Font	Schriftattribute (Namen, Größe, fett, kursiv etc.)	x	x
ForeColor	Vordergrundfarbe, Zeichenfarbe	x	x
Handle	Fensterhandle, an welches das Control gebunden ist		r
Location	Position der linken oberen Ecke	x	x
Locked	Sperren des Steuerelements gegen Veränderungen	x	
Modifiers	Zugriffsmodifizierer (*private*, *public*, ...)	x	x
Name	Bezeichner	x	
TabIndex	Tab-Reihenfolge	x	x
TabStop	Tabulatorstopp Ja/Nein	x	x
Tag	Hilfseigenschaft (speichert Info-Text)	x	x
Text	Beschriftung oder Inhalt	x	x
Visible	Sichtbar Ja/Nein	x	x

Einige dieser Eigenschaften bedürfen einer speziellen Erläuterung.

6.3.1 Font

Mit dieser Eigenschaft (die in Wirklichkeit ein Objekt ist) werden Schriftart, Schriftgröße und Schriftstil der *Text*-Eigenschaft eingestellt (gilt nicht für die Titelleiste eines Formulars). Die Einstellung zur Entwurfszeit ist kein Problem, da ja zusätzlich auch ein komfortabler Fontdialog (der von Windows) zur Verfügung steht.



Der Zugriff auf die *Font*-Eigenschaften per Programmcode ist aber nicht mehr ganz so einfach.

Beispiel 6.19: Zugriff auf die *Font*-Eigenschaften

C#

Sie können zwar die Schriftgröße einer *TextBox* lesen:

```
MessageBox.Show(textBox1.Font.Size.ToString();   // zeigt z.B. 8,25
```

Ein direktes Zuweisen/Ändern der Schriftgröße ist allerdings nicht möglich, da diese Eigenschaft schreibgeschützt ist:

```
textBox1.Font.Size = 12;                  // Fehler (read only)
```

oder

```
textBox1.Font.Size = textBox2.Font.Size;  // Fehler (read only)
```

Um die Schrifteigenschaften zu ändern, müssen Sie ein neues *Font*-Objekt erzeugen.

Beispiel 6.20: Ändern der Schriftgröße einer *TextBox*

C#

```
textBox1.Font = new Font(textBox1.Font.Name, 12);
```

Obiges Beispiel zeigt allerdings nur einen der zahlreichen Konstruktoren der *Font*-Klasse. Darunter gibt es natürlich auch welche, die das Zuweisen des Schriftstils (fett, kursiv etc.) ermöglichen.

Die möglichen Schriftstile sind in der *FontStyle*-Enumeration definiert (siehe folgende Tabelle):

Schriftstil	Beschreibung
Regular	normaler Text
Bold	fett formatierter Text
Italic	kursiv formatierter Text
Underline	unterstrichener Text
Strikeout	durchgestrichener Text

Beispiel 6.21: Inhalt einer *TextBox* fett formatieren

C#

```
textBox1.Font = new Font(textBox1.Font, FontStyle.Bold);
```

Was aber ist, wenn ein Text gleichzeitig mehrere Schriftstile haben soll? In diesem Fall müssen die einzelnen Schriftstile mit einer bitweisen ODER-Verknüpfung kombiniert werden.

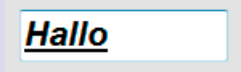
Beispiel 6.22: Mehrere Schriftstile zuweisen

C#

Der Inhalt einer *TextBox* wird mit der Schriftart „Arial“ und einer Schriftgröße von 16pt gleichzeitig fett, kursiv und mit Unterstreichung formatiert.

```
textBox1.Font = new Font("Arial", 16f, FontStyle.Bold | FontStyle.Underline |
                                       FontStyle.Italic);
```

Ergebnis



HINWEIS: Mehr zum *Font*-Objekt im Zusammenhang mit der Textausgabe auf einem Formular finden Sie im Grafik-Kapitel 27.

6.3.2 Handle

Die Handle-Eigenschaft ist wohl mehr für den fortgeschrittenen Programmierer bestimmt. Der Wert ist ein Windows-Handle und natürlich schreibgeschützt. Wenn das Handle noch nicht vorhanden ist, wird das Erstellen durch den Verweis auf diese Eigenschaft erzwungen.

Beispiel 6.23: Verwendung *Handle*

C#

Der Klick auf *button1* bewirkt, dass unter Verwendung des Handles von *button2* auch dieser mit einem Rechteck umrandet wird:

```
private void button1_Click(object sender, EventArgs e)
{
    ControlPaint.DrawFocusRectangle(Graphics.FromHwnd(button2.Handle),
                                    button2.ClientRectangle);
}
```

6.3.3 Tag

Diese Eigenschaft müssen Sie sich wie einen unsichtbaren „Merkzettel“ vorstellen, den man an ein Objekt „anheften“ kann. Im Eigenschaftenfenster erscheint *Tag* zwar als *string*-Datentyp, in Wirklichkeit ist *Tag* aber vom *object*-Datentyp, denn per Programmcode können Sie dieser Eigenschaft einen beliebigen Wert zuweisen.

Beispiel 6.24: Verwendung *Tag*

C#

Sie haben der *Image*-Eigenschaft einer *PictureBox* ein Bild zugewiesen und wollen beim Klick auf die *PictureBox* einen erklärenden Text in einem Meldungsfenster anzeigen lassen:

```
...
pictureBox1.Tag = "Das ist meine Katze!";
...
private void pictureBox1_Click(object sender, EventArgs e)
{
    MessageBox.Show(pictureBox1.Tag.ToString());  // zeigt "Das ist meine Katze!"
}
```

HINWEIS: Auch Steuerelemente, die im Komponentenfach abgelegt werden, wie *Timer*, *OpenFileDialog* etc. verfügen über eine *Tag*-Eigenschaft.

6.3.4 Modifiers

Beim visuellen Entwurfsprozess werden Steuerelemente vom Designer wie private Felder der *Form*-Klasse angelegt, ein Zugriff von außerhalb ist also zunächst nicht möglich.

Modifiers ist keine Eigenschaft im eigentlichen Sinn, sondern lediglich eine Option der Entwicklungsumgebung, die Sie dann brauchen, wenn Sie von einem Formular aus auf ein Steuerelement zugreifen wollen, das sich auf einem anderen Formular befindet. Angeboten werden die Modifizierer *public*, *protected*, *internal*, *protected internal* und *private*.

Beispiel 6.25: Verwendung von *Modifiers*

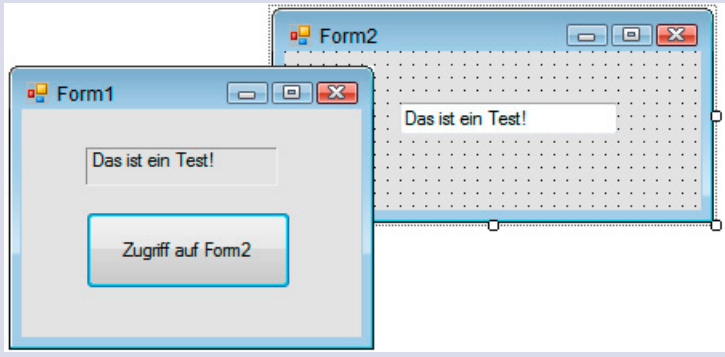
C#

Nach dem Klick auf einen auf *Form1* befindlichen *Button* soll der Inhalt einer auf *Form2* befindlichen *TextBox* in einem *Label* auf *Form1* angezeigt werden. Der folgende Code:

```
public partial class Form1 : Form
{
 ...
    private void button1_Click(object sender, EventArgs e)
    {
        Form2 f2 = new Form2();
        // f2.Show();          // nicht notwendig!
        label1.Text = f2.textBox1.Text;
    }
}
```

... funktioniert nur dann, wenn Sie vorher in *Form2* die *Modifiers*-Eigenschaft von *textBox1* auf *public*, *internal* oder *protected internal* gesetzt haben.

Ergebnis



6.4 Allgemeine Ereignisse von Komponenten

Wir wollen uns zunächst nur auf die Ereignisse beschränken, die für die meisten Komponenten gleichermaßen zutreffen, und auf einige grundsätzliche Programmieransätze eingehen.

6.4.1 Die Eventhandler-Argumente

Jedem Ereignis-Handler werden zumindest zwei Parameter übergeben:

- das *sender*-Objekt,
- die eigentlichen Parameter als Objekt mit dem kurzen Namen *e*.

Was können wir mit beiden Werten anfangen?

6.4.2 Sender

Da Ereignis-Handler einen beliebigen Namen erhalten und auch „artfremde" Komponenten sich ein und denselben Eventhandler teilen können, muss es ein Unterscheidungsmerkmal für den Aufrufer des Ereignisses geben. Über den *sender* ist es möglich, zu entscheiden, welches Objekt das Ereignis ausgelöst hat.

Zunächst werden Sie sicher enttäuscht sein, wenn Sie sich die einzelnen Eigenschaften bzw. Methoden des *sender*-Objekts ansehen. Lediglich eine Methode *GetType* ist zu finden. Doch keine Sorge, typisieren Sie das Objekt, so haben Sie Zugriff auf alle objekttypischen Eigenschaften und Methoden der das Ereignis auslösenden Komponente.

Beispiel 6.26: Ein gemeinsamer Ereignis-Handler *(TextChanged)* für mehrere Textboxen zeigt in der Kopfzeile des Formulars den Inhalt der jeweils bearbeiteten Textbox an.

C#

```
private void textBox_TextChanged(object sender, EventArgs e)
{
    this.Text = (sender as TextBox).Text;
}
```

Im obigen Beispiel wurde der *as*-Operator zur Typkonvertierung eingesetzt. Aber auch der übliche explizite *()*-Operator ist anwendbar[2].

[2] Im Unterschied zum *()*-Konvertierungsoperator ist der *as*-Operator nur für Referenz- bzw. Verweistypen anwendbar, wozu auch alle Controls gehören.

Beispiel 6.27: Vorgängerbeispiel mit dem *()*-Konvertierungsoperator

C#

```
private void textBox_TextChanged(object sender, EventArgs e)
{
    this.Text = ((TextBox) sender).Text;
}
```

Im Folgenden werden wir aber den *as*-Operator bevorzugen, da dieser *null* liefert, falls die Konvertierung misslingt. Beim *()*-Konvertierungsoperator würde hingegen eine Exception ausgelöst.

Das wäre zum Beispiel der Fall, wenn ein *TextChange* von einer *TextBox* und einer *ComboBox* kommt. Spätestens beim Testen der Anwendung gäbe es Ärger, da der Typ nicht immer stimmt. Mit dem *as*-Operator hingegen ist eine vorangestellte *null*-Abfrage möglich.

Beispiel 6.28: Vorgängerbeispiel mit *as*-Konvertierungsoperator und *null*-Abfrage

C#

```
private void control_TextChanged(object sender, EventArgs e)
{
    if (sender as TextBox != null)
          this.Text = (sender as TextBox).Text;
}
```

Brauchen Sie unbedingt den speziellen Objekttyp, können Sie zunächst mittels *GetType()* den Typ von *sender* feststellen, um dann, in Abhängigkeit vom Typ, unterschiedliche Aktionen auszuführen.

Beispiel 6.29: Verwendung von *sender*

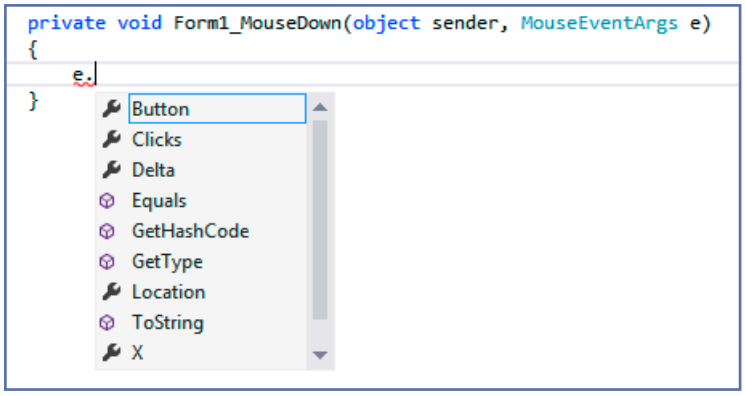
C#

Auch den folgenden Eventhandler teilen sich mehrere *Text*- und *ComboBox*en gemeinsam. Ändert sich der Inhalt einer *TextBox*, so wird dieser in der Titelleiste des Formulars angezeigt. Ändert sich aber die *Text*-Eigenschaft einer *ComboBox*, so wird nur deren Name angezeigt.

```
private void control_TextChanged(object sender, EventArgs e)
{
    switch (sender.GetType().Name)
    {
        case "TextBox": this.Text = (sender as TextBox).Text; break;
        case "ComboBox": this.Text = (sender as ComboBox).Name; break;
    }
}
```

6.4.3 Der Parameter e

Was sich hinter dem Parameter *e* versteckt, ist vom jeweiligen Ereignis abhängig. So werden bei einem *MouseDown*-Ereignis unter anderem die gedrückten Tasten und die Koordinaten des Mausklicks im Parameter *e* übergeben:



Teilweise werden über diesen Parameter auch Werte an das aufrufende Programm zurückgegeben.

Beispiel 6.30: Das Schließen des Formulars wird verhindert.

C#

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    e.Cancel = true;
}
```

6.4.4 Mausereignisse

Wenn sich der Mauscursor über einem Objekt befindet, können die folgenden Mausaktivitäten (teilweise mit Übergabeparametern) ausgewertet werden:

Ereignis	... tritt ein, wenn
Click	... auf das Objekt geklickt wird.
DblClick	... auf das Objekt doppelt geklickt wird.
MouseDown	... eine Maustaste niedergedrückt wird.
MouseUp	... eine Maustaste losgelassen wird.
MouseMove	... die Maus bewegt wird.
MouseEnter	... wenn die Maus in das Control hineinbewegt wird.
MouseLeave	... wenn die Maus aus dem Control hinausbewegt wird.

Bei einem *MouseDown* können Sie über *e.Button* unterscheiden, welcher Button gerade gedrückt wurde (*Left, Middle, Right*). Gleichzeitig können Sie über *e.x* bzw. *e.y* die Koordinaten bezüglich des jeweiligen Objekts ermitteln.

HINWEIS: Beachten Sie, dass jeder Doppelklick auch ein „normales" Klickereignis auslöst. Man sollte deshalb überlegt zu Werke gehen, wenn für ein Control beide Events gleichzeitig besetzt werden sollen.

Beispiel 6.31: *MouseDown*

C#

Beim Niederdrücken der rechten Maustaste über dem Formular wird eine *MessageBox* erzeugt, wenn sich die Maus in dem durch die Koordinaten 10,10 (linke obere Ecke) und 110,110 (rechte untere Ecke) bezeichneten Rechteck befindet:

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    if ((e.Button == MouseButtons.Right) &
            new Rectangle(10, 10, 100, 100).Contains(e.X, e.Y))
MessageBox.Show("Erfolg");
}
```

Beispiel 6.32: Ändern der *ListBox*-Hintergrundfarbe, wenn die Maus darüber bewegt wird.

C#

```
private void listBox1_MouseEnter(object sender, EventArgs e)
{
    (sender as ListBox).BackColor = Color.Blue;
}

private void listBox1_MouseLeave(object sender, EventArgs e)
{
    (sender as ListBox).BackColor = Color.White;
}
```

Tastaturereignisse

Wenn ein Steuerelement den Fokus hat, können für dieses Objekt die in der folgenden Tabelle aufgelisteten Keyboard-Events ausgewertet werden.

Ereignis	... tritt ein, wenn
KeyPress	... eine Taste gedrückt wird.
KeyDown	... die Taste nach unten bewegt wird (mit Intervall).
KeyUp	... eine Taste losgelassen wird.

KeyPress registriert das Zeichen der gedrückten Taste, während *KeyDown* und *KeyUp* auf alle Tasten der Tastatur (einschließlich Funktionstasten und Tastenkombinationen mit den Tasten *Umschalt*, *Alt* und *Strg*) reagieren können.

Beispiel 6.33: Verwendung *KeyUp*

C#

Beim Loslassen einer Zifferntaste innerhalb einer *TextBox* wird die Ziffer in eine *ListBox* übernommen (48 ... 57 sind die ANSI-Codes der Ziffern 0 ... 9).

```
private void textBox1_KeyUp(object sender, KeyEventArgs e)
{
    if ((e.KeyValue > 47) & (e.KeyValue < 58))
        listBox1.Items.Add((e.KeyValue-48).ToString());
}
```

6.4.5 KeyPreview

KeyPreview ist kein Ereignis, sondern eine Formulareigenschaft! *KeyPreview* steht aber im engen Zusammenhang mit den Tastaturereignissen und soll deshalb (ausnahmsweise) bereits an dieser Stelle erwähnt werden. Wenn *KeyPreview* den Wert *false* hat (Voreinstellung), werden die Ereignisse sofort zur Komponente weitergegeben. Hat *KeyPreview* aber den Wert *true*, so gehen, unabhängig von der aktiven Komponente, die Tastaturereignisse *KeyDown*, *KeyUp* und *KeyPress* zuerst an das Formular. Erst danach wird das Tastaturereignis an das Steuerelement weitergereicht. Damit kann an zentraler Stelle auf Tastaturereignisse reagiert werden.

Beispiel 6.34: *KeyPreview*

C#

Entsprechend dem vorhergehenden Beispiel soll beim Drücken einer Zifferntaste diese in eine *ListBox* übernommen werden, in diesem Fall jedoch bei **allen** Controls des aktuellen Fensters.

Setzen Sie im Eigenschaftenfenster (*F4*) die Eigenschaft *KeyPreview* des Formulars auf *true* und erzeugen Sie folgenden Event-Handler:

```
private void Form1_KeyUp(object sender, KeyEventArgs e)
{
    if ((e.KeyValue > 47) & (e.KeyValue < 58))
        listBox1.Items.Add((e.KeyValue - 48).ToString());
    e.Handled = true;
}
```

HINWEIS: Da in diesem Beispiel bei *KeyPreview=true* das Ereignis nur im Event-Handler des Formulars und nicht auch im aktuellen Steuerelement verarbeitet werden soll, haben wir *e.Handled* auf *true* gesetzt.

6.4.6 Weitere Ereignisse

Die folgenden Events finden Sie ebenfalls bei einer Vielzahl von Objekten:

Ereignis	... tritt ein, wenn
Change	... der Inhalt der Komponente geändert wird.
Enter	... die Komponente den Fokus erhält.
DragDrop	... das Objekt über der Komponente abgelegt wird.
DragOver	... das Objekt über die Komponente gezogen wird.
HelpRequested	... die Hilfe angefordert wird (*F1*).
Leave	... die Komponente den Fokus verliert.
Paint	... das Steuerelement gezeichnet wird.
Resize	... die Komponente in der Größe verändert wird.
Validate	... der Inhalt von Steuerelementen überprüft wird.

Beispiel 6.35: Erhält die *TextBox* den Eingabefokus, soll sich die Hintergrundfarbe ändern.

C#

```
private void textBox1_Enter(object sender, EventArgs e)
{
    textBox1.BackColor = Color.Yellow;
}
```

Beim Verlassen stellen wir die normale Farbe ein:

```
private void textBox1_Leave(object sender, EventArgs e)
{
    textBox1.BackColor = Color.White;
}
```

Im Zusammenhang mit dem Auftreten der Fokus-Ereignisse spielt häufig auch die Reihenfolge eine Rolle:

Enter → *GotFocus* → *Leave* → *Validating* → *Validated* → *LostFocus*

6.4.7 Validätsprüfungen

Das *Validate*-Ereignis ermöglicht es, zusammen mit der *CausesValidation*-Eigenschaft den Inhalt von Steuerelementen zu prüfen, **bevor** der Fokus das Steuerelement verlässt. Das *Validate*-Ereignis eignet sich besser zum Überprüfen der Dateneingabe als das *LostFocus*-Ereignis, da *LostFocus* erst **nach** dem Verschieben des Fokus eintritt.

Validate wird nur dann ausgelöst, wenn der Fokus in ein Steuerelement wechselt, bei dem die *CausesValidation*-Eigenschaft *true* ist (Standardeinstellung). *CausesValidation* sollte nur bei den Controls auf *false* gesetzt werden, deren Aktivierung keine Validätskontrolle auslösen soll, wie z. B. eine *Abbrechen*- oder eine *Hilfe*-Schaltfläche.

HINWEIS: Ist die Prüfung innerhalb des *Validating*-Events nicht erfolgreich, können Sie mit *e.Cancel* die weitere Ereigniskette (siehe oben) abbrechen.

Beispiel 6.36: *Validating*

C#

Der Fokus wandert nur dann zum nächsten Steuerelement, wenn in die *TextBox* mehr als fünf Zeichen eingegeben werden.

```
private void textBox2_Validating(object sender, CancelEventArgs e)
{
    if (textBox2.Text.Length < 5)
    {
        MessageBox.Show("Bitte mehr als 5 Zeichen eingeben!");
        e.Cancel = true;
    }
}
```

6.4.8 SendKeys

Mit diesem Objekt werden Tastatureingaben durch den Bediener simuliert, daher ist es zweckmäßig, es bereits an dieser Stelle im Zusammenhang mit Tastaturereignissen zu erwähnen. Zwei Methoden stehen zur Auswahl:

- *Send*
- *SendWait*

Während Erstere sich damit begnügt, die Tastatureingaben einfach an die aktive Anwendung zu senden, wartet *SendWait* auch darauf, dass die Daten verarbeitet werden. Insbesondere bei etwas langsameren Operationen kann es sonst schnell zu einem Fehlverhalten kommen.

Das Argument der beiden Methoden ist eine Zeichenkette. Jede Taste wird dabei durch mindestens ein Zeichen repräsentiert.

HINWEIS: Das Pluszeichen (+), Caret-Zeichen (^) und Prozentzeichen (%) sind für die UMSCHALT-, STRG- und ALT-Taste vorgesehen. Sondertasten sind in geschweifte Klammern einzuschließen.

Beispiel 6.37: *SendKeys*

C#

Die folgende Anweisung sendet die Tastenfolge *Alt+F4* an das aktive Fenster und bewirkt damit ein Schließen der Applikation.

```
private void button1_Click(object sender, EventArgs e)
{
    SendKeys.Send('%{F4}');
}
```

Häufig soll sich die „Tastatureingabe“ nicht auf das aktuelle Formular, sondern auf das aktive Steuerelement beziehen. Dann muss dieses Steuerelement vorher den Fokus erhalten.

Beispiel 6.38: *SendWait* mit *SetFocus*

C#

Die folgende Sequenz füllt das Textfeld *TextBox1* mit den Ziffern 12345678 und setzt danach die Ausführung fort.

```
private void button1_Click(object sender, EventArgs e)
{
    textBox1.Focus();
    SendKeys.SendWait('12345678');
}
```

HINWEIS: *SendKeys* macht es auch möglich, quasi „wie von Geisterhand“ andere Windows-Programme (z. B. den integrierten Taschenrechner) aufzurufen.

6.5 Allgemeine Methoden von Komponenten

Auch hier nur ein Auszug aus dem reichhaltigen Sortiment:

Methode	Erläuterung
Contains	... kontrolliert, ob ein angegebenes Steuerelement dem aktuellen untergeordnet ist
CreateGraphics	... erzeugt ein *Graphics*-Objekt zum Zeichnen im Steuerelement (mehr dazu in Kapitel 27)
DoDragDrop	... beginnt eine Drag&Drop-Operation
FindForm	... ruft das übergeordnete Formular ab
Focus	... setzt den Fokus auf das Objekt

(Fortsetzung nächste Seite)

(Fortsetzung)

Methode	Erläuterung
GetContainerControl	... ruft das übergeordnete Steuerelement ab
GetType	... ruft den Typ des Steuerelements ab
Hide	... verbirgt das Steuerelement
Invalidate	... veranlasst das Neuzeichnen des Controls
Refresh	... erneuert den Aufbau des Steuerelements
Scale	... skaliert das Steuerelement
SelectNextControl	... aktiviert das folgende Steuerelement
SetBounds	... setzt die Größe des Steuerelements
Show	... zeigt das Steuerelement nach dem Verbergen wieder an

HINWEIS: Über die an die Methoden zu übergebenden Parameter informieren Sie sich am besten per IntelliSense bzw. in der Dokumentation.

Beispiel 6.39: *SetBounds*

C#

Die folgende Anweisung verschiebt das aktuelle Formular an die Position 10,10 und verändert gleichzeitig die Größe auf 100 x 100 Pixel.

```
this.SetBounds(10, 10, 100, 100);
```

Beispiel 6.40: Skalieren einer *GroupBox* mit dem Faktor 1,5

C#

```
groupBox1.Scale(1.5, 1.5);
```

7 Windows Forms-Formulare

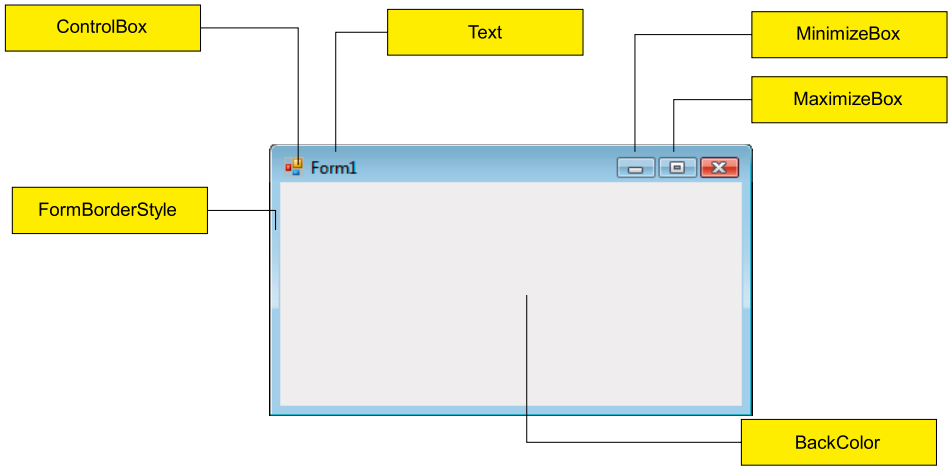
Fast jede Windows-Applikation läuft in einem oder mehreren Fenstern ab, die sozusagen als Container für die Anwendung fungieren. Schon deshalb ist das Formular (*Form*) das wichtigste Objekt, wir werden uns also damit etwas ausführlicher auseinandersetzen müssen.

HINWEIS: Die Ausführungen zu den allgemeinen Eigenschaften/Methoden und Ereignissen aus dem vorhergehenden Kapitel treffen auch auf die Windows-Formulare zu. Sie sollten also das vorhergehende Kapitel bereits durchgearbeitet haben.

7.1 Übersicht

Haben Sie ein neues Windows Forms-Anwendungsprojekt geöffnet, werden Sie bereits von einem ersten Formular im Designer begrüßt.

Wie ein Formular aussieht, brauchen wir sicher nicht weiter zu erklären. Welche Eigenschaften jedoch für das Aussehen verantwortlich sind, soll die folgende Abbildung verdeutlichen:



Für die weitere Orientierung sollen die folgenden Tabellen sorgen, die in Kürze die wichtigsten Eigenschaften, Methoden und Ereignisse des Formulars erläutern. In den weiteren Abschnitten wenden wir uns dann spezifischen Themen rund um das Formular zu.

7.1.1 Wichtige Eigenschaften des Form-Objekts

Eigenschaft	Beschreibung
AcceptButton *CancelButton*	Diese Eigenschaften legen fest, welche Buttons mit den Standardereignissen *Enter*-Taste bzw. *Esc*-Taste verknüpft werden. Diese Funktionalität wird häufig in Dialogboxen verwendet (siehe auch *DialogResult*). Löst der Nutzer eine der beiden Aktionen aus, wird der Ereigniscode der jeweiligen Taste verarbeitet.
ActiveControl	... bestimmt das gerade aktive Control, z. B.: `if (sender == this.ActiveControl) berechne();`
ActiveForm	... bestimmt das aktive Formular der Anwendung
ActiveMdiChild	... ermittelt das gerade aktive MDI-Child-Fenster
AutoScroll	... bestimmt, ob das Formular automatisch Scrollbars einfügen soll, wenn der Clientbereich nicht komplett darstellbar ist
BackgroundImage	... eine Grafik für den Hintergrund
ClientSize	... ermittelt die Größe des Formular-Clientbereichs
ContextMenu	... das Kontextmenü des Formulars
ControlBox	... Anzeige des Systemmenüs (*true*/*false*)
Controls	... eine Collection aller enthaltenen Controls
Cursor	... legt die Cursorform für das aktuelle Formular fest

Eigenschaft	Beschreibung
DesktopBounds	... legt Position und Größe des Formulars auf dem Desktop fest, z. B. `this.DesktopBounds = (new Rectangle.Create(10, 10, 100, 100));`
DesktopLocation	... legt die Position des Formulars fest

DialogResult	... über diesen Wert können Dialog-Statusinformationen an ein aufrufendes Programm zurückgegeben werden, z. B.
	`Form2 f2 = new Form2();`
	`if (f2.ShowDialog() == DialoResult.Abort) { ...}`

DoubleBuffered	... ermöglicht flackerfreie Darstellung von Grafiken (siehe Kapitel 8)
Dock	... setzt die Ausrichtung gegenüber einem übergeordneten Fenster
DockPadding	... setzt den Zwischenraum beim Docking von Controls
FormBorderStyle	... setzt den Formularrahmen. Dieser hat auch Einfluss auf das Verhalten (Tool-Window, Dialog etc.).
HelpButton	... soll der Hilfe-Button angezeigt werden?
Icon	... das Formular-Icon
IsMdiChild	... handelt es sich um ein MDI-Child-Fenster?
IsMdiContainer	... handelt es sich um ein MDI-Container-Fenster?
Location	... die linke obere Ecke des Formulars.
MaximizeBox *MinimizeBox*	... Anzeige der beiden Formular-Buttons (*true/false*)
MaximumSize *MinimumSize*	... setzt maximale bzw. minimale Maße für das Fenster
MdiChildren	... eine Collection der untergeordneten MDI-Child-Fenster
MdiParent	... ermittelt den MDI-Container
Menu	... das Hauptmenü des Formulars
Modal	... wird das Formular (Dialog) modal angezeigt?
Opacity	... Transparenz des Formulars in Prozent
OwnedForms	... eine Collection der untergeordneten Formulare
Owner	... das übergeordnete Formular
ShowInTaskbar	... soll das Formular in der Taskbar angezeigt werden?
Size	... die Formulargröße
StartPosition	... wo wird das Fenster beim ersten Aufruf angezeigt (zentriert etc.)?
Text	... der Text in der Titelleiste
TopMost	... soll das Formular an oberster Position angezeigt werden?
TransparencyKey	... welche Formularfarbe soll transparent dargestellt werden?
Visible	... ist das Formular sichtbar?
WindowState	... ist das Fenster maximiert, minimiert oder normal dargestellt?

Controls-Auflistung

Mit dieser Eigenschaft ist der Zugriff auf alle im Formular vorhandenen Steuerelemente möglich.

Beispiel 7.1: Alle Controls im aktuellen Formular um zehn Pixel nach links verschieben

C#

```
foreach (Control c in this.Controls)
{
    c.Left -= 10;
}
```

Der Zugriff auf Elemente der *Controls*-Auflistung eines Formulars ist nicht nur über den Index, sondern auch über den Namen des Elements möglich.

Beispiel 7.2: Zugriff auf ein spezielles Control über dessen Namen

C#

```
TextBox tb = (TextBox) Controls["textBox2"];
MessageBox.Show(tb.Text);
```

7.1.2 Wichtige Ereignisse des Form-Objekts

Neben den bereits im vorhergehenden Kapitel aufgelisteten Events sind für ein Formular die folgenden Events von Bedeutung:

Ereignis	... tritt ein, wenn
Activated	... das Formular aktiviert wird.
FormClosing	... das Formular geschlossen werden soll. Sie können den Vorgang über den Parameter *e.Cancel* abbrechen und so ein Schließen des Formulars verhindern.
FormClosed	... das Formular geschlossen ist.
Deactivate	... ein anderes Formular aktiviert wird.
Load	... das Formular geladen wird.
Paint	... das Formular neu gezeichnet werden muss.
Resize	... die Größe eines Formulars verändert wird.
HelpRequested	... Hilfe vom Nutzer angefordert wird (*F1*).
Layout	... die untergeordneten Controls neu positioniert werden müssen (Größenänderung des Formulars oder Hinzufügen von Steuerelementen).
LocationChanged	... sich die Position des Formulars ändert.
MdiChildActivated	... wenn ein MDI-Child-Fenster aktiviert wird.

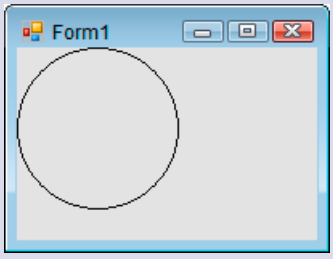
Beispiel 7.3: Verwenden des *Paint*-Events zum Zeichnen eines Kreises

C#

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.DrawEllipse(new Pen(Color.Black), 0, 0, 100, 100);
}
```

Ergebnis

Mit jedem Anzeigen oder jeder Größenänderung wird *Paint* ausgelöst. Das Ergebnis:



HINWEIS: Mehr zur Grafikprogrammierung finden Sie in Kapitel 9.

Beim Laden eines Formulars treten – nach Aufruf des Konstruktors – die Ereignisse in folgender Reihenfolge auf:

Move → *Load* → *Layout* → *Activated* → *Paint*

Beim Schließen hingegen haben wir es mit folgender Ereigniskette zu tun:

FormClosing → *FormClosed* → *Deactivate*

HINWEIS: Das Praxisbeispiel in Abschnitt 7.4.2 zeigt, wie Sie obige Ereignisketten selbst experimentell ermitteln können!

7.1.3 Wichtige Methoden des Form-Objekts

Die wichtigsten Methoden für Formulare sind im Folgenden zusammengestellt.

Methode	Beschreibung
Activate	... aktiviert das Formular
BringToFront	... verschiebt das Formular an die oberste Position (innerhalb der Anwendung)
Close	... schließt das Formular
CreateControl	... erzeugt ein neues Steuerelement

(Fortsetzung nächste Seite)

(Fortsetzung)

Methode	Beschreibung
CreateGraphics	... erstellt ein *Graphics*-Objekt für die grafische Ausgabe
DoDragDrop	... startet eine Drag & Drop-Operation
Focus	... setzt den Fokus auf das Formular
GetNextControl	... liefert das folgende Control in der Tab-Reihenfolge
Hide	... verbirgt das Formular
Invalidate	... erzwingt ein Neuzeichnen des Formularinhalts
PointToClient *RectangleToClient*	... rechnet Screen-Koordinaten in Fensterkoordinaten um (je nach Fensterposition)
PointToScreen *RectangleToScreen*	... rechnet Fensterkoordinaten in Screen-Koordinaten um
Refresh	... erzwingt ein Neuzeichnen des Fensters und der untergeordneten Controls
SelectNextControl	... verschiebt den Eingabefokus bei den untergeordneten Controls
Show	... zeigt das Fenster an
ShowDialog	... zeigt das Fenster als Dialogbox (modal) an

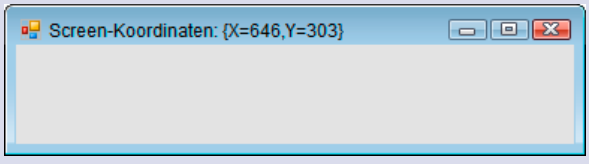
Beispiel 7.4: Umrechnen in Screen-Koordinaten

C#

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    this.Text = "Screen-Koordinaten: " + this.PointToScreen(new Point(e.X,
e.Y)).ToString();
}
```

Ergebnis

Die Anzeige hängt jetzt nicht nur von der relativen Mausposition, sondern auch von der absoluten Position des Formulars ab:



HINWEIS: Statt des Names des Formulars (z. B. *Form1*) können Sie auch *this* verwenden oder den Bezeichner völlig weglassen, da der Code ja in der aktuellen Klasse ausgeführt wird.

7.2 Praktische Aufgabenstellungen

Im Folgenden wollen wir die oben genannten Eigenschaften, Methoden und Ereignisse nutzen, um einige recht praktische Aufgabenstellungen im Umgang mit Windows-Formularen zu lösen.

7.2.1 Fenster anzeigen

Wie Sie aus der Startprozedur *Main()* heraus ein Fenster aufrufen, wurde im vorhergehenden Kapitel beschrieben. Wie Sie aus dem Hauptfenster heraus weitere Formulare aufrufen, soll Mittelpunkt dieses Abschnitts sein.

Zwei grundsätzliche Typen von Formularen müssen Sie unterscheiden:

- modale Fenster (Dialoge),
- nichtmodale Fenster.

Die Unterscheidung zwischen beiden Varianten findet erst beim Aufruf bzw. bei der Anzeige eines *Form*-Objekts statt. Zur Entwurfszeit wird diese Unterscheidung nicht getroffen, sieht man einmal von unterschiedlichen Rahmentypen (*FormBorderStyle*-Eigenschaft) ab.

Nichtmodale Fenster

Hierbei handelt es sich um ein Fenster, das den Fokus auch an andere Fenster abgeben bzw. das auch verdeckt werden kann.

Die Anzeige erfolgt mithilfe der Methode *Show*, die asynchron ausgeführt wird, das heißt, es wird mit der Programmverarbeitung nicht auf das Schließen des Formulars gewartet.

HINWEIS: Vor der Anzeige des Formulars muss dieses mit *new* instanziiert werden!

Beispiel 7.5: Instanziieren und Anzeigen eines weiteren Formulars

C#

```
private void button1_Click(object sender, EventArgs e)
{
    Form2 f2 = new Form2();
    f2.Text = "Mein zweites Formular";
    f2.Show();
}
```

Der „Nachteil“ dieser Fenster: Sie können zum einen verdeckt werden, zum anderen wissen Sie als Programmierer nie, wann der Anwender das Fenster schließt. Für die Eingabe von Werten und deren spätere Verarbeitung sind sie also ungeeignet.

Modale Fenster (Dialoge)

Abhilfe schaffen die Dialogfenster, auch modale Fenster genannt. Diese werden statt mit *Show* mit der Methode *ShowDialog* angezeigt. Die Programmausführung wird mit dem Aufruf der Methode an dieser Stelle so lange gestoppt, bis der Nutzer das Formular wieder geschlossen hat.

Beispiel 7.6: Anzeige eines Dialogfensters

C#

```
Form2 f2 = new Form2();
f2.Text = "Bitte tragen Sie Ihren Namen ein ...";
f2.ShowDialog();
MessageBox.Show(f2.textBox1.Text);  // Anzeige des Eingabewerts
```

HINWEIS: Bevor Sie auf Controls in *Form2* zugreifen können, müssen Sie deren *Modifiers*-Eigenschaft auf *public* festgelegt haben. Andernfalls können Sie nicht mit den Controls arbeiten.

Zu einer ordentlichen Dialogbox gehören im Allgemeinen auch ein OK- und ein Abbruch-Button. Auf diese Weise kann im aufrufenden Programm schnell entschieden werden, welcher Meinung der Anwender beim Schließen der Dialogbox war. Von zentraler Bedeutung ist in diesem Fall der Rückgabewert der Methode *ShowDialog*.

Beispiel 7.7: Auswerten des Rückgabewerts

C#

```
Form2 f2 = new Form2();
if (f2.ShowDialog() == DialogResult.Abort)
    {...}
```

Die möglichen Rückgabewerte:

DialogResult
Abort
Cancel
Ignore
No
None
OK
Retry
Yes

In der Dialogbox selbst stellen Sie den Rückgabewert entweder durch das direkte Setzen der Eigenschaft *DialogResult* ein oder Sie weisen den beiden Buttons (OK, Abbruch) die gewünschte *DialogResult*-Eigenschaft zu.

Beispiel 7.8: Verwendung von *DialogResult*

C#

Auf *Form1* befinden sich ein *Label* und ein *Button*. Über Letzteren wird ein Dialogfenster *Form2* aufgerufen, das die Schaltflächen „OK“ und „Abbruch“ besitzt. Außerdem hat *Form2* eine *TextBox*, deren *Modifiers*-Eigenschaft Sie auf *public* setzen.

```
public partial class Form1 : Form
{
    private void button1_Click(object sender, EventArgs e)
    {
        Form2 f2 = new Form2();
        f2.Text = "Bitte tragen Sie Ihren Namen ein ... ";
        if (f2.ShowDialog() == DialogResult.OK) label1.Text = f2.textBox1.Text;
    }
}
```

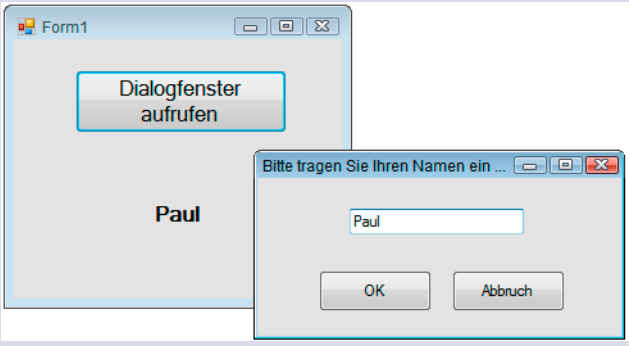
Der Code von *Form2*:

```
public partial class Form2 : Form
{
    public Form2()
    {
        InitializeComponent();

        button1.DialogResult = DialogResult.OK;
        button2.DialogResult = DialogResult.Abort;
    }
}
```

Die folgenden beiden Zuweisungen können Sie auch direkt im Eigenschaftenfenster von *button1* bzw. *button2* vornehmen:

Ergebnis

Nach dem Klick auf „OK“ wird der in *Form1* eingetragene Wert in *Form2* angezeigt.



7.2.2 Splash Screens beim Anwendungsstart anzeigen

Von vielen kommerziellen Anwendungen ist Ihnen sicher die Funktion eines Splash Screens bekannt. Der Hintergrund ist in vielen Fällen, dass die Zeit für das Laden von Programmmodulen, Datenbanken etc. für den Endanwender irgendwie sinnvoll überbrückt werden soll, ohne dass der Verdacht aufkommt, die Anwendung „hängt“. An einem recht einfachen Vertreter dieser Gattung wollen wir Ihnen die prinzipielle Vorgehensweise demonstrieren.

Beispiel 7.9: Einsatz eines Splash Screens

C#

Erstellen Sie zunächst ein neues Projekt und fügen Sie diesem neben dem Standardformular *Form1* ein weiteres Formular *SplashScreen* hinzu. Diesem gilt auch zunächst unsere Aufmerksamkeit.

Fügen Sie *SplashScreen* einen *Label* für die Begrüßungsmeldung und einen *Label* für mögliche Statusmeldungen hinzu. Setzen Sie weiterhin die *FormBorderStyle*-Eigenschaft auf den von Ihnen favorisierten Wert (z. B. *None*) sowie die Eigenschaft *StartPosition* auf *ScreenCenter*. Nicht vergessen dürfen wir auch die Eigenschaft *TopMost*, die wir tunlichst auf *true* setzen sollten, damit unser Formular auch im Vordergrund steht und nicht hinter dem Hauptformular der Anwendung verloren geht.

Die Klassendefinition erweitern Sie bitte um die im Folgenden fett hervorgehobenen Einträge:

```
...
    public partial class SplashScreen : Form
    {
```

Da wir nur ein Formular benötigen, definieren wir dieses gleich intern und statisch:

```
        static SplashScreen frmSplashScreen = null;
```

Mit der folgenden *Start*-Methode wird das Formular initialisiert und angezeigt. Da die Methode statisch ist, können wir diese direkt mit *SplashScreen.Start()* aufrufen.

```
        static public void Start()
        {
            if (frmSplashScreen != null)
                return;
            frmSplashScreen = new SplashScreen();
            frmSplashScreen.Show();
        }
```

Auch das Schließen des Formulars übernimmt eine statische Methode der *SplashScreen*-Klasse:

```
        static public void Stop()
        {
            frmSplashScreen.Close();
            frmSplashScreen = null;
        }
```

Möchten Sie Statusmeldungen anzeigen, ist dies über die statische Methode *SetMessage* möglich. Die Meldungen werden im *label2* angezeigt:

```
        static public void SetMessage(string msg)
        {
           frmSplashScreen.label2.Text = msg;
           Application.DoEvents();
        }
...
```

Selbstverständlich können Sie auch noch Fortschrittsanzeigen etc. in diesem Formular unterbringen, das Grundprinzip dürfte jedoch weitgehend gleich bleiben.

Was fehlt, ist die Integration des neuen Splash Screens in Ihre Anwendung. Öffnen Sie dafür die Datei *Program.cs* und nehmen Sie folgende Erweiterung vor:

```
    static class Program
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            SplashScreen.Start();
            Application.Run(new Form1());
        }
    }
```

Last but not least ist auch unser eigentliches Programm von Interesse. Wir wollen hektische und langwierige Datenbankoperationen beim Öffnen von *Form1* mit einer einfachen Schleife und der *Thread.Sleep*-Methode simulieren:

```
...
    public partial class Form1 : Form
    {
        private void Form1_Load(object sender, EventArgs e)
        {
            for (int i = 0; i < 100; i++)
            {
                Application.DoEvents();
                Thread.Sleep(50);
```

Hier besteht die Möglichkeit, den aktuellen Fortschritt im Splash Screen anzuzeigen:

```
                SplashScreen.SetMessage("Schritt " + i.ToString());
            }
```

Sind alle Ladeaktivitäten absolviert, sollten wir auch unseren Splash Screen wieder ausblenden:

```
            SplashScreen.Stop();
        }
```

Beim Start der Anwendung sollte jetzt zunächst unser Splash Screen erscheinen und anschließend *Form1*.

HINWEIS: Man kann sicher auch die Anzeige des Splash Screens in einen extra Thread auslagern, um die Aktualisierungen dieser Forms unabhängig von den anderen Aktivitäten der Anwendung zu realisieren. Die Alternative ist das regelmäßige Aufrufen der *DoEvents*-Methode, wie in obigem Beispiel demonstriert.

7.2.3 Eine Sicherheitsabfrage vor dem Schließen anzeigen

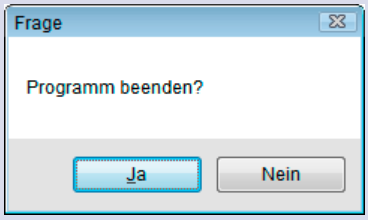
Für das Schließen der Anwendung bieten sich viele Möglichkeiten und so ist es sicher ratsam, die entsprechende Routine zentral zu organisieren. Dazu bietet sich das *FormClosing*-Ereignis an, wie es das folgende Beispiel zeigt.

Beispiel 7.10: Sicherheitsabfrage vor dem Schließen des Formulars

C#

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    e.Cancel = MessageBox.Show("Programm beenden?", "Frage",
MessageBoxButtons.YesNo) ==
                                DialogResult.No;
}
```

Ergebnis

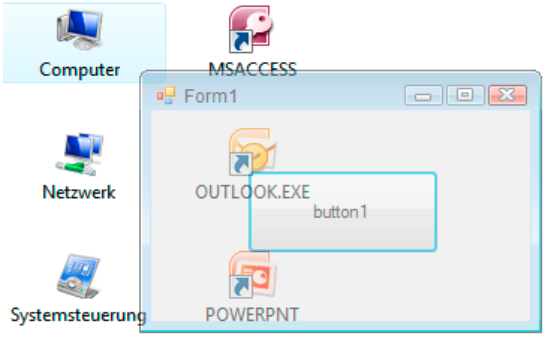


Die Neugier des Programmierers ist in vielen Fällen angebracht und so bietet der Parameter *e* ganz nebenbei auch Informationen darüber, warum das Ereignis ausgelöst wurde. Verantwortlich dafür ist der Member *CloseReason*, dessen einzelne Konstanten Sie der folgenden Tabelle entnehmen können:

Member	Beschreibung
`ApplicationExitCall`	Die Methode *Application.Exit* wurde im Programm aufgerufen.
`FormOwnerClosing`	Das Hauptformular wurde geschlossen.
`MdiFormClosing`	Das zentrale MDI-Form wurde geschlossen.
`None`	Hier weiß auch die API nicht weiter.
`TaskManagerClosing`	Der Taskmanager will die Anwendung schließen.
`UserClosing`	Eine Nutzeraktion (Formularschaltflächen) führt zum Schließen.
`WindowsShutDown`	Das Betriebssystem wird heruntergefahren.

7.2.4 Ein Formular durchsichtig machen

Sind Ihnen die bisherigen Formulare zu schlicht, können Sie Ihre Anwendung auch mit einem Transparenzeffekt aufpeppen. Setzen Sie dazu einfach die *Opacity*-Eigenschaft auf einen Wert zwischen 0 % (vollständige Transparenz) und 100 %. Das Ergebnis bei 50 % zeigt folgender Bildschirmausschnitt mit Blick auf den Desktop:

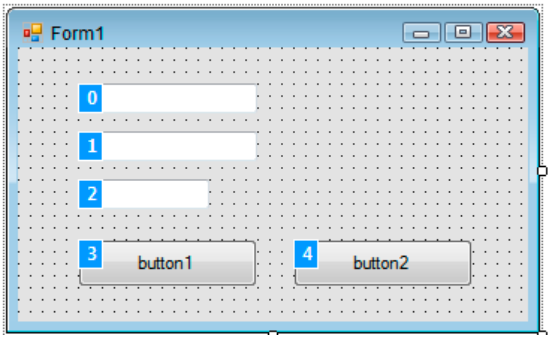


7.2.5 Die Tabulatorreihenfolge festlegen

Gerade bei Dialogboxen ist die Eingabereihenfolge von übergeordnetem Interesse. Was nützt dem Anwender eine Dialogbox, in der der Eingabefokus willkürlich zwischen den Text- und ComboBoxen hin und her springt?

Da der Wechsel von einem Eingabe-Control zum nächsten mit der Tabulatortaste erfolgt, spricht man auch von Tabulatorreihenfolge. Jedes sichtbare Steuerelement verfügt zu diesem Zweck über die Eigenschaften *TabIndex* und *TabStop*. Während mit *TabStop* lediglich festgelegt wird, ob das Control überhaupt den Fokus erhalten kann (mittels Tab-Taste), können Sie mit *TabIndex* Einfluss auf die Reihenfolge nehmen.

Visual Studio unterstützt Sie bei dieser Arbeit recht gut. Um den Überblick zu verbessern, können Sie die Tabulatorreihenfolge im Entwurfsmodus sichtbar machen. Aktivieren Sie diese über den Kontextmenüpunkt **Ansicht | Aktivierreihenfolge**. Ihr Formular dürfte danach zum Beispiel folgenden Anblick bieten:



Klicken Sie jetzt einfach in der gewünschten Reihenfolge in die kleinen Fähnchen, um die Tabulatorreihenfolge zu ändern.

7.2.6 Ausrichten von Komponenten im Formular

War es in der ersten Version von Visual Studio noch eine Qual bzw. ein riesiger Aufwand, Komponenten in einem Formular sauber auszurichten, stellt dies heutzutage kaum noch ein Problem dar. Zwei wesentliche Verfahren bieten sich an:

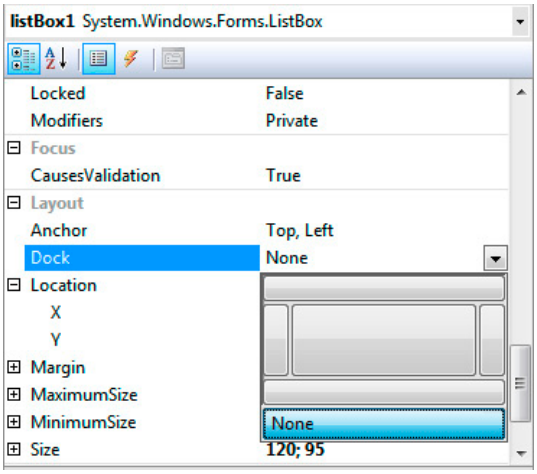
- **Docking** (Andocken an die Außenkanten einer übergeordneten Komponente)
- **Anchoring** (Ausrichten relativ zu den Außenkanten einer übergeordneten Komponente)

Verantwortlich für das Ausrichten sind die beiden naheliegenden Eigenschaften *Dock* und *Anchor*.

HINWEIS: Natürlich wollen wir der Vollständigkeit halber nicht vergessen, hier auch auf die Eigenschaften *Location* (linke obere Ecke) und *Size* (Breite, Höhe) hinzuweisen.

Dock

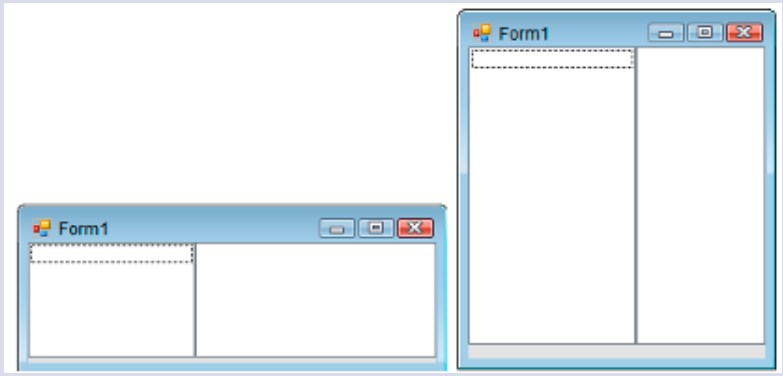
Öffnen Sie das Eigenschaftenfenster (F4) und wählen Sie die Eigenschaft *Dock*, steht Ihnen der Eigenschafteneditor zur Verfügung:



Eine Komponente lässt sich mit dieser Eigenschaft fest an den vier Außenkanten oder in der verbleibenden Clientfläche ausrichten. Dabei verändert sich die Größe der Komponente nur so, dass die Ausrichtung an den Außenkanten erhalten bleibt.

Beispiel 7.11: Ausrichten zweier Listboxen in einem Fenster

Die linke *ListBox* ändert lediglich ihre Höhe, die rechte *ListBox* füllt immer den gesamten freien Formbereich aus.



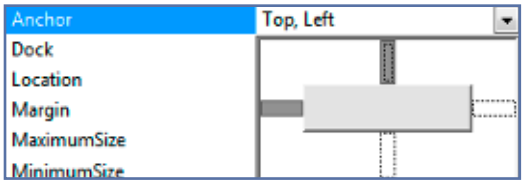
HINWEIS: Möchten Sie die Ausrichtung aufheben, wählen Sie im Eigenschaften-Editor die unterste Schaltfläche (*None*).

Damit stellt jetzt auch das Positionieren von Bildlaufleisten kein Problem mehr dar, einfach die Komponenten am rechten bzw. am unteren Rand ausrichten.

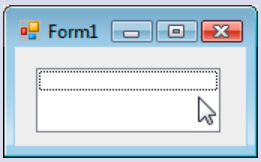
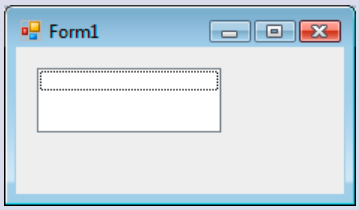
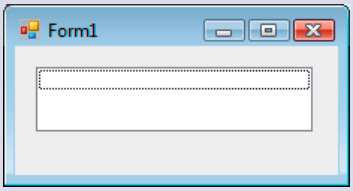
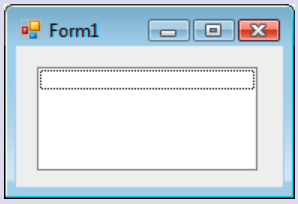
HINWEIS: Mit der Formulareigenschaft *Padding* können Sie einen Mindestabstand beim Docking vorgeben. Auf diese Weise lassen sich Ränder zu den Formularkanten definieren.

Anchor

Etwas anders als die *Dock*-Eigenschaft verhält sich die *Anchor*-Eigenschaft. Standardmäßig ist *Anchor* immer mit *Left*, *Top* aktiv, bei Größenänderungen des Formulars bleibt also der Abstand der Komponente zum linken und oberen Rand der umgebenden Komponente immer gleich. Auch hier steht ein eigener Eigenschafteneditor zur Verfügung:



Die folgenden Abbildungen zeigen Ihnen die Auswirkungen verschiedener *Anchor*-Einstellungen auf ein Formular:

Original-Fenstergröße	Fenster vergrößert (Anchor=Left,Top)
	
Fenster vergrößert (Anchor=Left,Top,Right)	**Fenster vergrößert (Anchor=Left,Top,Right,Bottom)**
	

Zur Laufzeit können Sie die *Anchor*-Eigenschaft mit den Werten der Enumeration *AnchorStyles* festlegen (siehe folgende Tabelle).

Konstante	Das Steuerelement ist ...
None	... nicht verankert.
Top	... am oberen Rand verankert.
Bottom	... am unteren Rand verankert.
Left	... am linken Rand verankert.
Right	... am rechten Rand verankert.

Da die *AnchorStyles*-Enumeration über ein *[Flags]*-Attribut verfügt, können die einzelnen Konstanten bitweise verknüpft werden.

Beispiel 7.12: Herstellen der Standardverankerung eines Buttons

C#

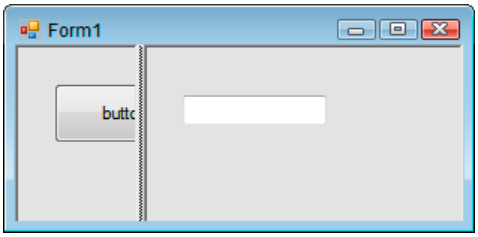
```
button1.Anchor = AnchorStyles.Left | AnchorStyles.Top;
```

7.2.7 Spezielle Panels für flexible Layouts

Auch hier geht es um die Bereitstellung eines flexiblen Formularlayouts. Allerdings haben wir es, im Unterschied zum vorhergehenden Abschnitt, mit keinen Formulareigenschaften mehr zu tun, sondern mit speziellen Panels, die Sie im „Container"-Segment der Toolbox vorfinden.

SplitContainer

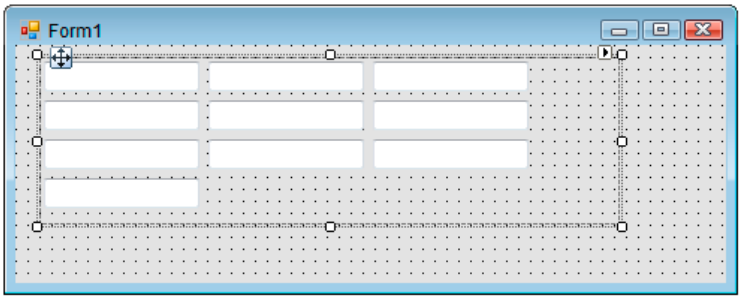
Diese Komponente gilt als Nachfolger für den *Splitter* und besteht aus zwei Panels, die durch einen zur Laufzeit veränderlichen Balken getrennt sind.



Wichtige Eigenschaften sind *Orientation* (horizontaler oder vertikaler Trennbalken), *IsSplitterFixed* (fester oder beweglicher Balken) und *BorderStyle* (in der Standardeinstellung *None* bleibt der Balken unsichtbar).

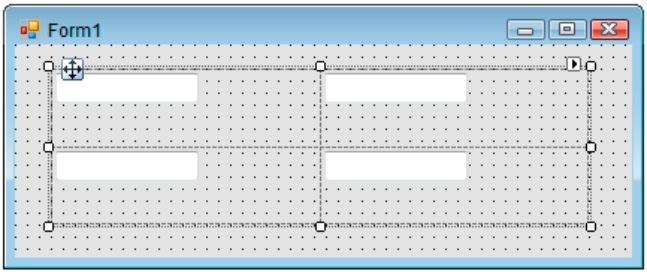
FlowLayoutPanel

Diese Komponente layoutet die auf ihr platzierten Steuerelemente dynamisch in horizontaler oder vertikaler Richtung. Das heißt, bei einer Größenänderung des Formulars werden die Steuerelemente automatisch „umgebrochen“ (`WrapContents = True`). Wichtig ist die *Anchor*-Eigenschaft, die die Ränder des Containers (`Top`, `Bottom`, `Left`, `Right`) bestimmt, an welche die Steuerelemente gebunden werden sollen.



TableLayoutPanel

Diese Komponente besorgt ein dynamisches Layout, das sich an einer Gitterstruktur orientiert. Wichtig sind die *Rows*- und die *Columns*-Eigenschaft, die über spezielle Dialoge (aufrufbar über den Smarttag rechts oben) den individuellen Bedürfnissen angepasst werden können.



7.2.8 Menüs erzeugen

An dieser Stelle wollen wir etwas vorgreifen, da *MenuStrip*- und *ContextMenuStrip*-Komponente (Nachfolger der veralteten *MainMenu*- und *ContextMenu*-Controls) eigentlich erst in das folgende Kapitel gehören, wo es um die Beschreibung der wichtigsten Steuerelemente geht. Doch die optische Verzahnung zwischen Menü und Formular ist so eng, dass wir bereits an dieser Stelle auf dieses Thema eingehen wollen.

Beide Komponenten haben gegenüber dem veralteten *MainMenu/ContextMenü* wesentlich verbesserte Features. So kann jedem Menüelement über seine *Image*-Eigenschaft auf einfache Weise eine Grafik zugewiesen werden. Menüeinträge können auch als *ComboBox*, *TextBox* oder *Separator* in Erscheinung treten.

MenuStrip

Möchten Sie ein „normales“ Menü erzeugen, platzieren Sie einfach eine *MenuStrip*-Komponente auf das Formular. Es handelt sich zunächst um eine nicht sichtbare Komponente, die lediglich im Komponentenfach zu sehen ist. Doch halt, auch in der Kopfzeile des Formulars tut sich etwas, sobald die Komponente markiert wird: Das Menü wird genau dort bearbeitet, wo es sich zur Laufzeit auch befindet. Klicken Sie also in den Text „Hier eingeben“ und tragen Sie beispielsweise „Datei“ ein. Automatisch werden bereits zwei weitere potenzielle Menüpunkte erzeugt (ein Menüpunkt auf der gleichen Ebene, ein untergeordneter Menüpunkt), die Sie auf die gleiche Weise bearbeiten können.

HINWEIS: Möchten Sie einen Trennstrich einfügen, genügt die Eingabe eines einzelnen Minuszeichens (-) als Beschriftung!

Beispiel 7.13: MenuStrip

C#

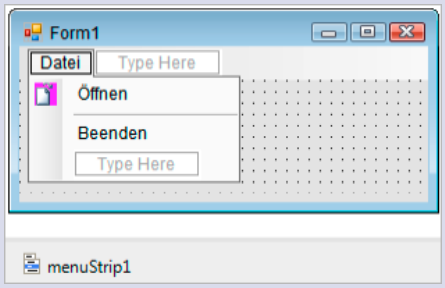
Die Abbildung am Ende des Beispiels zeigt ein Datei-Menü mit zwei Einträgen (*MenuItems*). Dazwischen befindet sich ein Trennstrich. Die kleine Grafik bei *Datei/Öffnen* wurde diesem Menüpunkt über dessen *Image*-Eigenschaft zugewiesen.

Um für einen Menüpunkt die Ereignisprozedur zu erzeugen, genügt ein Doppelklick und schon befinden Sie sich wieder im Code-Editor und programmieren die Funktionalität.

Die beiden Menüeinträge werden mit Code hinterlegt.

```
public partial class Form1 : Form
{
    ...
    private void öffnenToolStripMenuItem_Click(object sender, EventArgs e)
    {
        MessageBox.Show("Das ist nur ein Test!");
    }

    private void beendenToolStripMenuItem_Click(object sender, EventArgs e)
    {
        this.Close();
    }
}
```

Ergebnis



Anstatt eines normalen Menüeintrags können auch eine *TextBox* oder eine *ComboBox* verwendet werden.

Beispiel 7.14: *ComboBox* in *MenuStrip*

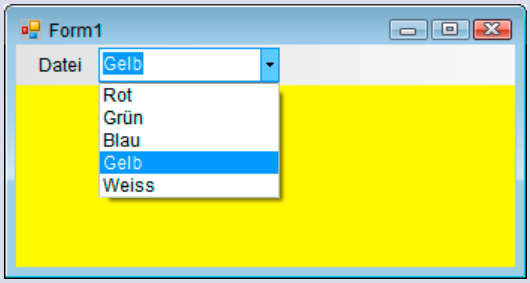
C#

Die oberste Menüebene des Vorgängerbeispiels wird um eine *ComboBox* erweitert, mit der man die Hintergrundfarbe des Formulars einstellen kann.

Die *Items*-Eigenschaft der *ComboBox* wird mit dem über das Eigenschaftenfenster erreichbaren Editor zeilenweise mit den Einträgen für die einzelnen Farben gefüllt. Zur Programmierung kann man das *SelectedIndexChanged*-Ereignis der *ComboBox* ausnutzen:

```
private void toolStripComboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    Color c = Color.White;
    switch (toolStripComboBox1.SelectedIndex)
    {
        case 0: c = Color.Red; break;
        case 1: c = Color.Green; break;
        case 2: c = Color.Blue; break;
        case 3: c = Color.Yellow; break;
        case 4: c = Color.White; break;
    }
    this.BackColor = c;
}
```

Ergebnis



ContextMenuStrip

Die Programmierung eines Kontextmenüs unterscheidet sich nur unwesentlich von der eines normalen Menüs. Der Entwurfsprozess ist nahezu identisch.

HINWEIS: Da sich ein Kontextmenü in der Regel auf ein bestimmtes visuelles Steuerelement bezieht, verfügen Letztere über eine *ContextMenuStrip*-Eigenschaft.

Beispiel 7.15: *ContextMenuStrip*

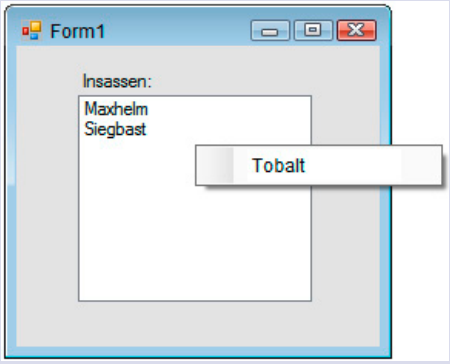
C#

Eine *ContextMenuStrip*-Komponente wird mit einer *TextBox* ausgestattet, um zur Laufzeit eine *ListBox* mit Einträgen zu füllen. Die *ContextMenuStrip*-Eigenschaft der *ListBox* ist mit dem Kontextmenü zu verbinden!

Nach Klick mit der rechten Maustaste auf die *ListBox* erscheint das Kontextmenü. Jeder neue Eintrag ist mittels *Enter*-Taste abzuschließen:

```
private void toolStripTextBox1_KeyUp(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Enter)
    {
        listBox1.Items.Add(toolStripTextBox1.Text);
        toolStripTextBox1.Text = String.Empty;
    }
}
```

Ergebnis



Weitere Eigenschaften von Menüeinträgen

Jeder Menüeintrag wird durch ein *ToolStripMenuItem*-Objekt dargestellt, hat also eigene Eigenschaften, von denen die *Text*-Eigenschaft die offensichtlichste ist. Auch viele der übrigen Eigenschaften erklären sich von selbst.

Beispiel 7.16: Mit der Eigenschaft *Checked* bestimmen Sie, ob vor dem Menüpunkt ein Häkchen angezeigt werden soll oder nicht.

C#

In der Ereignisprozedur können Sie den Wert entsprechend setzen oder auslesen:

```
private void testToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (testToolStripMenuItem.Checked)
    {
        MessageBox.Show("Das ist nur ein Test!");
        testToolStripMenuItem.Checked = false;
    }
}
```

Über die Eigenschaft *Visible* steuern Sie die Sichtbarkeit des Menüeintrags. Besser als das Ausblenden ist meist das Deaktivieren mit der *Enabled*-Eigenschaft. Über *ShortcutKeys* bzw. *ShowShortcutKeys* können Sie den Menüpunkt mit einer Tastenkombination verbinden (z. B. *Alt+G* oder *F7*).

7.3 MDI-Anwendungen

Ein Windows-Programm läuft oft als MDI-Applikation ab. Das heißt, innerhalb eines Rahmen- bzw. Mutterfensters können sich mehrere sogenannte Kindfenster tummeln[1]. Diese werden vom Mutterfenster verwaltet und so „an der Leine gehalten", dass sie z. B. auch dessen Bereich nicht verlassen können.

7.3.1 „Falsche" MDI-Fenster bzw. Verwenden von Parent

Formulare verfügen, wie auch die einfachen Controls, über die Eigenschaft *Parent*. Mithilfe dieser Eigenschaft können Sie ein Formular wie ein Control behandeln und in ein übergeordnetes Formular einfügen.

Das Resultat dieses Vorgehens: Das Formular kann den Clientbereich seines Parent nicht mehr verlassen, Sie können es aber wie gewohnt verschieben, skalieren oder auch schließen. Von besonderem Interesse dürfte die Möglichkeit sein, die untergeordneten Formulare wie Controls anzudocken. Damit steht der Programmierung von Toolbars etc. kaum noch etwas im Wege.

Beispiel 7.17: Einfügen des Formulars *Form2* in den Clientbereich von *Form1*

C#

```
public partial class Form1 : Form
{ ...
    private void button1_Click(object sender, EventArgs e)
    {
        Form2 f2 = new Form2();
        f2.TopLevel = false;
        f2.Parent = this;
        f2.Show();
    }
}
```

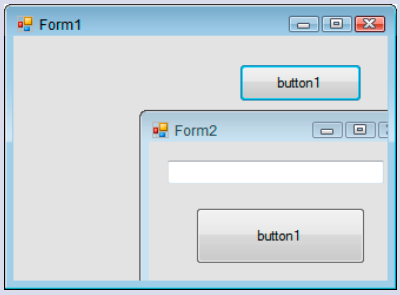
In *Form2* realisieren wir das Docking:

```
public partial class Form2 : Form
{ ...
    private void button1_Click(object sender, EventArgs e)
    {
        this.Dock = DockStyle.Left;
    }
}
```

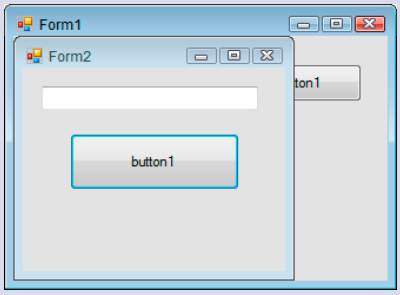
[1] Textverarbeitungsprogramme sind z. B. meist als Multiple Document Interface-Applikation aufgebaut. Die einzelnen Dokumente sind die Kindfenster.

Ergebnis

Das Endergebnis unserer Bemühungen:

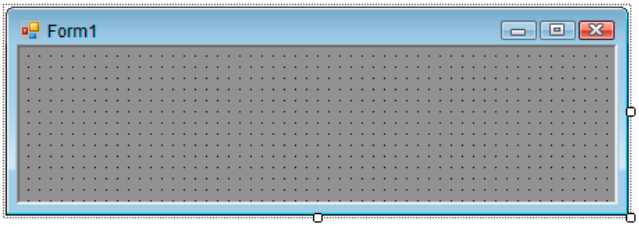


bzw. angedockt:



7.3.2 Die echten MDI-Fenster

Möchten Sie „echte“ MDI-Anwendungen programmieren, brauchen Sie wie im vorhergehenden Beispiel ebenfalls mindestens zwei Formulare, von denen jedoch eines als MDIContainer definiert ist (Eigenschaft *IsMdiContainer=True*). Nachfolgend ändert sich schon zur Entwurfszeit das Aussehen des Formulars:

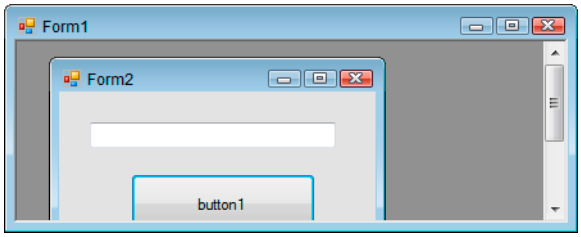


HINWEIS: Sie sollten keine weiteren Controls im MDIContainer platzieren, es sei denn, Sie richten diese mithilfe der *Dock*-Eigenschaft an den Außenkanten des Formulars aus (z. B. ein *Panel*).

7.3.3 Die Kindfenster

Die Kind- oder auch Child-Fenster werden über die Eigenschaft *MdiParent* kenntlich gemacht. Weisen Sie dieser Eigenschaft ein MDIContainer-Fenster zu, werden die Kindfenster automatisch in den Clientbereich des MDIContainers verschoben.

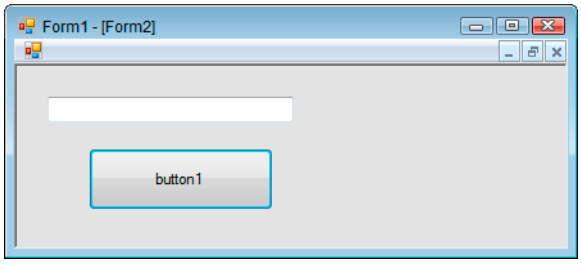
HINWEIS: Die *MDIParent*-Eigenschaft lässt sich nur zur Laufzeit zuweisen (also nicht über das Eigenschaftenfenster)!



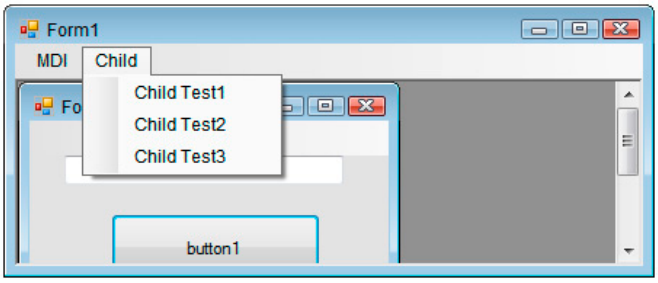
Die Umsetzung:

```
public partial class Form1 : Form
{ ...
    private void neuesChildToolStripMenuItem_Click(object sender, EventArgs e)
    {
        Form2 f2 = new Form2();
        f2.MdiParent = this;
        f2.Show();
    }
}
```

Vergrößern Sie ein MDI-Kindfenster auf Vollbild, so erscheint dessen Titel eingefasst in eckigen Klammern neben dem Titel des Hauptfensters:



Verfügen beide Fenster über eigene Menüs, so werden diese standardmäßig im Menü des MDIContainers kombiniert:

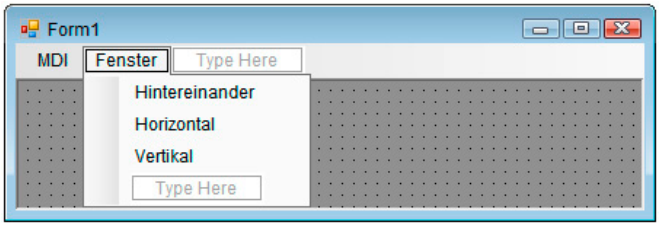


Gegenüber ihren konventionellen Kollegen besitzen die MDI-Kindfenster einige Einschränkungen, die durch das MDI-Konzept bedingt sind:

- MDI-Kindfenster werden nicht in der Windows-Taskleiste angezeigt.
- MDI-Kindfenster können den Clientbereich des MDIContainers nicht verlassen. Verschieben Sie die Fenster aus dem sichtbaren Bereich des MDIContainers, werden automatisch die nötigen Bildlaufleisten im Container angezeigt.
- Maximieren Sie ein MDI-Kindfenster, erreicht dieses maximal die Größe des verbleibenden Clientbereichs des MDIContainers (abzüglich Statuszeile, Menüleiste und Toolbar). Die Schaltflächen für Maximieren, Minimieren und Schließen des Kindfensters werden in diesem Fall in die Menüleiste des MDIContainers eingefügt.

7.3.4 Automatisches Anordnen der Kindfenster

Ein MDIContainer verfügt über die Methode *LayoutMdi*, mit der die vorhandenen Kindfenster nebeneinander, überlappend oder als Symbole angeordnet werden können. Üblicherweise entspricht dies dem in vielen MDI-Anwendungen vorhandenen *Fenster*-Menü:

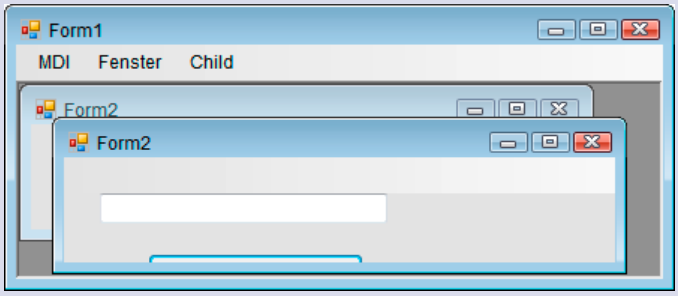


Beispiel 7.18: Hintereinander anordnen

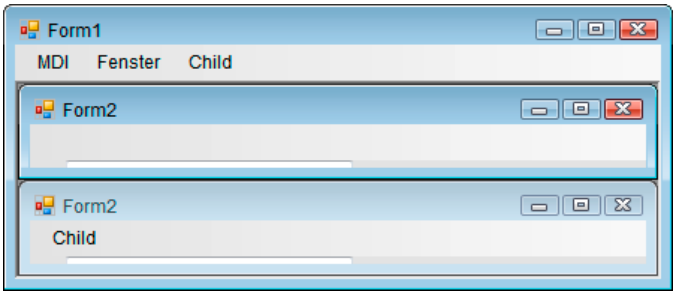
C#

```
private void hintereinanderToolStripMenuItem_Click(object sender, EventArgs e)
{
    this.LayoutMdi(MdiLayout.Cascade);
}
```

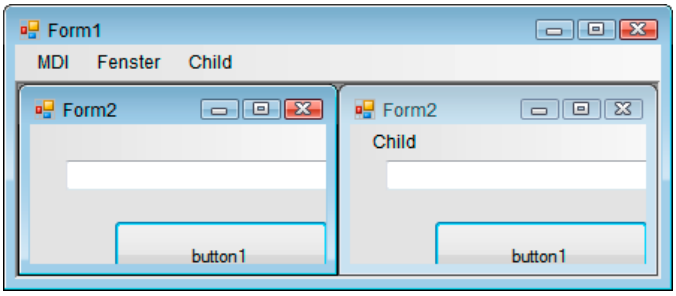
Ergebnis



Horizontal anordnen:



Vertikal anordnen:



7.3.5 Zugriff auf die geöffneten MDI-Kindfenster

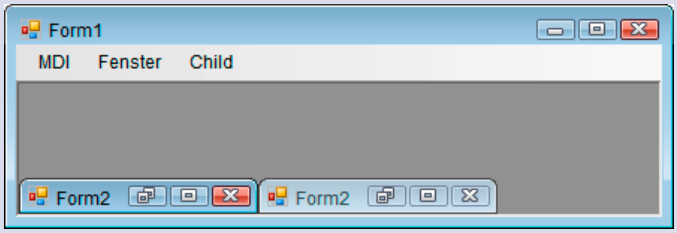
Über die Collection *MdiChildren* bietet sich die Möglichkeit, alle vorhandenen Kindfenster des MDIContainers zu verwalten.

Beispiel 7.19: Minimieren aller MDI-Kindfenster

C#

```
private void minimierenToolStripMenuItem_Click(object sender, EventArgs e)
{
    foreach (Form f in this.MdiChildren)
    {
        f.WindowState = FormWindowState.Minimized;
    }
}
```

Ergebnis



7.3.6 Zugriff auf das aktive MDI-Kindfenster

Neben der Liste aller Kindfenster ist meist auch das gerade aktuelle von besonderem Interesse für den Programmierer, muss er doch häufig auf Inhalte des aktuellen Fensters (z. B. Textfelder, Markierungen) zugreifen. Die per MDIContainer verfügbare Eigenschaft *ActiveMdiChild* lässt die aufkommende Freude recht schnell vergessen, liefert diese doch zunächst nur ein Objekt der Klasse *Form* zurück. Es ist also Ihre Aufgabe, das zurückgegebene Objekt entsprechend zu typisieren.

Beispiel 7.20: Abfrage und Typisieren des aktuellen MDI-Kindfensters per MDIContainer

C#

```
if (this.ActiveMdiChild != null)
{
    myMDIForm mdifrm = (myMDIForm)(this.ActiveMdiChild);
    mdifrm.label1.Text = "Ich bin aktiv";
}
```

7.3.7 Mischen von Kindfenstermenü/MDIContainer-Menü

Das Vermischen der Menüs von MDIContainer und MDI-Kindfenstern kann über spezielle Eigenschaften (*AllowMerge*, *MergeAction*, *MergeIndex*) gesteuert werden und ist (fast) eine „Wissenschaft“ für sich.

AllowMerge

Die Einstellungen der Eigenschaften *MergeAction* und *MergeIndex* wirken sich nur dann aus, wenn für den *MenuStrip* der Wert von *AllowMerge* auf *true* gesetzt ist, was standardmäßig der Fall ist. Setzen Sie den Wert auf *false*, erfolgt keine Mischung der Menüs.

MergeAction und MergeIndex

Mit der Eigenschaft *MergeAction* geben Sie vor, wie die Menüleisten von Container und Kind kombiniert werden, *MergeIndex* spezifiziert in einigen Fällen die Position des einzufügenden Menüelements.

7.4 Praxisbeispiele

7.4.1 Informationsaustausch zwischen Formularen

In diesem Beispiel wird das auch von fortgeschrittenen Programmierern häufig nachgefragte Thema „Wie greife ich von FormA auf FormB zu“ abgehandelt.

Überblick

Unter Visual Studio wird der Code zum Erzeugen des Haupt- bzw. Startformulars (standardmäßig *Form1*) von der IDE automatisch generiert. Um das Erzeugen weiterer Formulare nach dem Muster

```
Form2 f2 = new Form2();
```

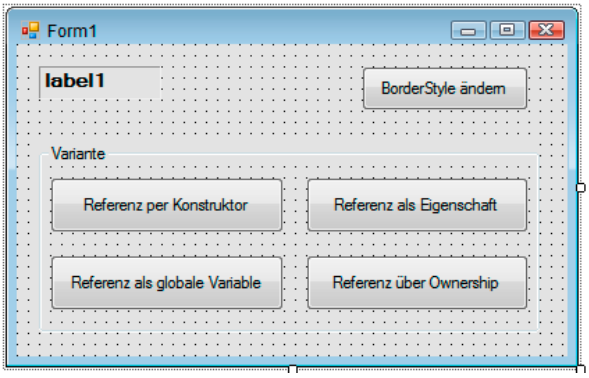
... muss sich der Programmierer selbst kümmern. Existiert die Formularinstanz, so können Sie gemäß den Regeln der OOP auf deren öffentliche Mitglieder auch von außerhalb zugreifen. Wenn beispielsweise im Code von *Form1* ein weiteres Formular (*Form2*) instanziiert wird, so ist es kein Problem, die öffentlichen Member von *Form2* zu verwenden. Etwas komplizierter wird es in umgekehrter Richtung, also wenn Sie von *Form2* (oder weiteren Kindformularen *Form3*, *Form4*, ...) auf das Hauptformular (oder andere Kindformulare) zugreifen wollen. Wir wollen folgende Varianten betrachten, die eine Beziehung zwischen *Form2* (Childform) und *Form1* (Parentform) ermöglichen:

- Erzeugen einer Referenz auf *Form1* im Konstruktor von *Form2*,
- Setzen einer Eigenschaft in *Form2*, die eine Referenz auf *Form1* einrichtet,
- Erzeugen einer globalen Variablen, die *Form1* referenziert und die von *Form2* (und weiteren untergeordneten Forms) benutzt werden kann,
- Verwenden des Owner-Mechanismus der Formulare.

In unserem Beispiel, für das wir fünf Formulare benötigen, werden wir diese vier Varianten vergleichen. Das erste Formular (*Form1*) wird als Hauptformular dienen und die anderen vier (*Form2*, *Form3*, *Form4*, *Form5*) sollen die untergeordneten Formulare sein. Letztere sind – der besseren Vergleichsmöglichkeiten wegen – mit der gleichen Bedienoberfläche ausgestattet und arbeiten auf identische Weise mit dem Hauptformular zusammen.

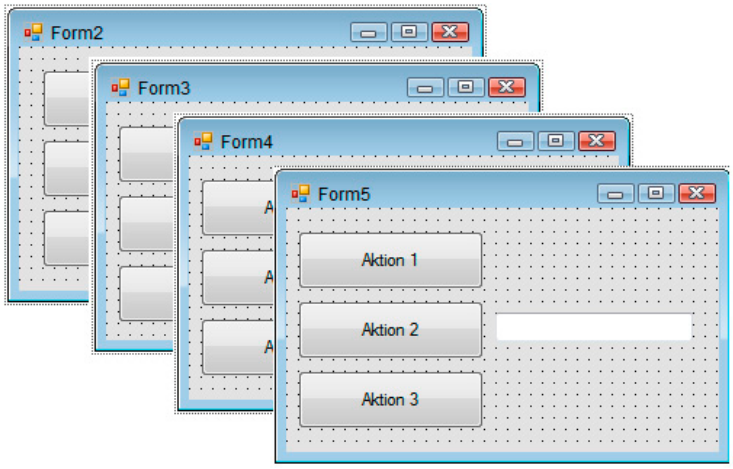
Bedienoberfläche Form1 (Hauptformular)

Öffnen Sie ein neues C#-Projekt als Windows Forms-Anwendung und gestalten Sie die abgebildete Oberfläche. Die in der *GroupBox* angeordneten Schaltflächen (*button2*, ..., *button5*) rufen je ein untergeordnetes Formular (*Form2*, ..., *Form5*) auf, das die gewünschte Variante demonstriert, wobei auf die beiden oben angeordneten Controls (*label1*, *button1*) des Hauptformulars zugegriffen wird.



Bedienoberfläche Form2 ... Form5 (untergeordnete Formulare)

Über das Menü **Projekt | Windows-Form hinzufügen...** ergänzen Sie das Projekt um vier weitere Formulare mit identischer Oberfläche (drei *Button*s und eine *TextBox*):



Die drei *Button*s sollen diverse Aktionen auf dem Hauptformular *Form1* auslösen:

- *button1* führt eine Instanzenmethode aus (Anzeige von Datum/Uhrzeit),
- *button2* soll *label1* mit dem Inhalt von *textBox1* füllen und
- *button3* löst das *Click*-Event für *button1* aus, wodurch der *BorderStyle* von *Form1* geändert wird.

Allgemeiner Code für Form 1

Zunächst treffen wir im Code von *Form1* einige Vorbereitungen für den späteren Zugriff:

```
public partial class Form1 : Form
{
        public Form1()
        {
            InitializeComponent();
        }
```

Den Lesezugriff auf die von außerhalb zu manipulierenden Controls als Eigenschaften offenlegen:

```
        internal Label lbl1
        {
            get
            { return label1;  }
        }

        internal Button btn1
        {
            get
            { return button1; }
        }
```

Irgendeine Methode:

```
        internal void machWas()
        {
            this.Text = DateTime.Now.ToString();
        }
```

Irgendeine Click-Aktion:

```
        private void button5_Click_1(object sender, EventArgs e)
        {
            if (this.FormBorderStyle.Equals(FormBorderStyle.None))
                this.FormBorderStyle = FormBorderStyle.Sizable;
            else
                this.FormBorderStyle = FormBorderStyle.None;
        }
        ...
}
```

Den Ereigniscode für *button1* ... *button4*, in dem die untergeordneten Formulare aufgerufen werden, ergänzen wir später.

Variante 1: Übergabe der Formular-Referenz im Konstruktor

Bei dieser Variante wird der Konstruktor von *Form2* so modifiziert, dass er eine Referenz auf *Form1* entgegennehmen und damit die private Zustandsvariable *m_Form* setzen kann.

```
public partial class Form2 : Form
{
        private Form1 m_Form;
```

```
        public Form2(Form1 frm)
        {
            InitializeComponent();
            m_Form = frm;   // Referenz auf Hauptformular setzen
        }
```

Eine Instanzenmethode in *Form1* ausführen:

```
        private void button1_Click(object sender, EventArgs e)
        {
            if (m_Form != null)
                m_Form.machWas();
            else
            throw new InvalidOperationException("Form1 Instanz nicht gesetzt!");
        }
```

Ein Steuerelement in *Form1* setzen:

```
        private void button2_Click(object sender, EventArgs e)
        {
            if (m_Form != null)
                m_Form.lbl1.Text = textBox1.Text;
            else
            throw new InvalidOperationException("Form1 Instanz nicht gesetzt!");
        }
```

Ein Ereignis in *Form1* auslösen:

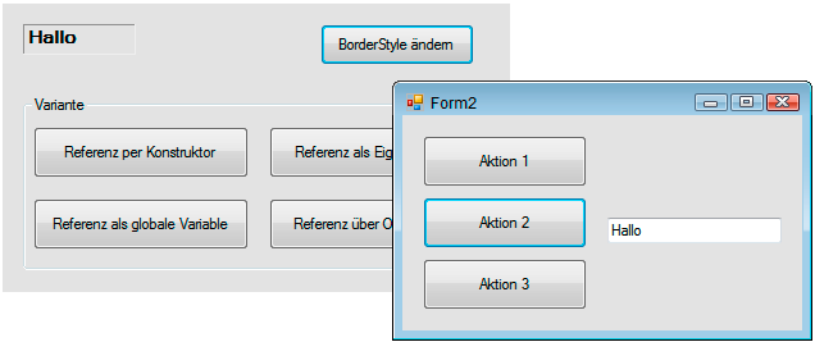
```
        private void button3_Click(object sender, EventArgs e)
        {
            if (m_Form != null)
                m_Form.btn1.PerformClick();  // Click-Event des Buttons auslösen
            else
                throw new InvalidOperationException("Form1 Instanz nicht gesetzt!");
        }
    }
```

Sie haben jetzt sicherlich bemerkt, dass alle *Click*-Events die Referenz für *m_Form* auf *null* testen und andernfalls einen Fehler auswerfen. Dies ist wichtig, weil der Benutzer von *Form2* sichergehen muss, dass eine gültige Referenz auf *Form1* vorliegt.

Nun müssen wir nur noch den Code von *Form1* ergänzen:

```
    private void button1_Click(object sender, EventArgs e)
    {
            Form2 frm = new Form2(this);  // Form1-Referenz übergeben!
            this.AddOwnedForm(frm);       // frm zum Besitz von Form1 hinzufügen
            frm.Show();
    }
```

Starten Sie die Anwendung, so wird zunächst nur das Hauptformular angezeigt. Ein Klick auf *button1* lässt *Form2* erscheinen. Klicken Sie hier auf „Aktion 1“, so werden in der Titelleiste des Hauptformulars Datum und Uhrzeit angezeigt. Geben Sie irgendetwas in die *TextBox* ein, klicken Sie auf „Aktion 2“ und das *Label* des Hauptformulars wird den Inhalt übernehmen. Ein Klick auf „Aktion 3“ lässt die Umrandung des Hauptformulars verschwinden und beim nächsten Klick wieder erscheinen.



An dieser Stelle scheint eine Bemerkung zur *AddOwnedForm()*-Methode angebracht: Sie können deren Aufruf auch weglassen, müssen dann aber zum Beispiel in Kauf nehmen, dass nach dem Minimieren des Hauptformulars das untergeordnete Formular an seinem Platz verbleibt und nicht – wie in unserem Fall – komplett verschwindet, um nach Vergrößern des Hauptformulars wieder aufzutauchen. Auch liegt *Form2* immer auf *Form1*, kann also von diesem nicht verdeckt werden.

Variante 2: Übergabe der Formular-Referenz als Eigenschaft

Die zweite Variante weist sehr starke Ähnlichkeiten zur ersten Variante auf, weshalb wir uns auf eine verkürzte Darstellung beschränken können. *Form3* hält die *Form1*-Referenz ebenfalls in der privaten Zustandsvariablen *m_Form*, lediglich an die Stelle der Übergabe im Konstruktor tritt jetzt die Übergabe in einer WriteOnly-Eigenschaft *Form* vom Typ *Form1*:

```
public partial class Form3 : Form
{
        private Form1 m_Form;
        ...
        internal Form1 Form
        {
            set
            {  m_Form = value; }
        }
        ...
}
```

Der restliche Code entspricht dem von *Form2*.

Nun müssen wir nur noch das *Click*-Event für *button2* des Hauptformulars hinzufügen, damit *Form3* angezeigt und getestet werden kann. Im Vergleich zur Vorgängervariante ist eine zusätzliche Codezeile erforderlich, in der die *Form1*-Referenz als *Form*-Eigenschaft dem untergeordneten Formular zugewiesen wird:

```
        private void button2_Click(object sender, EventArgs e)
        {
            Form3 frm = new Form3();
            this.AddOwnedForm(frm);
            frm.Form = this;          // Eigenschaft setzen!
            frm.Show();
        }
```

Beim Testen gibt es erwartungsgemäß keinerlei Unterschiede zur ersten Variante.

Variante 3: Übergabe der Formular-Referenz als globale Variable

Fügen Sie über **Projekt | Klasse hinzufügen...** eine neue statische Klasse (*GlobalRef*) mit folgendem Code hinzu:

```
internal static class GlobalRef
{
        public static Form1 g_Form;
}
```

Die globale Variable (*g_Form*) hält die Referenz auf das Hauptformular *Form1*. Der Zugriff von *Form4* aus:

```
public partial class Form4 : Form
{
   ...
   private void button1_Click(object sender, EventArgs e)
   {
        if (GlobalRef.g_Form != null)
           GlobalRef.g_Form.machWas();
        else
            throw new InvalidOperationException("Form1 Instanz nicht gesetzt!");
   }

      private void button2_Click(object sender, EventArgs e)
    {
        if (GlobalRef.g_Form != null)
            GlobalRef.g_Form.lbl1.Text = textBox1.Text;
        else
            throw new InvalidOperationException("Form1 Instanz nicht gesetzt!");
    }

    private void button3_Click(object sender, EventArgs e)
    {
        if (GlobalRef.g_Form != null)
            GlobalRef.g_Form.btn1.PerformClick();
        else
            throw new InvalidOperationException("Form1 Instanz nicht gesetzt!");
    }
}
```

Auch hier gibt es große Ähnlichkeiten zu den beiden Vorgängerversionen. Ein nicht zu unterschätzender Vorteil ist allerdings der globale Standort der *Form1*-Referenz, die nun nicht nur von *Form4*, sondern von jedem beliebigen Formular verwendet werden kann.

Schließlich noch der Aufruf von *Form4* im Code des Hauptformulars *Form1*:

```
    private void button3_Click(object sender, EventArgs e)
    {
        Form4 frm = new Form4();
        this.AddOwnedForm(frm);
        GlobalRef.g_Form = this;     // Setzen der globalen Referenz!
        frm.Show();
    }
```

Auch bei der dritten Variante werden Sie keinerlei Unterschiede zu den Vorgängern feststellen.

Variante 4: Übergabe der Formular-Referenz als Ownership

Diese vierte und letzte Version ist leider weitaus weniger bekannt als die bis jetzt besprochenen Versionen. Benutzt wird das in Windows Forms (*System.Windows.Forms.Control*-Klasse) eingebaute Ownership-Verhalten, nach dem ein Control eines oder mehrere andere „besitzen“ kann. In unserem Fall müssen wir *Form4* zum „Besitz“ (Ownership) der übergeordneten *Form1* hinzufügen (auf die Merkmale von Ownership-Beziehungen wurde bereits im Zusammenhang mit der Version 1 bzw. der *AddOwnedForm()*-Methode eingegangen).

Der folgende Code für *Form5* funktioniert nur, wenn eine *Owner*-Eigenschaft vorliegt und in den korrekten Typ (*Form1*) gecastet werden kann. Anderenfalls wird ein Fehler erzeugt.

```
public partial class Form5 : Form
{
    ...
    private void button1_Click(object sender, EventArgs e)
    {
        if ((this.Owner != null) && (this.Owner.GetType().Name == "Form1"))
            (this.Owner as Form1).machWas();
        else
            throw new InvalidOperationException("Form1 Instanze nicht gesetzt oder
ungültig!");
    }

    private void button2_Click(object sender, EventArgs e)
    {
        if ((this.Owner != null) && (this.Owner.GetType().Name == "Form1"))
            (this.Owner as Form1).lbl1.Text = textBox1.Text;
        else
            throw new InvalidOperationException("Form1 Instanze nicht gesetzt oder
ungültig!");
    }

    private void button3_Click(object sender, EventArgs e)
    {
        if ((this.Owner != null) && (this.Owner.GetType().Name == "Form1"))
             (this.Owner as Form1).btn1.PerformClick();
        else
            throw new InvalidOperationException("Form1 Instanze nicht gesetzt oder
ungültig!");
    }
}
```

Nun zum *Form1*-Code. Instanziierung und Aufruf von *Form5* sind ähnlich einfach wie bei den drei Vorgängerversionen:

```
    private void button4_Click(object sender, EventArgs e)
    {
         Form5 frm = new Form5();
         this.AddOwnedForm(frm);   // darf nicht weggelassen werden!
         frm.Show();
    }
```

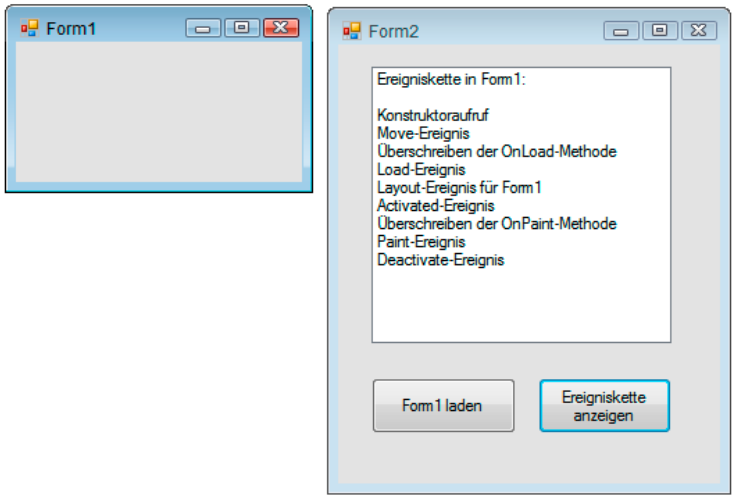
Auch der Test dieser Version führt zu exakt den gleichen Ergebnissen wie bei den Vorgängern.

7.4.2 Ereigniskette beim Laden/Entladen eines Formulars

In welcher Reihenfolge werden die Ereignisse beim Laden bzw. Entladen eines Formulars ausgelöst? Das durch ein eigenes Experiment zu erkunden, wirkt nachhaltiger als das pure Auswendiglernen. Ganz nebenbei werden im vorliegenden Beispiel auch solche Fragen beantwortet: Wie ändere ich das Startformular? Wie rufe ich von *Form2* aus *Form1* auf? Wie überschreibe ich die *OnLoad*-Ereignisprozedur des Formulars?

Bedienoberfläche (Form1 und Form2)

Da das Startformular (*Form1*) hier als „Messobjekt" dient, brauchen wir noch ein zweites Formular (*Form2*), das unser „Messgerät" aufnimmt, im konkreten Fall eine *ListBox* als Ereignislogger und zwei Buttons, mit denen das Laden von *Form1* und die Anzeige der Ereigniskette gestartet werden. *Form1* hingegen bleibt „nackt", das heißt, seine Oberfläche enthält keinerlei Steuerelemente (siehe Laufzeitansicht).



Warum genügt uns nicht ein einziges Formular? Die Antwort ist einfach: Durch die auf dem Formular befindlichen Steuerelemente und die in die *ListBox* vorzunehmenden Einträge vergrößert sich die Anzahl der ausgelösten Ereignisse (*Layout*, *Paint*, ...), die Sache wird unübersichtlich und eine klare Reihenfolge ist nur noch schwer zu erkennen.

Änderung des Startformulars

Nachdem Sie über das Menü **Projekt | Windows Form hinzufügen...** ein zweites Formular erzeugt haben, wollen Sie, dass dieses nach Start der Anwendung erscheint. Am einfachsten realisieren Sie das, indem Sie in der *Main*-Methode der Anwendung nicht mehr den Konst-

ruktor von *Form1* aufrufen (normalerweise hat das Visual Studio bereits für Sie erledigt), sondern den von *Form2*. Um den Code der *Main*-Methode editieren zu können, müssen Sie im Projektmappen-Explorer auf die Datei *Program.cs* doppelklicken und dann die im Folgenden fettgedruckte Änderung vornehmen:

```
namespace WindowsFormsApplication1
{
    static class Program
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form2());
        }
    }
}
```

Quellcode von Form1

```
public partial class Form1 : Form
{
```

Eine generische Liste übernimmt das Zwischenspeichern des Ereignislogs. Die Liste ist *public*, damit von *Form2* aus der Inhalt auf möglichst einfache Weise gelesen werden kann:

```
    public List<string> GList;
```

Das Erzeugen der Liste erfolgt im Konstruktor des Formulars (vor Aufruf von *InitializeComponent()*!).

```
    public Form1()
    {
        GList = new List<string>();
        GList.Add("Konstruktoraufruf");
        InitializeComponent();
    }
```

Was jetzt kommt, können Sie sich leicht selbst zusammenreimen: Für jedes der interessierenden Formularereignisse schreiben Sie einen Eventhandler, dessen Rahmencode Sie wie üblich über die Ereignisse-Seite des Eigenschaftenfensters (F4) automatisch generieren lassen. Die Reihenfolge der folgenden Eventhandler ist unwichtig.

```
        private void Form1_Move(object sender, EventArgs e)
        {
            GList.Add("Move-Ereignis");
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            GList.Add("Load-Ereignis");
        }
```

```
        private void Form1_Layout(object sender, LayoutEventArgs e)
        {
           Control c = e.AffectedControl;
           GList.Add ("Layout-Ereignis für " + c.Name );
        }

        private void Form1_Activated(object sender, EventArgs e)
        {
            GList.Add("Activated-Ereignis");
        }

        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            GList.Add( "Paint-Ereignis");
        }
        private void Form1_Resize(object sender, EventArgs e)
        {
            GList.Add("Resize-Ereignis");
        }

        private void Form1_FormClosing(object sender, FormClosingEventArgs e)
        {
            GList.Add("FormClosing-Ereignis");
        }

        private void Form1_FormClosed(object sender, FormClosedEventArgs e)
        {
            GList.Add("FormClosed-Ereignis");
        }

        private void Form1_Deactivate(object sender, EventArgs e)
        {
            GList.Add("Deactivate-Ereignis");
        }
```

Um herauszufinden, was passiert, wenn wir die *On...*-Ereignismethoden der Basisklasse des Formulars überschreiben (siehe Bemerkungen am Schluss), fügen wir noch folgende zwei Methoden hinzu:

```
        protected override void OnLoad(EventArgs e)
        {
            GList.Add("Überschreiben der OnLoad-Methode");
            base.OnLoad(e);    // Aufruf der Basisklassenmethode
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            GList.Add("Überschreiben der OnPaint-Methode");
            base.OnPaint(e);   // Aufruf der Basisklassenmethode
        }
```

Wenn Sie in den obigen beiden Methoden den Aufruf der Basisklassenmethode weglassen, wird das entsprechende Ereignis nicht ausgelöst!

```
    }
}
```

Quellcode von Form2

```
public partial class Form2 : Form
{ ...
```

Zunächst brauchen wir eine Referenz auf *Form1*:

```
    Form1 frm1 = null;
```

Nach Klick auf die erste Schaltfläche wird *Form1* erzeugt und geladen:

```
    private void button1_Click(object sender, EventArgs e)
    {
        frm1 = new Form1();
        frm1.Show();
    }
```

Ein Klick auf die zweite Schaltfläche bringt den Inhalt des Pufferspeichers *GList* aus *Form1* in der *ListBox* zur Anzeige:

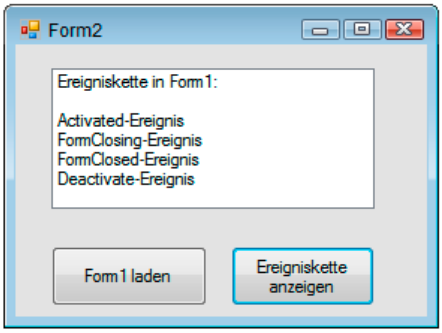
```
    private void button2_Click(object sender, EventArgs e)
    {
        listBox1.Items.Clear();
        listBox1.Items.Add("Ereigniskette in Form1:");
        listBox1.Items.Add("");
        for (int i = 0; i < frm1.GList.Count; i++)
                    listBox1.Items.Add(frm1.GList[i]);
        frm1.GList.Clear();
    }
}
```

Test

Vorhang auf für die verschiedensten Experimente! Untersuchen Sie zunächst die Ereigniskette beim Laden des Formulars (siehe obige Abbildung). Dazu klicken Sie zunächst auf den linken und dann auf den rechten Button.

Da ein Ereignis immer erst innerhalb der entsprechenden *On...*-Ereignismethode der Basisklasse ausgelöst wird, kommt in unserem Fall der in den beiden überschriebenen *On...*-Ereignismethoden enthaltene Code immer **vor** dem Code der entsprechenden Eventhandler zur Ausführung.

Nach dem Schließen von *Form1* klicken Sie erneut auf den rechten Button, um sich die Ereigniskette beim Entladen anzuschauen (siehe folgende Abbildung).



Verschieben Sie *Form1*, verändern Sie die Abmessungen, wechseln Sie den Fokus mit anderen Formularen – all diese Manipulationen finden ihren Niederschlag in einer typischen Ereigniskette.

Bemerkungen

- Beim Überschreiben von Basisklassenmethoden dürfen Sie den Aufruf der *base.On...*-Methode nicht vergessen, da ansonsten das entsprechende Ereignis nicht ausgelöst wird!
- Für das Implementieren von Initialisierungscode wird häufig der *Load*-Eventhandler des Formulars benutzt, auch die Autoren haben in den Beispielen dieses Buchs oft von dieser einfachen Möglichkeit Gebrauch gemacht. Laut Microsoft sollte man aber besser die das Ereignis auslösende *OnLoad*-Methode der Basisklasse *Form* überschreiben (siehe obiger Quellcode). Der Grund liegt im Multicast-Ereignismodell von .NET, das heißt, ein von einem Objekt (Subjekt) ausgelöstes Ereignis kann durchaus auch von mehreren anderen Objekten (Observern) abonniert werden, wobei die Reihenfolge der Abarbeitung unklar bzw. unübersichtlich werden kann. Ähnliche Überlegungen gelten z. B. auch für das *Paint*-Ereignis des Formulars und die das Ereignis auslösende *OnPaint*-Methode der Basisklasse (siehe dazu das Grafik-Kapitel 9).

8 Windows Forms-Komponenten

Nachdem Sie in den beiden vorhergehenden Kapiteln bereits die Grundlagen der Windows-Formulare kennengelernt haben, soll es Ziel dieses Kapitels sein, Ihnen einen Überblick über die wichtigsten visuellen Steuerelemente von Windows Forms-Anwendungen zu verschaffen. Da dieses Kapitel keine vollständige Referenz bereitstellt – diese Rolle kann die Ihnen zur Verfügung stehende Dokumentation viel effektiver übernehmen –, werden nur die aus der Sicht des Praktikers wichtigsten Eigenschaften, Ereignisse und Methoden in Gestalt von Übersichten und knappen Beispielen vorgestellt.

8.1 Allgemeine Hinweise

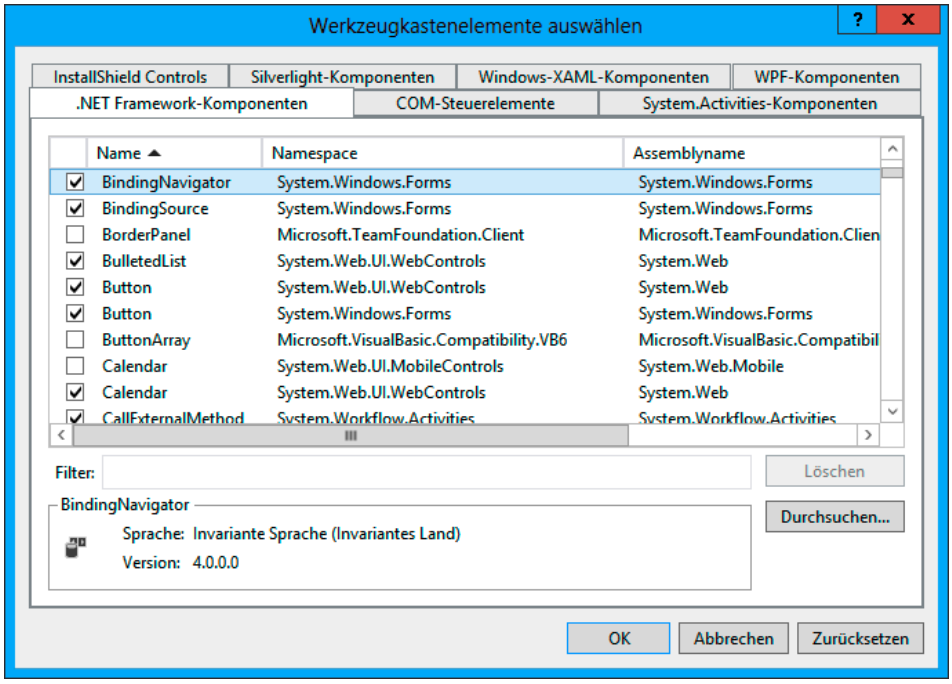
Visual Studio bietet Ihnen eine hervorragende Unterstützung bei der Entwicklung grafischer Benutzeroberflächen. Das grundlegende Handwerkszeug (Toolbox, Eigenschaftenfenster, ...) wurde Ihnen bereits im Einführungskapitel 1 im ausreichenden Umfang vermittelt. Die Vorgehensweise beim Oberflächenentwurf ist so intuitiv, dass sich weitere Erklärungen erübrigen. Stattdessen folgen einige grundsätzliche Hinweise.

8.1.1 Hinzufügen von Komponenten

Die Anzahl der Komponenten hat mit jeder neuen Version des .NET Frameworks zugenommen, sodass es aus Gründen der Übersichtlichkeit nicht mehr zu empfehlen ist, alle zusammen im Werkzeugkasten anzubieten. Außerdem wurden im Laufe der Zeit einige Steuerelemente durch neuere ersetzt (z. B. *MainMenu/ContextMenu* durch *MenuStrip/ContextMenuStrip*, *DataGrid* durch *DataGridView*, ...). Die älteren Versionen müssen aber aus Gründen der Abwärtskompatibilität auch weiterhin zur Verfügung stehen, sollten aber zunächst nicht mehr in der Toolbox auftauchen.

Wenn Sie das standardmäßige Angebot ändern wollen, können Sie dem Werkzeugkasten weitere Steuerelemente hinzufügen oder welche davon entfernen. Über der entsprechenden Kategorie des Werkzeugkastens klicken Sie im Kontextmenü *Elemente auswählen ...* Im Dia-

log „Werkzeugkastenelemente auswählen“ markieren Sie die gewünschten Steuerelemente bzw. entfernen bestimmte Häkchen.



8.1.2 Komponenten zur Laufzeit per Code erzeugen

Grundsätzlich haben Sie jederzeit die Möglichkeit, auf die Dienste des visuellen Designers zu verzichten und stattdessen den Code zur Erzeugung des Steuerelements eigenhändig zu schreiben. Obwohl das scheinbar mit einiger Mehrarbeit verbunden ist, kann es in einigen Fällen durchaus sinnvoll sein (z. B. beim Erzeugen von Steuerelemente-Arrays).

Beispiel 8.1: Ein Array mit drei Schaltflächen per Code erzeugen

C#

```
public partial class Form1 : Form
{
```

Die globalen Deklarationen:

```
    private const int n = 3;                // Anzahl der Buttons
    private const int xpos = 10, ypos = 10;  // linke obere Ecke des ersten Buttons

    private Button[] buttons = new Button[n]; // Array, welches Platz für n Buttons
bietet
```

Steuerelemente werden im Konstruktorcode erzeugt:

```
    public Form1()
    {
        InitializeComponent();
```

Das Button-Array durchlaufen:

```
        for (int i = 0; i < n; i++)
        {
```

Einen Button „nackt“ erzeugen:

```
            buttons[i] = new Button();
```

Eigenschaften zuweisen:

```
            buttons[i].Bounds = new Rectangle(new Point(xpos + i*100, ypos),
                                              new Size(100, 50));
            buttons[i].Text = "Button" + (i+1).ToString();
```

Gemeinsame Ereignisbehandlung anmelden:

```
            buttons[i].Click += new EventHandler(button_Click);
        }
```

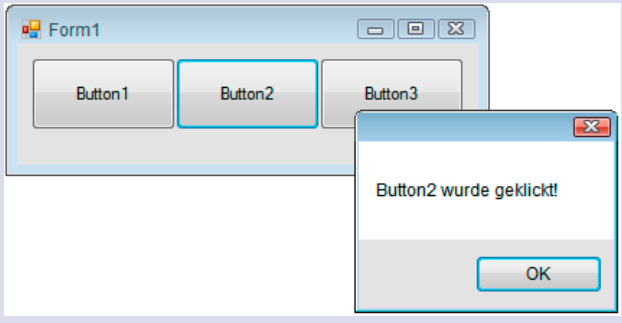
Alle Buttons zum Formular hinzufügen:

```
        this.Controls.AddRange(buttons);
    }
```

Gemeinsamer Eventhandler für *Click*-Ereignis:

```
    private void button_Click(object sender, EventArgs e)
    {
        Button btn = (Button) sender;
        MessageBox.Show(btn.Text + " wurde geklickt!");
    }
}
```

Ergebnis



Alle Buttons sind exakt nebeneinander ausgerichtet. Das Vergrößern des Arrays ist mit keinerlei Mehraufwand an Code verbunden.

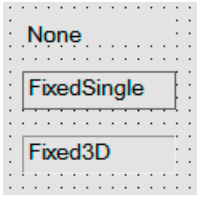
HINWEIS: Die sinnvolle Anwendung eines *CheckBox*-Arrays wird im Praxisbeispiel in Abschnitt 8.10.2gezeigt!

8.2 Allgemeine Steuerelemente

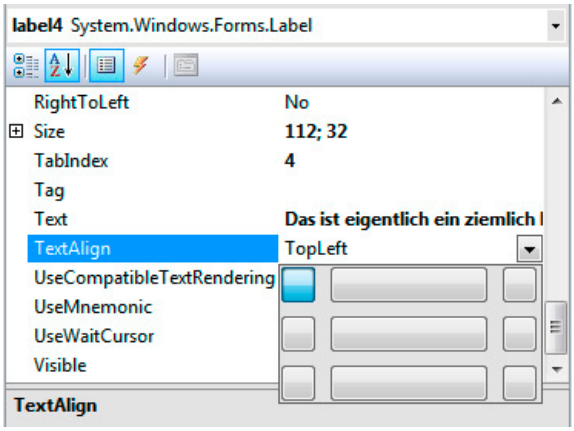
Diese Toolbox-Kategorie enthält standardmäßig die normalerweise am häufigsten benötigten Steuerelemente.

8.2.1 Label

Das harmlose, aber unverzichtbare *Label* dient, im Gegensatz zur *TextBox*, nur zur Anzeige von statischem (unveränderbarem) Text (*Text*-Property). Mit *BorderStyle* haben Sie die Wahl zwischen drei Erscheinungsbildern:

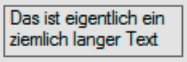


Hervorzuheben ist die *TextAlign*-Eigenschaft, welche die Ausrichtung des Textes auf insgesamt neun verschiedene Arten ermittelt bzw. setzt. Auch hier steht Ihnen ein komfortabler Property-Editor zur Verfügung:



Wenn die *AutoSize*-Eigenschaft auf *True* gesetzt wird, passt sich die Breite des Labels dem aufzunehmenden Text an. Bei *False* ist auch die Anzeige von mehrzeiligem Text möglich.

Beispiel 8.2: Der Zeilenumbruch bei *AutoSize* = *False* erfolgt „intelligent“, also beim nächsten passenden Leerzeichen.



HINWEIS: Möchten Sie ein &-Zeichen im *Label* verwenden, können Sie entweder die Eigenschaft *UseMnemonic* auf *False* setzen oder Sie müssen zwei &-Zeichen einfügen.

8.2.2 LinkLabel

Im Grunde handelt es sich bei dieser Komponente um ein *Label*, das mit etwas Funktionalität ergänzt wurde, um einen Hyperlink nachzubilden.

Konzeptionell kann dieses Control leider nur als missglückt angesehen werden. Mit viel Aufwand und Verspieltheit (Sie können mehrere Hyperlinks innerhalb des Controls definieren) wurde am Problem vorbei programmiert. Statt einer simplen *Link*-Eigenschaft, die den Hyperlink enthält und die sich auch zur Entwurfszeit zuweisen lässt, wurde noch eine Klasse integriert, die sich nur zur Laufzeit sinnvoll ansprechen lässt. Zu allem Überfluss muss auch noch die eigentliche Funktionalität, das heißt der Aufruf des Hyperlinks, selbst programmiert werden. Da ist man mit einem einfachen Label fast schneller am Ziel.

Wichtige Eigenschaften

Zur Gestaltung des optischen Erscheinungsbilds können Sie folgende Eigenschaften verwenden:

- *ActiveLinkColor* (der Hyperlink ist aktiv)
- *DisabledLinkColor* (der Hyperlink ist gesperrt)
- *LinkColor* (die Standardfarbe)
- *VisitedLinkColor* (der Hyperlink wurde bereits angeklickt)
- *LinkBehavior* (wie bzw. wann wird der Hyperlink unterstrichen)

Hyperlink einfügen

Verwenden Sie die *Links.Add*-Methode, um zur Laufzeit einen Hyperlink in das Control einzufügen. Übergabewerte sind der Bereich des Hyperlinks bezüglich der *Text*-Eigenschaft und der URL.

Beispiel 8.3: Aufruf der MS-Homepage

C#

Platzieren Sie einen Hyperlink mit dem Text „Microsoft im Internet“ auf dem Formular.

Weisen Sie im Formular-Konstruktor den Hyperlink zu:

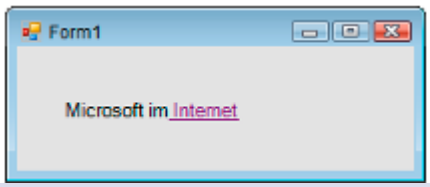
```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        linkLabel1.Links.Add(12, 9, "www.microsoft.com");
    }
```

Die Zahlenangaben bewirken, dass lediglich das Wort „Internet“ als Hyperlink verwendet wird.

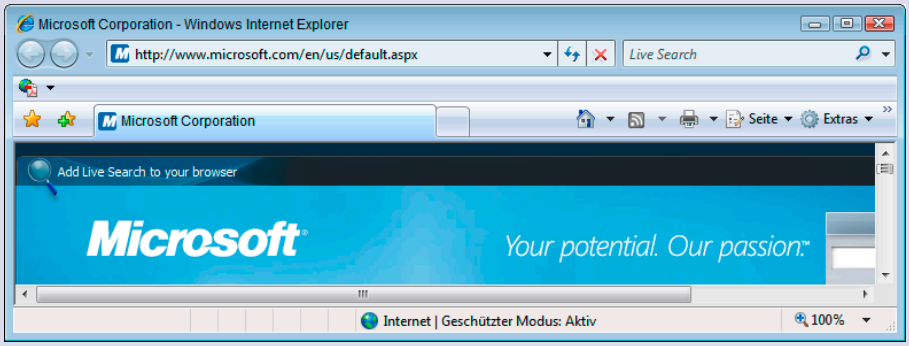
Mit dem Klick auf den *LinkLabel* wird das *LinkClicked*-Ereignis ausgelöst. Hier können Sie der Komponente zum einen mitteilen, dass der Hyperlink besucht wurde, zum anderen sind Sie dafür verantwortlich, den URL auszuwerten (wird im Parameter *e* übergeben) und aufzurufen:

```
    private void linkLabel1_LinkClicked(object sender,
LinkLabelLinkClickedEventArgs e)
    {
        linkLabel1.Links[linkLabel1.Links.IndexOf(e.Link)].Visited = true;
        System.Diagnostics.Process.Start(e.Link.LinkData.ToString());
    }
}
```

Ergebnis



Wie Sie sehen, erhält der eigentliche Hyperlink auch den Fokus, Sie können also auch mit der *Tabulator*- und der *Enter*-Taste arbeiten.



8.2.3 Button

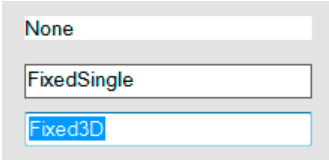
Dieses Steuerelement ist wohl aus (fast) keiner Applikation wegzudenken, über die Funktionalität brauchen wir deshalb kaum Worte zu verlieren. Die folgende Abbildung zeigt vier Vertreter dieser Gattung mit unterschiedlich gesetzter *FlatStyle*-Eigenschaft:

HINWEIS: Entgegen der üblichen Vorgehensweise legen Sie die *Default*-Taste in Dialogboxen nicht mehr über eine Eigenschaft des Buttons, sondern über die *AcceptButton*-Eigenschaft des Formulars fest. Das Gleiche gilt für *CancelButton*.

Mit der *Image*-Eigenschaft bietet sich ein weiteres gestalterisches Mittel, um die trostlosen Buttons etwas aufzupeppen. Alternativ können Sie als Quelle der Grafik auch eine *ImageList* verwenden, in diesem Fall wählen Sie die Grafik mit der *ImageIndex*-Eigenschaft aus.

8.2.4 TextBox

Im Unterschied zum *Label* besteht hier die Möglichkeit, den Text zur Laufzeit zu editieren oder zu markieren. All dies geschieht durch Zugriff auf die *Text*-Eigenschaft. Bei mehrzeiligem Text können Sie über die *Lines*-Eigenschaft auf die einzelnen Textzeilen zugreifen. Das äußere Erscheinungsbild wird, wie beim Label, im Wesentlichen durch die *BorderStyle*-Eigenschaft bestimmt:



Hervorzuheben sind weiterhin folgende Properties:

Eigenschaft	Beschreibung
AutoCompleteSource *AutoCompleteMode* *AutoCompleteCustomSource*	... ermöglichen automatisches Vervollständigen des einzugebenden Textes
TextAlign	... bestimmt die Ausrichtung des Textes (*Left*, *Right*, *Center*)
MaxLength	... legt die maximale Anzahl einzugebender Zeichen fest (Standardeinstellung 32.767 Zeichen)
ReadOnly	... das Control ist schreibgeschützt.

Mehrzeilige Textboxen

Wichtige Eigenschaften in diesem Zusammenhang:

Eigenschaft	Beschreibung
MultiLine	... erlaubt die Eingabe mehrzeiliger Texte (*True*). Für diesen Fall ist auch eine vertikale Scrollbar sinnvoll.
ScrollBars	... bestimmt, ob Bildlaufleisten enthalten sind. Die Eigenschaft zeigt nur bei *MultiLine=True* Wirkung.
AcceptsReturn	Ist diese Eigenschaft *True*, so können Sie mittels Enter-Taste einen Zeilenumbruch einfügen. Ein eventuell vorhandener *AcceptButton* wird damit außer Kraft gesetzt! Bleibt *WantReturns* auf *False,* müssten Sie nach wie vor *Strg+Enter* für einen Zeilenumbruch verwenden.
WordWrap	Damit bestimmen Sie, ob der Text im Eingabefeld am rechten Rand umgebrochen wird (*True*). Der Umbruch wird lediglich auf dem Bildschirm angezeigt, der Text selbst enthält keinerlei Zeilenumbrüche, die nicht eingegeben wurden. Wenn *WordWrap False* ist, entsteht eine neue Zeile nur dort, wo auch ein Zeilenumbruch in den Text eingefügt wurde.
Lines	Zwar gibt es auch eine *Text*-Eigenschaft, doch ist diese für die praktische Arbeit weniger gut geeignet. Sie arbeiten besser mit der *Lines*-Eigenschaft, die einen gezielten Zugriff auf einzelne Zeilen gestattet und die Sie im Stringlisten-Editor oder auch per Quellcode zuweisen können, z. B. Auslesen einer Zeile: `textBox2.Text = textBox1.Lines[3];`

Markieren von Text

SelectionLength, *SelectionStart*, *SelectedText* gelten für markierte Textausschnitte. *SelectionLength* bestimmt bzw. liefert die Zeichenzahl, *SelectionStart* ermittelt die Anfangsposition und *SelectedText* setzt bzw. ermittelt den Inhalt.

HINWEIS: Auf diese Properties kann nur zur Laufzeit zugegriffen werden, sie befinden sich **nicht** im Eigenschaftenfenster!

Alternativ können Sie auch die Methoden *SelectAll* bzw. *Select* verwenden.

Beispiel 8.4: Wenn man mit der Tab-Taste zur *TextBox* wechselt, wird das erste Zeichen markiert.

C#

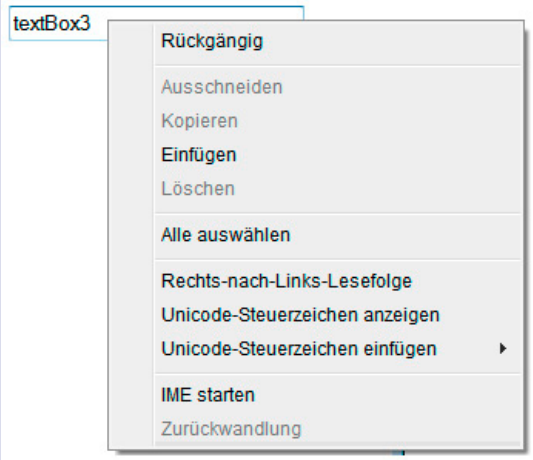
```
private void textBox2_Enter1(object sender, EventArgs e);
{
   textBox2.Select(0, 1);
}
```

oder

```
   textBox2.SelectionStart  = 0;
   textBox2.SelectionLength = 1;
```

Ergebnis

Übrigens können Sie zur Laufzeit für jedes Editierfeld ein umfangreiches Kontextmenü aufrufen, über das die wichtigsten Operationen direkt ausführbar sind:



PasswordChar

Diese Eigenschaft erlaubt das verdeckte Eingeben eines Passworts. Sie können das gewünschte Zeichen im Eigenschaften-Fenster oder per Quellcode zuweisen.

Beispiel 8.5: Passworteingabe

C#

```
textBox1.PasswordChar = '*';
```

Ergebnis

Automatisches Vervollständigen

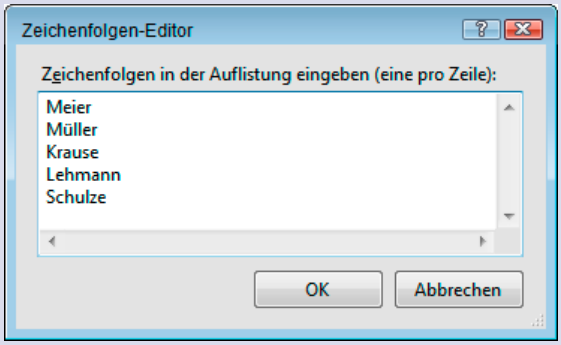
Die Eigenschaften *AutoCompleteCustomSource*, *AutoCompleteSource* und *AutoCompleteMode* einer *TextBox* bewirken das automatische Vervollständigen des eingegebenen Textes durch Vergleich mit einer vorhandenen Liste. Das ist besonders vorteilhaft bei sich häufig wiederholenden Eingabewerten, wie z. B. URLs, Adressen oder Dateinamen.

Voraussetzung für die Verwendung der *AutoCompleteCustomSource*-Eigenschaft ist, dass Sie der *AutoCompleteSource*-Eigenschaft den Wert *CustomSource* zuweisen. Das „Wie" der Textvervollständigung legen Sie mit der *AutoCompleteMode*-Eigenschaft (*None*, *Suggest*, *Append*, *SuggestAppend*) fest.

Beispiel 8.6: Automatisches Vervollständigen

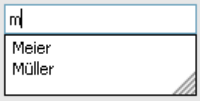
Entwurf

Einer *TextBox* (*AutoCompleteSource* = *CustomSource*, *AutoCompleteMode* = *Suggest*) wird die folgende *AutoCompleteCustomSource*-Eigenschaft zugewiesen:



Ergebnis

Nach Eingabe eines Buchstabens öffnet sich eine Liste mit passenden Einträgen (Auswahl über Maus oder Cursortasten):



HINWEIS: *AutoComplete* gibt es auch für die *ComboBox*!

8.2.5 MaskedTextBox

Dieses von der abstrakten *TextBoxBase* abgeleitete Steuerelement benutzt eine Maske, die es erlaubt, Benutzereingaben zu filtern.

Von zentraler Bedeutung ist, wen wundert es, die *Mask*-Eigenschaft. Dies ist ein String, der sich aus bestimmten Zeichen zusammensetzt, von denen die wichtigsten in der folgenden Tabelle erklärt werden. Einige Zeichen sind optional, sie können durch Eingabe eines Leerzeichens übergangen werden.

Zeichen	Beschreibung	Zeichen	Beschreibung
0	Ziffer zwischen 0 und 9	.	Dezimal-Trennzeichen
9	Ziffer oder Leerzeichen, optional	,	Tausender-Trennzeichen
#	Ziffer oder Leerzeichen mit Plus (+) oder Minus (-), optional	:	Zeit-Trennzeichen
L	Buchstabe (a-z, A-Z)	/	Datums-Trennzeichen
?	Buchstabe (a-z, A-Z), optional	<	Konvertiert alle folgenden Zeichen in Kleinbuchstaben
$	Währungssymbol	>	Konvertiert alle folgenden Zeichen in Großbuchstaben
&	Alphanumerisches Zeichen		

Dezimal- und Tausender-Trennzeichen sowie die Symbole für Datum, Zeit und Währung, entsprechen der standardmäßig eingestellten Kultur.

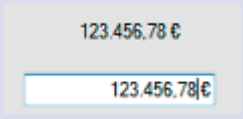
Beispiel 8.7: *MaskedTextBox*

C#

Ein Währungswert von 0 bis 999999 wird in eine *MaskedTextBox* eingegeben. Ist die Eingabe komplett, so wird der Inhalt in ein *Label* übernommen. Währungssymbol, Dezimal- und Tausender-Trennzeichen werden zur Laufzeit durch die kulturspezifischen Einstellungen ersetzt.

```
maskedTextBox1.Mask = "999,999.00 $";
...
private void maskedTextBox1_TextChanged(object sender, EventArgs e)
{
   if (maskedTextBox1.MaskCompleted) label1.Text = maskedTextBox1.Text;
}
```

Ergebnis

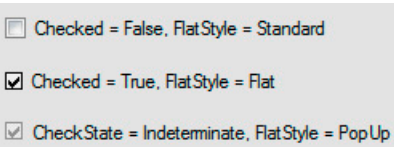


HINWEIS: Eine Leerzeichenkette als Maske wird den vorhandenen Inhalt unverändert belassen, die *MaskedTextBox* benimmt sich dann wie eine einzeilige „normale“ *TextBox*.

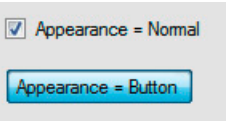
8.2.6 CheckBox

Bei der *CheckBox* entscheidet die *Checked*-Eigenschaft (*True/False*) darüber, ob das Häkchen gesetzt wurde oder nicht. Das äußere Erscheinungsbild kann über die *FlatStyle*-Eigenschaft geringfügig modifiziert werden. Für den Programmierer bietet sich zusätzlich die Möglichkeit, über *CheckState* alle drei Zustände zu bestimmen:

- *Checked*
- *Indeterminate*
- *Unchecked*



Die *Appearance*-Eigenschaft erlaubt es Ihnen, die *CheckBox* optisch in einen Button zu verwandeln, das Verhalten bleibt jedoch gleich:



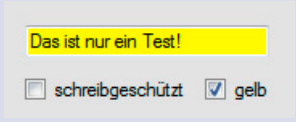
Eine Änderung der *Checked*-Eigenschaft löst das Ereignis *CheckedChanged* aus.

Beispiel 8.8: Mit zwei CheckBoxen werden Schreibschutz und Hintergrundfarbe einer *TextBox* eingestellt.

C#

```
private void checkBox_CheckedChanged(object sender, EventArgs e)
{
     if (checkBox1.Checked) textBox1.ReadOnly = true; else textBox1.ReadOnly =
false;

     if (checkBox2.Checked) textBox1.BackColor = Color.Yellow;
     else textBox1.BackColor = Color.White;
}
```

Ergebnis



Eine weitere interessante Eigenschaft ist *AutoCheck*. Ändert man ihren Wert von *True* nach *False*, so wird der Aktivierungszustand der *CheckBox* bei einem Klick nicht mehr automatisch umgeschaltet und der Programmierer muss sich selbst darum kümmern, z. B. durch Auswerten des *Click*-Ereignisses.

Beispiel 8.9: *Click*-Eventhandler einer *CheckBox* bei *AutoCheck = False*

C#

```
private void checkBox1_Click(object sender, EventArgs e)
{
    checkBox1.Checked = !checkBox1.Checked;
```

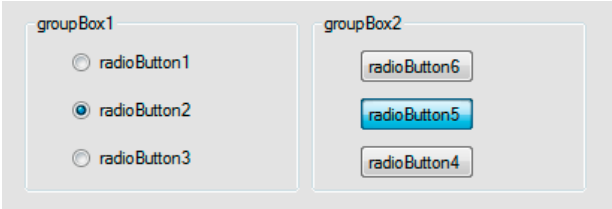
oder allgemeiner:

```
     CheckBox cb = (CheckBox)sender;
     cb.Checked = !cb.Checked;
}
```

8.2.7 RadioButton

Dieses Steuerelement dient zur Auswahl von Optionen innerhalb einer Anwendung. Im Unterschied zur *CheckBox* kann aber innerhalb einer Gruppe immer nur ein einziger *RadioButton* aktiv sein.

Meist fasst man mehrere *RadioButtons* mittels *GroupBox* (oder *Panel*) zu einer Optionsgruppe zusammen. Auch bei dieser Komponente können Sie die *Appearance*-Eigenschaft dazu nutzen, das Erscheinungsbild so zu ändern, dass für jeden *RadioButton* ein *Button* dargestellt wird:



Ausgewertet wird der Status über die *Checked*-Eigenschaft.

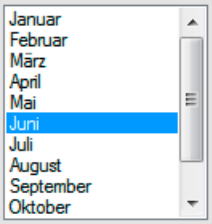
Beispiel 8.10: Ändern der Hintergrundfarbe des Formulars

C#

```
private void radioButton_CheckedChanged(object sender, EventArgs e)
{
    if (radioButton1.Checked) this.BackColor = Color.White;
    else if (radioButton2.Checked) this.BackColor = Color.Yellow;
    else if (radioButton3.Checked) this.BackColor = Color.Red;
}
```

8.2.8 ListBox

In einer *ListBox* kann eine Auflistung von Einträgen angezeigt werden, von denen der Benutzer mittels Maus oder Tastatur einen oder auch mehrere auswählen kann.



Die wichtigsten Eigenschaften zeigt die folgende Tabelle, auf datenbezogene Eigenschaften wird in Kapitel 11 näher eingegangen.

Eigenschaft	Beschreibung
Items *Items.Count*	... ist die Liste der enthaltenen Einträge, die Sie zur Entwurfszeit auch über den Zeichenfolgen-Editor eingeben können. Über *Items.Count* können Sie die Anzahl bestimmen.
Sorted	... legt fest, ob die Einträge alphabetisch geordnet erscheinen sollen (*True*/*False*)
SelectedIndex	... setzt bzw. ermittelt die Position (Index) des aktuellen Eintrags (-1, wenn nichts ausgewählt wurde)

Eigenschaft	Beschreibung
SelectionMode	... entscheidet, ob Einzel- oder Mehrfachauswahl zulässig ist
SelectedItem	... der Text des ausgewählten Eintrags

Die Methoden *Items.Add* und *Items.Remove/ItemsRemoveAt* fügen Einträge hinzu bzw. entfernen sie, *Items.Clear* löscht den gesamten Inhalt.

Beispiel 8.11: In die *ListBox* wird zehnmal „Hallo“ eingetragen, anschließend wird der zweite Eintrag über seinen Index gelöscht.

C#

```
for (int i= 1; i < 11; i++) listBox1.Items.Add(i.ToString + " Hallo");
listBox1.Items.RemoveAt(1);
```

Häufig will man auch den Inhalt eines (eindimensionalen) Arrays anzeigen.

Beispiel 8.12: Der Inhalt eines *String*-Arrays wird in eine *ListBox* übertragen.

C#

```
private void button1_Click(object sender, EventArgs e)
{
    string[] a = {"Sesam", "Weizen", "Hafer", "Gerste"};
    for (int i = 0; i <= a.GetUpperBound(0); i++) listBox1.Items.Add(a[i]);
    listBox1.SelectedIndex = 3;
}
```

Für den Programmierer ist das *SelectedIndexChanged*-Ereignis von besonderem Interesse, da es bei jedem Wechsel zwischen den Einträgen aufgerufen wird.

Beispiel 8.13: Nach Auswahl eines Eintrags aus einem Listenfeld wird dieser in eine *Textbox* übernommen.

C#

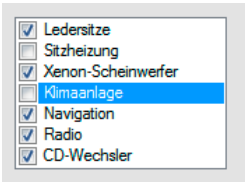
```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e);
{
   textBox1.Text = listBox1.SelectedItem.ToString();
}
```

Ob der Wechsel mittels Maus oder Tastatur erfolgt, ist in diesem Fall egal.

HINWEIS: Ist die Eigenschaft *SelectionMode* auf *MultiSimple* oder *MultiExtended* festgelegt, können Sie auch mehrere Einträge gleichzeitig auswählen. Über *SelectedItems* greifen Sie auf diese Einträge zu.

8.2.9 CheckedListBox

Hierbei handelt es sich um einen nahen Verwandten der „gemeinen“ Listbox mit dem Unterschied, dass die einzelnen Einträge gleichzeitig einen *True/False*-Wert (*Checked*) verwalten können.



Für das Setzen der Häkchen per Programm verwenden Sie die *SetItemChecked*-Methode.

Beispiel 8.14: Häkchen beim vierten Eintrag setzen

C#

```
checkedListBox1.SetItemChecked(3, true);
```

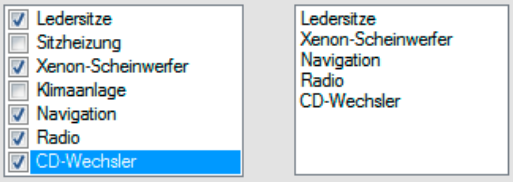
Über die *CheckedItems*-Collection haben Sie Zugriff auf alle markierten Einträge.

Beispiel 8.15: Anzeige aller markierten Einträge in einer weiteren *ListBox*

C#

```
for (int i = 0; i < checkedListBox1.CheckedItems.Count; i++)
     listBox1.Items.Add(checkedListBox1.CheckedItems[i]);
```

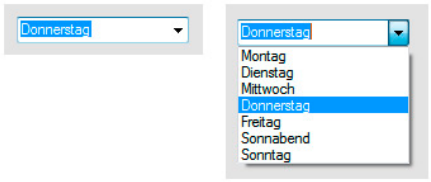
Ergebnis



HINWEIS: Da das Standardverhalten der *CheckedListBox* recht gewöhnungsbedürftig sein dürfte, sollten Sie besser die Eigenschaft *CheckOnClick* auf *True* setzen, damit nicht zweimal geklickt werden muss.

8.2.10 ComboBox

Eine *ComboBox* ist eine Mischung aus Text- und Listenfeld. Sie erlaubt also Eingaben und kann, im Unterschied zum Listenfeld, auch „aufgeklappt“ werden:



Hervorzuheben sind folgende Eigenschaften:

Eigenschaft	Beschreibung
DropDownStyle	... *Simple* (nur Texteingabe), *DropDown* (Texteingabe und Listenauswahl), *DropDownList* (nur Listenauswahl)
Items *Items.Count*	... ist die Liste der angezeigten Werte. Über *Items.Count* können Sie die Anzahl bestimmen.
MaxDropDownItems	... die maximale Höhe der aufgeklappten Listbox in Einträgen
Sorted	... legt fest, ob die Einträge alphabetisch geordnet erscheinen sollen (*True*/*False*)
SelectedIndex	... setzt bzw. ermittelt die Position (Index) des aktuellen Eintrags (-1, wenn nichts ausgewählt wurde)
SelectionMode	... entscheidet, ob Einzel- oder Mehrfachauswahl zulässig ist
Text	... der Text des ausgewählten Eintrags

Die Methoden *Items.Add* und *Items.Remove*/*ItemsRemoveAt* fügen Einträge hinzu bzw. entfernen sie, *Items.Clear* löscht den gesamten Inhalt.

Beispiel 8.16: *ComboBox* mit den Namen aller Monate füllen und den ersten Eintrag anzeigen

C#

```
  string[] monate = { "Januar", "Februar", "März", "April", "Mai", "Juni",
                      "Juli", "August", "September", "Oktober", "November",
"Dezember" };
  comboBox1.Items.AddRange(monate);
  comboBox1.SelectedIndex = 0;          // zeigt "Januar"
```

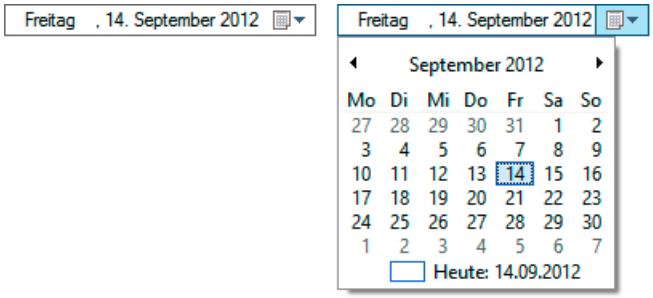
8.2.11 PictureBox

Dieses Control dient der Darstellung von fertigen Grafiken diverser Formate (BMP, GIF, TIFF, PNG ...). Außerdem kann mit Grafikmethoden in die *PictureBox* gezeichnet werden, weshalb man sie oft auch als „kleine Schwester“ des Formulars bezeichnet. Mittels *Image*-Eigenschaft können Sie diesem Control direkt eine Bildressource zuweisen. Über die *SizeMode*-Eigenschaft steuern Sie, wie die Grafik in den Clientbereich des Controls eingepasst wird bzw. ob sich das Control an die Grafik anpasst.

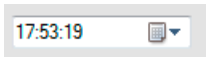
HINWEIS: Mehr zu dieser ebenso interessanten wie leistungsfähigen Komponente erfahren Sie im Kapitel „Grafikprogrammierung“ (Abschnitt 9.2.1).

8.2.12 DateTimePicker

Geht es um die platzsparende Auswahl eines Datums- oder eines Zeitwerts, sollten Sie sich mit der *DateTimePicker*-Komponente anfreunden. Diese funktioniert wie eine *ComboBox*, nach dem Aufklappen steht Ihnen ein recht komfortabler Kalender zur Verfügung:



Alternativ kann auch nur eine Uhrzeit bearbeitet werden (Eigenschaft *Format* = *Time*):



Auf alle Möglichkeiten der Konfiguration einzugehen, dürfte den Rahmen dieses Kapitels sprengen. Die wohl wichtigsten Eigenschaften dürften *MinDate*, *MaxDate* für die Beschränkung der Auswahl bzw. *Value* für den Inhalt des Controls sein.

Beispiel 8.17: Anzeige des Auswahlwerts

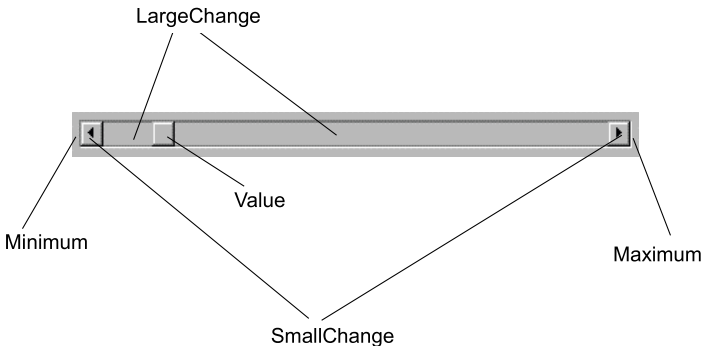
C#

```
private void dateTimePicker1_ValueChanged(object sender, EventArgs e)
{
   MessageBox.Show(dateTimePicker1.Value.ToLongDateString());
}
```

8.2.13 MonthCalendar

Funktionell ist diese Komponente dem *DateTimePicker* sehr ähnlich, der wesentliche Unterschied besteht darin, dass diese Komponente nicht auf- und zugeklappt werden kann und damit natürlich auch mehr Platz auf dem Formular verbraucht:

8.2.14 HScrollBar, VScrollBar



Diese Komponenten werden häufig zum Scrollen von Bild- und Fensterinhalten verwendet.

Maximum, *Minimum*, *Value*

Sie legen den größten bzw. kleinsten Einstellungswert fest bzw. bestimmen den aktuellen Wert (zwischen *Minimum* und *Maximum*).

LargeChange, *SmallChange*

Klickt man neben den „Schieber“, so wird der aktuelle Wert (*Value*) um *LargeChange* geändert, beim Klicken auf die Begrenzungspfeile hingegen nur um *SmallChange*.

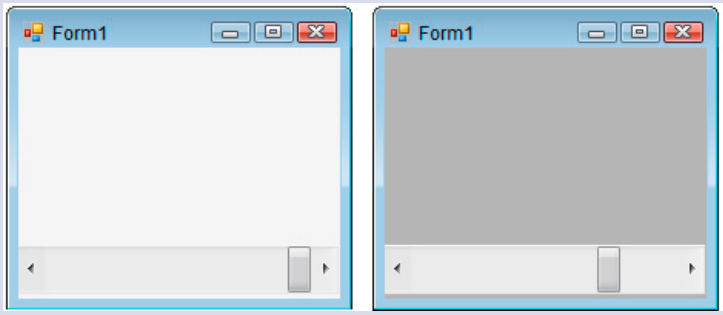
Eine wichtige Rolle spielt auch das *Scroll*-Ereignis, das immer dann eintritt, wenn die Position des Schiebers verändert wurde (*e.NewValue* enthält den neu gewählten Wert).

Beispiel 8.18: Einstellen der Formularhintergrundfarbe (Graustufe) mit einer horizontalen Scrollbar (*Minimum = 0, Maximum = 255*)

C#

```
private void hScrollBar1_Scroll(object sender, ScrollEventArgs e)
{
    this.BackColor = Color.FromArgb(255, e.NewValue, e.NewValue, e.NewValue);
}
```

Ergebnis



8.2.15 TrackBar

Neben den Scrollbars bietet sich auch die *TrackBar* für das schnelle Einstellen von Werten an. Hier zwei Varianten:

Die Breite kann bei *AutoSize = False* verändert werden. Ob die *TrackBar* horizontal oder vertikal erscheinen soll, können Sie mittels der *Orientation*-Eigenschaft festlegen. Die Anzahl der Teilstriche ergibt sich aus der Division der *Maximum*-Eigenschaft durch den Wert von *TickFrequency*. Bei der Standardeinstellung *Maximum = 10* und *TickFrequency = 1* ergeben sich also zehn Teilstriche.

Mit der *TickStyle*-Eigenschaft bestimmen Sie, auf welcher Seite die Teilstriche angezeigt werden sollen. Hier die Konstanten der *TickStyle*-Enumeration:

TickStyle	Erklärung
None	Keine Teilstriche
TopLeft	Teilstriche oben (bei horizontaler Ausrichtung) bzw. links (bei vertikaler Ausrichtung)
BottomRight	Teilstriche unten (bei horizontaler Ausrichtung) bzw. rechts (bei vertikaler Ausrichtung, Standardeinstellung)
Both	Teilstriche auf beiden Seiten

Da die übrigen wichtigen Eigenschaften (*Maximum*, *Minimum*, *Value*, *SmalChange*, *LargeChange*) sowie Ereignisse (*Scroll*, *ValueChanged*) weitgehend mit denen der Scrollbar übereinstimmen, möchten wir an dieser Stelle auf die vorhergehenden Abschnitte verweisen.

8.2.16 NumericUpDown

Wer hatte nicht schon die Aufgabe, einen Integer- oder Währungswert abzufragen, und war an den diversen Fehlermöglichkeiten gescheitert? *NumericUpDown* verspricht Abhilfe. Das Steuerelement entspricht der Kombination einer einzeiligen Textbox mit einer vertikalen Scrollbar.

Angefangen vom Definieren eines zulässigen Bereichs (*Maximum*, *Minimum*) über die Anzahl der Nachkommastellen (*DecimalPlaces*) bis hin zum Inkrement (*Increment*) bzw. zur Anzeige von Tausender-Trennzeichen (*ThousandsSeparator*) ist alles vorhanden, was das Programmiererherz begehrt. Last but not least soll die *Value*-Eigenschaft nicht vergessen werden, hier fragen Sie den eingegebenen Wert ab.

HINWEIS: Im Unterschied zu den Steuerelementen *ScrollBar* und *TrackBar* sind die gleichnamigen Eigenschaften von *NumericUpDown* vom Datentyp *decimal* anstatt *int*.

Von besonderem Interesse sind auch hier die Ereignisse *Scroll* (beim Klicken auf eine der beiden Schaltflächen) und *ValueChanged* (beim Ändern des Werts).

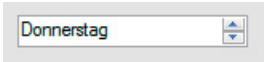
Beispiel 8.19: Meldung beim Überschreiten eines Limits

C#

```
private void numericUpDown1_ValueChanged(object sender, EventArgs e)
{
    if (numericUpDown1.Value > 1000)
MessageBox.Show("Wertebereichsüberschreitung!");
}
```

8.2.17 DomainUpDown

Für die Auswahl von Werten aus einer vorgegebenen Liste können Sie anstatt einer *ListBox* oder einer *ComboBox* auch eine *DomainUpDown*-Komponente verwenden. Dieses Control kombiniert die Features eines *NumericUpDown*-Steuerelements mit denen einer *ComboBox* (lässt sich allerdings nicht aufklappen). Rein äußerlich scheint es zunächst keinen Unterschied zu *NumericUpDown* zu geben, die Funktionalität ist jedoch grundsätzlich verschieden, bestehen doch die Einträge nicht aus Zahlen, sondern aus Strings (Reihenfolge kann mittels *Sort*-Eigenschaft alphabetisch sortiert werden).



Die zulässigen Auswahlwerte übergeben Sie der Komponente in der Collection *Items*, den gewählten Wert ermitteln Sie mit der *Text*-Eigenschaft oder Sie können auch den Listenindex mit *SelectedIndex* abrufen.

Beispiel 8.20: Der erste Listeneintrag wird angezeigt.

C#

```
domainUpDown1.SelectedIndex = 0;
```

8.2.18 ProgressBar

Ist Ihr Computer bei manchen Aufgaben nicht schnell genug, ist es sinnvoll, dem Nutzer ein Lebenszeichen zum Beispiel in Gestalt eines Fortschrittbalkens zu geben. Genau diese Aufgabe übernimmt die *ProgressBar*:

Die Eigenschaften *Minimum*, *Maximum* und *Value* sind vergleichbar mit denen der *ScrollBar*. Sie können *Value* einen Wert zuweisen, andererseits aber auch die Methode *PerformStep()* aufrufen.

Beispiel 8.21: Bei jedem Klick auf den *Button* bewegt sich der Balken weiter und hat nach zehnmaligem Klicken das Maximum erreicht.

C#

```
private void button1_Click(object sender, EventArgs e)
{
    progressBar1.Maximum = 100; progressBar1.Step = 10;
    progressBar1.PerformStep();
}
```

HINWEIS: Mit der *Style*-Eigenschaft (*Blocks*, *Continuous*, *Marquee*) können Sie das Aussehen des Balkens verändern.

8.2.19 RichTextBox

Wollten Sie nicht schon immer einmal Ihre Anwendungen mit einem kleinen Texteditor vervollständigen, der auch verschiedene Schriftarten, Schrift- und Absatzformate sowie Grafiken zulässt? Dann sollten Sie sich unbedingt das *RichTextBox*-Control ansehen.

HINWEIS: Mit RTF ist das „Rich Text"-Format gemeint, das als Austauschformat zwischen verschiedenen Textverarbeitungsprogrammen fungiert.

Viele Eigenschaften entsprechen denen der *TextBox*, so ist der gesamte Inhalt in der *Text*-Eigenschaft enthalten. Mit diversen *Selection...* -Eigenschaften lassen sich die markierten Textabschnitte formatieren.

Beispiel 8.22: Selektierten Text in Fett- und Kursivschrift formatieren

C#

```
richTextBox1.SelectionFont = new Font(richTextBox1.SelectionFont,
                                      richTextBox1.SelectionFont.Style ^
FontStyle.Bold);
richTextBox1.SelectionFont = new Font(richTextBox1.SelectionFont,
                                      richTextBox1.SelectionFont.Style ^
FontStyle.Italic);
```

Beispiel 8.23: Schriftgröße und -farbe ändern, Kursivschrift und markierte Zeile zentrieren

C#

```
richTextBox1.SelectionFont = new Font(richTextBox1.SelectionFont.Name, 20f);
richTextBox1.SelectionColor = Color.Red;
richTextBox1.SelectionAlignment = HorizontalAlignment.Center;
```

Ergebnis

Zum Laden bzw. Abspeichern von RTF-Dateien stehen die Methoden *LoadFile* und *SaveFile* zur Verfügung.

Beispiel 8.24: Laden der Datei *test.rtf* aus dem Anwendungsverzeichnis

C#

```
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        richTextBox1.LoadFile(Application.StartupPath + "\\test.rtf");
    }
    catch (System.IO.IOException ioe)
    {
         MessageBox.Show(ioe.Message);
    }
}
```

Bei der *RichTextBox* handelt es sich nur auf den ersten Blick um einen kompletten Texteditor, denn Sie haben sicherlich bereits festgestellt, dass Sie ja auch noch ein Menü bzw. eine Toolbar benötigen.

8.2.20 ListView

Die *ListView* ist eine ziemlich komplexe Komponente, sie ermöglicht Ihnen die Anzeige einer Liste mit Einträgen, die optional auch mit einem Icon zwecks Identifikation des Typs ausgestattet werden können, wozu zusätzlich eine oder mehrere *ImageList*-Komponenten erforderlich sind. Außerdem kann zu jedem Eintrag eine kleine Checkbox hinzugefügt werden, wodurch sich eine bequeme und übersichtliche Auswahlmöglichkeit ergibt.

Die Einsatzgebiete einer *ListView* sind äußerst vielgestaltig, z. B. Darstellen von Datenbankinhalten oder Textdateien. Außerdem kann die Komponente auch Nutzereingaben entgegennehmen, z. B. Dateiauswahl. Als Windows-Nutzer haben Sie garantiert schon mit diesem Control gearbeitet. Jeder einzelne Ordner auf dem Desktop ist im Grunde ein *ListView*-Objekt und fungiert als eine Art Container für eine Anzahl von einzelnen Items, die über Grafik und Text verfügen können.

ListViewItem

Das *ListViewItem*-Objekt repräsentiert einen einzelnen Eintrag (Item) in der *ListView*. Jedes *Item* kann mehrere *Subitems* haben, die zusätzliche Informationen bereitstellen.

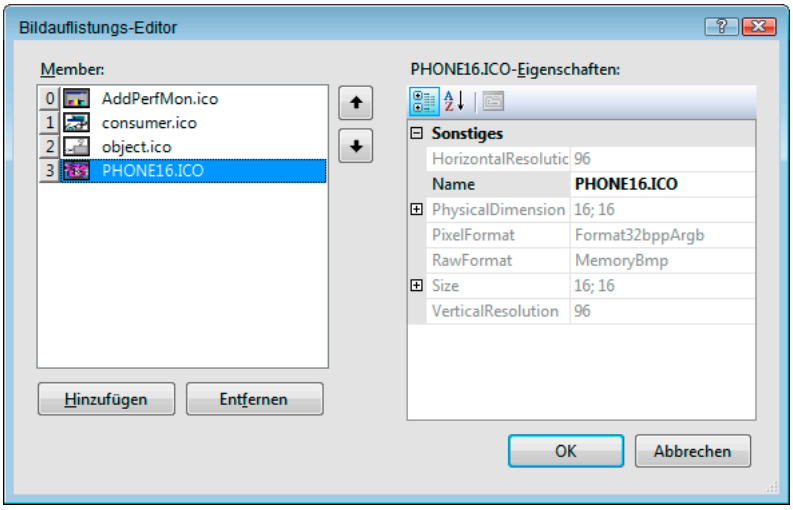
Die Anzeige der Items ist auf vier verschiedene Arten möglich:

- mit großen Icons,
- mit kleinen Icons,
- mit kleinen Icons in einer vertikalen Liste,
- Gitterdarstellung mit Spalten für Untereinträge (Detailansicht).

Die *ListView*-Komponente erlaubt einfache oder mehrfache Selektion, Letztere funktioniert ähnlich wie bei einer *ListBox*-Komponente.

ImageList

Meist wird die *ListView* zusammen mit einer (oder auch mehreren) *ImageList*-Komponenten eingesetzt, welche die Bilddaten speichern. Die Zuweisung erfolgt über die Eigenschaften *LargeImageList* bzw. *SmallImageList*. Jede *ImageList*-Komponente hat eine *Images*-Auflistung, die Sie über das Eigenschaften-Fenster (*F4*) erreichen und die Sie mithilfe des *Image-Auflistungs-Editor*s mit diversen Icons füllen können.



Übersichten zur ListView

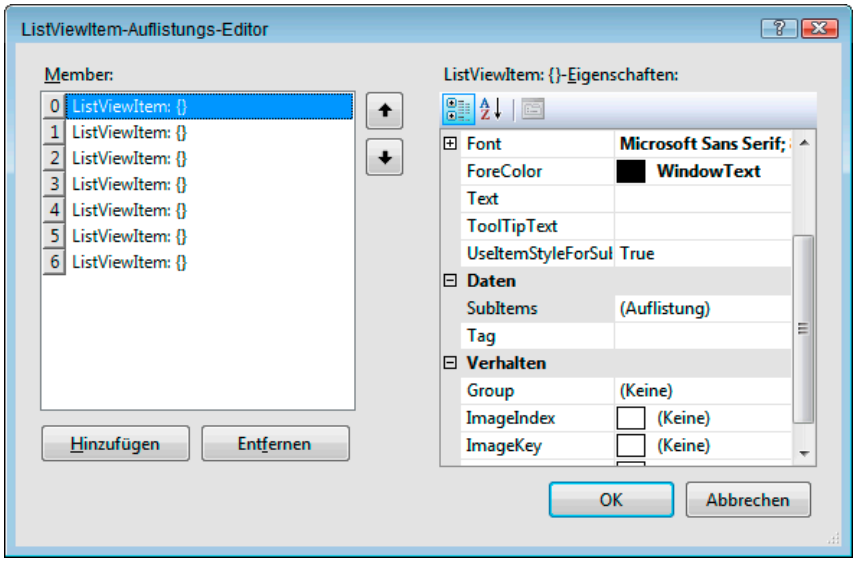
Eigenschaft	Bedeutung
View	... ermöglicht die Einstellung des Anzeigemodus (siehe oben): *List, SmallIcon, LargeIcon, Details*
LargeImageList *SmallImageList* *StateImageList*	Auswahl der *ImageList*-Objekte, welche die Bilddateien für große (32 x 32), kleine (16 x 16) Icons bzw. für die Darstellung der Checkbox bereitstellen (nur für *CheckBoxes = True*)
CheckBoxes	... Ein- bzw. Ausblenden der Checkbox (*True/False*)
CheckItems	... Zugriff auf die *CheckedListViewItemCollection*, um die aktivierten Einträge festzustellen
Columns	... Zugriff auf die *ColumnHeaderCollection* (nur für *View = View.Details*)
Items	... Hinzufügen/Entfernen von Einträgen durch Zugriff auf die *ListViewItemCollection*, die entsprechende Methoden bereitstellt
ImageIndex	... der Bildindex
LabelEdit	... erlaubt/verbietet das Editieren von Einträgen (*True*)
Sorting	... alphabetisches Sortieren der Einträge (*None*, *Ascending*, *Descending*)
AllowColumnReorder	nur für Detailansicht (*View = View.Details*): ... erlaubt Vertauschen der Spalten zur Laufzeit durch Anfassen mit der Maus (*True*)
FullRowSelect	... komplette Zeile wird selektiert (*True*)
GridLines	... Anzeige von Gitterlinien (*True*)
HeaderStyle	... versteckt Spaltentitel (*None*), zeigt ihn an (*NoneClickable*) bzw. Spaltentitel funktioniert wie ein Button (*Clickable*)
HideSelection	... selektierte Einträge bleiben markiert, wenn die *ListView* den Fokus verliert (*False*)

Methode	Beschreibung
BeginUpdate *EndUpdate*	... erlaubt das Hinzufügen von mehreren Items, ohne dass *ListView* jedes Mal neu gezeichnet wird
Clone	... überträgt Items in andere *ListView*
GetItemAt	... ermöglicht das Bestimmen des *Items* durch Klick mit der Maus auf ein *SubItem* (*View* = *View.Details*)
EnsureVisible	... ermöglicht die Anzeige eines gewünschten Eintrags nach Neuaufbau

Ereignis	... wird ausgelöst
BeforeLabelEdit *AfterLabelEdit*	... bevor bzw. nachdem ein Eintrag editiert wurde (nur bei *LabelEdit* = *True*), kann zur Gültigkeitsüberprüfung benutzt werden
ItemActivate	... wenn ein bestimmtes Item ausgewählt wurde, kann zum Auslösen von Aktionen benutzt werden
ColumnClick	... wenn in einen Spaltentitel geklickt wurde, kann z. B. zum Sortieren verwendet werden
ItemCheck	... wenn auf eine *CheckBox* geklickt wurde (nur für *CheckBoxes* = *True*)

Die Vorgehensweise scheint recht einfach zu sein:

- Verknüpfen Sie die *ListView* mit einer *ImageList* mit großen Icons (z. B. 32 x 32) per *LargeImageList*-Eigenschaft und mit einer weiteren *ImageList* die kleinen Icons (16 x 16) per *SmallImageList*-Eigenschaft.
- Neue Einträge erstellen Sie zur Entwurfszeit über die *Items*-Eigenschaft, die auch einen eigenen Editor bereitstellt. Wie Sie der folgenden Abbildung entnehmen können, verfügt jeder Eintrag wiederum über eine Auflistung *SubItems*, die den einzelnen Spalten in der Detailansicht entsprechen.

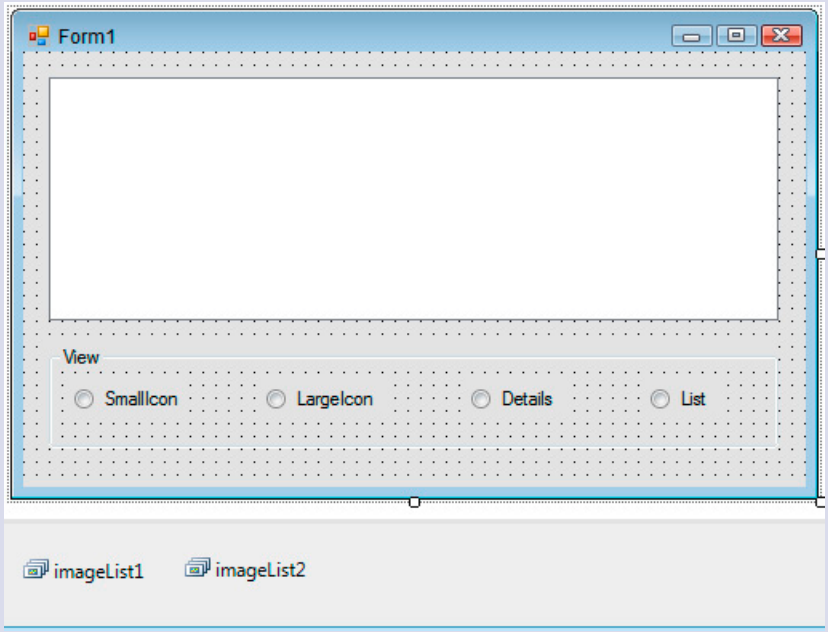


- Wichtig sind vor allem die Eigenschaften *ImageIndex* und *Text*, da beide das Aussehen des *Items* beeinflussen.
- Möchten Sie in der Detailansicht weitere Daten (Spalten) anzeigen, können Sie diese mit der *Columns*-Eigenschaft hinzufügen. Eingetragen werden später die *Text*-Eigenschaften der *SubItems*.
- Die Art der Anzeige legen Sie mit der *View*-Eigenschaft der *ListView* fest (Liste, Details etc.).

Beispiel 8.25: Stundenplan

C#

Es soll ein einfacher „Stundenplan“ erstellt werden. Die folgende Abbildung gibt einen Überblick über die Anordnung der Komponenten. Wichtig sind die *ListView*-Komponente (oben) und die beiden *ImageList*-Komponenten, die automatisch im Komponentenfach abgelegt werden.



Füllen Sie mit dem Image-Auflistungs-Editor (siehe oben) die erste *ImageList* mit „großen“ und die zweite *ImageList* mit „kleinen“ Icons. Weisen Sie der *LargeImageList*- und der *SmallImageList*-Eigenschaft der *ListView* die beiden *ImageList*-Komponenten zu.

Beim Laden von *Form1* wird dem *Click*-Ereignis der vier *RadioButton*s ein gemeinsamer Event-Handler zugewiesen:

```
private void Form1_Load(object sender, EventArgs e)
{
    radioButton1.Click += new EventHandler(this.commonClick);
    radioButton2.Click += new EventHandler(this.commonClick);
```

```
    radioButton3.Click += new EventHandler(this.commonClick);
    radioButton4.Click += new EventHandler(this.commonClick);
}
```

Der gemeinsame Event-Handler:

```
private void commonClick(object sender, System.EventArgs e)
{
    showListView();
}
```

Die Anzeige der *ListView*:

```
private void showListView()
{
   listView1.Items.Clear();
    if (radioButton1.Checked) listView1.View = View.SmallIcon;
    if (radioButton2.Checked) listView1.View = View.LargeIcon;
    if (radioButton3.Checked) listView1.View = View.Details;
    if (radioButton4.Checked) listView1.View = View.List;
```

Spalten für *Items* und *SubItems* definieren:

```
    listView1.Columns.Add("", -2, HorizontalAlignment.Left);
    listView1.Columns.Add("Montag", -2, HorizontalAlignment.Left);
    listView1.Columns.Add("Dienstag", -2, HorizontalAlignment.Left);
    listView1.Columns.Add("Mittwoch", -2, HorizontalAlignment.Center);
    listView1.Columns.Add("Donnerstag", -2, HorizontalAlignment.Left);
    listView1.Columns.Add("Freitag", -2, HorizontalAlignment.Center);
    listView1.LabelEdit = true;           // Editieren erlauben:
    listView1.AllowColumnReorder = true;  // Ändern der Spaltenanordnung erlauben
    listView1.CheckBoxes = true;
    listView1.FullRowSelect = true;
    listView1.GridLines = true;
    listView1.Sorting = SortOrder.Ascending;
```

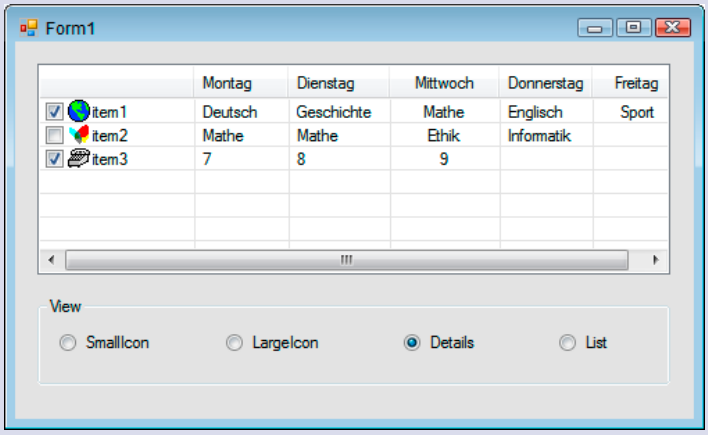
Schließlich drei *Items* mit Gruppen von *SubItems* erzeugen und zur *ListView* hinzufügen:

```
    ListViewItem item1 = new ListViewItem("item1", 0);
    item1.Checked = true;
    item1.SubItems.Add("Deutsch");
    item1.SubItems.Add("Geschichte");
    item1.SubItems.Add("Mathe");
    item1.SubItems.Add("Englisch");
    item1.SubItems.Add("Sport");
    listView1.Items.Add(item1);

    ListViewItem item2 = new ListViewItem("item2", 1);
    item1.Checked = true;
    item2.SubItems.Add("Mathe");
    item2.SubItems.Add("Mathe");
    item2.SubItems.Add("Ethik");
    item2.SubItems.Add("Informatik");
    item2.SubItems.Add("");
    listView1.Items.Add(item2);
```

```
    ListViewItem item3 = new ListViewItem("item3", 2);
    item3.Checked = true;
    item3.SubItems.Add("7");
    item3.SubItems.Add("8");
    item3.SubItems.Add("9");
    listView1.Items.Add(item3);
  }
```

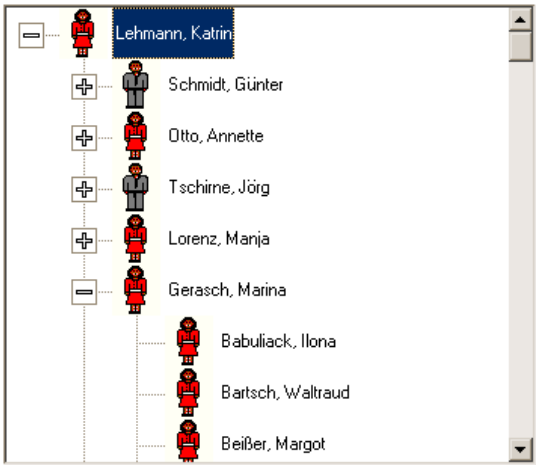
Ergebnis

Starten Sie das Programm und experimentieren Sie nun mit den verschiedenen Ansichten. Die Abbildung zeigt die Detailansicht:



8.2.21 TreeView

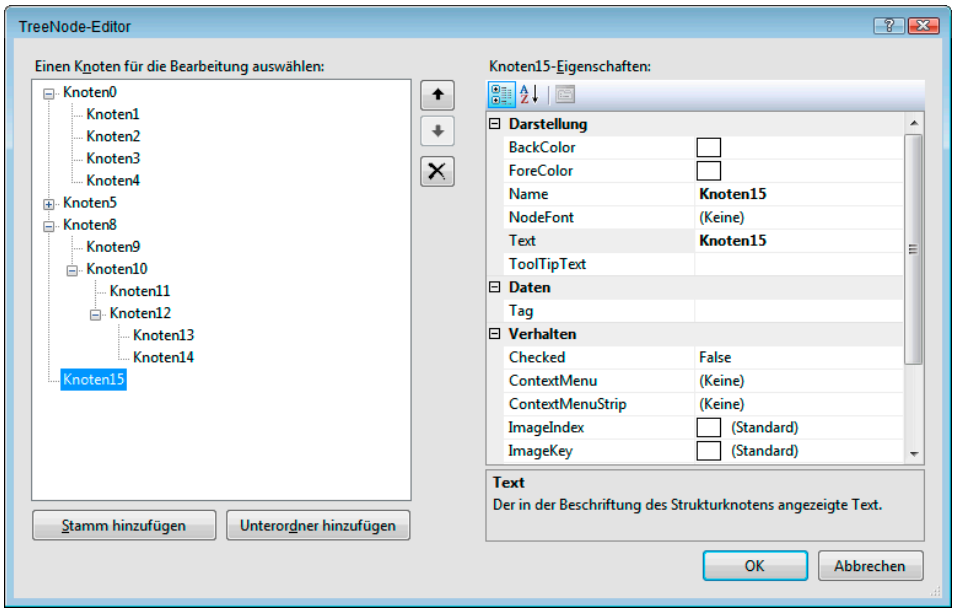
Wollen Sie den Anwendern Ihrer Programme mehr bieten als nur langweilige Listendarstellungen und primitive Eingabemasken? Oder müssen Sie hierarchische Abhängigkeiten grafisch darstellen?



Falls ja, dann kommen Sie wohl kaum um die Verwendung der *TreeView*-Komponente herum. Ganz abgesehen davon, dass hier eine Baumdarstellung wesentlich übersichtlicher ist als eine Tabelle, können Sie durch den Einsatz grafischer Elemente auch noch reichlich Eindruck schinden.

Wie auch die *ListView*-Komponente dient die *TreeView* „lediglich“ als Container für eine Liste von *TreeNodes* (Knoten), die jedoch im Gegensatz zur *ListView* hierarchisch angeordnet sind. Das heißt, ausgehend von einem Knoten werden weitere Unterknoten angeordnet usw.

Zur Entwurfszeit können Sie diese Knoten mit einem eigenen Editor (Eigenschaft *Nodes*) hinzufügen (ähnlich dem Erzeugen von Verzeichnissen auf einer Festplatte):



Knoten zur Laufzeit erzeugen

Zur Laufzeit gestaltet sich dieses Vorgehen schon etwas aufwendiger und erfordert vom Programmierer einiges Vorstellungsvermögen, gilt es doch, sich mithilfe von Collections und Methoden buchstäblich durch den Baum zu „hangeln“.

Beispiel 8.26: Dynamisches Füllen einer *TreeView*

C#

Zunächst den Baum löschen und die automatische Aktualisierung ausschalten (bessere Performance):

```
treeView1.BeginUpdate;
treeView1.Nodes.Clear;
```

Wir fügen den ersten (Root-)Knoten bzw. das erste Stammelement ein:

```
treeView1.Nodes.Add("Deutschland");
```

Da der erste Knoten den Index 0 in der *Nodes*-Collection erhält, können wir direkt über den Index auf diesen Knoten zugreifen und weitere untergeordnete Knoten erzeugen, die zur *Nodes*-Auflistung des Knotens hinzugefügt werden:

```
treeView1.Nodes[0].Nodes.Add("Bayern");
treeView1.Nodes[0].Nodes.Add("Sachsen");
treeView1.Nodes[0].Nodes.Add("Thüringen");
```

Das gleiche Prinzip auf die nächstfolgende Knotenebene angewendet:

```
treeView1.Nodes[0].Nodes[0].Nodes.Add("München");
treeView1.Nodes[0].Nodes[0].Nodes.Add("Nürnberg");
```

Doch eigentlich wird jetzt die Schreiberei etwas zu aufwendig. Bequemer ist die folgende Variante, bei der ein *TreeNode*-Objekt von der *Add*-Methode zurückgegeben wird:

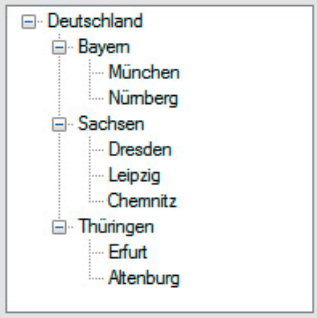
```
TreeNode n = treeView1.Nodes[0].Nodes[1];
n.Nodes.Add("Dresden");
n.Nodes.Add("Leipzig");
n.Nodes.Add("Chemnitz");
```

Es genügt also, dass wir uns auf einen bestimmten Knoten beziehen, um weitere Unterknoten zu erzeugen.

Zum Schluss noch die Aktualisierung einschalten, sonst ist nichts zu sehen:

```
treeView1.EndUpdate();
```

Ergebnis



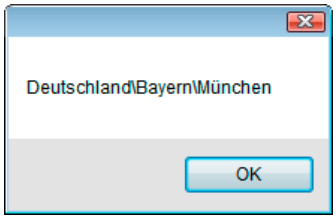
Auswerten des aktiven Knotens

Über das *AfterSelect*-Ereignis können Sie auswerten, welcher Knoten gerade aktiviert wurde. Der Knoten selbst wird als *e.Node*-Objekt als Parameter übergeben:

```
private void treeView1_AfterSelect(object sender, TreeViewEventArgs e)
{
```

```
        MessageBox.Show(e.Node.Text);
        MessageBox.Show(e.Node.FullPath);
    }
```

Während *Node.Text* lediglich die Beschriftung des aktuellen Knotens zurückgibt, enthält *Node.FullPath* den kompletten Pfad bis zur Root:



Wichtige Eigenschaften von TreeView

Die *TreeView* ist ein hochkomplexes Steuerelement und verfügt über eine Flut von Eigenschaften, Methoden und Ereignissen, bei denen man schnell die Übersicht verlieren kann. Gerade deshalb ist es sinnvoll, wenn man wenigstens die wichtigsten Klassenmitglieder kennt. Dabei müssen wir natürlich auch die untergeordnete Klasse *TreeNode*, die einen einzelnen Knoten innerhalb von *TreeView* repräsentiert, mit einbeziehen.

Eigenschaft	Beschreibung
CheckBoxes	... legt fest, ob CheckBoxen angezeigt werden (*True/False*)
FullRowSelect	... legt fest, ob die gesamte Breite farblich hervorgehoben wird (*True/False*)
HideSelection	... entfernt farbliche Hervorhebung des selektierten Knotens (*True/False*)
HotTracking	... legt fest, ob Knotenbezeichnung als Hyperlink erscheint (*True/False*)
ImageIndex	Index des Bildchens, das erscheint, wenn Knoten nicht selektiert ist
ImageList	... verweist auf die zugeordnete *ImageList*
Indent	... Einzugsbreite der untergeordneten Knoten (Pixel)
LabelEdit	... legt fest, ob Anwender Knotenbeschriftung editieren kann (*True/False*)
Nodes	... Auflistung mit allen *TreeNode*-Objekten der untergeordneten Ebene
SelectedImageIndex	... Index des Bildchens, das bei selektiertem Knoten angezeigt wird
SelectedNode	... liefert selektiertes *TreeNode*-Objekt
ShowLines	... gibt an, ob Linien zwischen den Knoten gezeichnet werden sollen (*True/False*)

Wichtige Methoden von TreeView

Methode	Beschreibung
BeginUpdate()	... deaktiviert das Neuzeichnen des Controls
CollapseAll()	... reduziert alle Knoten
EndUpdate()	... aktiviert das Neuzeichnen des Controls
ExpandAll()	... expandiert alle Knoten
Sort()	... sortiert alle Knoten

Wichtige Ereignisse von TreeView

Beim Aufklappen und Schließen der Knoten löst *TreeView* Ereignispärchen aus.

Ereignisse	Ereignis wird ausgelöst,
BeforeExpand *AfterExpand*	... bevor/nachdem Knoten geöffnet worden ist.
BeforeSelect *AfterSelect*	... bevor/nachdem Knoten ausgewählt wurde.
BeforeCollaps *AfterCollaps*	... bevor/nachdem Knoten geschlossen wurde.
NodeMouseClick	... wenn Benutzer auf einen Knoten klickt.

Wichtige Eigenschaften von TreeNode

Eigenschaft	Beschreibung
FirstNode	Das erste untergeordnete *TreeNode*-Objekt in der *Nodes*-Auflistung des aktuellen Knotens
Index	Position des aktuellen Knotens in der *Nodes*-Auflistung des übergeordneten Knotens
IsExpanded	*True*, falls *TreeNode*-Objekt expandiert ist
IsSelected	*True*, falls *TreeNode*-Objekt selektiert ist
LastNode	Das letzte untergeordnete *TreeNode*-Objekt in der *Nodes*-Auflistung des aktuellen Knotens
NextNode	Das nächste gleichrangige *TreeNode*-Objekt
Nodes	Auflistung, die alle *TreeNode*-Objekte der untergeordneten Ebene enthält
Parent	Das übergeordnete *TreeNode*-Objekt
PrevNode	Das vorhergehende gleichrangige *TreeNode*-Objekt

Wichtige Methoden von TreeNode

Methode	Beschreibung
Collapse()	Das *TreeNode*-Objekt wird reduziert.
Expand()	Das *TreeNode*-Objekt wird expandiert.
ExpandAll()	Alle untergeordneten *TreeNode*-Objekte werden expandiert.
Toggle()	*TreeNode* wechselt zwischen reduziertem und erweitertem Zustand.

8.2.22 WebBrowser

Seit der Einführung von Windows Forms 2.0 steht dieses hochkomplexe Steuerelement zur Verfügung (es handelt sich um eine verwaltete Wrapper-Klasse für das bekannte Internet Explorer ActiveX Control).

Die Programmierung eines voll funktionsfähigen Internetbrowsers erfordert erstaunlich wenig Aufwand.

Beispiel 8.27: *WebBrowser*

C#

Nachdem Sie das *WebBrowser*-Control von der Toolbox auf das Formular gezogen haben (*Dock = Bottom*), brauchen Sie nur noch eine *TextBox* für die URL und einen *Button* mit folgendem Ereigniscode hinzuzufügen:

```
private void button1_Click(object sender, EventArgs e)
{
      Uri aUri = new Uri(textBox1.Text);
      webBrowser1.Url = aUri;
}
```

Ergebnis

Die *Url*-Eigenschaft des *WebBrowser*-Controls ist vom Typ *Uri*. Diese Klasse repräsentiert einen *Uniform Resource Identifier* (URI) und kapselt den Zugriff auf die Bestandteile eines URI. Weitere wichtige Eigenschaften sind *IsWebWebBrowserContextMenuEnabled*, *SecurityLevel* und *WebBrowserShortcutEnabled*.

HINWEIS: Weitere Funktionen, wie die Aktualisierung der Anzeige etc., sind über das Kontextmenü des Webbrowsers verfügbar.

8.3 Container

Mit diesen Komponenten können Sie ein übersichtliches und anpassungsfähiges Outfit der Bedienoberfläche Ihrer Anwendung erreichen.

8.3.1 FlowLayout/TableLayout/SplitContainer

Diese Komponenten ermöglichen ein flexibles Formularlayout und wurden bereits im Abschnitt 8.2.6 des Vorgängerkapitels näher beschrieben.

8.3.2 Panel

Das auf den ersten Blick etwas unscheinbar wirkende Panel dürfte eines der wichtigsten Gestaltungsmittel für Dialogoberflächen sein. Die wichtigste Fähigkeit dieser Komponente: Sie kann weitere Komponenten in ihrem Clientbereich aufnehmen. Damit verhält sie sich fast wie ein Formular, es gibt eine *Controls*-Auflistung und sogar eine *AutoScroll*-Eigenschaft ist vorhanden!

Wie auch bei der *GroupBox* gilt:

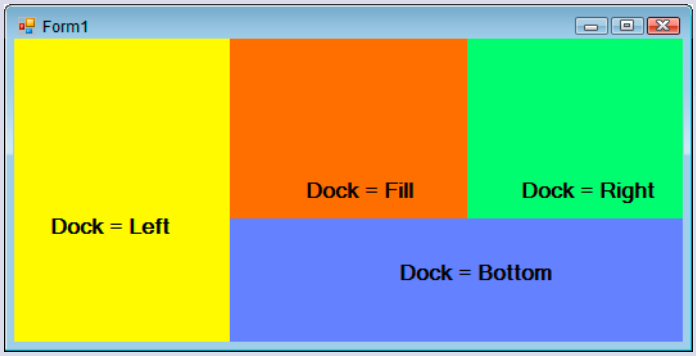
HINWEIS: Beachten Sie beim Entwurf, dass zuerst das *Panel* angelegt werden muss und erst dann Steuerelemente innerhalb desselben abgelegt werden können.

Oberflächen gestalten

Im Zusammenhang mit der *Dock*- bzw. *Anchor*-Eigenschaft bieten sich fast unbegrenzte Möglichkeiten, den Clientbereich des Formulars in einzelne Arbeitsbereiche aufzuteilen und dynamisch auf Größenänderungen zu reagieren.

HINWEIS: Nutzen Sie die *SplitContainer*-Komponente, kann der Nutzer zur Laufzeit die Größe der Panels ändern.

Beispiel 8.28: Mehrere mit Dock ausgerichtete *Panels*



8.3.3 GroupBox

Wenn man mehrere Steuerelemente mit einer *GroupBox* umgibt, können diese zu einer Einheit zusammengefasst werden. Insbesondere beim Gruppieren von *RadioButtons* dürfte die *GroupBox* die erste Wahl sein.

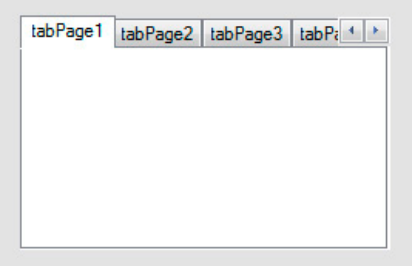
Wie auch das *Panel* verfügt die *GroupBox* über eine eigene *Controls*-Auflistung, zu der weitere Elemente mit *Add* bzw. *AddRange* hinzugefügt werden können.

Eine *GroupBox* reiht sich zwar mit ihrer *TabIndex*-Eigenschaft in die Tabulator-Reihenfolge der übrigen Steuerelemente des Formulars ein. Für die in der *GroupBox* enthaltenen Elemente gibt es jedoch eine eigene Reihenfolge, die mit *TabIndex = 0* beginnt.

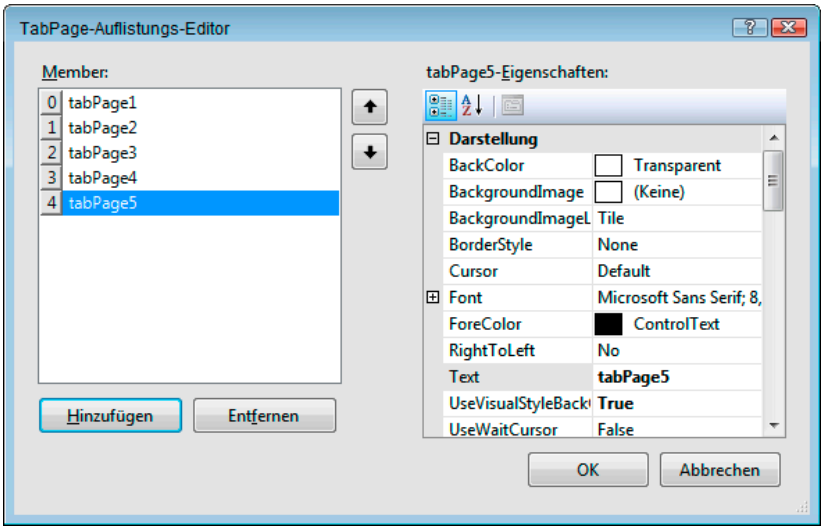
HINWEIS: Beachten Sie beim Entwurf, dass zuerst die *GroupBox* angelegt werden muss und erst dann Steuerelemente innerhalb derselben abgelegt werden können.

8.3.4 TabControl

Diese Komponente bietet die Funktionalität eines *Panel*s und die Möglichkeit, über ein Register auf weitere Seiten zuzugreifen:



Für das Zuweisen der Eigenschaften und für das Hinzufügen neuer Registerblätter steht Ihnen der „TabPage-Auflistungs-Editor“ zur Verfügung (erreichbar über die Eigenschaft *TabPages*):



Um per Code neue Registerkarten (*TabPage*-Objekte) hinzuzufügen, verwenden Sie (wie könnte es bei einem Container-Control auch anders sein) die *Add*-Methode der *Controls*-Auflistung des *TabControl*-Objekts.

Beispiel 8.29: Eine Registerkarte mit der Beschriftung „XYZ“ wird zu einem *TabControl* hinzugefügt.

C#

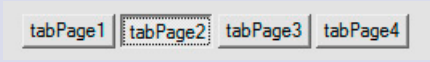
```
TabPage tabPg = new TabPage("XYZ");
tabControl1.Controls.Add(tabPg);
```

Wichtige Eigenschaften

Eigenschaft	Beschreibung
Alignment	... bestimmt Anordnung der Karteireiter
Appearance	... bestimmt Erscheinungsbild des *TabControl*s
ImageList	... die *ImageList* mit den auf den Registerkarten anzuzeigenden Bildern
ItemSize	... bestimmt die Größe der Registerkarten
MultiLine	... legt fest, ob Karteireiter mehrzeilig sein können
Padding	... bestimmt Abstand zwischen Beschriftung und Rand eines Karteireiters
RowCount	... liest Anzahl der Zeilen eines Karteireiters
SelectedIndex	... legt Index der aktuellen Registerkarte fest
SelectedTab	... bestimmt aktuell ausgewählte Registerkarte
TabPages	... liest Auflistung der Registerkarten des *TabControl*s

Beispiel 8.30: *Appearance*

Appearance = Buttons



Appearance = FlatButtons

Wichtige Ereignisse

Wenn sich die aktuelle Registerkarte ändert, haben wir es mit folgender Ereigniskette zu tun:

Deselecting → *Deselected* → *Selecting* → *Selected*

Ereignisse	Beschreibung
Deselecting / Deselected	Übergabe eines Objekts vom Typ *TabControlCancelEventArgs*
Selecting / Selected	Übergabe eines Objekts vom Typ *TabControlEventArgs*

Mit diesen Ereignissen können Sie auf einen Seitenwechsel reagieren (z. B. Sichern der Eingabewerte oder Initialisieren der enthaltenen Controls).

Mit der *Cancel*-Eigenschaft von *TabControlCancelEventArgs* können Sie den eingeleiteten Vorgang abbrechen (siehe auch *FormClosing*-Event).

Außer den oben aufgeführten Ereignissen ist noch das *SelectedIndexChanged*-Ereignis erwähnenswert, das bei Ändern der *SelectedIndex*-Eigenschaft auftritt.

8.3.5 ImageList

Um es gleich vorwegzunehmen, das *ImageList*-Steuerelement allein ist nur von begrenztem Nutzen. Es fungiert lediglich als (zur Laufzeit unsichtbarer) Container für mehrere (gleich große) Bitmaps, die Sie über ihren Index in der Liste ansprechen und in folgenden Steuerelementen anzeigen lassen können:

- *Label*, *Button*
- *ListView*, *TreeView*
- *ToolBar*
- *TabControl*

Um diese Controls mit der *ImageList* zu verbinden, setzen Sie einfach die jeweils vorhandene *ImageList*-Eigenschaft. Dies kann zur Entwurfszeit erfolgen, Sie dürfen aber auch zur Laufzeit diese Eigenschaft zuweisen.

Bevor Sie die Grafik in die Komponente einlesen, sollten Sie sich für die endgültige Größe (Eigenschaft *ImageSize*) entscheiden, da alle Grafiken auf diese Werte skaliert werden, egal wie groß sie vorher waren.

Über die *Images*-Collection rufen Sie zur Entwurfszeit einen eigenen Editor auf, in den Sie die Grafiken einfügen und, was auch wichtig ist, sie dann sortieren können.

Möchten Sie die Grafiken zur Laufzeit laden, verwenden Sie die *Add*-Methode der *Images*-Collection.

Beispiel 8.31: Hinzufügen von Grafiken zur Laufzeit

C#

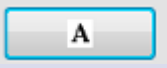
```
imageList1.Images.Add(new Bitmap("C:\\temp\\Bold.bmp"));
imageList1.Images.Add(new Icon("C:\\temp\\Italic.ico"));
```

Beispiel 8.32: Anzeige in einem Button

C#

```
button1.Text = String.Empty;
button1.ImageList = imageList1;
button1.ImageIndex = 0;
```

Ergebnis



HINWEIS: Verwenden Sie die Eigenschaft *ImageAlign*, um die Grafik in der jeweiligen Komponente auszurichten.

8.4 Menüs & Symbolleisten

In diesem Toolbox-Abschnitt finden Sie diverse komplexe Komponenten, die die bequeme Bedienbarkeit einer Anwendung ermöglichen.

8.4.1 MenuStrip und ContextMenuStrip

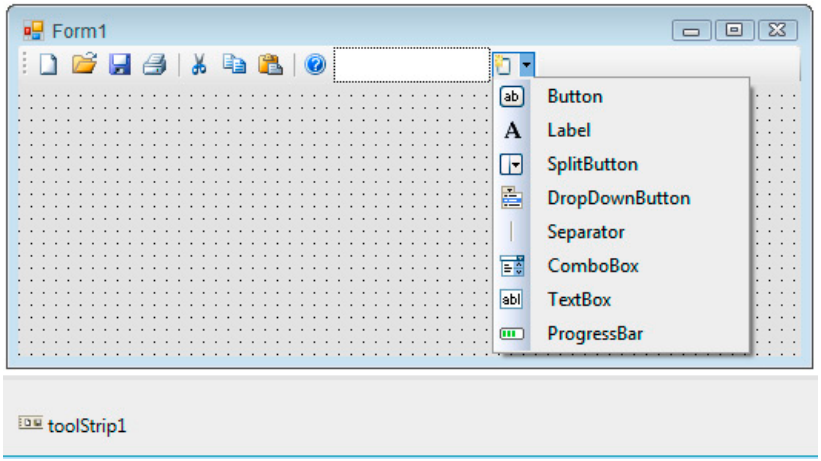
Auf beide Menü-Komponenten, die ab .NET 2.0 die veralteten *MainMenu* und *ContextMenu* abgelöst haben, wurde bereits in Kapitel 2 ausführlicher eingegangen. Im Zusammenhang damit sind u. a. die folgenden Objekte von Bedeutung:

- *ToolStripMenuItem*
- *ToolStripComboBox*
- *ToolStripTextBox*

HINWEIS: Beachten Sie, dass sowohl *ToolStripComboBox* als auch *ToolStripTextBox* keine Untermenüs enthalten können.

8.4.2 ToolStrip

Dieses Steuerelement ist Nachfolger der *ToolBar* und erlaubt einen komfortablen Entwurf von frei konfigurierbaren Werkzeugleisten, die man mit unterschiedlichen Schalt- und Anzeigeelementen (*Button*, *Label*, *SplitButton*, *ProgressBar* ...) besetzen kann.



8.4.3 StatusStrip

Hier handelt es sich um den Nachfolger der altbekannten *StatusBar*, wie sie in der Regel an den unteren Rand des Formulars angedockt wird. Genauso wie das *ToolStrip* kann man diese Komponente mit unterschiedlichen Schalt- und Anzeigeelementen (*Button*, *Label*, *SplitButton*, *ProgressBar*, ...) bestücken.

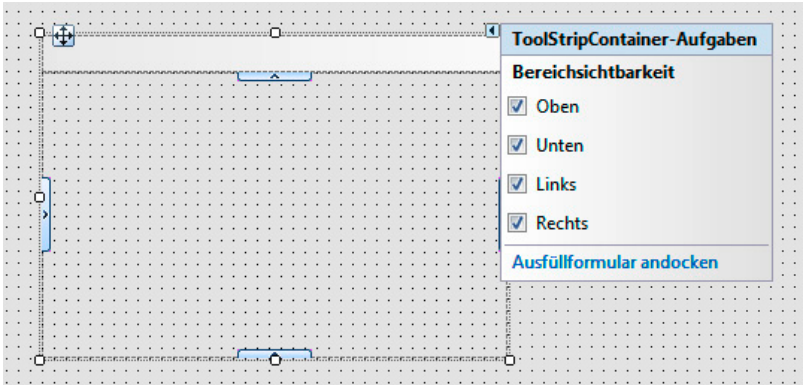
8.4.4 ToolStripContainer

Diese Container-Komponente übernimmt das flexible Positionieren von *MenuStrip* und *StatusStrip*, die zur Entwurfszeit in einen der vier Seitenbereiche (bzw. Panels) des *ToolStripContainer*s gezogen werden. Der Zentralbereich enthält die eigentlichen Steuerelemente des Formulars.

Die Seitenbereiche sind Objekte vom Typ *ToolStripPanel* und können durch Klick auf die entsprechende Markierung geöffnet werden.

In der Regel setzen Sie die *Dock*-Eigenschaft des *ToolStripContainer*s auf *Fill*, sodass diese Komponente den gesamten Clientbereich des Formulars einnimmt.

Alternativ können Sie aber auch im SmartTag-Fensterchen den Eintrag „Ausfüllformular andocken“ wählen, um festzulegen, an welchen Seitenrändern Menü- und Symbolleisten zur Laufzeit angedockt werden können.



8.5 Daten

Die in diesem Abschnitt enthaltenen Controls sollen in der Regel das Programmieren von ADO.NET-Anwendungen erleichtern. Ausführliche Beschreibungen und Anwendungsbeispiele finden Sie im Buch in den Kapiteln 14 und 15.

8.5.1 DataSet

Mit dieser Komponente erzeugen Sie eine Instanz eines typisierten oder aber eines nicht-typisierten *DataSet*s. Das *DataSet* ist das zentrale Objekt der ADO.NET-Technologie.

8.5.2 DataGridView/DataGrid

Dieses Control ist der Nachfolger des *DataGrid*. Letzteres steht nicht nur aus Kompatibilitätsgründen weiterhin zur Verfügung, es bietet auch den Vorteil, dass mehrere Tabellen und ihre Beziehungen gleichzeitig angezeigt werden können. Das *DataGridView* hingegen verfügt über ein deutlich umfangreicheres Objekt- und Ereignismodell, so können z. B. die Spaltenelemente auch aus anderen Komponenten bestehen, z. B. aus ComboBoxen.

HINWEIS: Das *DataGridView* ist nicht zwingend nur für den Einsatz in Datenbankanwendungen konzipiert, sondern für die tabellenförmige Darstellung nahezu beliebiger Daten geeignet.

8.5.3 BindingNavigator/BindingSource

Diese Steuerelemente sind im Zusammenhang mit der Datenbindung von Steuerelementen von Interesse und werden in Kapitel 11 besprochen.

8.5.4 Chart

Microsoft bietet ebenfalls eine eigene *Chart*-Komponente an, die in fast gleicher Form unter ASP.NET zur Verfügung steht.

So lassen sich mit der Komponente 35 verschiedene Diagrammtypen in 2D/3D-Darstellung anzeigen. Sie als Programmierer haben Einfluss auf Farben, Schatten, Betrachtungspunkte etc., es dürfte für jeden etwas dabei sein.

Beispiel 8.33: Chart mit Daten füllen

C#

```
        private void button1_Click(object sender, EventArgs e)
        {
```

Der Standardreihe eine neue Bezeichnung zuweisen:

```
            chart1.Series[0].Name = "Umsätze TFT";
```

Vier Datenpärchen übergeben:

```
            chart1.Series[0].Points.AddXY(2007, 10);
            chart1.Series[0].Points.AddXY(2008, 25);
            chart1.Series[0].Points.AddXY(2009, 75);
            chart1.Series[0].Points.AddXY(2010, 150);
```

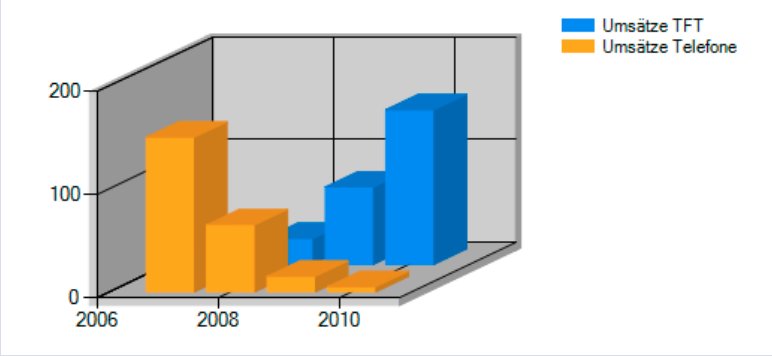
Wir erzeugen eine weitere Reihe und vergeben gleich eine Bezeichnung:

```
            chart1.Series.Add("Umsätze Telefone");
```

Auch hier wieder vier Datenpärchen zuweisen:

```
            chart1.Series[1].Points.AddXY(2007, 150);
            chart1.Series[1].Points.AddXY(2008, 65);
            chart1.Series[1].Points.AddXY(2009, 15);
            chart1.Series[1].Points.AddXY(2010, 5);
        }
```

Ergebnis



Wie Sie sehen, ist mit wenigen Zeilen ein recht ansprechendes Diagramm erstellt, das Sie zum Beispiel mit

```
chart1.Printing.PrintPreview();
```

in einer Druckvorschau anzeigen können oder auch gleich mit

```
chart1.Printing.Print(false);       // ohne Druckerauswahl
```

auf dem Standarddrucker ausgeben können.

Selbstverständlich ist auch eine direkte Datenbindung an diverse Datenquellen möglich, was bereits auf den Datenbankentwickler als Hauptzielgruppe hinweist.

8.6 Komponenten

In diesem Abschnitt sind einige häufig benötigte Controls enthalten, die die Arbeit des Programmierers unterstützen und die nicht direkt auf dem Formular, sondern im Komponentenfach ihren Platz finden.

8.6.1 ErrorProvider

Mit diesem Control können Sie auf einfache Weise eine Eingabevalidierung realisieren, um den Benutzer Ihrer Programme sofort auf ungültige Eingaben hinzuweisen (in diesem Fall erscheint ein Warnsymbol, z. B. neben einer *TextBox*).

8.6.2 HelpProvider

Hierbei handelt es sich um eine nicht sichtbare Komponente zur Einbindung der Hilfefunktionalität in ein Formular.

8.6.3 ToolTip

Über dieses Control steuern Sie, wie und vor allem wann Tooltips (die kleinen gelben Hinweisfähnchen) angezeigt werden. Der eigentliche Tooltip wird immer bei den jeweiligen Eigenschaften der Anzeige-Controls verwaltet.

8.6.4 BackgroundWorker

Diese Komponente erlaubt es Ihnen, auf einfache Weise einen Hintergrundthread zu starten (siehe im Buch Kapitel 10).

8.6.5 Timer

Diese Komponente löst in bestimmten Zeitabständen das *Tick*-Ereignis aus. Wesentlich ist die *Interval*-Eigenschaft, sie legt die Zeit (in Millisekunden) fest, der Wert 0 (null) ist nicht zulässig.

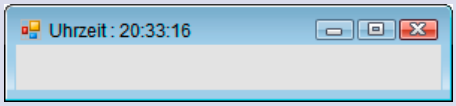
HINWEIS: Im Unterschied zu anderen Komponenten hat die *Enabled*-Eigenschaft standardmäßig den Wert *False*, zum Einschalten des *Timer*s müssen Sie diese Eigenschaft auf *True* setzen!

Beispiel 8.34: Die Uhrzeit wird per Timer im Formularkopf angezeigt (*Interval = 1000*).

C#

```
private void timer1_Tick(object sender, EventArgs e)
{
    this.Text = "Uhrzeit : " + DateTime.Now.ToLongTimeString();
}
```

Ergebnis



8.6.6 SerialPort

Dieses Control ermöglicht den Zugriff auf die serielle Schnittstelle des Rechners und ermöglicht damit die Steuerung von peripheren Geräten, die über ein solches RS 232-Interface verfügen.

8.7 Drucken

Relativ bescheiden ist das Angebot an Steuerelementen für alle Aktivitäten rund um die Druckausgabe (siehe dazu Kapitel 10), was jedoch nicht bedeutet, dass Sie beim Drucken nicht umfangreiche Unterstützung erfahren.

8.7.1 PrintPreviewControl

Dieses Steuerelement bündelt alle Aktivitäten rund um die Anzeige einer Druckvorschau. An dieser Stelle wollen wir allerdings nicht vorgreifen, sondern Sie gleich an das Kapitel 10 (Druckausgabe) verweisen.

8.7.2 PrintDocument

Mit dieser Komponente sind Sie in der Lage, beliebige Grafiken zu Papier zu bringen. Den kompletten Überblick über die Verwendung finden Sie ebenfalls im Kapitel 10.

8.8 Dialoge

Diese Komponenten kapseln die bekannten Windows-Dialoge.

8.8.1 OpenFileDialog/SaveFileDialog/FolderBrowserDialog

Hier handelt es sich um die Dialoge zum Öffnen und zum Abspeichern von Dateien sowie zum Blättern in Verzeichnissen, mehr dazu im Datei-Kapitel im Buch (Abschnitt 8.6) bzw. im Praxisbeispiel im buch in Abschnitt 9.8.1 (Zugriff auf Textdatei).

8.8.2 FontDialog/ColorDialog

Von Schriftartendialog und Farbdialog haben Sie sicher bereits während der Programmentwicklung Gebrauch gemacht, da die entsprechenden Eigenschaften bzw. Enumerationen bequem damit eingestellt werden können. Wie Sie diese Komponenten richtig einsetzen, zeigt das Grafik-Kapitel 9.

8.9 WPF-Unterstützung mit dem ElementHost

Möchten Sie im Rahmen Ihrer Windows Forms-Anwendung auch mit WPF-Controls (*Windows Presentation Foundation*) arbeiten, können Sie dies über den sogenannten *ElementHost* realisieren. Dazu fügen Sie Ihrem Projekt (nicht dem Form) zunächst ein WPF-Control hinzu. In dieses lassen sich beliebige WPF-Komponenten einfügen und per XAML-konfigurieren, das Control bildet lediglich den Container. Abschließend kann das neu erstellte WPF-Control einem *ElementHost* zugewiesen werden. Nutzen Sie dazu die *Child*-Eigenschaft oder das Aufgabenmenü des *ElementHost*-Controls.

Das Konzept dürfte Sie an die Benutzersteuerelemente erinnern. Auch hier wird zunächst in einem Container entwickelt, der abschließend in das Formular eingeblendet wird.

Ein durchgehendes Beispiel finden Sie im Praxisbeispiel in Abschnitt 8.10.4, WPF-Komponenten mit dem ElementHost anzeigen.

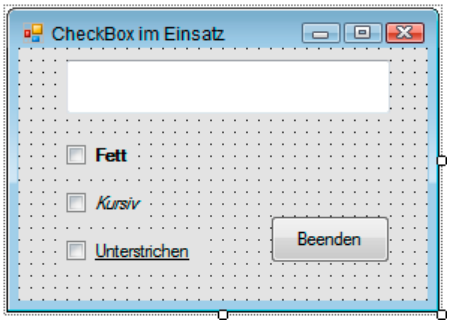
8.10 Praxisbeispiele

8.10.1 Mit der CheckBox arbeiten

Dieses kleine Beispiel demonstriert den Einsatz des *CheckBox*-Steuerelements anhand einer durchaus sinnvollen Aufgabenstellung: Der Schriftstil (fett, kursiv, unterstrichen) des Inhalts einer *TextBox* soll geändert werden. En passant wird auch der Umgang mit dem *Font*-Objekt erklärt.

Oberfläche

Nur eine *TextBox* (*MultiLine=True*), drei *CheckBox*en und ein *Button* sind erforderlich.



Quellcode

```
public partial class Form1 : Form
{ ...
```

Alle drei *CheckBox*en teilen sich einen gemeinsamen Eventhandler für das *CheckedChanged*-Ereignis. Geben Sie den Rahmencode komplett selbst ein (also nicht durch die IDE generieren lassen) und weisen Sie erst dann auf der „Ereignisse"-Seite des Eigenschaftenfensters diesen Eventhandler für jede der drei *CheckBox*en einzeln zu.

```
    private void CBChanged(object sender, EventArgs e)
    {
```

Der resultierende Schriftstil wird durch bitweise ODER-Verknüpfung schrittweise zusammengebaut:

```
        FontStyle ftStyle = FontStyle.Regular;
        if (checkBox1.Checked) ftStyle |= FontStyle.Bold;
        if (checkBox2.Checked) ftStyle |= FontStyle.Italic;
        if (checkBox3.Checked) ftStyle |= FontStyle.Underline;
```

Ein neues *Font*-Objekt wird auf Basis des geänderten Schriftstils erzeugt und der *TextBox* zugewiesen:

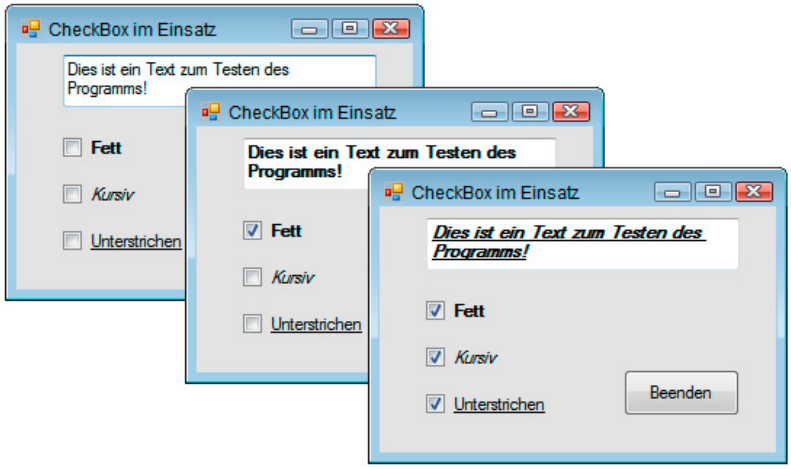
```
        textBox1.Font = new Font(textBox1.Font, ftStyle);
    }
```

Die „Beenden“-Schaltfläche:

```
    private void button1_Click(object sender, EventArgs e)
    {
        this.Close();
    }
}
```

Test

Da alle drei Fontstile miteinander kombiniert werden können, gibt es insgesamt acht Möglichkeiten (drei davon zeigt die folgende Abbildung).



8.10.2 Steuerelemente per Code selbst erzeugen

Der Windows Forms-Designer erlaubt zwar einen bequemen visuellen Entwurf von Benutzerschnittstellen, allerdings kann es sich manchmal durchaus lohnen, wenn man (zumindest teilweise) auf die Dienste des Designers verzichtet und die Steuerelemente „per Hand“ programmiert. Dabei gewinnt man nicht nur einen tieferen Einblick in die Prinzipien der objekt- und ereignisorientierten Programmierung, sondern kann auch elegante Oberflächen gestalten, die Aussehen und Funktionalität zur Laufzeit ändern. Ein besonderer Vorteil ergibt sich beim Entwurf von zahlreichen, funktionell gleichartigen, Steuerelementen (Control-Arrays), denn hier bietet Ihnen der Designer kaum Unterstützung.

Die folgende Demo löst das gleiche Problem wie im Vorgängerbeispiel (Zuweisung des Schriftstils für eine *TextBox*), wobei die Steuerelemente per Code erzeugt werden. Die drei *CheckBox*en sind dabei als Steuerelemente-Array organisiert.

Oberfläche

Gönnen Sie Ihrem Windows Forms-Designer eine Pause, das nackte Formular (*Form1*) genügt!

Quellcode

```
public partial class Form1 : Form
{
```

Die benötigten globalen Variablen:

```
    private TextBox tb = new TextBox();    // Verweis auf TextBox
    private CheckBox[] checkBoxes = null;  // Verweis auf CheckBox-Array
    private int anz = 0;                   // Anzahl der CheckBoxen
```

Initialisiertes Array mit den verwendeten Schriftstilen:

```
    private FontStyle[] ftStyles =
                { FontStyle.Regular, FontStyle.Bold, FontStyle.Italic,
FontStyle.Underline };
```

Im Konstruktor des Formulars werden die benötigten Steuerelemente erzeugt und mit ihren wesentlichen Eigenschaften initialisiert:

```
    public Form1()
    {
        InitializeComponent();
```

TextBox erzeugen:

```
        int xpos = 25, ypos = 10;       // linke obere Ecke der TextBox
        tb.Location = new Point(xpos, ypos);
        tb.Multiline = true;
        tb.Width = 200; tb.Height = 40;
```

Keinesfalls vergessen darf man das Hinzufügen des Steuerelements zur *Controls*-Auflistung des Formulars (ansonsten bleibt es verborgen):

```
        this.Controls.Add(tb);
```

Alle *CheckBox*en erzeugen:

```
        anz = ftStyles.Length-1;
        checkBoxes = new CheckBox[anz];
        for (int i = 0; i < anz; i++)
        {
            checkBoxes[i] = new CheckBox();
            checkBoxes[i].Location = new Point(xpos, ypos + 50 + i * 25);
            checkBoxes[i].Text = ftStyles[i+1].ToString();
```

Gemeinsame Ereignisbehandlung zuweisen:

```
            checkBoxes[i].CheckedChanged += new EventHandler(CBChanged);
        }
```

Alle *CheckBox*en zum Formular hinzufügen

```
        this.Controls.AddRange(checkBoxes);

        tb.Text = "Dies ist ein Text zum Testen des Programms!";
    }
```

Die Ereignisbehandlung für alle *CheckBox*en:

```
    private void CBChanged(object sender, EventArgs e)
    {
```

Zunächst normalen Schriftstil einstellen:

```
        FontStyle ftStyle = ftStyles[0];
```

Alle *CheckBox*en durchlaufen:

```
        for (int i = 0; i < anz; i++)
        {
            CheckBox cb = checkBoxes[i];
            if (cb.Checked)
```

Den resultierenden Schriftstil durch bitweises ODER zusammensetzen:

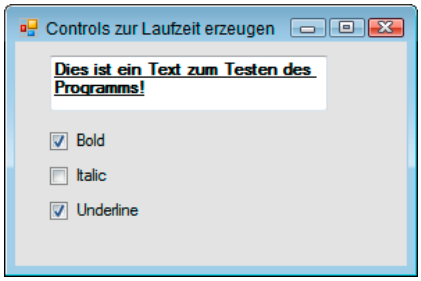
```
                ftStyle |= ftStyles[i + 1];
        }
```

Neues *Font*-Objekt erzeugen und der *TextBox* zuweisen:

```
        tb.Font = new Font(tb.Font, ftStyle);
    }
}
```

Test

Es gibt keine wesentlichen Unterschiede zum Vorgängerbeispiel. Aus Gründen der Einfachheit wurde auf eine deutsche Beschriftung der drei *CheckBox*en und auf eine „Beenden"-Schaltfläche verzichtet:



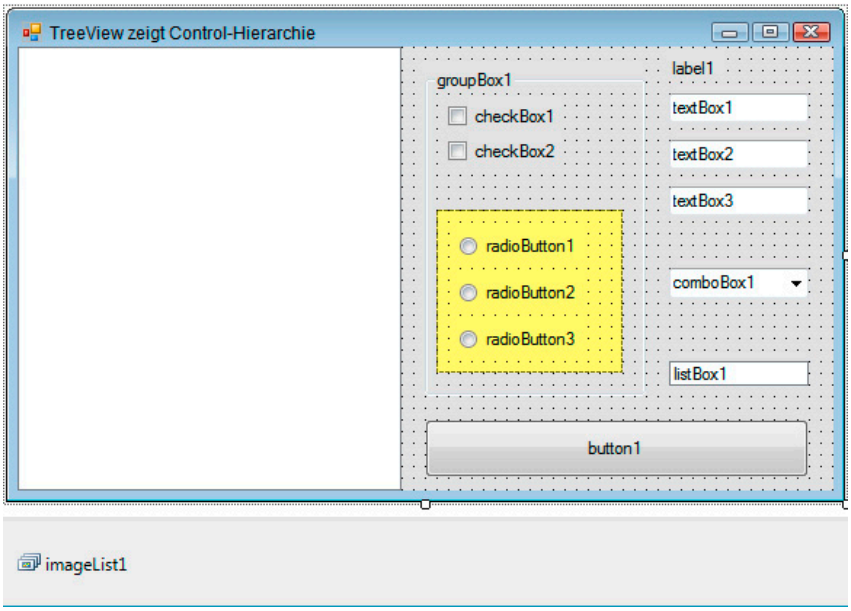
8.10.3 Controls-Auflistung im TreeView anzeigen

Das *TreeView*-Steuerelement gehört mit zu den komplexesten Windows Forms-Controls. Es ist zur Anzeige beliebig verschachtelter hierarchischer Strukturen, wie z. B. Verzeichnisbäume, bestimmt. Auch die *Controls*-Auflistung des Formulars liefert ein hervorragendes Beispiel, um den grundlegenden Umgang mit dieser Komponente zu demonstrieren.

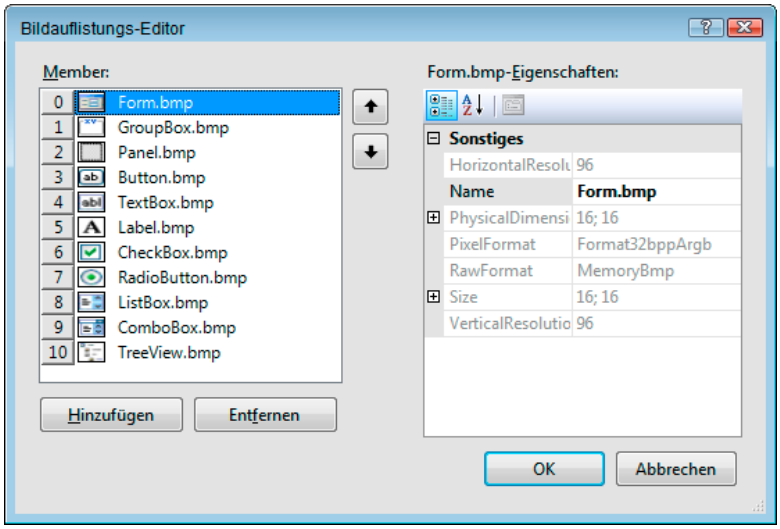
Oberfläche

Eine *TreeView*-Komponente (*Dock* = *Left*), eine *ListView*-Komponente sowie eine vom Prinzip her beliebige Anzahl von Steuerelementen gestalten die Oberfläche von *Form1*. Um eine aussagekräftige Tiefe der Hierarchieebenen zu erhalten, haben wir eine *GroupBox* mit aufgenommen, in welcher u. a. auch ein *Panel* enthalten ist. Beide Komponenten verfügen, genauso wie das Formular, über eigene *Controls*-Auflistungen.

Um die Strukturansicht attraktiv und übersichtlich zu gestalten, sollte jeder Knoten mit einem Icon ausgestattet werden, welches das entsprechende Steuerelement symbolisiert. Mit dem an die *ImageList* angeschlossenen „Image-Auflistungs-Editor“ ist das Hinzufügen der dafür benötigten Bilddateien (16 x 16 Pixel) im Handumdrehen erledigt.



HINWEIS: Vergessen Sie nicht, die *ImageList*-Eigenschaft der *TreeView*-Komponente mit der *ImageList* zu verbinden!



Quellcode

```
public partial class Form1 : Form
{
```

Alles passiert im Konstruktor des Formulars:

```
    public Form1()
    {
        InitializeComponent();
```

TreeView mit dem Wurzelknoten initialisieren (der entspricht *Form1*):

```
        this.treeView1.Nodes.Add(this.Name);
```

Array zum Speichern der Knoten (pro Control ein Knoten):

```
        TreeNode[] nodesForm = new TreeNode[this.Controls.Count];
```

Die folgende Anweisung ist dann überflüssig, wenn sich, wie in unserem Fall, das Bildchen für *Form1* an Position 0 in der *ImageList* befindet:

```
this.treeView1.Nodes[0].ImageIndex = 0;
```

Übrige Knoten erzeugen:

```
            this.setNodes(this.Controls, this.treeView1.Nodes[0]);
    }
```

Alle Controls des Formulars werden durchlaufen, um den *TreeView* aufzubauen. Um tatsächlich alle Steuerelemente zu erfassen, muss sich die *setNodes*- Methode immer wieder selbst aufrufen:

```
    private void setNodes(IList controls, TreeNode node)
    {
        foreach (Control ctrl in controls)
        {
            int index = node.Nodes.Add(new TreeNode(ctrl.Name));

            setImage(node, ctrl, index);  // das richtige Icon zuordnen

            if (ctrl.Controls.Count > 0)
            {
                setNodes(ctrl.Controls, node.Nodes[index]);  // rekursiver Aufruf!
            }
       }
    }
```

Die folgende Hilfsmethode lädt das zum Control passende Icon aus der *ImageList* und beschriftet den Knoten entsprechend:

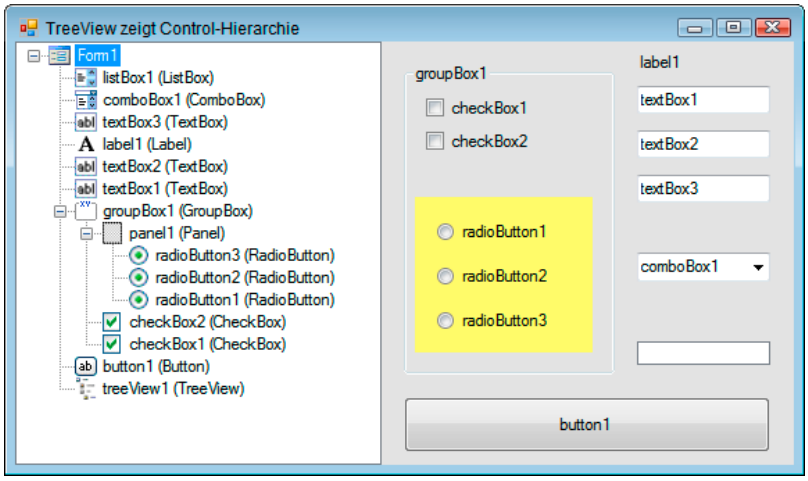
```
    private static void setImage(TreeNode node, Control ctrl, int index)
    {
        if (ctrl is GroupBox)
        {
            node.Nodes[index].ImageIndex = 1;
            node.Nodes[index].Text = ctrl.Name + " (GroupBox)";
        }
        else if (ctrl is Panel)
        {
            node.Nodes[index].ImageIndex = 2;
            node.Nodes[index].Text = ctrl.Name + " (Panel)";
        }
        else if (ctrl is Button)
        {
            node.Nodes[index].ImageIndex = 3;
            node.Nodes[index].Text = ctrl.Name + " (Button)";
        }
        else if (ctrl is TextBox)
        { ... usw.
```

Den Code für alle anderen Controls sparen wir uns hier (siehe Buchbeispiele). Es dürfte klar sein, dass der *ImageIndex* der Position entsprechen muss, auf welcher Sie das entsprechende Bildchen im „Image-Auflistungs-Editor“ abgelegt haben.

```
    }
}
```

Test

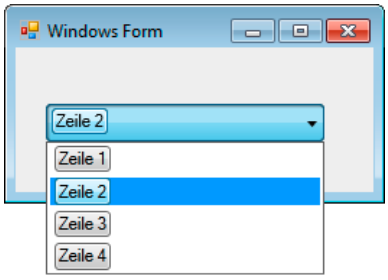
Unmittelbar nach Programmstart ist zunächst nur der Wurzelknoten (*Form1*) sichtbar. Nach Expandieren dieses Knotens erscheinen die Controls der ersten Ebene. Die zweite und dritte Ebene entstehen nach Expandieren der Knoten für *GroupBox* bzw. *Panel*.



8.10.4 WPF-Komponenten mit dem ElementHost anzeigen

Wozu WPF in einem Windows Forms-Kapitel? Die Antwort ist schnell gegeben, wenn die Migration nach WPF ansteht oder wenn Sie Effekte realisieren wollen, die sich mit den konventionellen Windows Forms-Controls nicht umsetzen lassen.

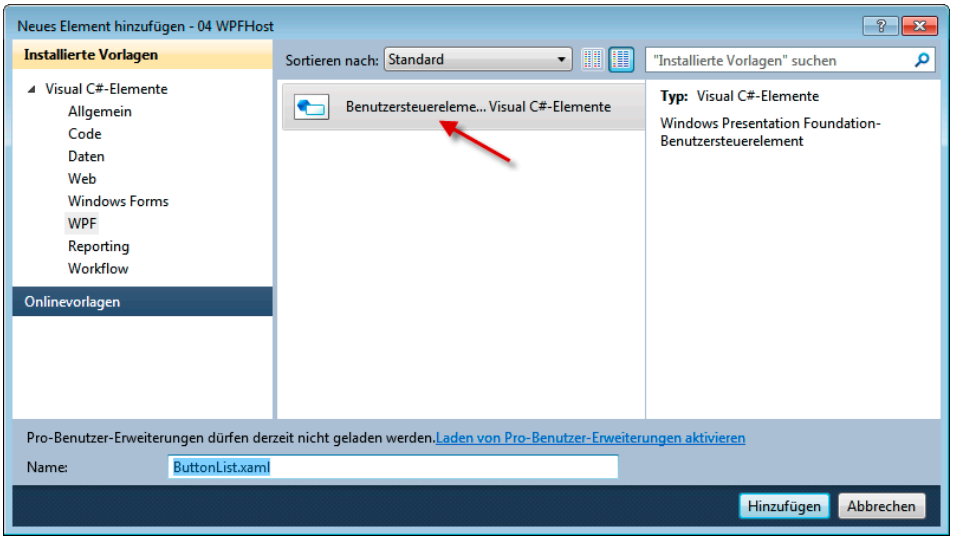
In unserem kleinen Praxisbeispiel soll eine WPF-*ComboBox* in ein Windows Forms-Formular eingefügt werden. Sicher nichts Spektakuläres, aber die WPF-Controls erlauben das beliebige Verschachteln von Komponenten und so wollen wir statt einfacher *ComboBox*-Einträge vier einzelne Schaltflächen anzeigen, die auch auf einen Mausklick reagieren (siehe folgende Laufzeitansicht)[1].



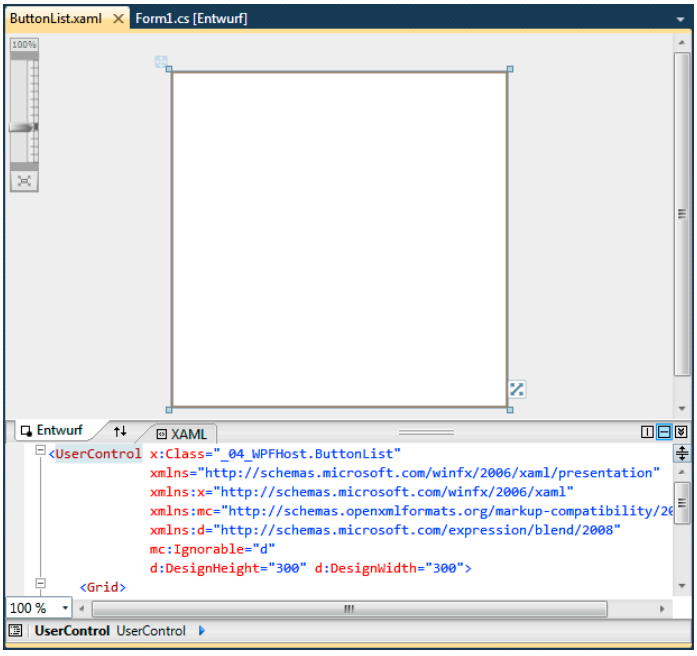
Oberfläche (WPF-Control)

In einem ersten Schritt erstellen Sie zunächst eine ganz gewöhnliche Windows Forms-Anwendung. Entgegen der sonstigen Vorgehensweise wenden wir uns noch nicht *Form1* zu, sondern fügen dem Projekt ein neues WPF-Benutzersteuerelement hinzu:

[1] Einen besonderen praktischen Nutzen hat das Beispiel sicher nicht, aber hier geht es ja um die WPF-Integration und nicht um ein unnötig kompliziertes Beispiel.



Nachfolgend öffnet sich dieses in der Entwurfsansicht:



Im oberen Teil des Designers sehen Sie den grafischen Editor, im unteren Teil den XAML-Editor, dem wir uns nachfolgend zuwenden wollen.

HINWEIS: Im Unterschied zu den Windows Forms werden die Oberflächen in WPF sprachneutral mittels XAML-Code beschrieben.

Erweitern Sie den bereits im Editor vorhandenen Code um die im Folgenden fett gedruckten XAML-Anweisungen:

```
<UserControl x:Class="_04_WPFHost.ButtonList"
...
     <Grid>
            <ComboBox>
                <ComboBoxItem Margin="2">
                    <Button>Zeile 1</Button>
                </ComboBoxItem>
                <ComboBoxItem Margin="2">
                    <Button Click="Button_Click">Zeile 2</Button>
                </ComboBoxItem>
                <ComboBoxItem Margin="2">
                    <Button>Zeile 3</Button>
                </ComboBoxItem>
            <ComboBoxItem Margin="2">
                <Button>Zeile 4</Button>
            </ComboBoxItem>
        </ComboBox>
    </Grid>
</UserControl>
```

An dieser Stelle nur soviel zum Code: Aufbauend auf einem Layout-Element (<*Grid*>) fügen wir eine *ComboBox* (<*ComboBox*>) mit Items (<*ComboBoxItem*>) hinzu, die wiederum Schaltflächen (<*Button*>) enthalten[2].

Beachten Sie, dass wir für den zweiten Button ein Ereignis definiert haben, das wir in der Codeansicht des WPF-Controls implementieren werden:

```
...
namespace _04_WPFHost
{
    public partial class ButtonList : UserControl
    {
...
```

Hier die hinzuzufügende Routine:

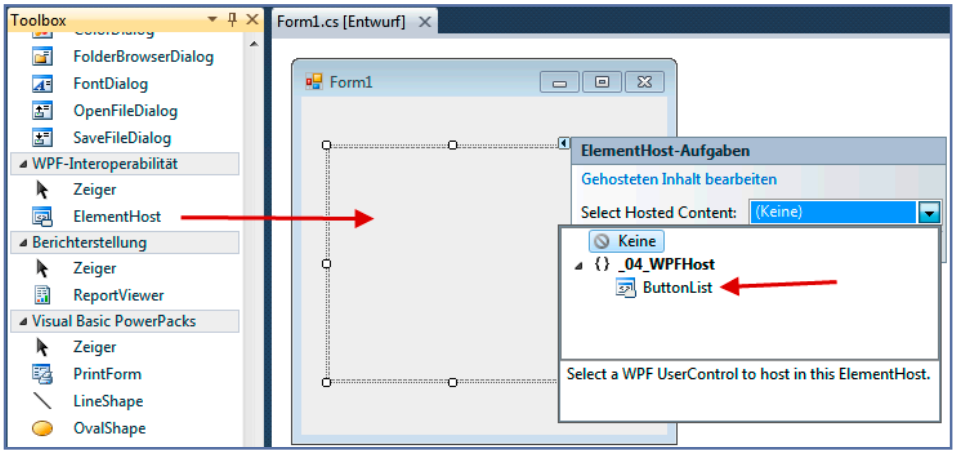
```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Button 2 wurde gedrückt!!!");
        }
    }
}
```

Schließen Sie nachfolgend das WPF-Benutzersteuerelement und kompilieren Sie die Anwendung. Dies stellt sicher, dass Sie im nächsten Schritt auch Zugriff auf das Control haben.

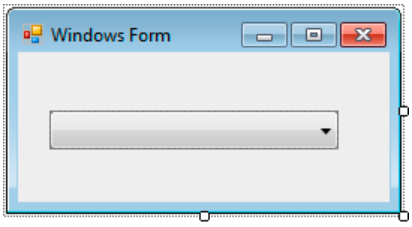
[2] Sie können diese Verschachtelung auch noch weiterführen und zum Beispiel ein Video oder ein Grid in die einzelnen Schaltflächen einfügen ...

Oberfläche (Windows Forms)

Öffnen Sie nun *Form1* und fügen Sie dem Formular ein *ElementHost*-Control hinzu. Über das Aufgabenmenü des Controls können Sie jetzt bequem das gewünschte WPF-Benutzersteuerelement, das wir gerade entworfen haben, hinzufügen:



Passen Sie nachfolgend noch die Größe des *ElementHost*-Controls an Ihre Wünsche an. Die Größe der jetzt enthaltenen WPF-ComboBox wird dadurch direkt beeinflusst.



Damit ist der Entwurf abgeschlossen und Sie können das Programm wie gewohnt starten. Wie bereits eingangs gezeigt, können Sie zur Laufzeit die *ComboBox* aufklappen und auf die enthaltenen Schaltflächen klicken. Der Klick auf die zweite Schaltfläche löst das definierte Ereignis im WPF-Benutzersteuerelement aus.

9 Grundlagen Grafikausgabe

Im vorliegenden Kapitel wollen wir uns grundlegend mit dem Erzeugen, Anzeigen und Verarbeiten von Grafiken im Rahmen von Windows Forms-Anwendungen beschäftigen. C# bietet dem Programmierer drei grundsätzliche Varianten an:

- Anzeige von „vorgefertigten" Grafikdateien (z. B. Bitmaps) in der *PictureBox*-Komponente
- Einsatz von Grafikmethoden (z. B. *DrawLine*, *DrawEllipse* ...) mit dem *Graphics*-Objekt
- Arbeiten mit GDI-Funktionen

Während bei der ersten Variante die Grafik bereits zur Entwurfszeit (Design Time) entsteht, wird sie bei den beiden Letzteren erst zur Laufzeit (Run Time) erzeugt. Erst hier kann man von eigentlicher „Grafikprogrammierung" sprechen, da diese ausschließlich per Quellcode (also ohne Zuhilfenahme der visuellen Entwicklungsumgebung von C#) funktioniert.

HINWEIS: Auch wenn in diesem Kapitel bereits die Grundlagen für die Druckausgabe gelegt werden, die eigentliche Beschreibung der Vorgehensweise und der nötigen Komponenten finden Sie erst in Kapitel 10.

Wem die Ausführungen dieses Kapitels nicht weit genug gehen oder wer bereits die Grundlagen der Grafikprogrammierung beherrscht, den verweisen wir gleich an das Kapitel 12 weiter. Dort gehen wir gezielt auf einige Spezialthemen rund um die Grafikprogrammierung ein.

9.1 Übersicht und erste Schritte

Nachdem mit dem GDI (*Graphics Design Interface*) die in Win32-Anwendungen übliche Grafikausgabe etwas in die Jahre gekommen war, wurde mit der Einführung des .NET-Frameworks auch eine neue Grafikschnittstelle unter dem Namen GDI+ implementiert.

9.1.1 GDI+ – ein erster Einblick für Umsteiger

GDI+ erfordert schon auf Grund seiner objektorientierten Schnittstelle ein ganz anderes Vorgehen als bei den alten GDI-Anwendungen. Umsteiger von „alten" Programmiersprachen haben es also erfahrungsgemäß etwas schwerer, sich mit den Konzepten von GDI+ anzufreunden. Aus diesem Grund zunächst einige Anmerkungen zu den Grundkonzepten von GDI+ für den Umsteiger.

Ein zentrales Grafikausgabe-Objekt

Im Gegensatz zur Vorgehensweise in GDI, bei der Sie die Grafikmethoden von einzelnen Komponenten genutzt haben, arbeiten Sie jetzt mit einem einzigen *Graphics*-Objekt, das Sie bestimmten Objekten zuordnen können bzw. von diesen ableiten. Nur dieses Objekt verfügt über relevante Grafikmethoden und -eigenschaften.

Die Grafikausgabe ist zustandslos

Eine der wichtigsten und zugleich einschneidendsten Änderungen betrifft die Organisation der Grafikausgabe. Haben Sie bisher zunächst die Parameter für Linien (Breite, Farbe, Style etc.), Pinsel oder Schriftarten gesetzt und nachfolgend mit diesen Grafikausgaben getätigt, ist dafür nun jede einzelne Grafikmethode zuständig. Das heißt, für eine Grafikmethode ist es vollkommen unerheblich, welche Linienart vorher genutzt wurde. Alle erforderlichen Parameter werden an die Methode bei **jedem** Aufruf übergeben. Die gleiche Vorgehensweise trifft auch auf Texte oder Pinsel zu.

Prinzipieller Ablauf

Wurde bisher nach dem Schema

- Grafikobjekte (Pen, Brush etc.) erzeugen,
- beliebige Zeichenfunktionen (z. B. LineTo) aufrufen,
- Grafikobjekte löschen

verfahren, müssen bei einer zustandslosen GDI+-Programmierung genau diese Operationen **für jeden** einzelnen Methodenaufruf ausgeführt werden, da GDI+ die GDI-Objekte im alten Zustand zurücklassen muss.

Beispiel 9.1: Zeichnen einer Linie im Formular

C#

```
    Graphics g;
```

Neuen Stift erzeugen:

```
      Pen mypen = new Pen(Color.Aqua);
```

Graphics-Objekt für das Formular erzeugen:

```
    g = this.CreateGraphics();
```

Linie mit dem neuen Stift zeichnen:

```
g.DrawLine(mypen, 10, 10, 100, 100);
```

Wichtige Features

Die wichtigsten Features auf einen Blick:

- GDI+ bietet eine verbesserte Farbverwaltung sowie mehr vordefinierte Farben.
- GDI+ unterstützt eine breite Palette an Bildformaten (.bmp, .gif, .jpeg, .exif, .png, .tiff, .ico, .wmf, .emf).
- Antialiasing-Funktionalität
- Farbverläufe für Pinsel
- Splines
- Bildtransformationen (Rotation, Translation, Skalierung)
- Gleitkommaunterstützung
- Unterstützung des Alpha-Kanals und damit auch Alpha-Blending
- ...

9.1.2 Namespaces für die Grafikausgabe

Bevor Sie im weiteren Verlauf mit der Fehlermeldung „Der Typ XYZ ist nicht definiert“ konfrontiert werden, möchten wir Ihnen die im Zusammenhang mit der Grafikausgabe relevanten Namespaces vorstellen:

- *System.Drawing*
- *System.Drawing.Drawing2D*
- *System.Drawing.Imaging*
- *System.Drawing.Printing*
- *System.Drawing.Design*
- *System.Drawing.Text*

Die ausführliche Behandlung der einzelnen Strukturen und Objekte erfolgt in den weiteren Abschnitten, hier nur eine kurze Übersicht.

System.Drawing

Dieser standardmäßig eingebundene Namespace enthält die meisten Klassen, Typen, Auflistungen etc., die Sie für die Basisfunktionen der Grafikausgabe benötigen.

Typ	Beschreibung
Color	... verwaltet ARGB-Farbwerte (Alpha-Rot-Grün-Blau), Konvertierungsfunktionen sowie diverse vordefinierte Farbkonstanten
Point *PointF*	... verwaltet 2-D-Koordinaten (x, y) als Integer- bzw. als Floatwert
Rectangle *RectangleF*	... verwaltet die Koordinaten eines Rechtecks als Integer- bzw. Floatwert
Size	... verwaltet die Größe eines rechteckigen Bereichs (Breite, Höhe)

Objekt	Beschreibung
Graphics	**Das zentrale Grafikausgabe-Objekt**
Pen	Objekt für die Definition des Linientyps
Brush, *Brushes,* *SolidBrush,* *TextureBrush*	Objekte für die Definition des Füllstils (Pinsel) von Objekten (Kreise, Rechtecke etc.)
Font, *FontFamily*	Objekt für die Definition von Schriftarten (Farbe, Größe etc.)
Bitmap, *Image*	Objekt für die Verwaltung von Bitmaps bzw. Grafiken

System.Drawing.Drawing2D

Dieser Namespace bietet Funktionen für Farbverläufe sowie Unterstützung für 2D- und Vektorgrafiken (Matrix für geometrische Transformationen).

System.Drawing.Imaging

Mithilfe dieses Namespace können Sie erweiterte Funktionen wie

- Direktzugriff auf Bitmap-Daten (Pointer),
- Unterstützung für Metafiles,
- Farbverwaltung,
- Grafikkonvertierung,
- Abfrage von Grafikeigenschaften

realisieren.

System.Drawing.Printing

Basisklassen für die Druckausgabe. Mehr dazu in Kapitel 10 (Druckausgabe).

System.Drawing.Design

Dieser Namespace enthält einige Klassen, die für die Entwurfszeit-Oberfläche genutzt werden.

System.Drawing.Text

Abfrage von Informationen über die installierten Schriftarten.

9.2 Darstellen von Grafiken

Bevor wir Sie in den folgenden Abschnitten mit Dutzenden von Objekten, Methoden und Eigenschaften überfrachten, wollen wir uns die wohl trivialste Form der Grafikdarstellung etwas näher ansehen. Die Rede ist von der *PictureBox*-Komponente, die in diesem Zusammenhang alle wesentlichen Aufgaben übernehmen kann. Untrennbar mit dieser Komponente ist auch das *Image*-Objekt verbunden, über das die eigentlichen Grafikdaten verwaltet werden (Laden, Manipulieren, Eigenschaften).

9.2.1 Die PictureBox-Komponente

Platzieren Sie zunächst eine *PictureBox* im Formular, können Sie diese wie jedes andere Objekt positionieren und die Größe bestimmen. Über die Eigenschaft *Image* lässt sich bereits zur Entwurfszeit eine Grafikdatei zuweisen, die nachfolgend fest in das Projekt und damit auch die EXE-Datei übernommen wird.

Wie diese Grafik formatiert, das heißt skaliert wird, bestimmen Sie mit der *SizeMode*-Eigenschaft:

Normal

CenterImage

StretchImage

Autosize

Wie Sie den obigen Abbildungen entnehmen können, skaliert die *Stretch*-Variante zwar die Grafik auf die gewünschte Größe, leider bleibt dabei aber das Seitenverhältnis der Grafik auf der Strecke.

Wie Sie die Proportion zur Laufzeit wiederherstellen können, zeigt das folgende Beispiel.

Beispiel 9.2: Bild proportional skalieren

C#

```
Single xy;
xy = (Single) pictureBox1.Image.Width / (Single) pictureBox1.Image.Height;
if (pictureBox1.Image.Width > pictureBox1.Image.Height)
  pictureBox1.Height = (int) (pictureBox1.Width / xy);
else
  pictureBox1.Width = (int) (pictureBox1.Height * xy);
```

Die Vorgehensweise: Sie weisen der Eigenschaft *SizeMode* den Wert *Stretch* zu und skalieren jeweils die Außenmaße der *PictureBox* so, dass die Grafik möglichst optimal angezeigt wird.

Über die *BorderStyle*-Eigenschaft können Sie zwischen *keinem*, *einfachem* und *dreidimensionalem* Rahmen wählen.

Damit sind auch schon alle wesentlichen Eigenschaften der *PictureBox* beschrieben. Ziemlich dürftig, werden Sie sicher sagen. Aber dieser Eindruck täuscht, wenn wir uns die bereits erwähnte Eigenschaft *Image* näher ansehen, bei der es sich um ein recht komplexes Objekt handelt.

9.2.2 Das Image-Objekt

Mit dem *Image*-Objekt bietet sich nicht nur die Möglichkeit, zur Entwurfszeit eine Grafik in die *PictureBox* einzubetten, sondern es stellt auch diverse Grafikbearbeitungsfunktionen und die dazu nötigen Informationen über die Grafik bereit.

Zwei Wege führen zum *Image*-Objekt:

- Haben Sie bereits zur Entwurfszeit eine Grafik geladen, können Sie direkt über *Picture.Image* auf die gewünschten Eigenschaften/Methoden zugreifen.
- Andernfalls müssen Sie zunächst ein *Image*-Objekt erzeugen und der *PictureBox* zuweisen.

Beispiel 9.3: Erzeugen eines neuen Image-Objekts (eine Bitmap 100 x 100 Pixel)[1]

C#

```
Image img;
img = new Bitmap(100, 100);
pictureBox1.Image = img;
```

Alternativ lässt sich ein *Image* auch aus einer Grafikdatei erzeugen:

```
Image img;
img = Image.FromFile("c:\\test.jpg");
pictureBox1.Image = img;
```

Womit wir auch schon beim Laden von Grafiken angekommen sind.

9.2.3 Laden von Grafiken zur Laufzeit

Wie bereits im vorhergehenden kurzen Beispiel gezeigt, stellt es kein Problem dar, ein *Image* aus einer bereits existierenden Datei zu laden. Je nach Dateityp handelt es sich beim *Image* nachfolgend um eine Pixel-, Vektorgrafik (WMF, EMF) oder um ein Icon (ICO).

HINWEIS: Doch Vorsicht: Nicht jedes Grafikformat unterstützt auch alle Grafikmethoden des *Image*-Objekts. Laden Sie beispielsweise eine WMF-Grafik in das Image, können Sie diese nicht mit der Funktion *RotateFlip* drehen oder spiegeln.

Beispiel 9.4: Grafikformat bestimmen

C#

Statt mit

```
Image img;
img = Image.FromFile("c:\\test.bmp");
pictureBox1.Image = img;
```

... können Sie auch direkt den gewünschten *Image*-Typ erzeugen:

```
Image img;
img = Bitmap.FromFile("c:\\test.bmp");
pictureBox1.Image = img;
```

Das Resultat ist in beiden Fällen das Gleiche.

[1] Dies ist nur eine Variante!

9.2.4 Sichern von Grafiken

Ähnlich einfach wie das Laden ist auch das Speichern von Grafiken. Mit der *Image*-Methode *Save* können Sie die Grafik in einem der unterstützten Dateiformate sichern.

Beispiel 9.5: Speichern im PNG-Format

C#

```
pictureBox1.Image.Save("c:\\test.png");
```

Auf diese Weise können Sie auch einen Dateikonverter programmieren, Sie brauchen nicht einmal eine *PictureBox* dafür.

Beispiel 9.6: Konvertieren vom BMP- ins PNG-Format

C#

```
using System.Drawing.Imaging;
...
      Image img;
      img = Bitmap.FromFile("c:\\test.bmp");
      img.Save("c:\\test.png", ImageFormat.Png);
```

Spezielle Einstellungen

Leider, und das scheint bei fast allen universellen Bibliotheken der Fall zu sein, ist die praktische Verwendung teilweise recht eingeschränkt bzw. umständlich. Möchten Sie beispielsweise im JPEG-Format speichern, ist dies kein Problem, doch was, wenn Sie auch den Kompressionsfaktor beeinflussen wollen?

In diesem Fall kommen Sie um etwas mehr Programmierung nicht herum.

Beispiel 9.7: Festlegen eines Kompressionsfaktors von 60 % für die zu speichernde JPEG-Datei

C#

```
 private void button1_Click(object sender, EventArgs e)
 {
   EncoderParameters myEncoderParameters = new EncoderParameters(1);
   myEncoderParameters.Param[0] = new EncoderParameter(Encoder.Quality, (long) 60);
   pictureBox1.Image.Save("c:\\test.jpg", GetEncoderInfo("image/jpeg"),
myEncoderParameters);
   pictureBox2.Image = Bitmap.FromFile("c:\\test.jpg");
 }
```

Die notwendige Hilfsfunktion für die Rückgabe des passenden *ImageCodecInfo*-Objekts:

```
 private ImageCodecInfo GetEncoderInfo(String mt)
 {
   ImageCodecInfo[] encoders = ImageCodecInfo.GetImageEncoders();
   for (int i=0; i < encoders.Length - 1; i++)
   if (encoders[i].MimeType == mt) return encoders[i];
   return null;
 }
```

9.2.5 Grafikeigenschaften ermitteln

Breite und Höhe der Grafik

Breite und Höhe der Grafik können Sie über die Eigenschaften *Width* und *Height* des *Image*-Objekts abrufen.

HINWEIS: Die Maße des *Image*-Objekts, das heißt der Grafik, stehen in keinem Zusammenhang mit den Maßen der *PictureBox*.

Auflösung

Die vertikale bzw. horizontale Auflösung der Grafik können Sie mit den Eigenschaften *VerticalResolution* bzw. *HorizontalResolution* abfragen. Beide geben einen Wert in *dpi* (Punkte pro Inch) zurück.

Grafiktyp

Den aktuell geladenen Grafiktyp fragen Sie über die Eigenschaft *Rawformat* ab. Rückgabewerte sind die verschiedenen Bildtypen, die von der *Image*-Komponente unterstützt werden (BMP, GIF etc.).

Interner Bitmap-Aufbau

Um welchen Typ von Bitmap (Farbtiefe) bzw. um welches Speicherformat (RGB, ARGB etc.) es sich handelt, verrät Ihnen die Eigenschaft *PixelFormat*. An dieser Stelle handelt es sich jedoch lediglich um eine schreibgeschützte Eigenschaft. Möchten Sie Einfluss auf das Speicherformat nehmen, sollten Sie sich näher mit dem Kapitel 12 beschäftigen.

9.2.6 Erzeugen von Vorschaugrafiken (Thumbnails)

Sicher sind Ihnen unter Windows auch schon die kleinen Vorschaugrafiken in den Explorerfenstern aufgefallen. Bevor Sie jetzt an aufwendige Algorithmen und Funktionen zur Skalierung von Bitmaps denken, vergessen Sie es besser wieder. GDI+ unterstützt Sie an dieser Stelle mit einer einfach verwendbaren Methode:

Syntax:

```
Image GetThumbnailImage(int thumbWidth,int thumbHeight,
                        Image.GetThumbnailImageAbort callback,IntPtr callbackData);
```

Die beiden ersten Parameter dürften leicht verständlich sein, *callback* erwartet einen Verweis auf einen Delegate (optional *null*). *CallbackData* übergeben Sie grundsätzlich *IntPtr.Zero*.

Leider müssen Sie sich um die Proportionen des zu erzeugenden Image selbst kümmern. Übergeben Sie beispielsweise 100 für Breite und Höhe und haben Sie eine Ausgangsgrafik mit anderem Höhen-/Seitenverhältnis, wird die Vorschaugrafik gestaucht, was sicher nicht

sehr professionell aussieht. Fragen Sie also vorher das Höhen-/Seitenverhältnis der Grafik ab und setzen Sie Breite und Höhe entsprechend.

Beispiel 9.8: Funktion zum Erzeugen einer proportionalen Vorschaugrafik

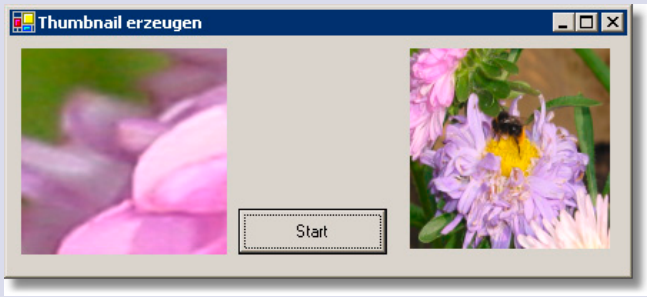
C#

```
private Image GetThumb(int size,Image img)
{
  Single xy = (Single) (img.Width / img.Height);
  if (xy > 1)
     return img.GetThumbnailImage(size, (int) (size / xy), null, IntPtr.Zero);
  else
     return img.GetThumbnailImage((int)(size * xy), size, null, IntPtr.Zero);
}
```

Der Aufruf ist einfach:

```
private void button1_Click(object sender, EventArgs e)
{
   pictureBox2.Image = GetThumb(80, pictureBox1.Image);
}
```

Ergebnis



HINWEIS: Enthält die Ausgangsgrafik bereits eine Thumbnail-Grafik (z. B. bei JPG-Format), wird diese für die Methode *GetThumbnailImage* genutzt, um nicht die ganze Bitmap in den Speicher zu laden. Dies ist allerdings maximal bis zu einer Auflösung von 120 x 120 Pixeln sinnvoll. Benötigen Sie größere Thumbnails, sollten Sie besser die Methode *DrawImage* verwenden, um eine verkleinerte Kopie der Ausgangsbitmap zu erzeugen.

9.2.7 Die Methode RotateFlip

Mit der Methode *RotateFlip* stehen Ihnen rudimentäre Manipulationsfunktionen für die Grafikausgabe zur Verfügung (drehen, spiegeln). Übergeben Sie der Methode einfach die gewünschte Konstante und aktualisieren Sie gegebenenfalls die übergeordnete *PictureBox*-Komponenten, schon ist die Grafik gedreht oder gespiegelt.

Beispiel 9.9: Drehen um 180°, vertikal kippen

C#

```
pictureBox1.Image.RotateFlip(RotateFlipType.Rotate180FlipY);
pictureBox1.Refresh();
```

Ein „Schweizer Taschenmesser“, was die Funktionalität anbelangt, könnte man auf den ersten Blick denken, doch wer sich die Liste der verschiedenen Möglichkeiten (Konstanten) einmal näher ansieht, wird feststellen, dass einige Varianten schlicht überflüssig sind. Ausgangspunkt für die Beispielgrafiken ist folgende „aufwendige“ Grafik:

Konstante	Beschreibung	Resultat
Rotate180FlipNone	Drehung um 180 Grad	
Rotate180FlipX	Drehung um 180 Grad, horizontal kippen	
Rotate180FlipXY	Drehung um 180 Grad, horizontal und vertikal kippen	
Rotate180FlipY	Drehung um 180 Grad, vertikal kippen	
Rotate270FlipNone	Drehung um -90 Grad	
Rotate270FlipX	Drehung um -90 Grad, horizontal kippen	
Rotate270FlipXY	Drehung um -90 Grad, horizontal und vertikal kippen	
Rotate270FlipY	Drehung um -90 Grad, vertikal kippen	
Rotate90FlipNone	Drehung um 90 Grad	
Rotate90FlipX	Drehung um 90 Grad, horizontal kippen	
Rotate90FlipXY	Drehung um 90 Grad, horizontal und vertikal kippen	
Rotate90FlipY	Drehung um 90 Grad, vertikal kippen	
RotateNoneFlipX	horizontal kippen	
RotateNoneFlipXY	horizontal und vertikal kippen	
RotateNoneFlipY	vertikal kippen	

Eine Rotation um 180° (*Rotate180FlipNone*) entspricht einem Kippen in vertikaler und horizontaler Richtung (*RotateNoneFlipXY*). Auch bei der Kombination *Rotate90FlipXY* bzw. *Rotate270FlipXY* scheint mit den Entwicklern der Spieltrieb durchgegangen zu sein. Das gleiche Ergebnis kann auch mit *Rotate270FlipNone* bzw. mit *Rotate90FlipNone* erreicht werden.

Unangefochtener „Spitzenreiter“ in der Tabelle ist *Rotate180FlipXY*.

9.2.8 Skalieren von Grafiken

Eine Grafik mit GDI+ zu skalieren, stellt eines der kleinsten Probleme dar[2]. Mit einer einzigen Zeile Code können Sie beispielsweise die Größe einer Grafik verdreifachen.

Beispiel 9.10: Bitmap auf dreifache Größe skalieren

C#

```
pictureBox1.Image = new Bitmap(pictureBox1.Image,
                    new Size(pictureBox1.Image.Width * 3, pictureBox1.Image.Height
* 3));
```

Wir nehmen diese Anweisung jetzt noch einmal auseinander bzw. deklarieren die Objekte für eine bessere Übersicht:

Eine temporäre Bitmap deklarieren:

```
Bitmap b;
```

Eine Variable für die neue Größe deklarieren:

```
Size s;
```

Die neue Größe bestimmen:

```
        s.Width = pictureBox1.Image.Width * 3;
        s.Height = pictureBox1.Image.Height * 3;
```

Die neue Bitmap erzeugen:

```
        b = new Bitmap(PictureBox1.Image, s);
```

Die Bitmap dem Image zuweisen:

```
        pictureBox1.Image = b;
```

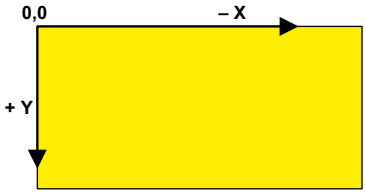
Damit sind die wichtigsten Möglichkeiten von *PictureBox* und *Image* aufgezählt, wir können uns der eigentlichen Grafikprogrammierung, das heißt dem Erzeugen von Grafiken, zuwenden.

[2] Sieht man einmal von potenziellen Speicherproblemen bei großen Bitmaps ab.

9.3 Das .NET-Koordinatensystem

Bevor Sie sich mit dem Zeichnen von Linien, Kreisen oder der Ausgabe von Bitmaps beschäftigen, sollten Sie einen Blick auf das .NET-Koordinatensystem werfen.

Auf den ersten Blick bietet sich nichts Ungewohntes, der Koordinatenursprung für alle Grafikmethoden liegt in der linken oberen Ecke des jeweils gewählten Ausgabefensters (das kann auch ein Steuerelement sein). Positive x-Werte werden nach rechts, positive y-Werte nach unten abgetragen:



Die Maßeinheit für Ausgaben ist standardmäßig das Pixel. Alternativ kann es jedoch bei einer Druckausgabe von Vorteil sein, die Maßeinheit zum Beispiel in Millimeter zu ändern. Nutzen Sie dazu die Eigenschaft *PageUnit* des *Graphics*-Objekts. Folgende Werte sind zulässig:

Wert	Beschreibung
Display	1/100 Zoll
Document	1/300 Zoll
Inch	Zoll (2,54 cm)
Millimeter	Millimeter
Pixel	(Standard) Bildschirmpixel
Point	1/72 Zoll

GDI+ unterscheidet im Zusammenhang mit der Grafikausgabe drei verschiedene Koordinatensysteme:

- globale Koordinaten,
- Seitenkoordinaten und
- Gerätekoordinaten.

9.3.1 Globale Koordinaten

Hierbei handelt es sich um das ursprüngliche Koordinatensystem. Dieses wird über bestimmte Transformationen in Seiten- und damit auch Gerätekoordinaten umgewandelt.

Ausgangspunkt bei der Maßeinheit Pixel: Ein Pixel in globalen Koordinaten entspricht zunächst einem Pixel auf dem endgültigen Ausgabegerät (Bildschirm, Drucker).

Beispiel 9.11: Globale Koordinaten

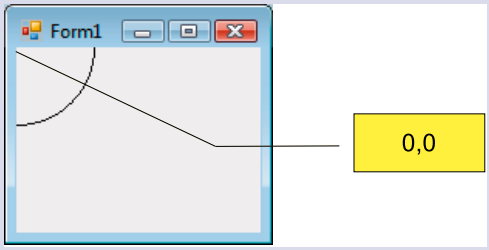
C#

Es soll ein Kreis mit dem Radius 50 (Pixel) auf dem Formular gezeichnet werden (an dieser Stelle müssen wir leider etwas vorgreifen, mit den folgenden Anweisungen wird im *Paint*-Ereignis des Formulars der Kreis gezeichnet):

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
   e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
}
```

Ergebnis

Das Ergebnis auf dem Bildschirm:



9.3.2 Seitenkoordinaten (globale Transformation)

Möchten Sie den Nullpunkt des Koordinatensystems verschieben, sodass der gesamte Kreis sichtbar wird, müssen Sie den Nullpunkt verschieben, das heißt, eine globale Transformation durchführen.

Translation (Verschiebung)

Mithilfe der Methode *TranslateTransform* lässt sich diese Aufgabe bewältigen.

Beispiel 9.12: Translation

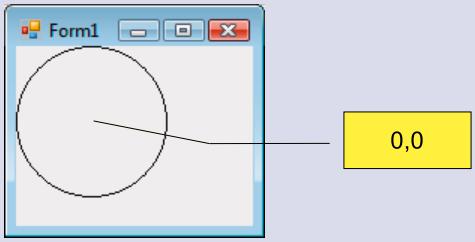
C#

Verschieben des Nullpunkts um jeweils 50 Pixel in x- und y-Richtung

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
   e.Graphics.TranslateTransform(50, 50);
   e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
}
```

Ergebnis

Das Ergebnis der Grafikausgabe:



Skalierung (Vergrößerung/Verkleinerung)

Ähnlich verhält es sich mit einer Skalierung zwischen globalen und Seitenkoordinaten: Möchten Sie beispielsweise den Kreis auf dem Bildschirm doppelt so groß ausgeben, können Sie dies mit der Eigenschaft *PageScale* steuern.

Beispiel 9.13: Skalierung

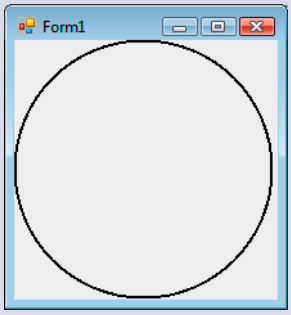
C#

Ein Pixel global soll zwei Pixeln Seiteneinheit entsprechen:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.TranslateTransform(50, 50);
    e.Graphics.PageScale = 2;
    e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
}
```

Ergebnis

Wie Sie sehen, sind alle Zeichenanweisungen, inklusive der Strichstärke, von dieser Skalierung betroffen:



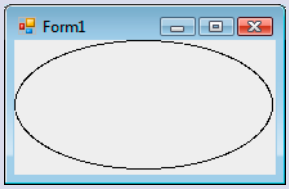
Alternativ können Sie mit *ScaleTransform* auch unterschiedliche Skalierungsfaktoren für x und y einführen.

Beispiel 9.14: Skalierung in x-Richtung (zweifach)

C#

```
e.Graphics.ScaleTransform(2, 1);
e.Graphics.TranslateTransform(50, 50);
e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
```

Ergebnis



HINWEIS: Sollen auch Images im korrekten Verhältnis wiedergegeben werden, müssen Sie *ScaleTransform* verwenden!

Rotation

Als letzte Variante bleibt das Drehen des Koordinatensystems. Verantwortlich dafür ist die Methode *RotateTransform*, der Sie einfach den Drehwinkel übergeben.

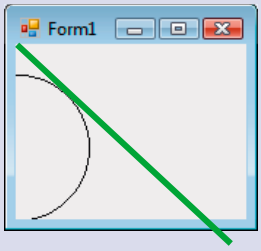
Beispiel 9.15: Drehen des Koordinatensystems um 45° (Uhrzeigersinn)

C#

```
e.Graphics.RotateTransform(45);
e.Graphics.TranslateTransform(50, 50);
e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
```

Ergebnis

Die folgende Abbildung zeigt das Resultat, die eingezeichnete Linie deutet die Position der x-Achse (y = 0) an:



Dass nicht nur einfache Objekte wie Linien oder Kreise von den Transformationen betroffen sind, zeigt das folgende Beispiel.

Beispiel 9.16: Bitmap-Transformation

C#

Drehen einer Bitmap um 25°, Skalieren um 100 %, Verschieben des Nullpunkts in das Formular (50, 50)

```
e.Graphics.TranslateTransform(50, 50);
e.Graphics.RotateTransform(25);
e.Graphics.ScaleTransform(2, 2);
e.Graphics.DrawImage(pictureBox1.Image, 0, 0);
```

Ergebnis

9.3.3 Gerätekoordinaten (Seitentransformation)

Das dritte Koordinatensystem bezieht sich auf das endgültige Ausgabegerät, z. B. ein Drucker oder ein Bildschirm. Diese Geräte verfügen über vorgegebene Auflösungen, die in Dots per Inch (dpi) angegeben werden. Beispielsweise verfügt der Bildschirm meist über eine Auflösung von 96 dpi, Laserdrucker werden mit Auflösungen von 300 bis weit über 1000 dpi angeboten.

Solange Sie nicht mit Geräteeinheiten arbeiten, werden Sie je nach Geräteauflösung zu unterschiedlichen Ergebnissen kommen. Aus diesem Grund unterstützt das .NET-Koordinatensystem auch gerätespezifische Einheiten wie Millimeter oder Inch. Über die Eigenschaft *PageUnit* können Sie die Einheit für Ihr Koordinatensystem ändern.

Beispiel 9.17: Bildschirmanzeige in Millimetern

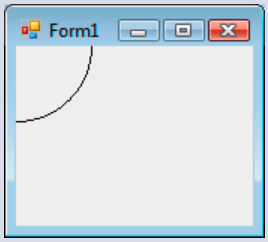
C#

```
e.Graphics.PageUnit = GraphicsUnit.Millimeter;
e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
```

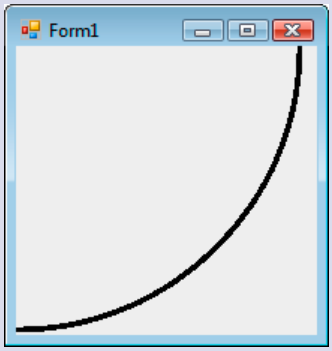
Ergebnis

Die beiden folgenden Abbildungen zeigen das Ergebnis für Pixel bzw. Millimeter:

Pixel



Millimeter



Frage: Wie groß ist bei der rechten Abbildung der Radius in Pixeln?

Die Antwort: Bei einer Bildschirmauflösung von 96 dpi ergibt sich ein Faktor von ca. 3,7 Pixeln pro Millimeter. Multipliziert mit dem Radius von 50 mm erhalten Sie 188 Pixel. Sie können das berechnete Ergebnis zum Beispiel mit Paintbrush „nachmessen“.

HINWEIS: Weitere Anwendungsbeispiele für Koordinatentransformationen finden Sie in den Kapiteln 10 und 12.

9.4 Grundlegende Zeichenfunktionen von GDI+

Im vorhergehenden Abschnitt hatten Sie ja bereits ersten Kontakt mit einigen Zeichenfunktionen, auf die wir nun in diesem Abschnitt näher eingehen wollen.

9.4.1 Das zentrale Graphics-Objekt

Wie bereits angedeutet, sind mit GDI+ die Zeiten des unbekümmerten Programmierens vorbei. Um den ausufernden Möglichkeiten etwas Einhalt zu gebieten und eine einheitliche Schnittstelle für die Grafikausgabe zu schaffen, wurde die *Graphics*-Klasse eingeführt.

Nur Objekte dieses Typs verfügen über Grafikausgabemethoden und die nötigen Parameter. Alle anderen Komponenten, und dazu zählen auch Formulare und Drucker, können lediglich *Graphics*-Objekte zur Verfügung stellen.

Wie erzeuge ich ein Graphics-Objekt?

Drei wesentliche Varianten bieten sich an:

- Für Formulare können Sie das *Paint* nutzen, der übergebene Parameter *e* stellt auch ein *Graphics*-Objekt zur Verfügung. -Ereignis
- Für alle anderen Komponenten nutzen Sie die *CreateGraphics*-Methode des jeweiligen Objekts. Diese gibt ein *Graphics*-Objekt zurück.
- Möchten Sie auch auf *Images* (BMP etc.) per *Graphics*-Objekt zugreifen, verwenden Sie einfach die *Graphics.FromImage*-Methode.

Beispiel 9.18: Zeichnen mit *Paint*-Ereignis

C#

```
private void Form1_Paint(object sender,PaintEventArgs e)
{
  e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100) ;
}
```

Beispiel 9.19: Verwenden von *CreateGraphics*

C#

```
private void button1_Click(object sender, EventArgs e)
{
  Graphics g;
  g = this.CreateGraphics();
  g.DrawEllipse(new Pen(Color.Black), 0, 0, 100, 100);
}
```

Um die Ausgabe auf einem Button statt auf dem Formular vorzunehmen, genügt es, wenn Sie das *Graphics*-Objekt mit *button1.CreateGraphics* erzeugen.

Beispiel 9.20: Verwenden von *Graphics.FromImage*

C#

```
private void button1_Click(object sender, EventArgs e)
{
   Graphics g;
   g = Graphics.FromImage(pictureBox1.Image);
   g.DrawEllipse(new Pen(Color.Black), 0, 0, 100, 100);
   pictureBox1.Refresh();
}
```

HINWEIS: Für alle Grafikausgaben gilt: Nach dem Verdecken eines Formulars müssen Sie sich um das Aktualisieren der Bildschirmanzeige selbst kümmern. Einzige Ausnahme ist die *PictureBox*, wenn Sie, wie im letzten Beispiel gezeigt, über *Graphics* in das *Image* schreiben.

Die Invalidate-Methode

Nutzen Sie das *Paint*-Ereignis für die Ausgabe der Grafik und muss diese zwischenzeitlich aktualisiert werden, können Sie die Methode *Invalidate* zum Aktualisieren der Anzeige verwenden.

Beispiel 9.21: Mittels *Timer* soll ein kontinuierlich größer werdender Kreis gezeichnet werden.

C#

Globale Variable für den Offset:

```
private int x;
```

Das *Tick*-Ereignis:

```
private void timer1_Tick(object sender, EventArgs e)
{
   x++; this.Invalidate();
}
```

Die eigentliche Zeichenroutine im *Paint*-Ereignis:

```
private void Form1_Paint(object sender,PaintEventArgs e)
{
   e.Graphics.DrawEllipse(new Pen(Color.Black), 0, 0, 100 + x, 100 + x);
}
```

Im Weiteren wollen wir uns mit den wichtigsten Grafikgrundoperationen beschäftigen, auch wenn Sie bereits einige in den vorhergehenden Beispielen kennengelernt haben.

Die Eigenschaft ResizeRedraw

So bequem die Verwendung des *Paint*-Ereignisses auch ist, einen Nachteil werden Sie schnell bemerken, wenn das Formular skaliert bzw. kurzzeitig verdeckt wurde. In diesem Fall werden nur die neuen bzw. die verdeckten Flächen neu gezeichnet. Dies kann allerdings dazu führen, dass Ihre Grafik nicht mehr wie gewünscht ausgegeben wird. Um diesem Missstand abzuhelfen, steht über das Formular die Eigenschaft *ResizeRedraw* zur Verfügung. Ist diese auf *true* gesetzt, wird immer der komplette Clientbereich neu gezeichnet.

Beispiel 9.22: Auswirkung von *ResizeRedraw*

C#

Zeichnen einer Ellipse im Clientbereich des Formulars:

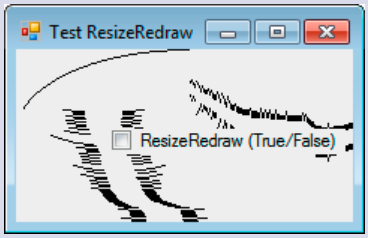
```
private void Form1_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   g.DrawEllipse(new Pen(Color.Black),0, 0,
this.ClientSize.Width,this.ClientSize.Height);
}
```

Per *CheckBox* die *ResizeRedraw-Eigenschaft* beeinflussen:

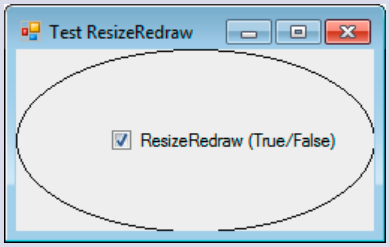
```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
    this.ResizeRedraw = checkBox1.Checked;
}
```

Ergebnis

ResizeRedraw=false:



ResizeRedraw=true:



9.4.2 Punkte zeichnen/abfragen

Tja, da sieht es gleich ganz düster aus. Eine entsprechende Funktion existiert zunächst nicht. Einzige Ausnahme: die *GetPixel-/SetPixel*-Methoden des *Bitmap*-Objekts.

Syntax:

```
Color GetPixel(int x, int y)
```

Syntax:

```
void SetPixel(int x, int y, Color color)
```

Beiden Methoden übergeben Sie die gewünschten Koordinaten, *GetPixel* gibt Ihnen einen *Color*-Wert zurück, *SetPixel* müssen Sie einen *Color*-Wert übergeben.

HINWEIS: Mehr zu Farben und dem *Color*-Objekt finden Sie in Abschnitt 9.5.3.

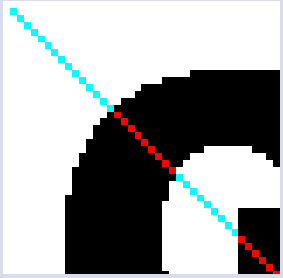
Beispiel 9.23: Setzen von Pixeln in Abhängigkeit von der vorhergehenden Farbe

C#

```
private void button1_Click(object sender, EventArgs e)
    {
      Bitmap b = (Bitmap) pictureBox1.Image;
      for (int x = 1; x <=50; x++)
      {
        if (b.GetPixel(x, x).ToArgb() == Color.Black.ToArgb())
          b.SetPixel(x, x, Color.Red);
        else
          b.SetPixel(x, x, Color.Aqua);
      }
      pictureBox1.Refresh();
    }
```

Ergebnis

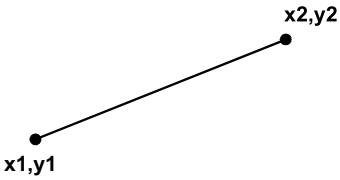
Die folgende Ausschnittsvergrößerung zeigt das Resultat:



9.4.3 Linien

Das Zeichnen von Linien gestaltet sich mit GDI+ relativ einfach, sieht man einmal davon ab, dass Sie bei **jedem** Aufruf von *DrawLine* auch einen geeigneten *Pen* übergeben müssen. An dieser Stelle wollen wir zunächst noch nicht auf die verschiedenen Stifttypen eingehen, mit denen Sie sowohl Farbe, Muster, Endpunkte etc. beeinflussen können. Mehr Infos zu diesem Thema finden Sie erst in Abschnitt 9.5.

Die weiteren Parameter der *DrawLine*-Methode: Entweder Sie übergeben jeweils ein Paar x,y-Koordinaten oder Sie verwenden gleich die vordefinierte Struktur *Point* (eine x- und eine y-Koordinate).



Beispiel 9.24: Verwendung von *DrawLine*

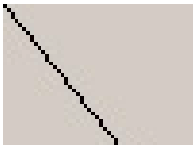
C#

```
Graphics g = this.CreateGraphics();
g.DrawLine(new Pen(Color.Black), 10, 10, 100, 100);
```

Mit *New Pen(Color.Black)* erzeugen wir einen schwarzen Stift mit der Linienbreite 1.

9.4.4 Kantenglättung mit Antialiasing

Im Zusammenhang mit der Ausgabe von Linien kommen wir auch schnell mit einem ersten „Problem“ in Berührung. Die Rede ist von der „Treppchen“- bzw. „Stufen“-Bildung von Linien, die einen von 0° bzw. 90° verschiedenen Winkel aufweisen:



Ursache ist die begrenzte Bildschirmauflösung, es können nur die Pixel gesetzt werden, die in das xy-Raster passen. Zwangsläufig kommt es bei einer Abweichung von Raster und gewünschtem Linienverlauf zu Sprüngen, die sich als hässliche Treppchen bemerkbar machen.

Mithilfe der Antialiasing-Technik können diese Effekte vermindert werden. Hintergrund dieser Technik ist ein „Glätten“ des Linienrands durch Auffüllen mit Pixeln, deren Farbe aus Linien- **und** Hintergrundfarbe berechnet wird.

GDI+ bietet Ihnen zu diesem Zweck die Methode *SetSmoothingMode*, mit der Sie die Form der Kantenglättung beeinflussen können. Zwei Extreme sind möglich:

- *SmoothingModeHighSpeed* (schnell, aber kantig)
- *SmoothingModeHighQuality* (langsam(er), aber sauber)

Beispiel 9.25: Ein- und Ausschalten der Kantenglättung

C#

```
Graphics g = this.CreateGraphics();
g.SetSmoothingMode(SmoothingModeHighQuality);
g.DrawLine(new Pen(Color.Black), 10, 10, 100, 100);
g.SetSmoothingMode(SmoothingModeHighSpeed);
```

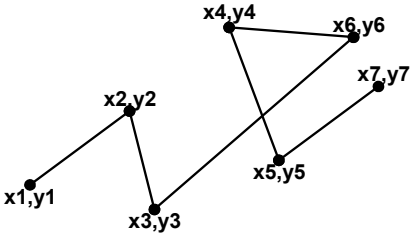
Ergebnis

Die folgende Abbildung zeigt das Ergebnis:

HINWEIS: Diese Form der Kantenglättung wirkt sich nicht auf Textausgaben, sondern nur auf alle Arten von Linien aus. Für Textausgaben verwenden Sie die Eigenschaft *TextRenderingHint*.

9.4.5 PolyLine

Möchten Sie mehr als eine Linie zeichnen und stimmen die Endpunkte mit den Anfangspunkten der folgenden Linien überein, verwenden Sie zum Zeichnen von Linien am besten die Methode *DrawLines*. Übergabewert ist wie gewohnt ein initialisierter *Pen* sowie ein Array von Punkten (*Point* oder *PointF*).



Beispiel 9.26: Zeichnen von zwei Linien

C#

```
      Graphics g = this.CreateGraphics();
      PointF[] punkte = {new PointF(0, 0), new PointF(100, 100), new PointF(20,
80)};
      g.DrawLines(new Pen(Color.Black), punkte);
```

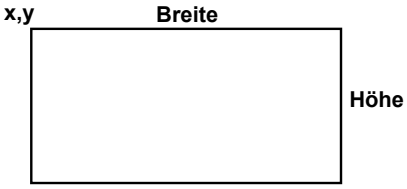
HINWEIS: Selbstverständlich können Sie auch ein frei definiertes Array verwenden, das zur Laufzeit mit Werten gefüllt wird.

9.4.6 Rechtecke

Für das Zeichnen von Rechtecken können Sie entweder die Methode *DrawRectangle* (nur Rahmen) oder *FillRectangle* (gefülltes Rechteck) verwenden.

DrawRectangle

Diese Methode erwartet als Übergabeparameter einen *Pen* sowie die Koordinaten (als x, y, Breite, Höhe oder *Rectangle*-Struktur).



Beispiel 9.27: Zeichnen von Rechtecken

C#

```
Graphics g = this.CreateGraphics();
g.DrawRectangle(new Pen(Color.Black), 10, 10, 100, 100);
```

Alternativ können Sie auch die folgenden Anweisungen verwenden:

```
Graphics g = this.CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 100, 100);
g.DrawRectangle(new Pen(Color.Black), rec);
```

FillRectangle

Im Unterschied zur *DrawRectangle*-Methode erwartet *FillRectangle* einen initialisierten *Brush*, das heißt die Information, wie das Rechteck zu füllen ist.

HINWEIS: Die Methode füllt lediglich das Rechteck, es wird kein Rahmen gezeichnet!

Beispiel 9.28: *Rotes Rechteck zeichnen*

C#

```
Graphics g = this.CreateGraphics();
g.FillRectangle(new SolidBrush(Color.Red), 10, 10, 100, 100);
```

Wie auch bei *DrawRectangle* können Sie alternativ eine *Rectangle*-Struktur übergeben. Dies bietet sich beispielsweise an, wenn man das Rechteck auch mit einem Rahmen versehen will.

Beispiel 9.29: Verwendung *Rectangle*

C#

```
Graphics g = this.CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 100, 100);

g.FillRectangle(new SolidBrush(Color.Red), rec);
g.DrawRectangle(new Pen(Color.Black), rec);
```

DrawRectangles/FillRectangles

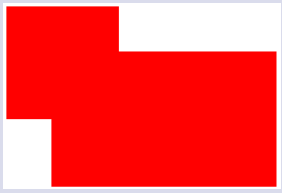
Nicht genug der Pein, neben den beiden genannten Methoden können Sie auch *DrawRectangle*s und *FillRectangle*s verwenden, um gleich mehrere Rechtecke auf einmal zu zeichnen. Übergeben wird in diesem Fall ein Array von *Rectangle*-Strukturen.

Beispiel 9.30: Zeichnen zweier gefüllter Rechtecke

C#

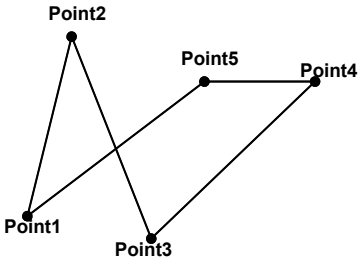
```
Graphics g = this.CreateGraphics();
Rectangle[] rec = {new Rectangle(10, 10, 100, 100),
                   new Rectangle(50, 50, 200, 120)};

g.FillRectangles(new SolidBrush(Color.Red), rec);
```

Ergebnis



9.4.7 Polygone

Möchten Sie ein n-Eck zeichnen, nutzen Sie die Methode *DrawPolygon*. Soll dieses Vieleck auch gefüllt werden, können Sie dies mit *FillPolygon* realisieren. Beide Funktionen arbeiten ähnlich wie die Funktionen zum Zeichnen von Rechtecken, der einzige Unterschied: *DrawPolygon* bzw. *FillPolygon* erwarten als Übergabeparameter ein Array von *Point*-Strukturen (x- und y-Koordinate) sowie jeweils einen *Pen* bzw. einen *Brush*.

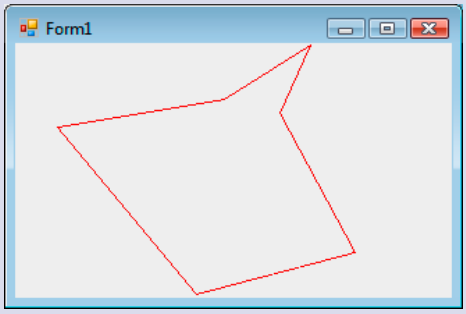


Beispiel 9.31: Zeichnen eines Vielecks

C#

```
Graphics g = this.CreateGraphics();
Point[] Ps = new Point[6];
Ps[0] = new Point(30, 60);   Ps[1] = new Point(150, 40);
Ps[2] = new Point(212, 1);   Ps[3] = new Point(190, 50);
Ps[4] = new Point(244, 150); Ps[5] = new Point(130, 180);
g.DrawPolygon(new Pen(Color.Red), Ps);
```

Ergebnis



HINWEIS: Die letzte Linie, zurück zum ersten Punkt, wird automatisch mit gezeichnet, um die Figur zu schließen.

9.4.8 Splines

Möchten Sie Diagramme zeichnen, stehen Sie häufig vor dem Problem, dass die entstehende Kurve ziemlich eckig ist, da Sie aus Zeitgründen nur wenige Stützpunkte berechnet haben. Das lineare Verbinden ist in diesem Fall nicht der ideale Weg. Besser ist hier die Verwendung von Splines, das heißt Linienzügen, die bei einem Wechsel des Anstiegs weiche Übergänge realisieren.

GDI+ bietet Ihnen die Methode *DrawCurve*, der Sie zunächst die gleichen Parameter wie *DrawLines* übergeben können (Stift, Point-Array).

Syntax:

```
void DrawCurve(Pen pen, Point[] points, int Start, int Anzahl, float Spannung);
```

Zusätzlich können Sie bestimmen, ab welchem Punkt in der Liste die Spline-Kurve gezeichnet wird bzw. wie viele Punkte gezeichnet werden. Der letzte Parameter bestimmt die Spannung, das heißt, wie der Übergang zwischen zwei Teilstrecken hergestellt wird. Ein Beispiel soll für mehr Klarheit sorgen.

Beispiel 9.32: Spline zeichnen

C#

Neues *Graphics*-Objekt erzeugen:

```
Graphics g = this.CreateGraphics();
```

Array für die Punkte erstellen:

```
PointF[] Ps = new PointF[6];
```

Die Spannung, verändern Sie diesen Wert wie gewünscht:

```
Single u = 0.6F;

Ps[0] = new PointF(0, 100);
Ps[1] = new PointF(100, 0);
Ps[2] = new PointF(200, 100);
Ps[3] = new PointF(300, 200);
Ps[4] = new PointF(400, 100);
Ps[5] = new PointF(500, 0);
```

Zunächst zeichnen wir zum Vergleich eine „normale“ Kurve:

```
g.DrawLines(new Pen(Color.Black, 2), Ps);
```

Und jetzt die Spline-Kurve:

```
g.DrawCurve(new Pen(Color.Red, 2), Ps, 0, 5, u);
```

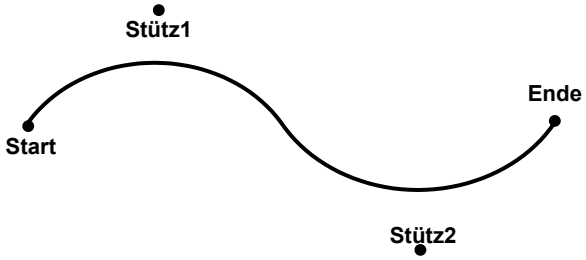
Ergebnis

9.4.9 Bézierkurven

Für einfache Zeichnungen dürften die bisherigen Zeichenfunktionen ausreichen, kompliziertere Gebilde bekommen Sie mit der Bézierfunktion in den Griff. Wie bei einer „normalen“ Linie übergeben Sie Anfangs- und Endpunkt, zusätzlich jedoch noch zwei Stützpunkte, mit denen die Linie wie ein Gummiband „gezogen“ werden kann.

Syntax:

```
void DrawBezier(Pen pen, Point Start, Point Stütz1, Point Stütz2, Point Ende);
```



Beispiel 9.33: Zeichnen einer Bézierkurve

C#

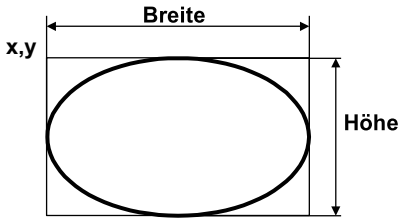
```
Graphics g = this.CreateGraphics();
Point start, Stütz1, stütz2, ende;

start = new Point(0, 100);
Stütz1 = new Point(100, 0);
stütz2 = new Point(200, 200);
ende = new Point(300, 100);
g.DrawBezier(new Pen(Color.Black, 2), start, Stütz1, stütz2, ende);
```

Ergebnis

HINWEIS: Mit *DrawBeziers* können Sie mehrere Bézierkurven gleichzeitig zeichnen.

9.4.10 Kreise und Ellipsen

Kreise und Ellipsen zeichnen Sie mit *DrawEllipse* bzw. *FillEllipse*, wenn Sie eine Kreis-/Ellipsenfüllung erzeugen wollen. Übergabeparameter ist eine *Rectangle*-Struktur bzw. die linke obere Ecke sowie die Breite und Höhe der Ellipse.



HINWEIS: Kreisbögen und Kuchenstücke zeichnen Sie mit eigenen Methoden.

Beispiel 9.34: Zeichnen einer Ellipse

C#

```
Graphics g = this.CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 150, 100);
g.DrawEllipse(new Pen(Color.Black, 2), rec);
```

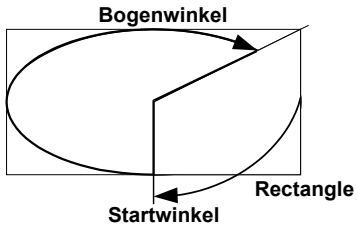
9.4.11 Tortenstück (Segment)

Die Methode *DrawPie* zeichnet ein Tortenstück, das durch eine Ellipse und zwei Linien begrenzt ist. Die Begrenzungslinien werden über den Startwinkel bzw. den Bogenwinkel bestimmt.

Syntax:

```
void DrawPie(Pen pen, RectangleF rect, float startwinkel, float bogenwinkel);
```

HINWEIS: Winkel werden im Uhrzeigersinn abgetragen.



HINWEIS: Mit *FillPie* erstellen Sie ein gefülltes Tortenstück.

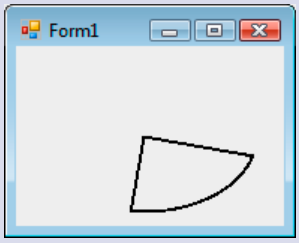
Beispiel 9.35: Tortenstück zeichnen

C#

```
Graphics g = this.CreateGraphics();

Rectangle rec = new Rectangle(10, 10, 150, 100);
g.DrawPie(new Pen(Color.Black, 2), rec, 10, 90);
```

Ergebnis

Zeichnet folgendes Tortenstück:



Kuchendiagramme

Da die Funktionen Winkel als Parameter erwarten, gestaltet sich das Zeichnen von Kuchendiagrammen besonders leicht. Ausgehend von einer einheitlichen *Rectangle*-Struktur brauchen Sie lediglich die Winkelangaben zu variieren.

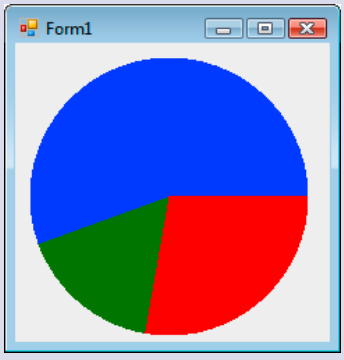
Beispiel 9.36: Zeichnen eines Kuchendiagramms (100°, 60°, 200°)

C#

```
Graphics g = this.CreateGraphics();

Rectangle rec = new Rectangle(10, 10, 200, 200);
g.FillPie(new SolidBrush(Color.Red), rec, 0, 100);
g.FillPie(new SolidBrush(Color.Green), rec, 100, 60);
g.FillPie(new SolidBrush(Color.Blue), rec, 160, 200);
```

Ergebnis

Das Ergebnis kann sich durchaus sehen lassen:

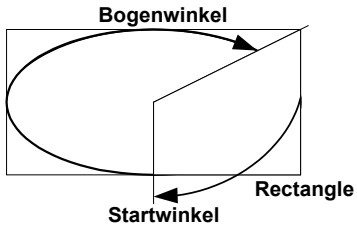


9.4.12 Bogenstück

Die Methode *DrawArc* zeichnet im Gegensatz zur Methode *DrawPie* nur den Bogen, nicht die Verbindungen zum Ellipsenmittelpunkt. Aus diesem Grund kann diese Figur auch nicht gefüllt werden.

Syntax:

```
DrawArc(Pen pen, RectangleF rect, float startwinkel, float bogenwinkel);
```



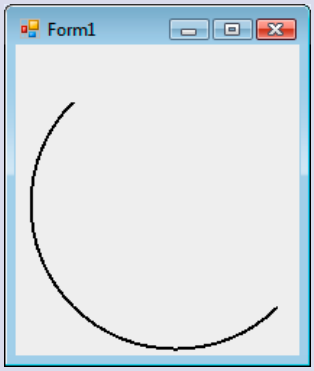
HINWEIS: Positive Winkelangaben werden im Uhrzeigersinn abgetragen.

Beispiel 9.37: Bogenstück zeichnen

C#

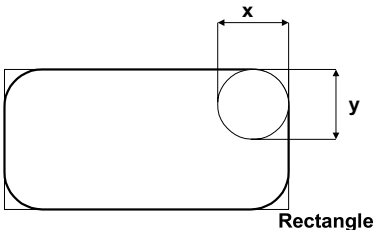
```
Graphics g = this.CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 200, 200);

g.DrawArc(new Pen(Color.Black, 2), rec, 45, 180);
```

Ergebnis



9.4.13 Wo sind die Rechtecke mit den runden Ecken?

Ersatzlos gestrichen! Es bleibt Ihnen also nichts anderes übrig, als sich eine eigene Funktion zu schreiben, oder Sie nutzen die Möglichkeiten von GDI.



Beispiel 9.38: Eine „selbst gestrickte" *RoundRect*-Methode

C#

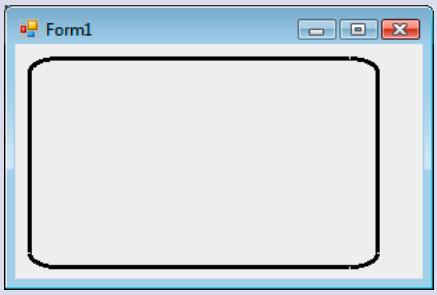
```
private void RoundRect(Graphics g, Pen p, Rectangle rect, int x, int y)
{
  g.DrawArc(p, new Rectangle(rect.Left, rect.Top, 2 * x, 2 * y), 180, 90);
  g.DrawArc(p, new Rectangle(rect.Left + rect.Width - 2 * x, rect.Top, 2 * x, 2 *
y),
            270, 90);
  g.DrawArc(p, new Rectangle(rect.Left + rect.Width - 2 * x,
            rect.Top + rect.Height - 2 * y, 2 * x, 2 * y), 0, 90);
  g.DrawArc(p, new Rectangle(rect.Left, rect.Top + rect.Height - 2 * y,
            2 * x, 2 * y), 90, 90);
  g.DrawLine(p, rect.Left + x, rect.Top, rect.Right - x, rect.Top);
  g.DrawLine(p, rect.Left + x, rect.Bottom, rect.Right - x, rect.Bottom);
  g.DrawLine(p, rect.Left, rect.Top + y, rect.Left, rect.Bottom - y);
  g.DrawLine(p, rect.Right, rect.Top + y, rect.Right, rect.Bottom - y);
}
```

Beim Aufruf der Methode übergeben Sie ein *Graphics*-Objekt, einen *Pen*, eine *Rectangle*-Struktur für das umgebende Rechteck sowie die Breite und Höhe der „Ecken":

```
Graphics g = this.CreateGraphics();
RoundRect(g, new Pen(Color.Black, 3), new Rectangle(10, 10, 250, 150), 20, 10);
```

Ergebnis

Wie Sie sehen können, ist das Ergebnis schon recht brauchbar:



9.4.14 Textausgabe

Neben den grafischen Primitiven wie Kreis oder Linie wird auch Text unter Windows als Grafik ausgegeben. GDI+ bietet dafür die recht universelle Methode *DrawString*. Leider gibt es reichlich Variationen dieser Methode, was es dem Anfänger sicher nicht einfacher macht. Wir beschränken uns auf zwei wichtige Vertreter:

Syntax:

```
void DrawString(string s, Font font, Brush brush, PointF xy,
                StringFormat format);
```

... Ausgabe eines Textes *String* mit der Schrift *Font* und der Füllung *Brush* an der Stelle *XY*. Wenn notwendig, können Sie weitere Formatierungsanweisungen in *StringFormat* übergeben.

Syntax:

```
void DrawString(string s, Font font, Brush brush,
                RectangleF layoutRectangle, StringFormat format);
```

... Ausgabe eines Textes *String* mit der Schrift *Font* und der Füllung *Brush* in einem Rechteck *Rectangle*.

HINWEIS: Mehr Details über das Erstellen von *Font*- bzw. *Brush*-Objekten finden Sie im Abschnitt 9.5.

Beispiel 9.39: Ausgabe eines Textes an der Position 70,70

C#

```
    Graphics g = this.CreateGraphics();
    g.DrawString("Textausgabe", new Font("Arial", 18), new SolidBrush(Color.Black),
70, 70);
```

Texteigenschaften

Haben Sie einen String, wie zum Beispiel

```
s = "Rotationswinkel 27°"
```

... und möchten Sie diesen **zentriert** ausgeben, brauchen Sie Informationen darüber, wie hoch und wie breit der auszugebende Text ist. Die Methode *MeasureString* ist in diesem Zusammenhang interessant.

Für einen vorgegebenen String mit dem gewählten Font werden Breite und Höhe in einer *SizeF*-Struktur zurückgegeben. Während *SizeF.Height* der Schrifthöhe in der jeweils gewählten Skalierung entspricht, ist *SizeF.Width* von der Anzahl der Buchstaben und der Schriftart abhängig:

Die Ausgabeposition berechnet sich wie folgt:

```
Graphics g = this.CreateGraphics();
Font myfont = new Font("Arial", 36);
SizeF mySize = g.MeasureString("Textausgabe", myfont);
g.DrawString("Textausgabe", myfont, new SolidBrush(Color.Black),
            (this.Width - mySize.Width) / 2, 150);
```

Ausgabe von mehrzeiligem Text

Für die Ausgabe von mehrzeiligem Text nutzen Sie eine überladene Variante von *DrawString*, die zusätzlich eine *Rectangle*-Struktur als Parameter akzeptiert.

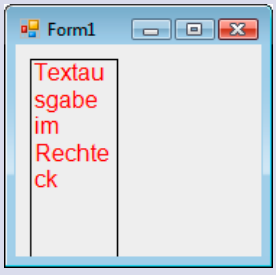
HINWEIS: Möchten Sie die Anzahl der entstehenden Zeilen und Spalten ermitteln, können Sie ebenfalls die Methode *MeasureString* nutzen.

Beispiel 9.40: Ausgabe von mehrzeiligem Text

C#

```
Graphics g = this.CreateGraphics();
Font myfont = new Font("Arial", 12);
g.DrawString("Textausgabe im Rechteck", myfont, Brushes.Red,
             new RectangleF(10, 10, 60, 180));
g.DrawRectangle(new Pen(Color.Black), new Rectangle(10, 10, 60, 180));
```

Ergebnis

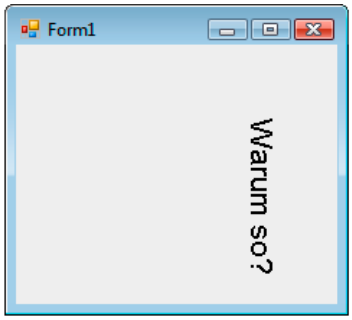


Textattribute

Über den Parameter *StringFormat* können Sie zusätzliche Formatierungsanweisungen an die *DrawString*-Methode übergeben. Auf die einzelnen Parameter wollen wir an dieser Stelle nicht eingehen, da Sie diese im Normalfall auch kaum gebrauchen werden. Lediglich der

Parameter *DirectionVertical* dürfte Ihr Interesse wecken, zu vermuten ist, dass Sie damit einen senkrechten Text ausgeben können.

Sicher, die Autoren haben immer etwas zu „meckern“, aber warum *StringFormatFlags.DirectionVertical* einen Text erzeugt, der die folgende Ausrichtung hat, scheint doch etwas merkwürdig[3]:



Ausgabequalität

Wem die Darstellungsqualität des Textes partout nicht passt bzw. wer sich nach einer Antialiasing-Funktion umsieht, wird bei der Eigenschaft *TextRenderingHint* fündig.

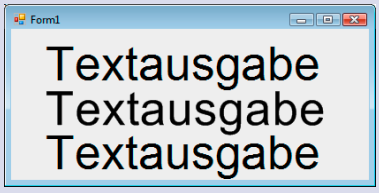
Beispiel 9.41: Antialiasing einschalten

C#

```
   Graphics g = this.CreateGraphics();
   g.RotateTransform(15);
   g.DrawString("Textausgabe", new Font("Arial", 40), new SolidBrush(Color.Black),
30, 10);
   g.TextRenderingHint = TextRenderingHint.AntiAlias;
   g.DrawString("Textausgabe", new Font("Arial", 40), new SolidBrush(Color.Black),
30, 60);
   g.TextRenderingHint = TextRenderingHint.ClearTypeGridFit;
   g.DrawString("Textausgabe", new Font("Arial", 40), new SolidBrush(Color.Black),
30, 110);
```

Ergebnis

Das Ergebnis zeigt die folgende Abbildung, die beste Darstellung dürfte der Wert *AntiAlias* liefern.



[3] Die Autoren haben bisher kein deutsches Buch oder eine technische Zeichnung gefunden, wo eine derartige Textausrichtung vorkommt. Unsere amerikanischen Freunde hätten es bei der schlichten Angabe eines Drehwinkels belassen sollen.

HINWEIS: Bevor Sie jetzt alle Texte auf diese Weise ausgeben, vergessen Sie bitte nicht, dass dafür auch jede Menge Rechenzeit benötigt wird.

Und wo bleibt eine Methode zum Drehen von Text?

Sie werden auch bei intensivster Suche weder beim *Font*-Objekt noch bei der Methode *DrawString* einen Weg finden, gedrehten Text auszugeben. Wer das Kapitel bis hier aufmerksam gelesen hat, wird sich an die *Graphics*-Methode *RotateTransform* erinnern[4]. Für alle anderen gilt: „Gehe zurück zum Abschnitt 9.3.2“!

HINWEIS: Möchten Sie nacheinander Text mit verschiedenen Winkeln ausgeben, entscheiden Sie mit dem optionalen Parameter *MatrixOrder*, ob die Winkel inkrementell oder absolut angegeben werden.

Beispiel 9.42: Rotation von Text um 15°

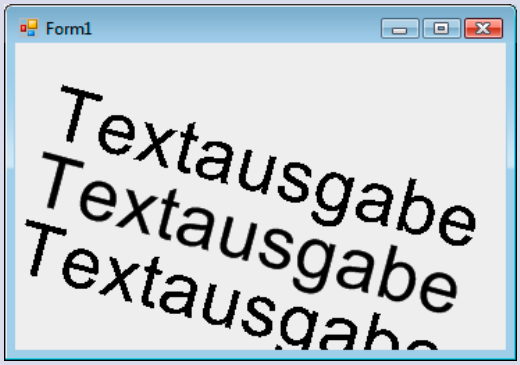
C#

```
    Graphics g = this.CreateGraphics();
    g.RotateTransform(15);
    g.DrawString("Textausgabe", new Font("Arial", 18), new SolidBrush(Color.Black),
 70, 70);
```

Ergebnis

Kommen wir noch einmal auf unser Beispiel aus dem vorhergehenden Abschnitt zurück.

Testen Sie das Beispiel einmal mit gedrehtem Text, werden Sie folgendes Ergebnis erhalten:



[4] Diese Form der Umsetzung scheint allerdings doch etwas „sinnfrei“ zu sein, berechnen Sie doch bitte einmal schnell die neuen x,y-Koordinaten, wenn Sie einen Text im Winkel von 90° an einem Rechteck ausrichten wollen.

Beispiel 9.43: Absolute Angabe

C#

```
g.RotateTransform(10, System.Drawing.Drawing2D.MatrixOrder.Prepend);
```

Beispiel 9.44: Inkrementelle Angabe

C#

```
g.RotateTransform(10, System.Drawing.Drawing2D.MatrixOrder.Append);
```

9.4.15 Ausgabe von Grafiken

Unter „Ausgabe von Grafiken“ möchten wir an dieser Stelle die Wiedergabe von fertigen Grafiken (Bitmaps, Icons, Metafiles) auf einem *Graphics*-Objekt verstehen.

Syntax:

```
void DrawImageUnscaled(Image image, int x, int y);
```

... zeichnet die angegebene Grafik an der durch x und y angegebenen Position.

HINWEIS: Die anderen Varianten von *DrawImageUnscaled* können Sie getrost vergessen, trotz angekündigtem Beschneidens der Grafik erfolgt dies nicht.

Beispiel 9.45: Verwendung von *DrawImageUnscaled*

C#

```
Graphics g = this.CreateGraphics();
g.DrawImageUnscaled(pictureBox1.Image, 0, 0);
```

Skalieren

Für die skalierte Ausgabe von Grafiken können Sie *DrawImage* verwenden:

Syntax:

```
void DrawImage(Image image, int x, int y, int width, int height);
```

Syntax:

```
void DrawImage(Image image, int x, int y, Rectangle srcRect, GraphicsUnit srcUnit);
```

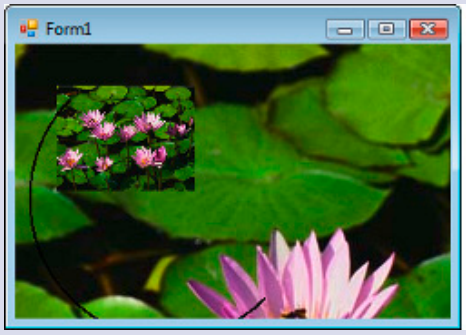
Während die erste Variante neben der Position auch die Breite und Höhe erwartet, können Sie bei der zweiten Syntaxvariante zusätzlich die Einheiten (*GraphicsUnit*) angeben (z. B. beim Drucker).

Beispiel 9.46: Verkleinerte, aber proportionale Ausgabe eines Image

C#

```
Single faktor;
Graphics g = this.CreateGraphics();
faktor = PictureBox1.Image.Height / PictureBox1.Image.Width;
g.DrawImageUnscaled(PictureBox1.Image, 0, 0);
g.DrawImage(PictureBox1.Image, 30, 30, 100, 100 * faktor);
```

Ergebnis



9.5 Unser Werkzeugkasten

Nachdem Sie sich im vorhergehenden Abschnitt mit diversen Grafikmethoden herumgeschlagen haben, wollen wir Ihnen jetzt die einzelnen „Werkzeuge“ des Grafikprogrammierers sowie deren Konfigurationsmöglichkeiten näher vorstellen.

9.5.1 Einfache Objekte

Zunächst sollten wir kurz auf einige grundlegende Strukturen im Zusammenhang mit der Grafikausgabe eingehen.

Point, FPoint

Für die Angabe von Koordinaten wird bei vielen Grafikmethoden ein *Point*-Objekt (Integer) bzw. ein *PointF* (Gleitkommawert) verwendet. Die beiden wichtigsten Eigenschaften: x, y.

Beispiel 9.47: Deklaration eines neuen *Point*-Objekts (x = 10, y = 10)

C#

```
Point p = new Point(10, 10);
```

Beispiel 9.48: Direktes Verändern der x-Koordinate

C#

```
Point p = new Point(10, 10);
p.X = 100;
```

Beispiel 9.49: Verschieben des Punkts

C#

```
Point p = new Point(10, 10);
p.Offset(10, 10);
```

Beispiel 9.50: Vergleich von zwei Punktkoordinaten

C#

```
Point p1 = new Point(10, 10);
Point p2 = new Point(10, 10);
if (p1.Equals(p2)) MessageBox.Show("Punkte sind gleich");
```

Beispiel 9.51: Konvertieren in einen Gleitkomma-Point (PointF)

C#

```
Point p1 = new Point(10, 10);
PointF p2;
p2 = Point.op_Implicit(p1);        // Variante 1
p2 = (PointF) p1;                  // Variante 2
```

Size, FSize

Ähnlich wie *Point* verwaltet *Size* bzw. *SizeF* ein Koordinatenpaar, nur dass es sich in diesem Fall um die Breite (*Width*) bzw. Höhe (*Height*) handelt.

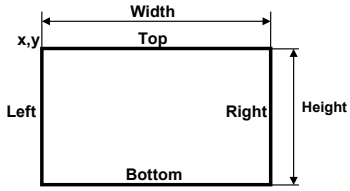
Beispiel 9.52: Deklarieren und Verändern der Breite

C#

```
Size mysize = new Size(100, 80);
mysize.Width = 130;
```

Rectangle, FRectangle

Das dritte Grundobjekt ist im Grunde die Kombination der beiden vorhergehenden Objekte. Die folgende Skizze zeigt die verschiedenen Formen des Zugriffs auf die Koordinatenpaare:



Beispiel 9.53: Deklarieren und Verwenden von *Rectangle*

C#

```
Graphics g = this.CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 200, 100);
g.FillRectangle(new SolidBrush(Color.Red), rec);
```

9.5.2 Vordefinierte Objekte

Vielleicht hatte irgendein Programmierer bei Microsoft ein Einsehen und entschied sich dafür, wenigstens einige Grafikobjekte vorzudefinieren, das heißt, ohne dass Sie diese erst mit viel Aufwand und umfangreichen Parameterlisten initialisieren müssen.

Vier Gruppen können Sie unterscheiden:

- vordefinierte Pinsel (Brushes),
- vordefinierte Stifte (Pens),
- vordefinierte Farben (Colors),
- vordefinierte Grafiken (Icons).

Vordefinierte Pinsel

Neben *Brushes* (alle relevanten Farben) können Sie auch *SystemBrushes* (die vordefinierten Systemfarben) verwenden.

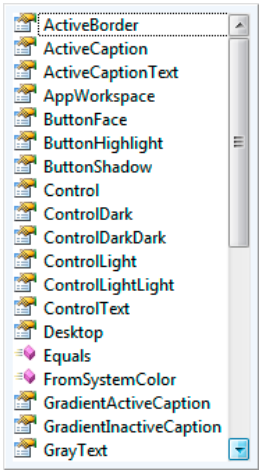
Beispiel 9.54: Erzeugen einer leicht gelb getönten Fläche (Farbe der Hints)

C#

```
Graphics g = this.CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 200, 100);
g.FillRectangle(SystemBrushes.Info, rec);
```

Vordefinierte Stifte

Über das *SystemPens*-Objekt bzw. dessen Eigenschaften rufen Sie vordefinierte Stifte (Breite = 1 Pixel) für alle Windows-Farben ab:



Beispiel 9.55: Verwendung eines System-Pens

C#

```
Graphics g = this.CreateGraphics();
g.DrawRectangle(SystemPens.ControlText, 10, 10, 100, 100);
```

Vordefinierte Farben

Farben können Sie entweder über *SystemColors* (die Farben der Windows-Elemente) oder über *Color* abrufen.

Beispiel 9.56: Verwendung von *Color*

C#

```
Graphics g = this.CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 200, 200);
g.FillPie(new SolidBrush(Color.Red), rec, 0, 100);
```

Vordefinierte Icons

Über *SystemIcons* können Sie die wichtigsten System-Icons direkt abrufen[5]:

Eigenschaft
Application
Asterisk
Error
Exclamation
Hand

[5] Diese unterscheiden sich etwas je nach System, deshalb hier keine Abbildungen.

Eigenschaft
Information
Question
Warning
WinLogo

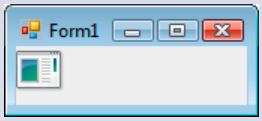
Beispiel 9.57: Verwendung eines System-Icons

C#

```
Graphics g = this.CreateGraphics();
g.DrawIcon(SystemIcons.Application, 0, 0);
```

Ergebnis

Die Grafikausgabe:



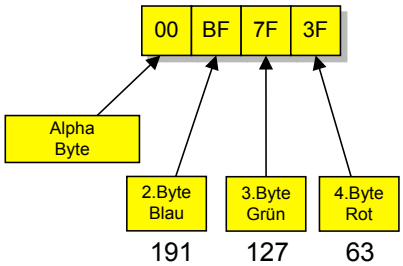
9.5.3 Farben/Transparenz

Farben setzen sich auch unter Windows aus der additiven Überlagerung der drei Grundfarben Rot, Grün und Blau zusammen (RGB). Da jeder Farbanteil in 256 Farbstufen unterteilt ist, ergibt sich eine maximale Anzahl von ca. 16 Mio. Farben. Bei dieser Auflösung ist das menschliche Auge nicht mehr in der Lage, einzelne Abstufungen wahrzunehmen, man spricht von Echtfarben.

ARGB-Farben

GDI+ verwendet für die Verwaltung von Farbinformationen sogenannte ARGB-Werte, das heißt 4 Byte- bzw. Integer-Werte. Auch die bereits vordefinierten Farbwerte (siehe vorhergehender Abschnitt) verwenden dieses Farbmodell.

Um zu verstehen, wie die Farbwerte gespeichert werden können, müssen wir wissen, dass zur Darstellung **eines** Bytes **zwei** Hexziffern benötigt werden. Eine einzelne Hexziffer wird durch eines der 16 Zeichen 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F dargestellt. Die drei niederwertigen Bytes geben die RGB-Farbintensität für Blau, Grün und Rot an.



Beispiel 9.58: Farbwerte

C#

Da pro Byte 256 Werte gespeichert werden können, entspricht der Wert 0x00FF0000 einem reinen Blau mit voller Intensität, der Wert 0x0000FF00 einem reinen Grün und der Wert 0x000000FF einem reinen Rot. 0x00000000 gibt Schwarz und 0x00FFFFFF Weiß an. In obiger Abbildung wird eine graublaue Farbe definiert.

HINWEIS: Möchten Sie einen Graustufenwert erzeugen, müssen die einzelnen Farbanteile jeweils den gleichen Wert aufweisen.

Was ist mit dem höchstwertigen Byte?

Bei ARGB-Werten wird in diesem Byte die Information über die Transparenz dieser Farbe gespeichert. Ein Wert von 255 entspricht der vollen Deckkraft, 0 entspricht vollständiger Transparenz. Mithilfe der Methode *FromArgb* können Sie einen beliebigen Farbwert mit der gewünschten Transparenz erzeugen.

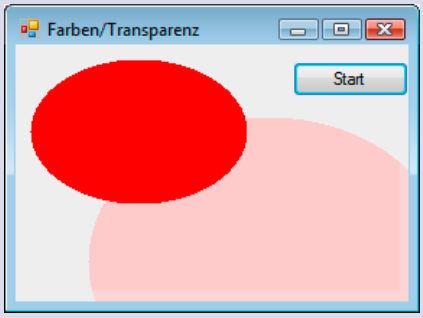
Beispiel 9.59: Erzeugen von zwei teiltransparenten Ellipsen

C#

```
Graphics g  = this.CreateGraphics();
Color c1, c2;
c1 = Color.FromArgb(125, Color.Red);
g.FillEllipse(new SolidBrush(c1), new Rectangle(10, 10, 150, 100));
c2 = Color.FromArgb(12, Color.Red);
g.FillEllipse(new SolidBrush(c2), new Rectangle(50, 50, 250, 200));
```

Ergebnis

Das Ergebnis auf dem Formular:



Rufen Sie obige Anweisungen mehrfach auf, addieren sich die Farbwerte immer weiter, bis eine vollständige Deckung erreicht wird.

Möchten Sie zwei Farbwerte vergleichen, verwenden Sie die Methode *ToArgb*:

```
if (c1.ToArgb() == c2.ToArgb()) MessageBox.Show("Beide Farben sind gleich!");
```

9.5.4 Stifte (Pen)

Für Grafikmethoden, die Linien verwenden, benötigen Sie ein initialisiertes *Pen*-Objekt. Dieses enthält unter anderem Informationen über:

- die Farbe,
- die Dicke,
- die Linienenden,
- die Verbindung zweier zusammenhängender Linien,
- den Füllstil.

Bevor wir Sie mit einer Fülle von Eigenschaften erschlagen, sollen einige Beispiele für mehr Klarheit sorgen.

Einfarbige Stifte

Einfarbige Stifte werden mit einer der vordefinierten Farben oder einem ARGB-Wert als Parameter erzeugt.

Beispiel 9.60: Erzeugen eines roten Stifts

C#

```
Pen myPen = new Pen(Color.Red);
```

Beispiel 9.61: Erzeugen eines einfarbigen Pens mit 50 % Transparenz und 10 Pixeln Breite

C#

```
Pen myPen = new Pen(Color.FromArgb(128, 17, 69, 137), 10);
```

Beispiel 9.62: Zeichnen von zwei aufeinanderfolgenden roten Linien (15 Pixel breit) mit einem Pfeil am Anfang und einem runden Ende

C#

```
Graphics g  = this.CreateGraphics();
```

Zunächst einen einfarbigen *Pen* der Stärke 15 erzeugen:

```
Pen myPen = new Pen(Color.Red, 15);
```

Die Punkte für die Linienenden festlegen:

```
PointF[] punkte = new PointF[] {new PointF(10, 10), new PointF(100, 100),
                                new PointF(50, 150)};
```

Den Linienstart festlegen (Pfeil):

```
   myPen.EndCap = LineCap.Round;
```

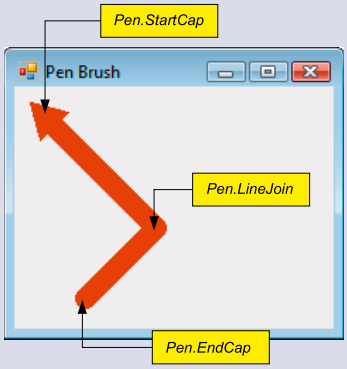
Das Linienende festlegen (rund):

```
myPen.StartCap = LineCap.ArrowAnchor;
```

Die Verbindung der zwei Teillinien festlegen (rund) und Linien zeichnen:

```
myPen.LineJoin = LineJoin.Round;
g.DrawLines(myPen, punkte);
```

Ergebnis



Einige wichtige Linieneigenschaften auf einen Blick:

Eigenschaft	Beschreibung
Alignment	... beschreibt die Position (*Center, Inset, Outset, Left, Right*) der Linie bezüglich der gedachten Ideallinie zwischen zwei Punkten 
Color	... die Füllfarbe des Stifts
LineJoin	... beschreibt den Übergang (*Bevel, Miter, MiterClipped, Round*) zwischen zwei aufeinanderfolgenden Linien. Siehe dazu vorhergehendes Beispiel.
StartCap, EndCap	... beschreibt die Form der Linienenden (*Flat, Round, Square ...*). Siehe dazu vorhergehendes Beispiel.
DashCap	... beschreibt die Form der einzelnen Punkte/Linienabschnitte bei gestrichelten Linien (*Flat, Round Triangle*)
DashStyle	... beschreibt die Form von gestrichelten Linien (*Custom, Dash, DashDot, DashDotDot, Dot, Solid*)
PenType	... beschreibt die Stiftart (*SolidColor, Hatchfill, LinearGradient, PathGradient, TextureFill*). Diese Eigenschaft ist schreibgeschützt.

Stifte mit Füllung

Im Unterschied zu den bisher vorgestellten Stiften, die lediglich unterschiedliche Linienmuster bzw. Linienfarben aufweisen konnten, lassen sich auch Stifte erzeugen, die mit einem *Brush* statt mit einer Farbe initialisiert werden.

Beispiel 9.63: Erzeugen eines Stifts mit Brush als Füllung

C#

```
Graphics g = this.CreateGraphics();
Pen mypen = new Pen(brushes.Azure, 10);
g.DrawLine(mypen, 10, 10, 100, 100);
```

Auf den ersten Blick werden Sie keinen Unterschied zu den bisherigen Pens erkennen, da wir einen *SolidBrush* verwendet haben.

Etwas anders sieht die Linie allerdings aus, wenn Sie zum Beispiel einen *HatchBrush* (siehe auch folgender Abschnitt) für die Linienfüllung verwenden.

Beispiel 9.64: *HatchBrush* für Linie erzeugen

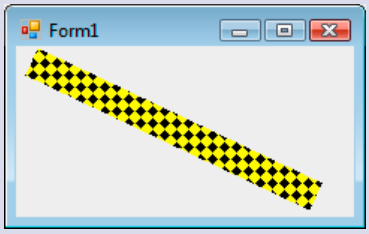
C#

Vergessen Sie nicht, den entsprechenden Namespace einzubinden:

```
using System.Drawing.Drawing2D;
...
  Graphics g = this.CreateGraphics();
  HatchBrush myBrush = new HatchBrush(HatchStyle.SolidDiamond, Color.Black,
Color.Yellow);
  Pen myPen = new Pen(myBrush, 20);
  g.DrawLine(myPen, 10, 10, 200, 100);
```

Ergebnis

Das Ergebnis unterscheidet sich doch wesentlich von den bisher bekannten Linienarten:



Wie Sie sehen, können Sie fast beliebige Stiftfüllungen erzeugen, Sie müssen nur einen entsprechenden Brush (siehe folgender Abschnitt) zur Verfügung stellen.

HINWEIS: Verwechseln Sie nicht Stifte mit Pinseln. Mit Stiften können Sie niemals eine Figur füllen, mit Pinseln können Sie keine Figuren zeichnen, sondern nur füllen.

9.5.5 Pinsel (Brush)

Damit sind wir bei einem der wohl komplexesten Zeichenobjekte angelangt. Neben dem bereits mehrfach in diesem Kapitel verwendeten *SolidBrush* und dem im vorhergehenden Abschnitt bereits angesprochenen *HatchBrush* finden Sie noch weitere drei Vertreter dieser Gattung.

Alle Pinsel auf einen Blick:

Typ	Beschreibung
SolidBrush	... ein Pinsel mit einheitlicher Farbe und ohne Muster
HatchBrush	... ein Pinsel mit einem Linienmuster sowie Vordergrund- (Linie) und Hintergrundfarbe
TextureBrush	... ein Pinsel mit Images/Bitmaps als Füllmuster
LinearGradientBrush	... ein Pinsel mit einem Farbverlauf als Füllung (Verlauf zwischen zwei Farben)
PathGradientBrush	... ein Pinsel mit mehreren Farbverläufen (Verlauf zwischen einer Farbe und mehreren anderen)

Gefüllte Objekte, basierend auf den oben genannten Pinseln, erzeugen Sie mit den entsprechenden *Fill*xyz-Methoden (z. B. *FillRectangle*) des *Graphics*-Objekts.

9.5.6 SolidBrush

Das Erzeugen eines *SolidBrush* dürfte Sie kaum vor Probleme stellen. Übergeben Sie eine Farbe oder verwenden Sie die Methode *FromArgb*, um eine neue Farbe zu definieren.

Beispiel 9.65: Roter Pinsel

C#

```
SolidBrush mybrush = new SolidBrush(Color.Red);
```

Beispiel 9.66: Teilweise transparenter Pinsel mit selbst definierter Farbe

C#

```
SolidBrush mybrush = new SolidBrush(Color.FromArgb(123, 10, 17, 36));
```

9.5.7 HatchBrush

Bevor Sie sich näher mit dieser Pinselvariante beschäftigen können, müssen Sie den notwendigen Namespace *System.Drawing.Drawing2D* einbinden.

Ein *HatchBrush* besteht aus einem Vordergrundmuster mit eigener Farbe und einer Hintergrundfarbe. Der Konstruktor:

Syntax:

```
HatchBrush(HatchStyle hatchstyle, Color foreColor, Color backColor);
```

Übergeben können Sie unter anderem folgende Konstanten:

HatchStyle
BackwardDiagonal
Cross
DarkDownwardDiagonal
DarkHorizontal
DarkUpwardDiagonal
DarkVertical
DashedDownwardDiagonal
DashedHorizontal
DashedUpwardDiagonal
DashedVertical
DiagonalBrick
DiagonalCross
Divot
DottedDiamond
DottedGrid
ForwardDiagonal

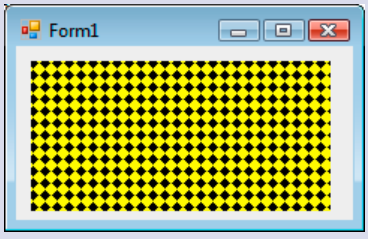
Beispiel 9.67: Erzeugen und Verwenden einer *HatchBrush*

C#

```
   Graphics g = this.CreateGraphics();
   HatchBrush myBrush = new HatchBrush(HatchStyle.SolidDiamond, Color.Black,
 Color.Yellow);
   g.FillRectangle(myBrush, 10, 10, 200, 100);
```

Ergebnis

Das erzeugte Rechteck:



HINWEIS: Eine Skalierung des Ausgabegeräts (*ScaleTransform*) hat keinen Einfluss auf die Mustergröße.

9.5.8 TextureBrush

Mit einem *TextureBrush* lassen sich Flächen mit Bitmap-Mustern füllen. Ist die Bitmap zu klein, um die gesamte Fläche auszufüllen, wird diese wiederholt dargestellt.

Beispiel 9.68: Basierend auf den System-Icons soll ein Rechteck mit Error-Symbolen gefüllt werden.

C#

```
Graphics g = this.CreateGraphics();
TextureBrush myBrush = new TextureBrush(SystemIcons.Error.ToBitmap());
g.FillRectangle(myBrush, 10, 10, 200, 100);
```

Ergebnis

Die Ausgabe:

Natürlich stellt es auch kein Problem dar, die Grafik aus einer Datei zu laden.

Beispiel 9.69: Laden der Datei *Bitmap1.bmp* als *TextureBrush*

C#

```
Graphics g = this.CreateGraphics();
Image img  = new Bitmap("bitmap1.bmp");

TextureBrush myBrush = new TextureBrush(img);
g.FillRectangle(myBrush, 10, 10, 200, 100);
```

9.5.9 LinearGradientBrush

Mit dem *LinearGradientBrush* bringen Sie im wahrsten Sinne des Wortes mehr Farbe in Ihre Anwendungen. Das Grundprinzip: Sie geben zwei Farben und eine Richtung (daher das „linear“) an und GDI+ berechnet Ihnen den zugehörigen Farbverlauf.

An den Konstruktor müssen Sie folgende Werte übergeben:

Syntax:

```
LinearGradientBrush(Rectangle rect, Color startfarbe, Color endfarbe,
                    LinearGradientMode linearGradientMode);
```

Rectangle gibt ein Rechteck an (es sind auch zwei *Point*-Werte zulässig), in dem der Farbverlauf berechnet wird. Die Betonung liegt auf „berechnet“. Welche Ausgabefläche Sie später mit dem neuen *Brush* füllen, ist eine ganz andere Frage.

Start- und Endfarbe sind normale ARGB-Color-Werte mit Transparenzangabe. Das heißt, wenn Sie beispielsweise zwei gleiche Farben, aber unterschiedliche Alpha-Werte angeben, können Sie einen Farbverlauf mit zu- bzw. abnehmender Transparenz realisieren.

Folgende *LinearGradientMode*-Werte sind zulässig:

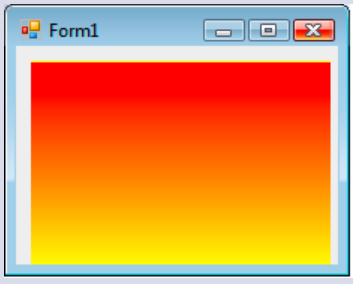
Konstante	Beschreibung
BackwardDiagonal	Farbverlauf von rechts oben nach links unten
ForwardDiagonal	Farbverlauf von links oben nach rechts unten
Horizontal	Farbverlauf von links nach rechts
Vertical	Farbverlauf von oben nach unten

Beispiel 9.70: Ein vertikaler Farbverlauf von Rot nach Gelb soll realisiert werden.

C#

```
Graphics g = this.CreateGraphics();
Rectangle rect = new Rectangle(10, 10, 200, 135);
LinearGradientBrush myBrush = new LinearGradientBrush(rect, Color.Red,
                                  Color.Yellow, LinearGradientMode.Vertical);
g.FillRectangle(myBrush, new Rectangle(10, 10, 200, 185));
```

Ergebnis



Wer ganz genau hinsieht, wird allerdings schnell „ein Haar in der Suppe“ finden. Die erste Zeile des Ausgaberechtecks ist nicht rot, sondern gelb. Die Ursache ist vermutlich die Wiederholung des Farbverlaufs, wenn der berechnete Verlauf kleiner als die Ausgabefläche ist.

Nehmen Sie folgende Änderung vor, um die gelbe Zeile zu entfernen:

```
Rectangle rect = new Rectangle(10, 10, 200, 185);
LinearGradientBrush myBrush = new LinearGradientBrush(rect, Color.Red,
                                  Color.Yellow, LinearGradientMode.Vertical);

g.FillRectangle(myBrush, new Rectangle(10, 10, 200, 180));
```

9.5.10 PathGradientBrush

Hier haben wir es mit einer Spezialform von Gradienten zu tun, die es uns erlauben, recht eindrucksvolle Farbeffekte zu realisieren. Ausgehend von einer zentralen Farbe können Sie innerhalb eines *Path*-Objekts (siehe dazu Abschnitt 9.5.12) mehrere Farbverläufe zu verschiedenen Farben realisieren.

Beispiel 9.71: Erzeugen und Verwenden eines *PathGradientBrush*

C#

```
Graphics g = this.CreateGraphics();
```

Zunächst erzeugen wir das *Path*-Objekt:

```
GraphicsPath path = new GraphicsPath();
path.AddLine(10, 10, 300, 10);
path.AddLine(300, 10, 250, 200);
path.AddLine(250, 200, 150, 250);
path.AddLine(150, 250, 50, 200);
```

Jetzt können wir mit diesem *Path* den *PathGradientBrush* erzeugen und konfigurieren:

```
PathGradientBrush myBrush = new PathGradientBrush(path);
```

Ein Array mit den Farben für die jeweiligen Eckpunkte im *Path*:

```
Color[] myColors = new Color[] {Color.Yellow, Color.Green, Color.Red,
                                Color.Cyan, Color.Blue};
```

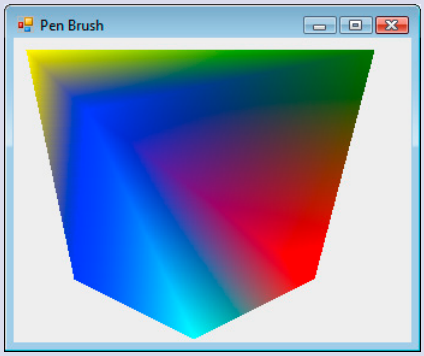
Die Farbe im Mittelpunkt[1] und an den Ecken des *Path*:

```
myBrush.CenterColor = Color.Blue;
myBrush.SurroundColors = myColors;
```

Wir geben den *Path* aus:

```
myBrush.CenterPoint = new PointF(60, 60);
g.FillPath(myBrush, path);
```

Ergebnis



9.5.11 Fonts

Wie bei Stiften und Pinseln handelt es sich auch bei Schriften (Fonts) um Objekte, die Sie zunächst erzeugen müssen. Sage und schreibe dreizehn verschiedene Konstruktorüberladungen werden Ihnen angeboten, wir belassen es bei den beiden folgenden Varianten:

Syntax:

```
Font(string familyName, float emSize);
```

Syntax:

```
Font(string familyName, float emSize, FontStyle style);
```

Was mit *familyName* und *emSize* gemeint ist, dürfte klar sein. Über *style* können Sie eine Kombination der folgenden *FontStyle*-Werte zuweisen:

- *Bold* (fett)
- *Italic* (kursiv)
- *Regular* (normal)
- *Strikeout* (durchgestrichen)
- *Underline* (unterstrichen)

Kombinieren können Sie die Werte durch die Verknüpfung mit dem OR-Operator.

HINWEIS: Die Farbe bzw. die Füllung der Schriftart legen Sie erst bei der Ausgabe mittels *DrawString* über einen entsprechenden *Brush* fest. Damit können Sie auch Farbverläufe oder Transparenzeffekte bei Fonts erreichen.

Genug der Theorie, ein praktisches Beispiel soll die Verwendung zeigen.

Beispiel 9.72: Erzeugen und Verwenden eines neuen Font-Objekts

C#

```
Graphics g = this.CreateGraphics();
Font f = new Font("Arial", 24, FontStyle.Italic | FontStyle.Bold);
g.DrawString("Test Schriftarten", f, new SolidBrush(Color.Blue), 10, 10);
```

Ergebnis

Ein Blick auf die Eigenschaften des *Font*-Objekts macht uns neugierig: Mit *Bold, Italic, Size* etc. stehen uns alle Möglichkeiten zur Konfiguration der Schriftart zur Verfügung. Doch ach: Alle wünschenswerten Eigenschaften sind schreibgeschützt, Sie müssen also wohl oder übel eine neue Schriftart erzeugen.

9.5.12 Path-Objekt

Path-Objekte fassen mehrere Zeichenoperationen (*DrawLine, DrawString, DrawEllipse* ...) bzw. mehrere grafische Primitive (Linien, Kreise etc.) quasi in einem Objekt zusammen. Sie können *Path*s am besten mit der Gruppieren-Funktion eines Grafikprogramms vergleichen. Ähnlich wie mit der Gruppe können Sie den kompletten *Path* mit einer Anweisung auf einem *Graphics*-Objekt wiedergeben (*DrawPath* oder *FillPath*). Welche Linientypen bzw. welcher Füllstil benutzt wird, entscheiden Sie erst beim Zeichnen auf dem *Graphics*-Objekt, nicht beim Erstellen des Objekts.

Beispiel 9.73: Erzeugen eines einfachen *Path* (nur zwei Ellipsen und zwei Linien) und Wiedergabe auf dem *Form*-Objekt

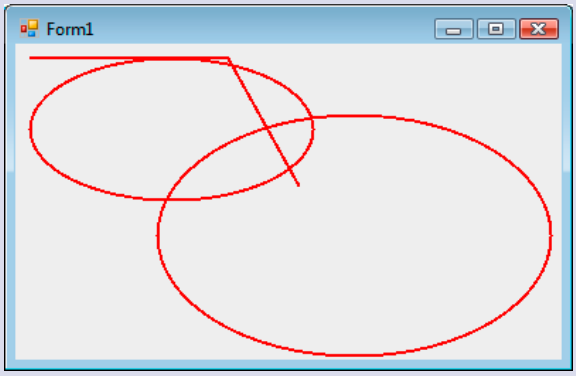
C#

```
using System.Drawing.Drawing2D;
...
   Graphics g = this.CreateGraphics();
   GraphicsPath myPath = new GraphicsPath();

   myPath.AddEllipse(10, 10, 200, 100);
   myPath.AddEllipse(100, 50, 278, 170);
   myPath.AddLine(10, 10, 150, 10);
   myPath.AddLine(150, 10, 200, 100);
```

Wie Sie sehen, wird erst bei der Wiedergabe des *Path*-Objekts ein entsprechender *Pen* zugeordnet:

```
      g.DrawPath(new Pen(Color.Red, 2), myPath);
```

Ergebnis



Füllen

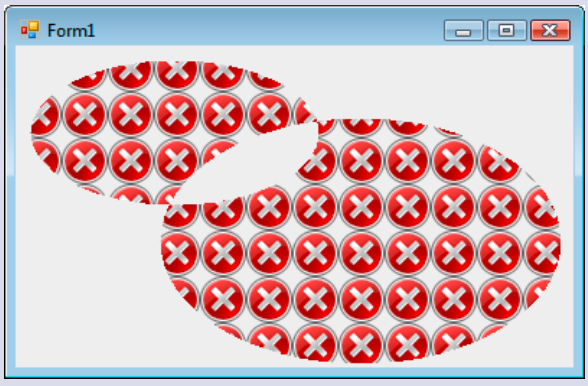
Sie können ein beliebig zusammengesetztes *Path*-Objekt mit einem Pinsel Ihrer Wahl füllen lassen. Ein Beispiel hatten Sie ja bereits in Abschnitt 9.5.10 (PathGradientBrush) kennengelernt. Das Grundprinzip ist immer gleich. Sie erzeugen ein *Path*-Objekt, fügen diesem die gewünschten Grafikanweisungen hinzu und geben dann diesen *Path* auf einem *Graphics*-Objekt aus.

Beispiel 9.74: Füllen mit einem Bitmap-Muster

C#

```
Graphics g = this.CreateGraphics();
GraphicsPath myPath = new GraphicsPath();
TextureBrush myBrush = new TextureBrush(SystemIcons.Error.ToBitmap());
myPath.AddEllipse(10, 10, 200, 100);
myPath.AddEllipse(100, 50, 278, 170);
g.FillPath(myBrush, myPath);
```

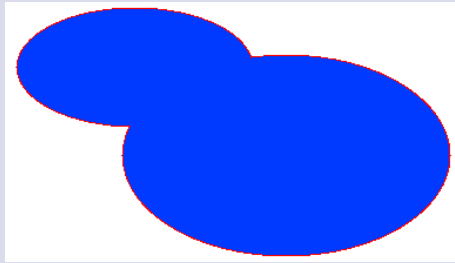
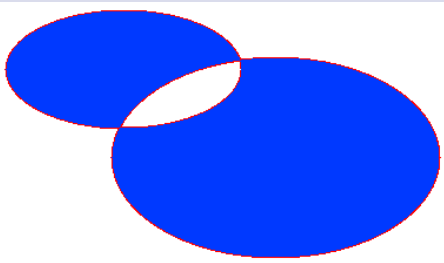
Ergebnis



Fillmode

In diesem Zusammenhang ist Ihnen sicher aufgefallen, dass der sich überschneidende Bereich beider Ellipsen nicht gefüllt worden ist.

Wie überlappende bzw. sich schneidende Objekte gefüllt werden, bestimmen Sie mit der Eigenschaft *FillMode*. Statt vieler Worte über die Funktionsweise dürften die beiden folgenden Abbildungen mehr über die Funktionsweise aussagen:

FillMode = FillMode.Winding	*FillMode = FillMode.Alternate*
	

Sehen wir uns mit diesem Wissen noch einmal unser Einstiegsbeispiel an, verwenden jedoch statt der *DrawPath*- die *FillPath*-Methode.

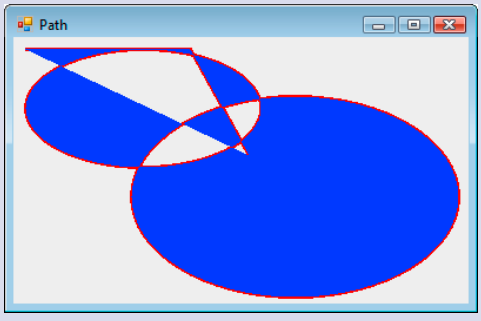
Beispiel 9.75: Füllen eines Paths

C#

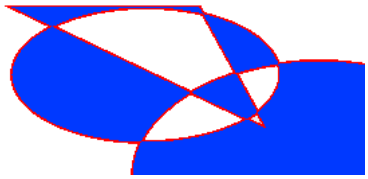
```
Graphics g = this.CreateGraphics();
GraphicsPath myPath = new GraphicsPath();
myPath.AddEllipse(10, 10, 200, 100);
myPath.AddEllipse(100, 50, 278, 170);
myPath.AddLine(10, 10, 150, 10);
myPath.AddLine(150, 10, 200, 100);
myPath.CloseAllFigures();
g.DrawPath(new Pen(Color.Red, 3), myPath);
g.FillPath(Brushes.Blue, myPath);
```

Ergebnis

Im Ergebnis werden Sie feststellen, dass beim Füllen auch die beiden Linien eine Rolle spielen.



Zwischen den beiden Endpunkten der Linien wird zwar keine Linie gezeichnet, eine Füllung kommt unter Berücksichtigung von *Fillmode* jedoch zustande. Soll aus den beiden Linien eine in sich geschlossene Fläche erzeugt werden, rufen Sie die Methode *CloseAllFigures* auf, die die beiden Linien zu einem Dreieck verbindet (achten Sie auf die neue Verbindungslinie zwischen den beiden Linienendpunkten):



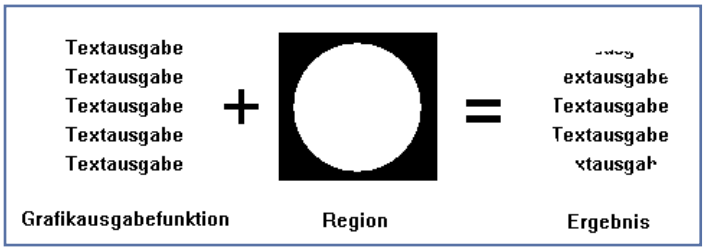
HINWEIS: Ein weiteres Einsatzgebiet für *Path*-Objekte findet sich im Zusammenhang mit dem Clipping, das wir im folgenden Abschnitt näher betrachten wollen.

9.5.13 Clipping/Region

In Ihren Programmen sind Sie es gewohnt, dass Zeichenoperationen, die über den Clientbereich des Formulars bzw. der Komponente hinausgehen, einfach abgeschnitten werden. Damit haben Sie auch schon die einfachste Form von Clipping kennengelernt.

Für bestimmte visuelle Effekte ist es häufig wünschenswert, dass diese Ausgabebereiche nicht nur rechteckig, sondern zum Beispiel auch mal rund oder aus verschiedenen grafischen Primitiven zusammengesetzt sind.

Die Abbildung zeigt das Ergebnis von Grafikausgaben in einem runden Clipping-Bereich:



Ein Clipping-Bereich ist immer einem *Graphics*-Objekt zugewiesen, mithilfe der Methode *SetClip* können Sie diesen Bereich verändern.

Unter GDI+ bieten sich mehrere Varianten an:

- Übernahme von anderem *Graphics*-Objekt (*Graphics.SetClip(Graphics)*),
- Clipping in einem Rechteck (*Graphics.SetClip(Rectangle)*),
- Clipping in einem *Path* (*Graphics.SetClip(GraphicsPath)*),
- Clipping in einer *Region* (*Graphics.SetClip(Region)*).

Im Folgenden beschränken wir uns auf die letzte Variante, die Vorgehensweise ist auch bei einem *Path* oder einem Rechteck immer gleich.

Regions

Regions können, im Zusammenhang mit dem Clipping, quasi wie eine Schablone betrachtet werden. Da Sie *Regions* recht einfach zusammensetzen können (mit unterschiedlichen Verknüpfungen), dürften sie die erste Wahl für die meisten Clipping-Aufgaben sein.

Das Erzeugen einer Region ist relativ simpel. Über den Konstruktor des *Region*-Objekts können Sie bereits entscheiden, worauf die Region aufgebaut wird:

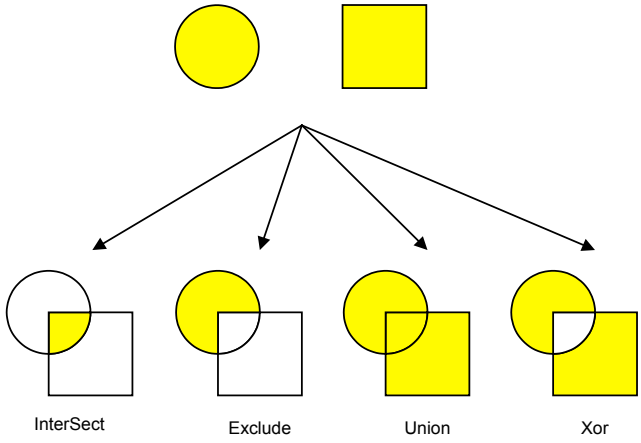
- ein *Path*-Objekt,
- ein Rechteck,
- ein anderes *Region*-Objekt.

Beispiel 9.76: Erzeugen und Füllen einer rechteckigen Region

C#

```
using System.Drawing.Drawing2D;
...
 Graphics g = this.CreateGraphics();
 SolidBrush blueBrush = new SolidBrush(Color.Blue);
 Region Reg1 = new Region(new Rectangle(100, 100, 200, 200));
 g.FillRegion(blueBrush, Reg1);
```

Zwei Regionen können Sie mithilfe der *Region*-Methoden auch kombinieren:



Beispiel 9.77: Kombination zweier rechteckiger Regionen

C#

```
Graphics g = this.CreateGraphics();

SolidBrush blueBrush = new SolidBrush(Color.Blue);
Region Reg1 = new Region(new Rectangle(100, 100, 200, 200));
Region Reg2 = new Region(new Rectangle(50, 50, 150, 150));
```

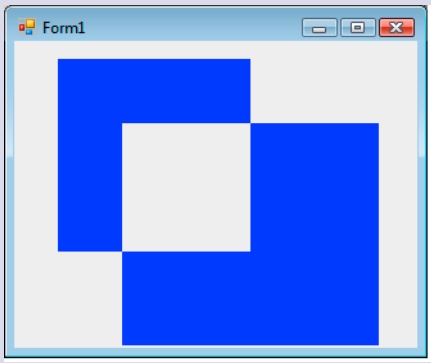
Region2 wird mit Region1 XOR-verknüpft:

```
Reg2.Xor(Reg1);
```

Die Region füllen:

```
g.FillRegion(blueBrush, Reg2);
```

Ergebnis



Clipping

Nach diesem Kurzeinstieg in die Arbeit mit Regionen können wir uns wieder dem Clipping zuwenden. Nutzen Sie Regionen oder Paths, um den Ausgabebereich von Grafikoperationen (Füllen, Zeichnen) in einem *Graphics*-Objekt zu beschränken. Alle über den Clippingbereich hinausgehenden Zeichenoperationen werden abgeschnitten.

Beispiel 9.78: Erzeugen eines Clipping-Bereichs, der aus zwei Regionen besteht

C#

```
Graphics g = this.CreateGraphics();

Region Reg1 = new Region(new Rectangle(100, 100, 200, 200));
Region Reg2 = new Region(new Rectangle(50, 50, 150, 150));
```

Verbinden beider Regionen:

```
Reg2.Union(Reg1);
```

Clipping-Bereich festlegen:

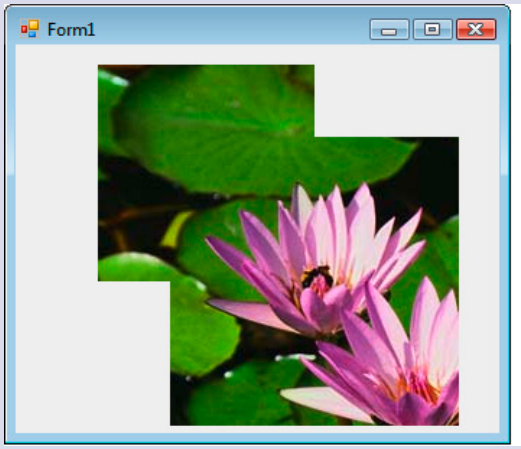
```
g.SetClip(Reg2, CombineMode.Replace);
```

Normales Zeichnen einer Bitmap, die normalerweise die gesamte Fensterfläche füllen würde:

```
g.DrawImage(PictureBox1.Image, 0, 0);
```

Ergebnis

Im Ergebnis wird die Grafikausgabe auf den gewünschten Clipping-Bereich eingeschränkt:

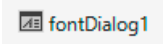


9.6 Standarddialoge

Im Zusammenhang mit der Bearbeitung von Grafiken stehen Ihnen zwei wesentliche Dialoge zur Verfügung:

- Schriftauswahl
- Farbauswahl

9.6.1 Schriftauswahl



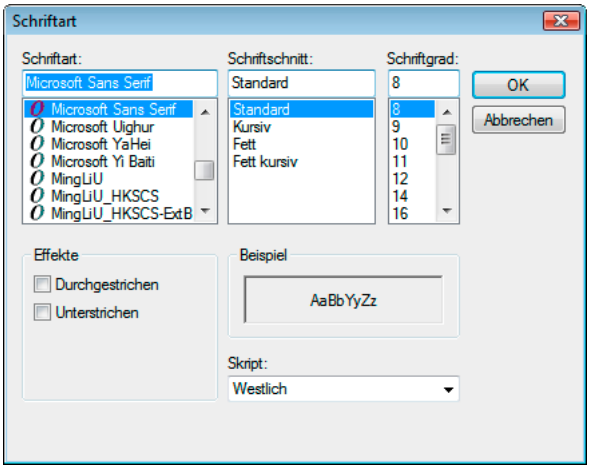
Der wohl jedem bekannte Dialog zur Auswahl einer Schrift bzw. deren Parameter (Größe, Farbe etc.) lässt sich über die entsprechende Komponente *FontDialog* in Ihre Anwendung einbinden.

Die Anzeige erfolgt zur Laufzeit mittels *ShowDialog*-Methode:

```
if (fFontDialog1.ShowDialog() == DialogResult.OK)
 {  ...  }
```

Über den Rückgabewert der Methode können Sie den Status beim Beenden der Dialogbox ermitteln (*DialogResult.OK* oder *DialogResult.Abort*).

Die zweifelsfrei interessanteste Eigenschaft dieser Komponente ist *Font*. Sie können diese Eigenschaft sowohl vor dem Aufruf des Dialogs initialisieren als auch nach der Anzeige der Dialogbox auswerten.



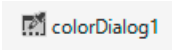
Das Zuweisen der Fontattribute, zum Beispiel an einen Button, gestaltet sich absolut simpel:

```
button1.Font = fontDialog1.Font;
```

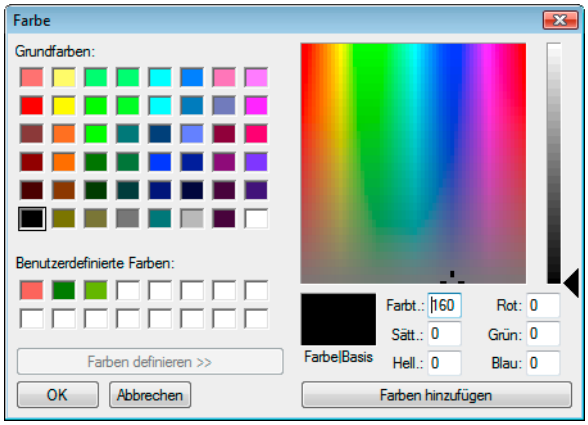
Außer *Font* sind die in der folgenden Tabelle aufgeführten Eigenschaften für das Verhalten bzw. das Aussehen der Dialogbox von Bedeutung:

Eigenschaft	Beschreibung
AllowScriptChange	Skript (Westlich, Arabisch, Türkisch etc.) kann geändert werden.
AllowSimulations	Windows (GDI) darf Schriften simulieren.
AllowVectorFonts	Auswahl von Vektorschriftarten zulassen
AllowVerticalFonts	Vertikale Fonts sind zulässig.
Color	Die aktuelle Schriftfarbe
FixedPitchOnly	Es werden nur Schriftarten mit fester Zeichenbreite angezeigt.
FontMustExist	Es können nur vorhandene Schriftarten ausgewählt werden.
MaxSize, MinSize	Maximaler und minimaler Schriftgrad, der ausgewählt werden kann
ShowColor	Anzeige der Farbauswahl-Combobox
ShowEffects	Anzeige der Effekte (Durchstreichen, Unterstreichen, Farbe)
ShowHelp	Anzeige des Hilfe-Buttons

9.6.2 Farbauswahl



Für die Auswahl von Farben zur Laufzeit können Sie den Standarddialog *ColorDialog* verwenden. Wie auch beim *FontDialog* genügt der einfache Aufruf der Methode *ShowDialog*, um den Dialog anzuzeigen:



Über den Rückgabewert der Methode können Sie den Status beim Beenden der Dialogbox ermitteln (*DialogResult.OK* oder *DialogResult.Abort*).

Im Mittelpunkt der Komponente steht die Eigenschaft *Color* (ein ARGB-Wert), die Sie sowohl vor dem Aufruf des Dialogs initialisieren als auch nach der Anzeige der Dialogbox auswerten können.

Beispiel 9.79: Verändern der Formularfarbe

C#

```
colorDialog1.Color = this.BackColor;
if (colorDialog1.ShowDialog() == DialogResult.OK)
    this.BackColor = colorDialog1.Color;
```

Die wichtigsten Eigenschaften des Dialogs zeigt die folgende Tabelle:

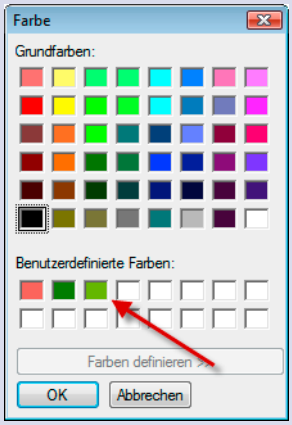
Eigenschaft	Bemerkung
AllowFullOpen, *FullOpen*	Ein-/Ausblenden der rechten Seite des Dialogs zum Definieren eigener Farben zulassen
AnyColor	Anzeigen aller verfügbaren Farben
Color	Ruft die von den Benutzern ausgewählte Farbe ab oder legt diese fest
CustomColors	Definieren oder Abfragen von nutzerdefinierten Farben (siehe Beispiel)
FullOpen	Ein-/Ausblenden der rechten Seite des Dialogs zum Definieren eigener Farben beim Öffnen des Dialogs
SolidColorOnly	Nur Volltonfarben können ausgewählt werden.

Beispiel 9.80: Zuweisen von nutzerdefinierten Farben

C#

```
colorDialog1.AllowFullOpen = false;
colorDialog1.CustomColors = new int[] {6975964, 231202, 1294476};
if (colorDialog1.ShowDialog() == DialogResult.OK) this.BackColor =
colorDialog1.Color;
```

Ergebnis



HINWEIS: Wer weitere und detailliertere Informationen über die Grafikausgabe mit GDI+ sucht, der sollte sich das Kapitel 12 zu Gemüte führen.

9.7 Praxisbeispiele

9.7.1 Ein Graphics-Objekt erzeugen

In klassischen Programmiersprachen ist es üblich, mit Methoden direkt auf die Zeichenoberfläche eines Formulars oder eines *Picture*-Controls zuzugreifen. Als .NET-Programmierer müssen Sie umdenken.

Zugriff auf alle wesentlichen Grafikmethoden erhalten Sie über ein Graphics-Objekt. Im Vergleich mit anderen .NET-Objekten hat es allerdings die Besonderheit, dass man es nicht mit dem *new*-Konstruktor erzeugen kann. Woher also nehmen wir es? Das vorliegende Beispiel zeigt Ihnen vier Möglichkeiten:

- Nutzung des im Paint-Event des Formulars übergebenen *Graphics*-Objekts,
- Nutzung des in der überschriebenen *OnPaint*-Methode übergebenen *Graphics*-Objekts,
- Erzeugen eines neuen *Graphics*-Objekts mit der *CreateGraphics*-Methode des Formulars,
- Nutzung des im *Paint*-Event einer *PictureBox* (oder einer anderen Komponente, die über dieses Ereignis verfügt) übergebenen *Graphics*-Objekts.

Lassen Sie uns also ein wenig experimentieren!

Zunächst soll uns ein nacktes Windows Form genügen, auf das wir verschiedenfarbige Ellipsen zeichnen wollen. Später ergänzen wir noch weitere Steuerelemente (*Button*, *PictureBox*).

Variante 1: Verwendung des Paint-Events

Die in der *System.Windows.Forms.Form*-Basisklasse implementierte *OnPaint*-Methode wird automatisch nach jedem Freilegen und Verdecken des Fensters aufgerufen. Sie löst das *Paint*-Ereignis aus, das wir in einem *Paint*-Eventhandler abfangen und behandeln wollen.

Über das Argument des Events ist ein *Graphics*-Objekt verfügbar:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillEllipse(new SolidBrush(Color.Red), 10, 30, 200, 100);  // rote Ellipse
}
```

Test

Die rote Ellipse erscheint sofort nach Programmstart und ist auch nach Freilegen und Verdecken des Fensters zu sehen (siehe folgende Abbildung).

Variante 2: Überschreiben der OnPaint-Methode

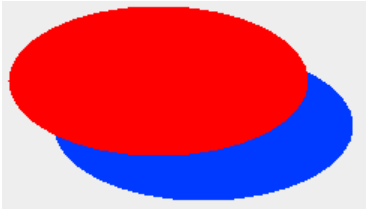
Dies ist die in der .NET-Dokumentation favorisierte Realisierung, bei der Sie keinen neuen Eventhandler verwenden müssen, sondern lediglich die *OnPaint*-Methode der Basisklasse überschreiben. Wir wollen nach diesem Prinzip eine versetzte blaue Ellipse zeichnen.

Implementieren Sie die Überschreibung wie folgt:

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillEllipse(new SolidBrush(Color.Blue), 40, 60, 200, 100);  // blaue Ellipse
    base.OnPaint(e);  // Aufruf der Basisklassenmethode
}
```

Test

An der Reihenfolge (unten Blau, oben Rot) erkennen Sie, dass die überschriebene *OnPaint*-Methode zuerst abgearbeitet wurde und erst anschließend der bereits vorhandene *Paint*-Eventhandler:



HINWEIS: Wenn Sie die Anweisung *base.OnPaint(e)* auskommentieren, wird das *Paint*-Ereignis nicht mehr ausgelöst und nur noch die blaue Ellipse erscheint!

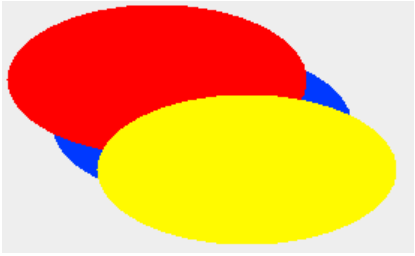
Variante 3: Graphics-Objekt mit CreateGraphics erzeugen

Diese Variante nutzt die Möglichkeit, über die *CreateGraphics*-Methode des Formulars ein neues *Graphics*-Objekt zu erzeugen. Allerdings benötigen wir hier einen *Button*, um das Zeichnen (versetzte gelbe Ellipse) zu demonstrieren.

```
    private void button1_Click(object sender, EventArgs e)
    {
        Graphics g = this.CreateGraphics();
        g.FillEllipse(new SolidBrush(Color.Yellow), 70, 90, 200, 100);  // gelbe
Ellipse
    }
```

Test

Die gelbe Ellipse erscheint erst nach Klick auf den Button. Im Unterschied zur roten und blauen Ellipse (Variante 1 und 2) verschwindet diese Ellipse wieder, nachdem das Formular vorübergehend verdeckt wurde.



Variante 4: Verwendung des Graphics-Objekts einer PictureBox

Bei einer *PictureBox* – wie bei vielen anderen Komponenten auch – können Sie über die *CreateGraphics*-Methode auf die Zeichenfläche zugreifen. Sinnvoller ist allerdings auch hier die Nutzung des im *Paint*-Event übergebenen *Graphics*-Objekts, da Sie sich dann um die Restaurierung des Bildinhalts nicht weiter zu kümmern brauchen.

Ergänzen Sie die Oberfläche des Testformulars um eine *PictureBox* und erzeugen Sie einen Eventhandler für das *Paint*-Ereignis der *PictureBox*:

```
    private void pictureBox1_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.FillEllipse(new SolidBrush(Color.Red), 10, 30, 200, 100);
    }
```

9.7.2 Zeichenoperationen mit der Maus realisieren

Dieses Beispiel zeigt Ihnen eine Möglichkeit, wie Sie einfache Zeichenoperationen (Linie, Ellipse, Rechteck) in ein eigenes Programm integrieren können. Dreh- und Angelpunkt ist die Verwendung einer Hintergrundbitmap, mit deren Hilfe wir einen Gummiband-Effekt beim Zeichnen realisieren.

Oberfläche

Ein Windows Form und eine *ToolStrip*-Komponente zur Auswahl der Zeichenoperation. Zusätzlich fügen Sie noch eine *ColorDialog*-Komponente zur Auswahl der Malfarbe ein.

Wir haben die Oberfläche bewusst einfach gehalten, hier geht es um das Handling der Maus-Events und die Verwendung einer Hintergrundbitmap und nicht um Schönheit im Detail.

Quelltext

```
public partial class Form1 : Form
{
```

Zunächst eine Enumeration definieren:

```
    enum Figuren : int  { Linie, Ellipse, Rechteck }
```

Eine Statusvariable für die Zeichenoperation:

```
    private Figuren Figur = Figuren.Linie;
```

Die Bitmap und das *Graphics*-Objekt für die Hintergrundbitmap:

```
    private Bitmap bmp;
    private Graphics bckg;
```

Der Malstift:

```
    private Pen p;
```

Start- und Endpunkt der Zeichenoperation:

```
    private Point p1;
    private Point p2;
```

Im Konstruktor erzeugen wir zunächst die Hintergrundgrafik in der maximal nötigen Größe:

```
public Form1()
{
    InitializeComponent();
    Size maxsize = SystemInformation.PrimaryMonitorMaximizedWindowSize;
    bmp = new Bitmap(maxsize.Width, maxsize.Height);
    bckg = Graphics.FromImage(bmp);
```

Mit Hintergrundfarbe füllen:

```
    bckg.Clear(this.BackColor);
```

Zeichenstift initialisieren:

```
    p = new Pen(Color.Black);
}
```

Mit dem Drücken der Maustaste beginnt der Zeichenvorgang, wir merken uns die Position:

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    p1 = e.Location;
}
```

Jede Mausbewegung bei gedrückter linker Maustaste erfordert das Wiederherstellen der Grafik vor dem Zeichenvorgang und das erneute Zeichnen mit den neuen Mauskoordinaten:

```
private void Form1_MouseMove(object sender, MouseEventArgs e)
{
    p2 = e.Location;
    if (e.Button == MouseButtons.Left)
    {
        Graphics g = CreateGraphics();
        g.DrawImage(bmp, 0, 0);
        Zeichne(g);
        g.Dispose();
    }
}
```

Erst wenn die Maustaste losgelassen wird, fügen wir das gerade gewählte Zeichenobjekt mit den aktuellen Koordinaten in die Hintergrundbitmap ein:

```
private void Form1_MouseUp(object sender, MouseEventArgs e)
{
    Zeichne(bckg);
}
```

Die eigentliche Zeichenroutine unterscheidet die einzelnen Zeichenobjekte:

```
private void Zeichne(Graphics dst)
{
    switch (Figur)
    {
        case Figuren.Linie:
```

```
            {
                    dst.DrawLine(p, p1, p2);
                    break;
            }
            case Figuren.Rechteck:
            {
                    dst.DrawRectangle(p, p1.X, p1.Y, p2.X - p1.X, p2.Y - p1.Y);
                    break;
            }
            case Figuren.Ellipse:
            {
                    dst.DrawEllipse(p, p1.X, p1.Y, p2.X - p1.X, p2.Y - p1.Y);
                    break;
            }
        }
    }
```

Auch nach einem Verdecken des Fensters soll die Grafik wiederhergestellt werden:

```
    private void Form1_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.DrawImage(bmp, 0, 0);
    }
```

Auswahl der Zeichenobjekte über den *ToolStrip*:

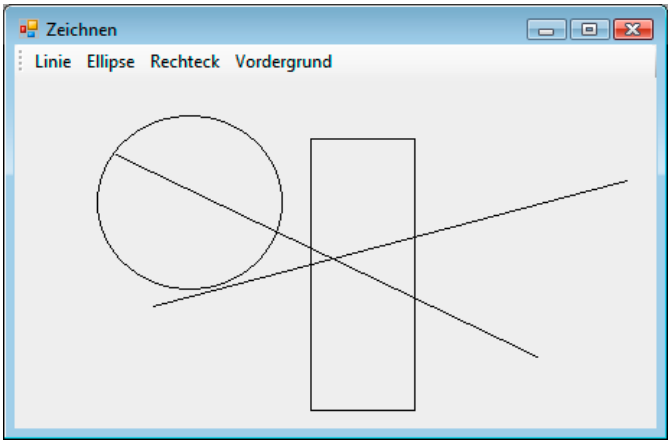
```
    private void toolStripButton1_Click(object sender, EventArgs e)
    {
        Figur = Figuren.Linie;
    }

    private void toolStripButton3_Click(object sender, EventArgs e)
    {
        Figur = Figuren.Ellipse;
    }

    private void toolStripButton2_Click(object sender, EventArgs e)
    {
        Figur = Figuren.Rechteck;
    }
```

Auswahl der Malfarbe:

```
    private void toolStripButton4_Click(object sender, EventArgs e)
    {
        colorDialog1.AllowFullOpen = true;
        if (colorDialog1.ShowDialog() == DialogResult.OK) p = new
Pen(colorDialog1.Color);
    }
}
```

Test

Nach dem Start können Sie Ihren künstlerischen Fähigkeiten freien Lauf lassen:



10 Druckausgabe

In diesem Kapitel wollen wir uns ausgiebig mit den Möglichkeiten beschäftigen, unter C# etwas aufs Papier zu bringen. Vier grundsätzliche Varianten bieten sich an:

- Drucken über die *PrintDocument*-Komponente,
- Drucken mithilfe von OLE-Automation,
- Drucken mit den Crystal Report-Komponenten,
- Drucken mit Reporting Services.

Hier beschränken wir uns auf die beiden ersten Möglichkeiten.

HINWEIS: Zunächst jedoch sollten Sie sich mit dem vorhergehenden Kapitel (Grafikprogrammierung) eingehend beschäftigt haben, da wir auf diesen Grundlagen aufbauen werden.

10.1 Einstieg und Übersicht

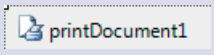
Bevor Sie mit viel Faktenwissen, endlosen Tabellen etc. gepeinigt werden, möchten wir Ihnen an einem Kurzbeispiel das Grundkonzept der Druckausgabe über *PrintDocument* demonstrieren.

10.1.1 Nichts geht über ein Beispiel

Beispiel 10.1: Druckausgabe eines 10 x 10 cm großen Rechtecks auf dem Standarddrucker

C#

Fügen Sie zunächst in Ihr Formular eine *PrintDocument*-Komponente ein:



Ergänzen Sie dann das *PrintPage*-Ereignis um folgende Zeilen:

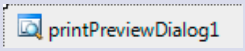
```
private void printDocument1_PrintPage(object sender,
                                      System.Drawing.Printing.PrintPageEventArgs e)
{
    e.Graphics.PageUnit = GraphicsUnit.Millimeter;
    e.Graphics.FillRectangle(new SolidBrush(Color.Blue), 30, 30, 100, 100);
 }
```

Fügen Sie nun noch einen *Button* ein, mit dem Sie die *Print*-Methode von *PrintDocument1* aufrufen:

```
private void button1_Click(object sender, EventArgs e)
{
     printDocument1.Print();
}
```

Das war es auch schon, nach dem Klick auf den Button dürfte sich Ihr Drucker in Bewegung setzen. Doch was ist der Vorteil einer derartigen ereignisorientierten Programmierung beim Drucken? Die Antwort finden Sie, wenn Sie statt der Druckausgabe zunächst eine Druckvorschau am Bildschirm realisieren möchten.

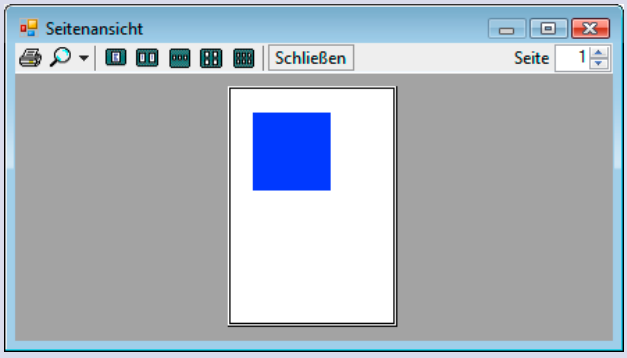
Fügen Sie einfach eine *PrintPreviewDialog*-Komponente in das Formular ein und verknüpfen Sie diese über die Eigenschaft *Document* mit der bereits vorhandenen *PrintDocument*-Komponente.



Der folgende Aufruf zeigt Ihnen bereits die Druckvorschau mit dem Rechteck an:

```
private void button2_Click(object sender, EventArgs e)
{
    printPreviewDialog1.ShowDialog();
}
```

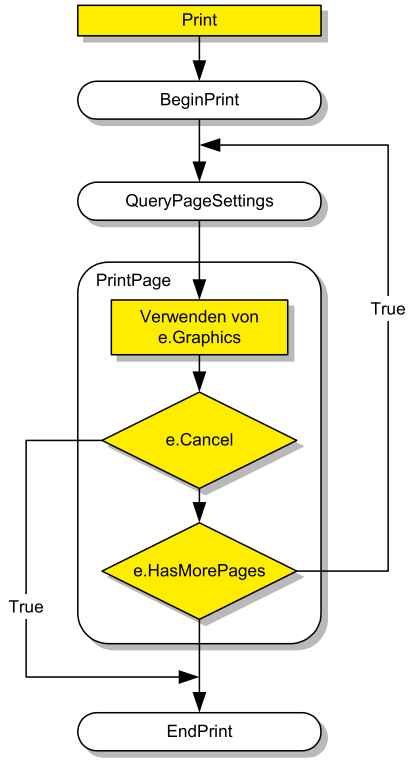
Das erzeugte Druckvorschaufenster:



HINWEIS: Eine Trennung beim Ausgabemedium (Papier, Druckvorschau am Bildschirm) gibt es nicht mehr, Sie entwickeln lediglich **eine** Ausgabelogik im *PrintPage*-Ereignis. Alle Ausgaben erfolgen systemneutral über ein dort bereitgestelltes *Graphics*-Objekt.

10.1.2 Programmiermodell

Wie Sie bereits dem vorhergehenden Beispiel entnehmen konnten, handelt es sich um ein ereignisorientiertes Programmiermodell. Die folgende Skizze soll Ihnen das Grundprinzip noch einmal verdeutlichen:



Mit dem Aufruf der *Print*-Methode wird zunächst das *BeginPrint*-Ereignis von *PrintDocument* ausgelöst. Hier bietet sich Ihnen die Möglichkeit, diverse Einstellungen einmalig zu konfigurieren. Nachfolgend wird das Ereignis *QueryPageSettings* vor dem Druck jeder Seite aufgerufen. Darauf folgt das wohl wichtigste Ereignis: *PrintPage*. Über den Parameter *e* erhalten Sie Zugriff auf das *Graphics*-Objekt des Druckers. Weiterhin legen Sie hier fest, ob weitere Seiten gedruckt werden sollen (*e.HasMorePages*) oder ob der Druck abgebrochen (*e.Cancel*) werden soll. Steht der Druck weiterer Seiten an, wird die Ereigniskette, wie oben abgebildet, erneut durchlaufen.

Damit wird auch klar, dass Sie selbst dafür verantwortlich sind, welche Seite zu welchem Zeitpunkt gedruckt werden soll. Insbesondere im Zusammenhang mit den Druckersetup-Dialogen werden wir noch einigen Aufwand treiben müssen, aber das sind Sie ja bereits nicht anders gewohnt.

10.1.3 Kurzübersicht der Objekte

Folgende Komponenten stehen Ihnen im Zusammenhang mit der Druckausgabe in Windows Forms-Anwendungen zur Verfügung[1]:

Komponente	Beschreibung
PrintDocument	Der Dreh- und Angelpunkt der Druckausgabe. Über dieses Objekt bestimmen Sie den gewünschten Drucker, die Papierausrichtung, die Auflösung, die zu druckenden Seiten usw. Über das *PrintPage*-Ereignis erhalten Sie Zugriff auf das *Graphics*-Objekt des Druckers. Weiterhin steuern Sie hier den Druckverlauf (Anzahl der Seiten, Seitenauswahl, Abbruch). Mehr zu dieser Komponente finden Sie in den beiden folgenden Abschnitten.
PrintDialog	Der Windows-Standarddialog zur Auswahl eines Druckers sowie der wichtigsten Druckparameter (Seiten, Exemplare, Auflösung)
PageSetupDialog	Der Windows-Standarddialog zur Konfiguration der Druckausgabe (Seitenausrichtung, Papierausrichtung, Seitenränder)
PrintPreviewDialog	Eine komplette Druckvorschau, mit Navigationstasten, Zoom etc.
PrintPreviewControl	Eine Alternative für den *PrintPreviewDialog*. Bei dieser Komponente ist lediglich der Preview-Bereich vorhanden, für die Ansteuerung und Konfiguration sind Sie selbst verantwortlich.

Alle Komponenten können über die *Document*-Eigenschaft mit der *PrintDocument*-Komponente verknüpft werden. Sie müssen also die Parameter nicht „von Hand“ übergeben.

10.2 Auswerten der Druckereinstellungen

In den folgenden Abschnitten wollen wir versuchen, Ihnen die „Vorzüge“ der relativ unübersichtlichen Objektstruktur zu ersparen. Aus diesem Grund werden wir auf eine Auflistung von Eigenschaften und Methoden für die einzelnen Objekte verzichten, stattdessen stellen wir die zu lösende Aufgabe in den Vordergrund.

[1] Auf das *Chart*-Control gehen wir im Rahmen des Praxisbeispiels in Abschnitt 32.6.2 gesondert ein.

10.2.1 Die vorhandenen Drucker

Einen Überblick, welche Drucker auf dem aktuellen System installiert sind, können Sie sich über die Collection *InstalledPrinters* verschaffen.

Beispiel 10.2: Ausgabe aller Druckernamen in einer ComboBox und Markieren des aktiven Druckers

C#

```
...
using System.Drawing.Printing;
...
    private void Form1_Load(object sender, EventArgs e)
    {
       PrintDocument doc = new PrintDocument();
```

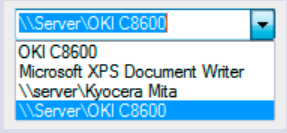
Füllen der *ComboBox*:

```
       foreach (String Printername in PrinterSettings.InstalledPrinters)
           comboBox1.Items.Add(Printername);
```

Auswahl des aktiven Druckers:

```
       comboBox1.Text = doc.PrinterSettings.PrinterName;
    }
```

Ergebnis



10.2.2 Der Standarddrucker

Möchten Sie überprüfen, ob der aktuell gewählte Drucker gleichzeitig auch der System-Standarddrucker ist, können Sie dies mithilfe der Eigenschaft *IsDefaultPrinter* realisieren.

Beispiel 10.3: Test auf Standarddrucker

C#

```
if (printDocument1.PrinterSettings.IsDefaultPrinter)
   MessageBox.Show("Standarddrucker");
```

Ergebnis

Den Standarddrucker erkennen Sie in der Systemsteuerung an einem kleinen Häkchen, auch wenn dieser nicht verfügbar ist:

10.2.3 Verfügbare Papierformate/Seitenabmessungen

Geht es um die Abfrage, welche Papierarten der Drucker unterstützt, können Sie einen Blick auf die *PaperSizes*-Collection werfen. Diese gibt Ihnen nicht nur Auskunft über die Blattgröße (*Height, Width*), sondern auch über die Blattbezeichnung (*PaperName*) und den Typ (*Kind*).

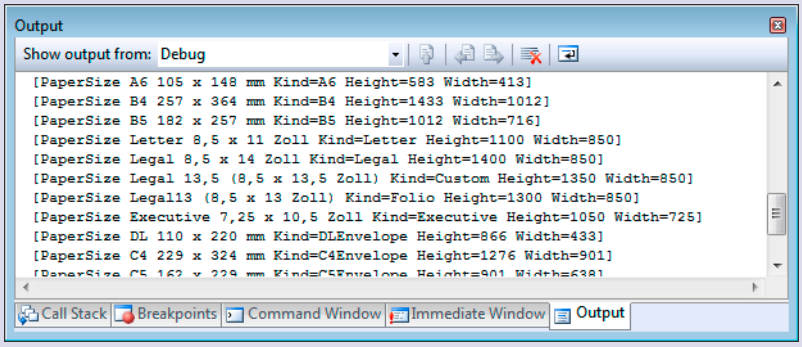
Beispiel 10.4: Anzeige aller Papierformate im Ausgabefenster

C#

```
using System.Drawing.Printing;
using System.Diagnostics;
...
    private void button1_Click(object sender, EventArgs e)
    {
        foreach (PaperSize ps in printDocument1.PrinterSettings.PaperSizes)
            Debug.WriteLine(ps);
    }
```

Ergebnis

Die Anzeige im Ausgabefenster:



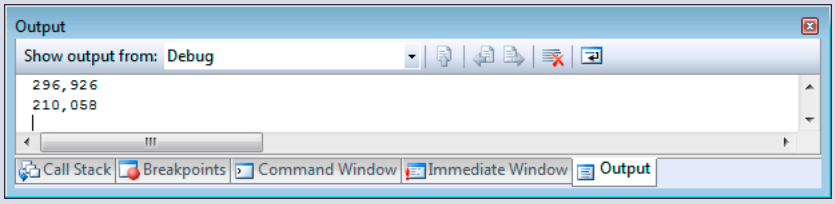
HINWEIS: Die Blattabmessungen werden in 1/100 Zoll zurückgegeben! Der Umrechnungsfaktor in Millimetern ist 0,254.

Beispiel 10.5: Anzeige der aktuellen Blattabmessungen in Millimetern

C#

```
Debug.WriteLine(printDocument1.PrinterSettings.DefaultPageSettings.PaperSize.
Height*0.254);
Debug.WriteLine(printDocument1.PrinterSettings.DefaultPageSettings.PaperSize.
Width*0.254);
```

Ergebnis



Gleichzeitig steht Ihnen mit *System.Drawing.Printing.PaperKind* eine Aufzählung der Standardpapierformate zur Verfügung (Auszug):

Element	Beschreibung
A2	A2 (420 x 594 mm)
A3	A3 (297 x 420 mm)
A3Extra	A3 Extra (322 x 445 mm)
A3ExtraTransverse	A3 Extra quer (322 x 445 mm)
A3Rotated	A3 gedreht (420 x 297 mm)
A3Transverse	A3 quer (297 x 420 mm)
A4	A4 (210 x 297 mm)

10.2.4 Der eigentliche Druckbereich

Leider druckt nicht jeder Drucker bis zu den Blatträndern. Aus diesem Grund ist es wichtig, den eigentliche Druckbereich und insbesondere den Offset des Druckbereichs zu bestimmen. Verwenden Sie dazu die Eigenschaften *HardMarginX*, *HardMarginY* sowie *PrintableArea*.

HINWEIS: Vergessen Sie in diesem Zusammenhang die Eigenschaft *Margins* ganz schnell wieder, es handelt sich lediglich um theoretische Seitenränder, die Sie selbst definieren können.

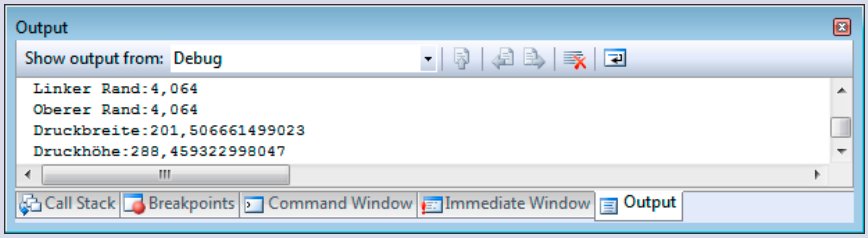
Beispiel 10.6: Anzeige der physikalischen Blattränder

C#

```
private void Form1_Load(object sender, EventArgs e)
{
   PrintDocument pd = new PrintDocument();
   Debug.WriteLine("Linker Rand:" + pd.DefaultPageSettings.HardMarginX * 0.254);
   Debug.WriteLine("Oberer Rand:" + pd.DefaultPageSettings.HardMarginY * 0.254);
   Debug.WriteLine("Druckbreite:" + pd.DefaultPageSettings.PrintableArea.Width *
0.254);
   Debug.WriteLine("Druckhöhe:" + pd.DefaultPageSettings.PrintableArea.Height *
0.254);
}
```

Ergebnis

Die Anzeige im Ausgabefenster:



10.2.5 Die Seitenausrichtung ermitteln

Die aktuelle Blatt- bzw. Seitenausrichtung können Sie über die Eigenschaft *Landscape* abfragen.

Beispiel 10.7: Abfrage der Seitenausrichtung

C#

```
using System.Drawing.Printing;
...
if (printDocument1.PrinterSettings.DefaultPageSettings.Landscape)
{  ...  };
```

10.2.6 Ermitteln der Farbfähigkeit

Ob Ihr aktuell gewählter Drucker auch in der Lage ist, mehr als nur Schwarz zu Papier zu bringen, lässt sich mit der Eigenschaft *SupportsColor* ermitteln.

Beispiel 10.8: Test auf Farbunterstützung

C#

```
using System.Drawing.Printing;
...
if (printDocument1.PrinterSettings.SupportsColor)
{  ...  };
```

10.2.7 Die Druckauflösung abfragen

Möchten Sie sich über die physikalische Druckauflösung des aktiven Druckers informieren, sollten Sie sich mit der Eigenschaft *PrinterResolution* näher beschäftigen.

Beispiel 10.9: Die horizontale Druckauflösung (readonly)

C#

```
Debug.WriteLine(printDocument1.DefaultPageSettings.PrinterResolution.X);
```

Beispiel 10.10: Die vertikale Druckauflösung (readonly)

C#

```
ebug.WriteLine(printDocument1.DefaultPageSettings.PrinterResolution.Y);
```

HINWEIS: Die Rückgabewerte entsprechen Punkten pro Zoll (Dots per Inch: dpi).

Alternativ können Sie über die *Kind*-Eigenschaft einen der folgenden Werte abrufen:

Kind	Beschreibung
Custom	Benutzerdefinierte Auflösung
Draft	Auflösung in Entwurfsqualität
High	Hohe Auflösung
Low	Niedrige Auflösung
Medium	Mittlere Auflösung

Beispiel 10.11: Ausgabe der Druckauflösung

C#

```
Debug.WriteLine(printDocument1.DefaultPageSettings.PrinterResolution.Kind);
```

10.2.8 Ist beidseitiger Druck möglich?

Ob der Drucker duplexfähig ist, das heißt, ob er beidseitig drucken kann, ermitteln Sie über die Eigenschaft *CanDuplex*.

Beispiel 10.12: Duplexfähigkeit bestimmen

C#

```
using System.Drawing.Printing;
...
if (PrintDocument1.PrinterSettings.CanDuplex)
{ ... }
```

10.2.9 Einen „Informationsgerätekontext“ erzeugen

Wer bisher mit GDI-Funktionen gearbeitet hat, dem wird auch der Begriff „Informationsgerätekontext“ nicht unbekannt sein. Der Hintergrund: Bei einem Drucker wird für die Abfrage von Gerätemerkmalen (Auflösung, Seitenränder etc.) häufig ein DC benötigt, der zum Beispiel im Zusammenhang mit der Funktion *GetDeviceCaps* genutzt wird. Dieses DC ist nur für die **Abfrage** von Werten vorgesehen.

Den DC selbst erhalten Sie nur über einen kleinen Umweg: Mithilfe der Methode *CreateMeasurementGraphics* erzeugen Sie zunächst ein *Graphics*-Objekt und dieses stellt bekanntlich die Methode *GetDC* zur Verfügung.

Beispiel 10.13: Wir testen die Grafikfähigkeit des aktuellen Druckers.[2]

C#

```
using System.Drawing.Printing;
using System.Runtime.InteropServices;

public partial class Form1 : Form
{
```

Die Konstanten für die möglichen Rückgabewerte der API-Funktion:

```
    private const int TECHNOLOGY = 2;
    private const int DT_PLOTTER = 0;
    private const int DT_RASPRINTER = 2;
    private const int DT_CHARSTREAM = 4;
    private const int DT_METAFILE = 5;
```

Einbinden der API-Funktion:

```
    [DllImport("gdi32.dll")]
    private static extern int GetDeviceCaps(IntPtr hdc, int nIndex);
```

[2] Nicht jeder Drucker muss auch voll grafikfähig sein, hier müssen Sie teilweise Einschränkungen erwarten.

```
    private void Form1_Load(object sender, EventArgs e)
    {
```

Informationsgerätekontext erzeugen:

```
        PrintDocument pd = new PrintDocument();
        Graphics g = pd.PrinterSettings.CreateMeasurementGraphics();
        IntPtr dc = g.GetHdc();
```

Abfrage der Grafikfähigkeit und Auswertung:

```
        switch (GetDeviceCaps(dc, TECHNOLOGY))
        {
            case DT_PLOTTER:
                label1.Text = "Plotter"; break;
            case DT_CHARSTREAM:
                label1.Text = "Zeichen"; break;
            case DT_METAFILE:
                    label1.Text = "Metafile"; break;
            case DT_RASPRINTER:
                    label1.Text = "Rasterdrucker"; break;
        }
        g.ReleaseHdc(dc);
    }
}
```

Installieren Sie ruhig einmal den „Generic/Text Only"-Drucker als Standarddrucker und lassen Sie dann das Programm laufen. Dieser Druckertreiber kann nur Zeichen verarbeiten, keine Grafiken.

HINWEIS: Vergessen Sie nicht, den DC wieder freizugeben. Dies muss innerhalb der aktuellen Ereignisroutine geschehen. Sie können den Wert **nicht** in einer globalen Variablen speichern!

10.2.10 Abfragen von Werten während des Drucks

Statt wie in den vorhergehenden Beispielen mit der *PrintDocument*-Komponente, nutzen Sie besser den im *PrintPage*-Ereignis angebotenen Parameter *e*. Über diesen erhalten Sie Zugriff auf die gewünschten Eigenschaften.

Beispiel 10.14: Papiergröße während des Drucks bestimmen

C#

```
private void printDocument1_PrintPage(object sender, PrintPageEventArgs e)
{
   Debug.WriteLine(e.PageSettings.PaperSize);
}
```

10.3 Festlegen von Druckereinstellungen

Nachdem wir im vorhergehenden Abschnitt recht passiv mit den Druckeroptionen umgegangen sind und uns auf das reine Auslesen beschränkt haben, wollen wir uns im Weiteren um das Konfigurieren des Druckers kümmern.

10.3.1 Einen Drucker auswählen

Der wohl erste Schritt, wenn mehr als ein Drucker zur Verfügung steht, ist die Auswahl des Druckers. Zwei Varianten bieten sich an:

- Verwendung der Eigenschaft *PrinterName*,
- Verwendung der *PrintDialog*-Komponente (siehe Abschnitt 10.4).

HINWEIS: Nach dem Setzen der Eigenschaft bzw. vor dem endgültigen Drucken sollten Sie mit der Eigenschaft *IsValid* überprüfen, ob die Konfiguration auch realisierbar ist.

Beispiel 10.15: Setzen der *PrinterName*-Eigenschaft und nachfolgende Prüfung mit *IsValid*

C#

```
printDocument1.PrinterSettings.PrinterName = comboBox1.Text;
if (printDocument1.PrinterSettings.IsValid)
  printPreviewDialog1.ShowDialog();
```

Beispiel 10.16: Verwendung des *QueryPageSettings*-Ereignisses zur Auswahl eines Druckers

C#

```
private void printDocument1_QueryPageSettings(object sender,
QueryPageSettingsEventArgs e)
 {
      e.PageSettings.PrinterSettings.PrinterName = "FRITZFax Drucker";
 }
```

10.3.2 Drucken in Millimetern

Sie werden hoffentlich nicht auf die Idee kommen, Zeichnungen in Pixeln auf dem Drucker auszugeben. Je nach Modell ist sonst Ihre Grafik mikroskopisch klein oder riesengroß. Bleibt die Frage, wie Sie die Maßeinheit auf Millimeter umstellen können. Die Lösung ist schnell gefunden, über die Eigenschaft *PageUnit* können Sie eine der folgenden Maßeinheiten auswählen:

Konstante	Beschreibung
Display	Eine Einheit entspricht 1/75 Zoll.
Document	Eine Einheit entspricht 1/300 Zoll.
Inch	Eine Einheit entspricht 1 Zoll.
Millimeter	Eine Einheit entspricht einem Millimeter.
Pixel	Eine Einheit entspricht einem Gerätepixel.
Point	Eine Einheit entspricht 1/72 Zoll (Point).

Beispiel 10.17: Setzen der Maßeinheit im *PrintPage*-Ereignis

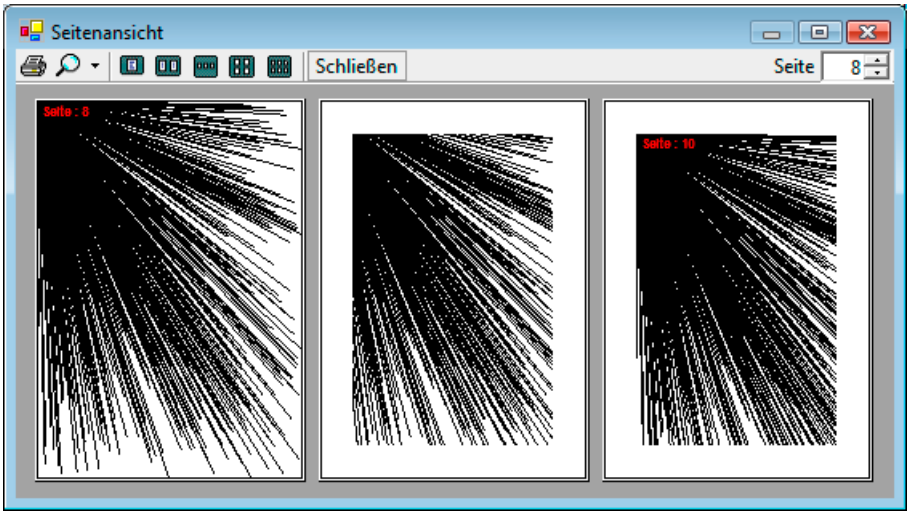
C#

```
private void printDocument1_PrintPage(object sender, PrintPageEventArgs e)
{
   e.Graphics.PageUnit = GraphicsUnit.Millimeter;
   e.Graphics.DrawLine(new Pen(Color.Black, 10), 50, 100, 150, 200);
 ...
}
```

10.3.3 Festlegen der Seitenränder

Tja, welche Ränder meinen Sie denn? Geht es um Seitenränder wie zum Beispiel in Microsoft Word, nutzen Sie die Eigenschaft *Margins*. Allerdings bedeutet das Festlegen per Code oder mithilfe der Dialogbox *PageSetupDialog* noch lange nicht, dass diese Ränder auch zwingend eingehalten werden. Dafür sind Sie im *PrintPage*-Ereignis selbst verantwortlich.

Die folgende Abbildung soll Ihnen die Problematik verdeutlichen. In jedem der drei Fälle werden, beginnend mit der Koordinate 0,0 (linke obere Ecke), Zufallslinien gezeichnet, die maximal die Abmessungen des Blatts erreichen.



Beispiel 10.18: Variante 1

C#

Variante 1 zeigt deutlich, dass die eingestellten Seitenränder (100,100,100,100) vollkommen ignoriert werden:

```
Graphics g = e.Graphics;
g.PageUnit = GraphicsUnit.Display;
for (i = 0; i <= 500; i++)
     g.DrawLine(p, 0, 0, Rnd.Next(e.PageBounds.Width),
                         Rnd.Next(e.PageBounds.Height));
```

Beispiel 10.19: Variante 2

C#

Diese Variante berücksichtigt bereits die eingestellten Seitenränder durch die Verwendung eines Clipping-Bereichs:

```
Graphics g = e.Graphics;
g.PageUnit = GraphicsUnit.Display;
g.SetClip(e.MarginBounds);
for (i = 0; i <=500; i++)
      g.DrawLine(p, 0, 0, Rnd.Next(e.PageBounds.Width),
                 Rnd.Next(e.PageBounds.Height));
```

Beispiel 10.20: Variante 3

C#

Wir bringen auch den Koordinatenursprung an die richtige Position:

```
Graphics g = e.Graphics;
g.PageUnit = GraphicsUnit.Display;
g.SetClip(e.MarginBounds);
g.TranslateTransform(e.MarginBounds.Left, e.MarginBounds.Top);
for (i = 0; i <= 500; i++)
       g.DrawLine(p, 0, 0, Rnd.Next(e.PageBounds.Width),
                  Rnd.Next(e.PageBounds.Height));
```

Damit brauchen Sie sich beim Zeichnen eigentlich nur noch um die Breite und Höhe des bedruckbaren Bereichs (*e.MarginBounds.Width* bzw. *e.MarginBounds.Height*) zu kümmern. Die linke obere Ecke ist bereits korrekt gesetzt.

HINWEIS: Die Eigenschaft *Margins* hebt natürlich keine physikalischen Grenzen auf. Wenn der Drucker einen entsprechenden Offset aufweist, müssen Sie diesen auch berücksichtigen (siehe Abschnitt 10.2.4).

10.3.4 Druckjobname

Was im Normalfall eher sekundär ist, kann in Netzwerkumgebungen bzw. Multiuser-Umgebungen für mehr Übersicht sorgen. Über die Eigenschaft *DocumentName* können Sie vor dem Drucken einen aussagekräftigen Druckjobnamen festlegen, der im Druckerspooler angezeigt wird.

Beispiel 10.21: Ändern des Druckjobnamens

C#

```
printDocument1.DocumentName = "Mein erster C#.NET-Druckversuch";
```

10.3.5 Anzahl der Kopien

Die Anzahl der Druckkopien kann zum einen mithilfe des Dialogs *PrintDialog*, zum anderen auch per Code festgelegt werden. Nutzen Sie die Eigenschaft *Copies*.

Beispiel 10.22: Drei Kopien

C#

```
printDocument1.PrinterSettings.Copies = 3;
```

HINWEIS: Mit *MaximumCopies* können Sie einen Maximalwert für die Druckdialoge vorgeben!

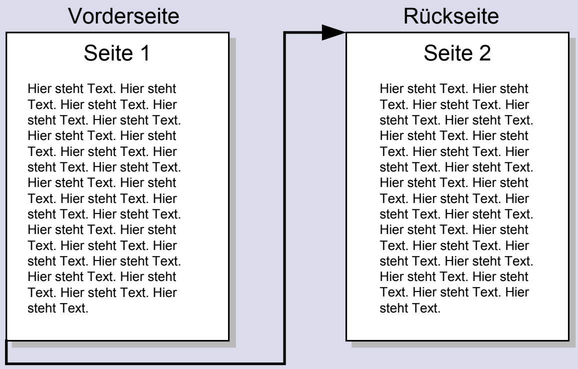
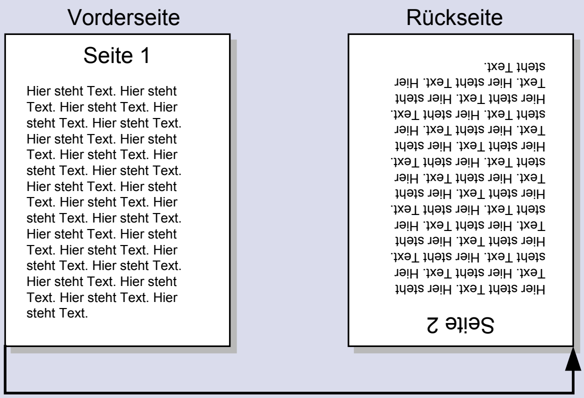
Beispiel 10.23: Maximal fünf Kopien zulassen

C#

```
printDocument1.PrinterSettings.MaximumCopies = 5;
```

10.3.6 Beidseitiger Druck

Geht es um das beidseitige Bedrucken von Papier, was aus ökologischer Sicht sicher sinnvoller ist, müssen Sie sich zunächst vergewissern, ob der Drucker auch über dieses Feature verfügt (siehe Abschnitt 10.2.8). Nachfolgend können Sie über die *Duplex*-Eigenschaft den gewünschten Wert einstellen.

Konstante	Beschreibung
Default	Die Standardeinstellungen des Druckers werden genutzt.
Simplex	Der „normale“ einseitige Druck
Horizontal	
Vertical	

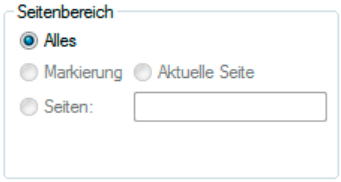
Beispiel 10.24: Einstellen der *Duplex*-Eigenschaft

C#

```
using System.Drawing.Printing;
...
printDocument1.PrinterSettings.Duplex = Duplex.Horizontal;
```

10.3.7 Seitenzahlen festlegen

Die Überschrift dürfte auf den ersten Blick etwas missverständlich klingen, da Sie doch selbst über den zu druckenden Inhalt entscheiden. Wenn Sie sich jedoch an den Druckerdialog erinnern, sind dort auch Optionen für die Seitenauswahl möglich:



Leider ist die Unterstützung dieser Option ein nicht ganz leicht verdaulicher Brocken.

Zunächst einmal unterscheiden Sie die vier gewählten Optionen (Alles, Markierung, Seiten, Aktuelle Seite) mithilfe der folgenden Konstanten über die *PrintRange*-Eigenschaft.

Konstante	Beschreibung
AllPages	Alle Seiten drucken
Selection	Die ausgewählten Seiten drucken (per Userauswahl)
SomePages	Die Seiten zwischen *FromPage* und *ToPage* sollen gedruckt werden.
CurrentPage	Die aktuelle Seite drucken (was die aktuelle Seite ist, bestimmt Ihr Programm)

Ein Beispiel zeigt die Auswertung der vier Varianten im Zusammenhang.

Beispiel 10.25: Berücksichtigung des vorgegebenen Druckbereichs

C#

```
using System.Drawing.Printing;

public partial class Form1 : Form
{
```

Eine Variable für die aktuell zu druckende Seite:

```
    int page;
```

Die aktuelle Seite bei Mehrfachauswahl:

```
    int selectedindex;
```

Die maximal druckbaren Seiten (Dokumentlänge):

```
    const int maxpages = 30;
```

Beim Programmstart füllen wir zunächst eine *ListBox* mit den möglichen Seitenzahlen (1...30):

```
    private void Form1_Load(object sender, EventArgs e)
    {
        for (int i = 0; i <= maxpages; i++)
                listBox1.Items.Add("Seite " + i.ToString());
    }
```

Druckerdialog anzeigen und im Erfolgsfall die Druckvorschau öffnen:

```
    private void button1_Click(object sender, EventArgs e)
    {
        if (printDialog1.ShowDialog() == DialogResult.OK)
                printPreviewDialog1.ShowDialog();
    }
```

Vorbereiten des „Druckvorgangs“:

```
    private void printDocument1_BeginPrint(object sender,

System.Drawing.Printing.PrintEventArgs e)
    {
        page = 1;
        selectedindex = 0;
```

Zur Sicherheit prüfen wir, ob auch mindestens eine Seite ausgewählt wurde (nur bei Seitenauswahl):

```
        switch (printDialog1.PrinterSettings.PrintRange)
        {
            case PrintRange.Selection:
                    if (listBox1.SelectedItems.Count == 0)
                        e.Cancel = true;
                break;
        }
    }
```

Der eigentliche Druckvorgang:

```
    private void printDocument1_PrintPage(object sender,

System.Drawing.Printing.PrintPageEventArgs e)
    {
        int printpage = 0;
```

Je nach Auswahl im Druckdialog müssen wir nun die aktuelle Seite bestimmen:

```
        switch (e.PageSettings.PrinterSettings.PrintRange)
        {
```

Es soll die aktuelle Seite gedruckt werden (der Wert wird per *NumericUpDown* bestimmt):

```
            case PrintRange.CurrentPage:
                    printpage = (int) numericUpDown1.Value;
                    break;
```

Es soll ein Seitenbereich gedruckt werden:

```
            case PrintRange.SomePages:
                    printpage = page + e.PageSettings.PrinterSettings.FromPage - 1;
                    break;
```

Es sollen alle Seiten gedruckt werden:

```
            case PrintRange.AllPages:
                    printpage = page;
                    break;
```

Eine Seitenauswahl (*ListBox*) soll gedruckt werden:

```
            case PrintRange.Selection:
                    printpage = listBox1.SelectedIndices[selectedindex];
                    selectedindex++;
                    break;
        }
```

Hier können Sie die Seite auswerten und die Drucklogik unterbringen:

```
        switch (printpage)
        {
            case 1:
                    break;
            case 2:
                    break;
            // ...
        }
```

Unser Beispiel zeigt stattdessen die aktuelle Seitenzahl an:

```
        e.Graphics.DrawString("Seite : " + printpage.ToString(),
               new Font("Arial", 20, FontStyle.Bold, GraphicsUnit.Millimeter),
                        Brushes.Black, 70, 50);
```

Eine Seite weiter:

```
        page++;
```

Je nach Auswahl im Druckerdialog bestimmen wir jetzt, ob es noch weitere Seiten gibt:

```
        switch (e.PageSettings.PrinterSettings.PrintRange)
        {
            case PrintRange.Selection:
                    e.HasMorePages = selectedindex <
 listBox1.SelectedIndices.Count;
                    break;
            case PrintRange.CurrentPage:
                    e.HasMorePages = false;
                    break;
            case PrintRange.SomePages:
                    e.HasMorePages = (printpage <
 e.PageSettings.PrinterSettings.ToPage);
                    break;
            case PrintRange.AllPages:
                    e.HasMorePages = (page <= maxpages);
                    break;
        }
    }
 }
```

HINWEIS: Über die Eigenschaften *MinimumPage* und *MaximumPage* können Sie maximale Grenzen für die Auswahl des Druckbereichs festlegen.

Beispiel 10.26: Druckbereich maximal von Seite 1 bis Seite 10

C#

```
printDocument1.PrinterSettings.MinimumPage = 1;
printDocument1.PrinterSettings.MaximumPage = 10;
```

10.3.8 Druckqualität verändern

Unter diesem Punkt verstehen wir zum einen die Einstellung der dpi-Zahl des Druckers, zum anderen die Optionen bei der Ausgabe von Grafiken (*Antialiasing*, *CompositingQuality*).

Beispiel 10.27: Setzen der Druckauflösung

C#

```
printDocument1.DefaultPageSettings.PrinterResolution =
          printDocument1.PrinterSettings.PrinterResolutions[2];
```

10.3.9 Ausgabemöglichkeiten des Chart-Controls nutzen

An dieser Stelle wollen wir uns einem Spezialfall zuwenden. Im Mittelpunkt stehen die *Chart*-Komponente und deren Möglichkeiten zur Druckausgabe. Diese sind zwar auf den ersten Blick recht überschaubar, allerdings dürften Sie damit auch alle wichtigen Anwendungsfälle problemlos abdecken. Unser Interesse gilt vor allem der *Printing*-Eigenschaft des *Chart*-Controls. Diese bündelt alle Aktivitäten rund um die Druckausgabe.

Folgende Methoden werden bereitgestellt:

Methode	Beschreibung
PageSetup	... zeigt den bekannten Page Setup-Dialog an. Mehr zur Verwendung dieses Dialogs finden Sie in Abschnitt 10.4.2.
Print	... druckt das vorliegende Diagramm. Übergeben Sie als Parameter *true*, wird der bekannte Druckdialog zur Druckerauswahl angezeigt (siehe Abschnitt 10.4.1).
PrintPaint	Ausgabe des Diagramms auf einem *Graphics*-Objekt (siehe Praxisbeispiel in Abschnitt 10.6.2).
PrintPreview	Statt der direkten Druckauswahl wird eine Druckvorschau angezeigt.

Neben obigen Methoden steht über die *Printing*-Eigenschaft auch ein *PrintDocument* zur Verfügung, über das Sie die Einstellungen des Druckers auslesen oder auch beeinflussen können, wie Sie es in den beiden vorhergehenden Abschnitten bereits im Detail gesehen haben.

Beispiel 10.28: Verwendung von *Printing.PrintDocument*

C#

Anzeige, ob es sich um den Standarddrucker handelt:

```
Text = chart1.Printing.PrintDocument.PrinterSettings.IsDefaultPrinter.ToString();
```

HINWEIS: Ein Praxisbeispiel für die Integration eines Diagramms in einen eigenen Report finden Sie in Abschnitt 10.6.2, Diagramme mit dem Chart-Control drucken.

10.4 Die Druckdialoge verwenden

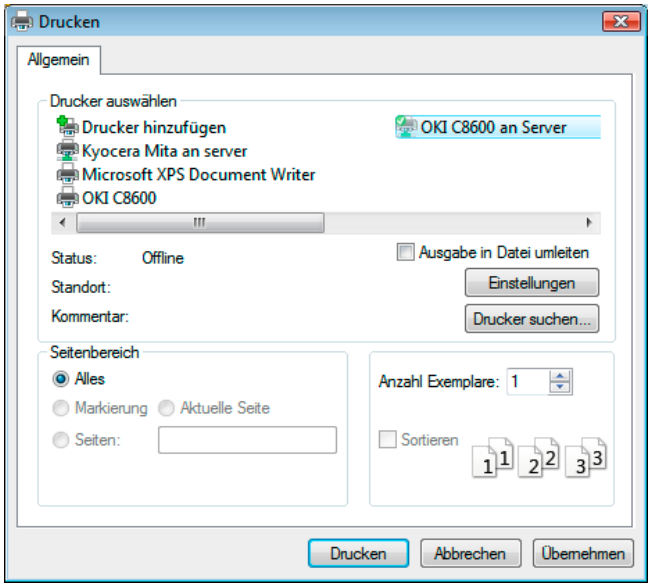
Im Folgenden wollen wir Ihnen kurz die drei wesentlichen Druckdialoge

- *PrintDialog*,
- *PageSetupDialog*,
- *PrintPreviewDialog*

und deren wichtigste Parameter im Zusammenspiel mit der Druckausgabe vorstellen.

10.4.1 PrintDialog

Der allgemein bekannte Standarddruckdialog wird mit der Komponente *PrintDialog* eingebunden.



Die Komponente selbst können Sie mittels *Document*-Eigenschaft direkt an ein *PrintDocument*-Control binden. Alle gewählten Parameter werden automatisch an *PrintDocument* übergeben.

Eigenschaft	Beschreibung
AllowPrintToFile	... aktiviert das Kontrollkästchen „Ausgabe in Datei“
AllowSelection	... aktiviert das Optionsfeld „Seiten von ... bis ...“
AllowSomePages	... aktiviert das Optionsfeld „Seiten“
ShowHelp	... aktiviert die Schaltfläche „Hilfe“
ShowNetwork	... aktiviert die Schaltfläche „Netzwerk“ (nur in der Theorie)
PrinterSettings	Über diese Eigenschaft können Sie Standardwerte vorgeben sowie die Einstellungen des Dialogfelds abfragen.
PrintToFile	... fragt den Wert des Kontrollkästchens „Ausgabe in Datei“ ab

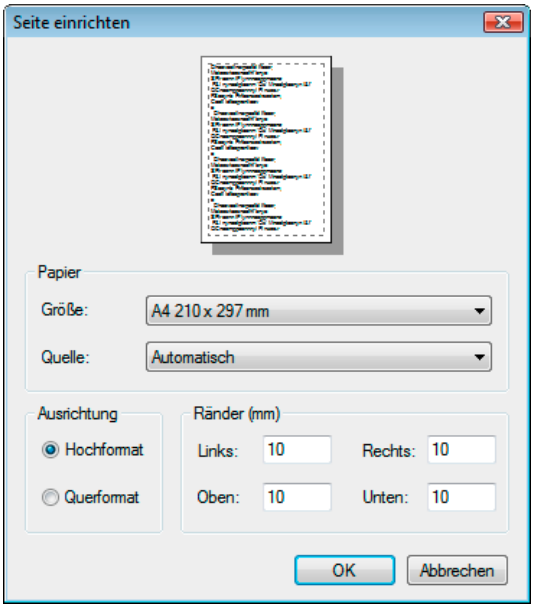
Beispiel 10.29: Anzeige des Dialogs und Abfrage des gewählten Druckers

C#

```
if (printDialog1.ShowDialog() = DialogResult.OK)
    MessageBox.Show(printDialog1.PrinterSettings.PrinterName, "Hinweis",
                    MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
```

10.4.2 PageSetupDialog

Aus vielen Programmen dürfte Ihnen der folgende Dialog bekannt sein, mit dem Sie einen Menüpunkt „Seite einrichten“ realisieren können.



Auch diese *PageSetupDialog*-Komponente können Sie mittels *Document*-Eigenschaft direkt an eine *PrintDocument*-Komponente binden, um die eingestellten Parameter automatisch zu übernehmen.

Eigenschaft	Beschreibung
AllowMargins	... aktiviert den Bereich „Ränder (mm)“
AllowOrientation	... aktiviert den Bereich „Orientierung“
AllowPaper	... aktiviert den Bereich „Papier“
AllowPrinter	... aktiviert die Schaltfläche „Drucker...“
ShowHelp	... aktiviert die Schaltfläche „Hilfe“
MinMargins	... legt die minimalen Werte für die Ränder fest
PageSettings *PrinterSettings*	... hier können Sie Standardwerte vorgeben bzw. die Werte abfragen.

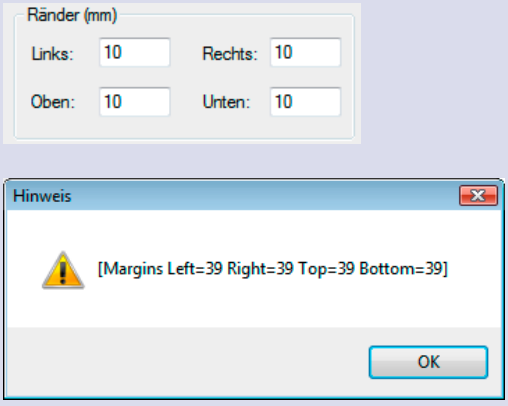
HINWEIS: Über das Ereignis *HelpRequest* können Sie auf den Button „Hilfe“ reagieren!

Beispiel 10.30: Aufruf der Dialogbox und Anzeige der neu gesetzten Ränder

C#

```
if (pageSetupDialog1.ShowDialog()== DialogResult.OK)
      MessageBox.Show(pageSetupDialog1.PageSettings.Margins.ToString(), "Hinweis",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
```

Ergebnis



Probleme mit den Rändern

Doch wo viel Licht, da ist auch Schatten, ein kleiner Bug hat sich in die Komponente eingeschlichen, der aber wahrscheinlich nur in den lokalisierten Varianten von Visual Studio auftritt:

HINWEIS: Die Werte der eingestellten Ränder stimmen nicht mit den Werten der Eigenschaft *Margins* überein (aus einem Zoll Vorgabewert werden in der Anzeige 10 Millimeter und aus diesen wiederum korrekte 0,39 Zoll). Eine fragwürdige Umrechnung.

Deshalb der folgende Workaround:

Rufen Sie **vor** dem Aufruf der Dialogbox jedes Mal die folgenden Anweisungen auf:

```
pageSetupDialog1.PageSettings.Margins.Left =
   (int) (pageSetupDialog1.PageSettings.Margins.Left * 2.54);
pageSetupDialog1.PageSettings.Margins.Top =
   (int) (pageSetupDialog1.PageSettings.Margins.Top * 2.54);
pageSetupDialog1.PageSettings.Margins.Right =
   (int) (pageSetupDialog1.PageSettings.Margins.Right * 2.54);
pageSetupDialog1.PageSettings.Margins.Bottom =
    (int) (pageSetupDialog1.PageSettings.Margins.Bottom * 2.54);
pageSetupDialog1.ShowDialog();
```

Nach dem Aufruf stehen Ihnen die Seitenränder wieder in der korrekten 1/100-Zoll-Angabe zur Verfügung.

10.4.3 PrintPreviewDialog

Im Grunde ist die Verwendung des *PrintPreviewDialog* recht simpel. Nach dem Einfügen der Komponente brauchen Sie diese lediglich über die *Documents*-Eigenschaft mit der *PrintDocument*-Komponente zu verknüpfen und die Methode *ShowDialog* aufzurufen.

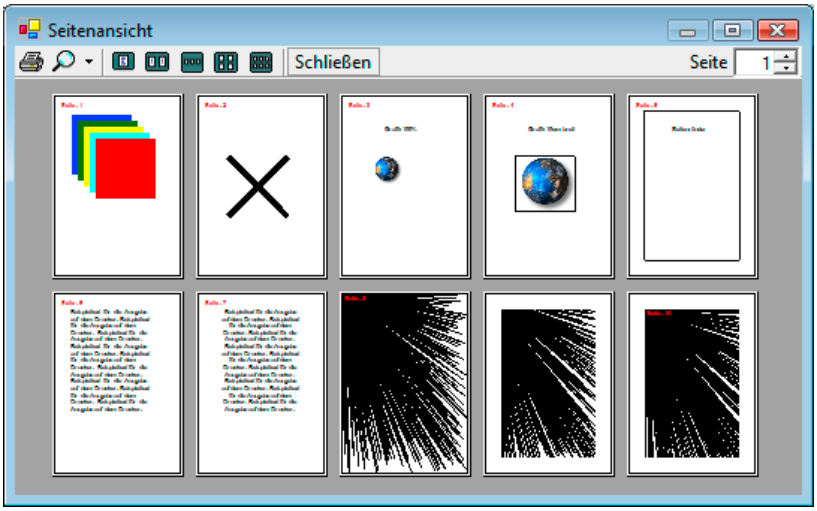
Bis auf die Eigenschaft *UseAntialias* (Verbessern der Anzeigequalität) können Sie kaum weitere Einstellungen vornehmen. Die Ausnahme stellt die Eigenschaft *PrintPreviewControl* dar, mit der Sie direkt das Aussehen und Verhalten der Vorschau beeinflussen können.

Beispiel 10.31: Gleichzeitige Anzeige von zehn Seiten

C#

```
printPreviewDialog1.PrintPreviewControl.Rows = 2;
printPreviewDialog1.PrintPreviewControl.Columns = 5;
printPreviewDialog1.ShowDialog();
```

HINWEIS: Mehr zur Konfiguration und Verwendung der *PrintPreviewControl*-Komponente finden Sie im folgenden Abschnitt 10.4.4.



Weiterhin dürfte die Eigenschaft *WindowState* in Zusammenhang mit der Anzeige der Dialogbox von Interesse sein. Hiermit steuern Sie die Art der Anzeige (Vollbild etc.).

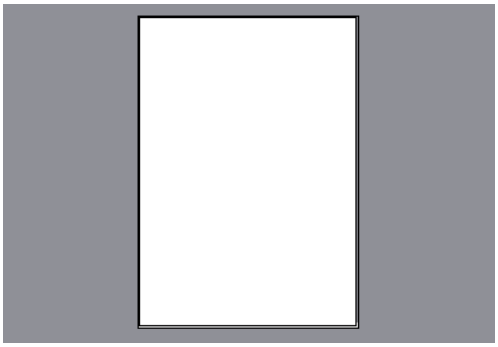
Beispiel 10.32: Vollbildanzeige aktivieren

C#

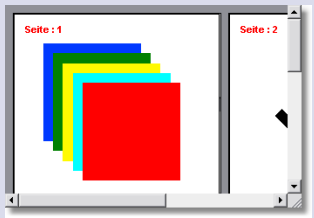
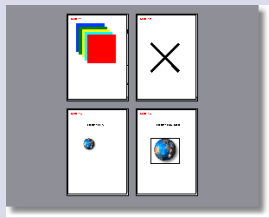
```
printPreviewDialog1.WindowState = FormWindowState.Maximized;
printPreviewDialog1.ShowDialog();
```

10.4.4 Ein eigenes Druckvorschaufenster realisieren

Wem die *PrintPreview*-Komponente zu wenig Eingriffsmöglichkeiten bietet, der kann sich mit der *PrintPreviewControl*-Komponente eine eigene Druckvorschau zusammenbauen.



Bis auf den reinen Druckvorschaubereich können Sie sich um alle Einstellungen und optischen Spielereien selbst kümmern. Die folgende Tabelle listet die wichtigsten Eigenschaften auf:

Eigenschaft	Beschreibung
AutoZoom	Ist der Wert auf *True* gesetzt, werden die Seiten so skaliert, dass die vorgegebene Anzahl von Seiten flächenfüllend dargestellt wird.
	AutoZoom = False / *AutoZoom = True*
	 
BackColor	... die Hintergrundfarbe für die Druckvorschau
Columns	... die Anzahl von Spalten, das heißt, wie viele Seiten nebeneinander dargestellt werden
Rows	... die Anzahl von Zeilen, das heißt, wie viele Seiten untereinander dargestellt werden
Document	... die Verknüpfung zum *PrintDocument*-Objekt
StartPage	... die Seitenzahl der linken oberen Seite. Durch Verändern dieses Werts können Sie die weiteren Seiten anzeigen.
Zoom	... legt explizit einen Zoomfaktor fest

Wie Sie die *PrintPreviewControl*-Komponente im Zusammenhang verwenden, zeigt Ihnen das Praxisbeispiel in Abschnitt 10.6.1 am Kapitelende.

10.5 Drucken mit OLE-Automation

Wir wollen versuchen, mithilfe der bekannten Office-Programme Druckausgaben zu realisieren. Schnell kommt der Verdacht auf, dass der Programmierer versucht, das Brett an der dünnsten Stelle anbohren zu wollen. Doch warum sollen nicht die Möglichkeiten von Office-Programmen genutzt werden, wenn doch häufig der Wunsch besteht, Reportausgaben nachträglich zu bearbeiten oder in umfangreichere Dokumentationen aufzunehmen? Nicht zuletzt bieten sich gerade die Office-Anwendungen an, wenn es um eine sinnvolle Archivierung von Dokumenten geht.

Unser Favorit ist, wie sollte es auch anders sein, Microsoft Word, ein Allround-Talent, was die Gestaltung von ansprechenden Druckausgaben angeht. Als zweite Variante bietet sich insbesondere bei der Verwendung von Desktop-Datenbanken Microsoft Access an. Hier kann der integrierte Report-Generator zeigen, was er kann.

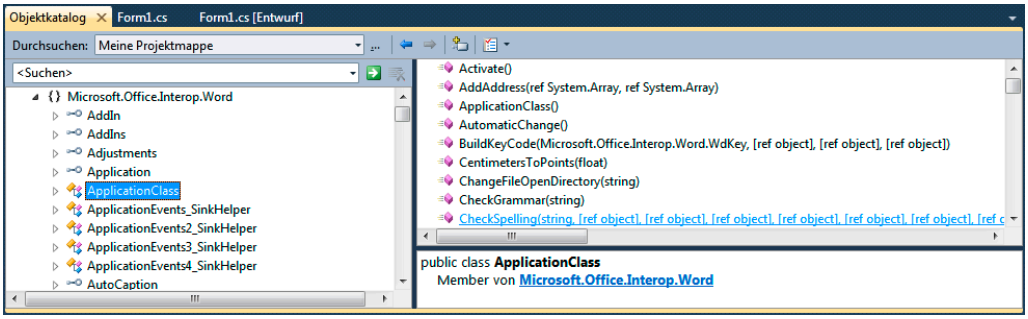
Für die im Weiteren vorgestellten Verfahren ist es sinnvoll, wenn wir kurz auf die grundlegenden Möglichkeiten und Funktionen der OLE-Automation eingehen.

HINWEIS: Ein wesentlicher Nachteil der im Folgenden beschriebenen Verfahrensweise soll natürlich nicht unerwähnt bleiben. Geben Sie Ihre Anwendungen an andere Anwender weiter, muss auf dem jeweiligen Rechner natürlich auch die OLE-Anwendung (Access oder Word) installiert sein.

10.5.1 Kurzeinstieg in die OLE-Automation

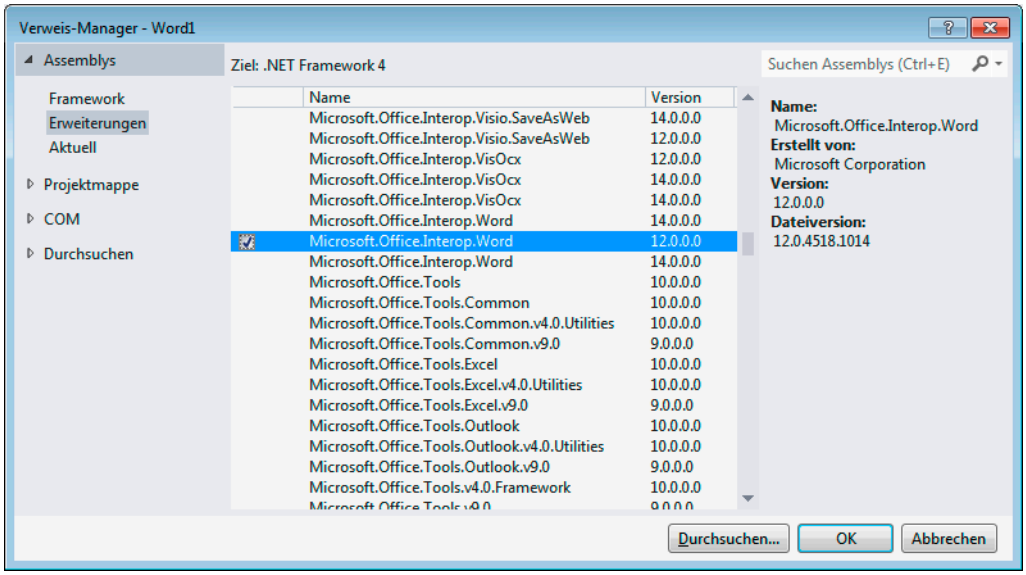
Über OLE-Automation lassen sich Objekte anderer Applikationen (z. B. Word oder Access) von Ihrem .NET-Programm quasi „fernsteuern“. Nach der Definition einer entsprechenden Objektvariablen können Sie auf Eigenschaften und Methoden dieser Objekte genauso zugreifen, als ob es sich um ein normales C#-Objekt handeln würde.

Wichtigstes Hilfsmittel für den OLE-Programmierer ist der in Visual Studio integrierte Objektkatalog (siehe folgende Abbildung).



Im Objektkatalog werden neben den Klassen alle Methoden, Eigenschaften, Ereignisse und Konstanten des jeweiligen Objekts angezeigt.

Welche Objekte angezeigt werden, hängt von den Verweisen ab, die Sie unter *Projekt/Verweise hinzufügen...* eingebunden haben:



Programmieren der OLE-Automation

Das Grundprinzip besteht darin, dass Sie in C# eine Instanz der gewünschten Klasse erzeugen. Mit diesem Objekt können Sie dann wie mit jedem anderen Objekt arbeiten.

HINWEIS: Wer bereits mit älteren C#-Versionen (< 4.0) gearbeitet hat, wird sich sicher noch an die grauenhafte Arbeit mit dem Erzeugen von Objekten und der Übergabe von optionalen Parametern erinnern. Vergessen Sie dies alles, seit C# 4.0 können Sie auf derartige Handstände verzichten.

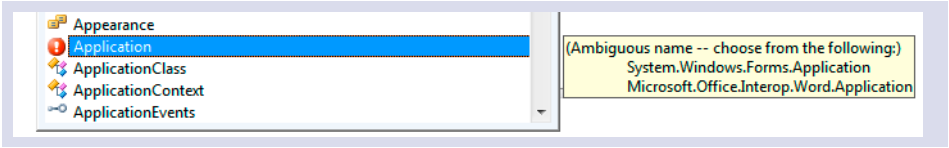
Voraussetzung für das Erstellen einer Instanz ist ein Verweis auf die entsprechende Klasse. Um neue Verweise zu erstellen, müssen Sie unter **Projekt | Verweise hinzufügen...**

- die gewünschte COM-Klassenbibliothek (z. B. *Microsoft Word 12.0 Object Library*) auswählen
- oder alternativ eine der vorhandenen PIAs (*Primary Interop Assemblies*) nutzen.

Im vorliegenden Fall nutzen wir die PIA *Microsoft.Office.Interop.Word* (siehe obige Abbildung).

HINWEIS: Binden Sie den neuen Namespace (*using*) direkt ein, kann es zu Überschneidungen der COM-Objektnamen mit den C#-Objekten kommen.

Beispiel 10.33: Hier ist nicht eindeutig, welches *Application* gemeint ist.



Besser Sie verwenden in diesem Fall einen Aliasnamen für die Assembly, wie es das folgende Beispiel zeigt:

Beispiel 10.34: Verwendung Aliasnamen

C#

Erstellen einer Objektvariablen *myWord* als Instanz des *Word.Application*-Objekts:

```
using Word = Microsoft.Office.Interop.Word;

namespace WindowsFormsApplication3
{
...
     var myWord = new Word.Application();
...
```

10.5.2 Drucken mit Microsoft Word

Verwenden Sie Word für die Druckausgabe, können Sie zwei verschiedene Varianten einsetzen:

- Sie entwerfen die komplette Seite mit Word und fügen an den relevanten Stellen sogenannte Platzhalter (Formularfelder) ein. Diese werden später aus dem C#-Programm heraus gezielt aufgerufen und mit neuen Inhalten gefüllt. Der Vorteil dieser Variante: Das Word-Dokument kann quasi wie eine Vorlage genutzt werden, der Aufwand ist minimal. Nachteil: Sie müssen die Datei zur Laufzeit in den Word-Editor laden.
- Der komplette Report bzw. das komplette Word-Dokument wird zur Laufzeit aus C# heraus generiert. Vorteil: Das Erstellen von Listen ist mit dieser Variante wesentlich einfacher als mit vorgefertigten Dokumenten. Nachteil: jede Menge Quellcode.

Der Nachteil der zweiten Variante kann jedoch schnell wieder wettgemacht werden, lassen Sie einfach Word für sich arbeiten.

Was ist gemeint? Die Antwort findet sich unter dem Word-Menüpunkt **Extras | Makro | Aufzeichnen**[3]. Öffnen Sie Word und rufen Sie den genannten Menübefehl auf. Alle weiteren Aktionen, die Sie durchführen (Text eingeben, formatieren, Tabellen erstellen etc.) werden durch den Makrorekorder aufgezeichnet. Sie müssen nur noch das aufgezeichnete Makro in Ihr C#-Programm einfügen und „geringfügig“ anpassen.

[3] Oder für Opfer der Versionen ab 2007: **Entwicklertools | Makro aufzeichnen** (eventuell müssen Sie die Registerkarte *Entwicklertools* zunächst über die Optionen einblenden).

Beispiel 10.35: Transformieren eines aufgezeichneten Word-Makros in C#

C#

Sie erstellen bei eingeschaltetem Makrorekorder ein neues Dokument und geben Sie eine 16 Punkt große Überschrift ein. Das Resultat ist folgender Bandwurm (Word-VBA):

```
Sub Makro1()
  Documents.Add Template:="E:\Programme\Ms\Vorlagen\Normal.dot", NewTemplate
:=False
    With Selection.Font
        .Name = "Times New Roman"
        .Size = 16
        .Bold = False
        .Italic = False
        .Underline = wdUnderlineNone
        .StrikeThrough = False
        .DoubleStrikeThrough = False
        .Outline = False
        .Emboss = False
        .Shadow = False
        .Hidden = False
        .SmallCaps = False
        .AllCaps = False
        .ColorIndex = wdAuto
        .Engrave = False
        .Superscript = False
        .Subscript = False
        .Spacing = 0
        .Scaling = 100
        .Position = 0
        .Kerning = 0
        .Animation = wdAnimationNone
    End With
    Selection.TypeText Text:="Überschrift"
    Selection.HomeKey Unit:=wdLine
End Sub
```

Aus diesem Quellcode-Haufen filtern wir uns erst einmal die relevanten Daten heraus:

```
Sub Makro1()
  Documents.Add
  Selection.Font.Size = 16
  Selection.TypeText Text:="Überschrift"
End Sub
```

Das sieht doch schon viel freundlicher aus, das Resultat beider Makros ist dasselbe. Kopieren Sie nun den Quellcode in Ihre C#-Anwendung. Erster Schritt ist jetzt das Erzeugen einer Word-Instanz bzw. einer *Word.Application*-Instanz:

```
using Word = Microsoft.Office.Interop.Word;
...
private void button1_Click(object sender, EventArgs e)
{
   var myWord = new Word.Application();
```

Word muss extra einblendet werden:

```
    myWord.Visible = true;
```

Und hier kommen die eigentlichen Anweisungen:

```
    myWord.Documents.Add();
    myWord.Selection.Font.Size = 16;
    myWord.Selection.TypeText("Überschrift");
}
```

Der ehemalige Makrocode ist fett hervorgehoben.

HINWEIS: Das obige C#-Listing ist erst ab C# 4.0 lauffähig. In älteren Versionen müssen Sie sich damit abfinden, dass Sie immer alle Parameter an die jeweiligen Methoden übergeben müssen. Dies erfordert zum Teil einen beträchtlichen Mehraufwand und ein intensives Studium der VBA-Hilfe.

Beispiel 10.36: Umsetzung ab C# 2008

C#

```
using Microsoft.Office.Interop.Word;
...
      ApplicationClass myWord = new ApplicationClass();
      object n = System.Reflection.Missing.Value;
      myWord.Visible = true;
      myWord.Documents.Add(ref n, ref n, ref n, ref n);
      myWord.Selection.Font.Size = 16;
      myWord.Selection.TypeText("Überschrift");
```

HINWEIS: Beachten Sie, dass in diesem Fall alle Methodenparameter mittels Referenz (*ref*) zu übergeben sind.

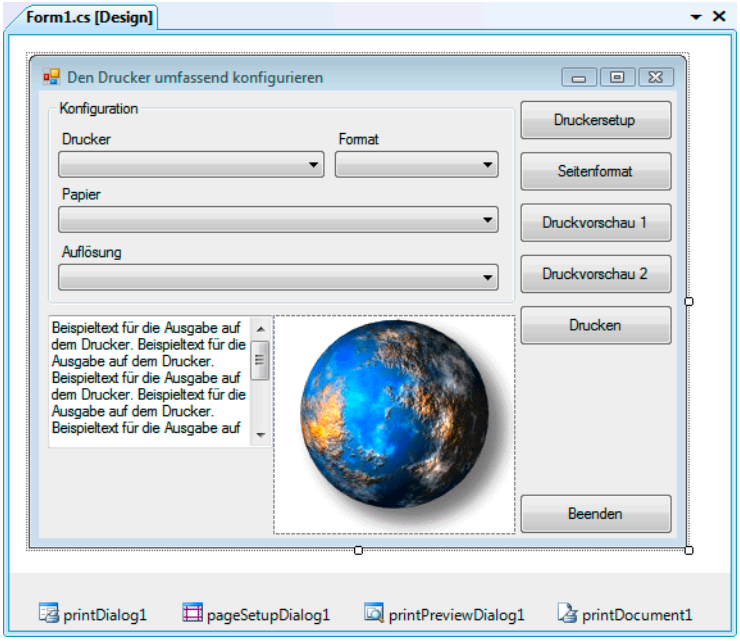
10.6 Praxisbeispiele

10.6.1 Den Drucker umfassend konfigurieren

Das Ziel dieses Beispiels ist eine umfassende Darstellung des Zusammenspiels der einzelnen Druckerkomponenten sowie deren Konfiguration per Code bzw. per Dialogbox. Insgesamt zehn Beispielseiten verdeutlichen die verschiedenen Möglichkeiten der Gestaltung des Druckbilds.

Oberfläche (Hauptformular Form1)

Entwerfen Sie eine Oberfläche entsprechend folgender Abbildung:

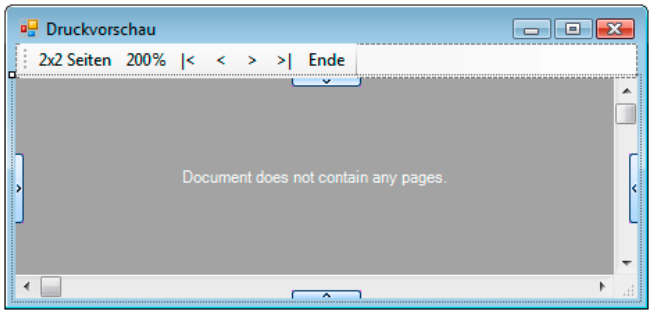


Verknüpfen Sie die vier nicht sichtbaren Komponenten (siehe unterer Bildrand) über die *Documents*-Eigenschaft mit dem *printDocument1*.

Sowohl die *TextBox* als auch die *PictureBox* dienen uns lediglich als Container für einen zu druckenden Text bzw. eine zu druckende Grafik.

Oberfläche (Druckvorschau Form2)

Mit der folgenden Oberfläche wollen wir keinen Schönheitspreis gewinnen, es geht lediglich um die Darstellung des Grundprinzips. Welche Komponenten Sie für die Oberflächengestaltung nutzen, bleibt Ihrer Fantasie überlassen. Wichtig ist vor allem das *PrintPreview*-Control:



Quelltext (Form1)

```
using System.Drawing.Printing;

public partial class Form1 : Form
{
```

Eine globale Variable erleichtert uns die Anzeige bzw. den Druck der richtigen Seite:

```
    private int page;
```

Die folgende Routine aktualisiert die *ComboBox*en nach Änderungen über die Standarddialoge:

```
    private void aktualisieren()
    {
```

Der aktuelle Drucker:

```
        comboBox1.Text = printDocument1.PrinterSettings.PrinterName;
```

Die verschiedenen Papierformate:

```
        comboBox2.Items.Clear();
        foreach (PaperSize ps in printDocument1.PrinterSettings.PaperSizes)
           comboBox2.Items.Add(ps);
        comboBox2.Text = printDocument1.DefaultPageSettings.PaperSize.ToString();
```

Die Seitenausrichtung:

```
        if (printDocument1.DefaultPageSettings.Landscape)
           comboBox3.SelectedIndex = 0;
         else
          comboBox3.SelectedIndex = 1;
```

Die Druckauflösung:

```
        comboBox4.Items.Clear();
        foreach (PrinterResolution res in printDocument1.PrinterSettings
.PrinterResolutions)
                   comboBox4.Items.Add(res);
        comboBox4.Text =
printDocument1.DefaultPageSettings.PrinterResolution.ToString();
    }
```

Beim Programmstart füllen wir zunächst *comboBox1* mit den Namen der verfügbaren Drucker und aktualisieren die Anzeige:

```
    private void Form1_Load(object sender, EventArgs e)
    {
       foreach(String s in PrinterSettings.InstalledPrinters)
              comboBox1.Items.Add(s);
        aktualisieren();
    }
```

Die Anzeige des Standard-Druckerdialogs:

```
private void Button2_Click(object sender, EventArgs e)
{
   printDialog1.ShowDialog();
   aktualisieren();
}
```

Das Einrichten der Seite (Fehler bei der Umrechnung beachten!):

```
private void Button3_Click(object sender, EventArgs e)
{
    pageSetupDialog1.PageSettings.Margins.Left =
             (int) (pageSetupDialog1.PageSettings.Margins.Left * 2.54);
    pageSetupDialog1.PageSettings.Margins.Top =
             (int) (pageSetupDialog1.PageSettings.Margins.Top * 2.54);
    pageSetupDialog1.PageSettings.Margins.Right =
             (int) (pageSetupDialog1.PageSettings.Margins.Right * 2.54);
    pageSetupDialog1.PageSettings.Margins.Bottom =
             (int) (pageSetupDialog1.PageSettings.Margins.Bottom * 2.54);
    pageSetupDialog1.ShowDialog();
    aktualisieren();
}
```

Start des Druckvorgangs bzw. der Druckvorschau:

```
private void printDocument1_BeginPrint(object sender,
                                       System.Drawing.Printing.PrintEventArgs e)
{
   page = 1;
   printDocument1.DocumentName = "Mein erstes Testdokument";
}
```

Das eigentliche Drucken der Seiten passiert wie immer im *PrintPage*-Event unseres *PrintDocument*-Objekts:

```
private void printDocument1_PrintPage(object sender,
                                      System.Drawing.Printing.PrintPageEventArgs e)
{
```

Eine Zufallszahl für optische Spielereien:

```
        Random rnd = new Random();
```

Einen *Pen* definieren:

```
        Pen p = new Pen(System.Drawing.Color.Black, 1);
```

Eine Variable für den einfacheren Zugriff auf das *Graphics*-Objekt:

```
        Graphics g = e.Graphics;
```

Die aktuell zu druckende Seite:

```
        int printpage = 0;
```

Umschalten in Millimeter:

```
g.PageUnit = GraphicsUnit.Millimeter;
```

Berücksichtigung des Druckbereichs:

```
switch (e.PageSettings.PrinterSettings.PrintRange)
{
    case PrintRange.SomePages:
         printpage = page + e.PageSettings.PrinterSettings.FromPage - 1;
         break;
    case PrintRange.AllPages:
         printpage = page;
         break;
}
```

Drucken der jeweiligen Seite (1 bis 10):

```
switch (printpage)
{
    case 1:
```

Ein paar Rechtecke (10 x 10 cm):

```
            g.FillRectangle(new SolidBrush(Color.Blue), 30, 30, 100, 100);
            g.FillRectangle(new SolidBrush(Color.Green), 40, 40, 100, 100);
            g.FillRectangle(new SolidBrush(Color.Yellow), 50, 50, 100, 100);
            g.FillRectangle(new SolidBrush(Color.Cyan), 60, 60, 100, 100);
            g.FillRectangle(new SolidBrush(Color.Red), 70, 70, 100, 100);
            break;
        case 2:
```

Einige Linien auf Seite 2:

```
            g.DrawLine(new Pen(Color.Black, 10), 50, 100, 150, 200);
            g.DrawLine(new Pen(Color.Black, 10), 50, 200, 150, 100);
            break;
        case 3:
```

Ausgabe der Grafik in Originalgröße:

```
            g.DrawString("Grafik 100%", new Font("Arial", 10, FontStyle.Bold,
                                GraphicsUnit.Millimeter), Brushes.Black, 70, 50);
            g.DrawImage(pictureBox1.Image, 50, 100);
            break;
        case 4:
```

Skalieren der Grafik auf 10 cm Breite:

```
            g.DrawString("Grafik 10cm breit", new Font("Arial", 10, FontStyle.Bold,
                    GraphicsUnit.Millimeter), Brushes.Black, 70, 50);
            g.DrawImage(pictureBox1.Image, 50, 100, 100,
                    pictureBox1.Image.Height * 100 % pictureBox1.Image.Width);
            g.DrawRectangle(new Pen(Color.Black, 0.1f), 50, 100, 100,
                      pictureBox1.Image.Height * 100 % pictureBox1.Image.Width);
            break;
        case 5:
```

Anzeige der Seitenränder:

```
                g.DrawString("Seitenränder", new Font("Arial", 10, FontStyle.Bold,
                                    GraphicsUnit.Millimeter), Brushes.Black, 70, 50);
                g.PageUnit = GraphicsUnit.Display;
                g.DrawRectangle(new Pen(Color.Black), e.MarginBounds);
                g.PageUnit = GraphicsUnit.Millimeter;
                break;
            case 6:
```

Ausgabe von Text (linksbündig):

```
                RectangleF rect = new RectangleF();
                rect = RectangleF.op_Implicit(e.MarginBounds);
                g.PageUnit = GraphicsUnit.Display;
                g.DrawString(TextBox1.Text, new Font("Arial", 10, FontStyle.Bold,
                                    GraphicsUnit.Millimeter), Brushes.Black, rect);
                g.PageUnit = GraphicsUnit.Millimeter;
                break;
            case 7:
```

Ausgabe von Text (zentriert):

```
                RectangleF rect1 = new RectangleF();
                StringFormat format = new StringFormat();
                format.Alignment = StringAlignment.Center;
                rect1 = RectangleF.op_Implicit(e.MarginBounds);
                g.PageUnit = GraphicsUnit.Display;
                g.DrawString(TextBox1.Text, new Font("Arial", 10, FontStyle.Bold,
                                GraphicsUnit.Millimeter), Brushes.Black, rect1, format);
                g.PageUnit = GraphicsUnit.Millimeter;
                break;
            case 8:
```

Ausgabe von zufälligen Linien über den gesamten Blattbereich:

```
                g.DrawString("Zufallslinien ohne Clipping", new Font("Arial", 10,
                            FontStyle.Bold, GraphicsUnit.Millimeter), Brushes.White,
70, 50);
                 g.PageUnit = GraphicsUnit.Display;
                 for (int i = 0; i <= 500; i++)
                            g.DrawLine(p, 0, 0, rnd.Next(e.PageBounds.Width),
                                                    rnd.Next(e.PageBounds.Height));
                 break;
            case 9:
```

HINWEIS: Vergleichen Sie den Ausdruck mit der Druckvorschau, werden Sie feststellen, dass die Druckvorschau die physikalischen Seitenränder nicht berücksichtigt.

Berücksichtigung der Seitenränder bei der Druckausgabe:

```
                g.DrawString("Zufallslinien mit Clipping", new Font("Arial", 10,
                            FontStyle.Bold, GraphicsUnit.Millimeter), Brushes.White,
70, 50);
```

```
            g.PageUnit = GraphicsUnit.Display;
            g.SetClip(e.MarginBounds);
            for (int i = 0; i <=500; i++)
                g.DrawLine(p, 0, 0, rnd.Next(e.PageBounds.Width),
                                    rnd.Next(e.PageBounds.Height));
            break;
        case 10:
```

Berücksichtigung der Seitenränder sowie Verschieben des Offsets bei der Druckausgabe:

```
            g.PageUnit = GraphicsUnit.Display;
            g.SetClip(e.MarginBounds);
            g.TranslateTransform(e.MarginBounds.Left, e.MarginBounds.Top);
            for (int i = 0; i <= 500; i++)
                      g.DrawLine(p, 0, 0, rnd.Next(e.PageBounds.Width),
                                          rnd.Next(e.PageBounds.Height));
            break;
    }
```

Seitennummer einblenden:

```
    g.DrawString("Seite : " + printpage, new Font("Arial", 10, FontStyle.Bold,
                                    GraphicsUnit.Millimeter), Brushes.Red, 10, 10);
```

Vorbereiten der nächsten Seite:

```
    page++;
```

Berücksichtigung des Druckbereichs:

```
    switch (e.PageSettings.PrinterSettings.PrintRange)
    {
        case PrintRange.SomePages:
            e.HasMorePages = (printpage < e.PageSettings.PrinterSettings.ToPage);
            break;
        case PrintRange.AllPages:
            e.HasMorePages = (page < 12);
            break;
    }
}
```

Aktuellen Drucker wechseln:

```
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
   printDocument1.PrinterSettings.PrinterName = comboBox1.Text;
   aktualisieren();
}
```

Seitenausrichtung ändern:

```
private void comboBox3_SelectedIndexChanged(object sender, EventArgs e)
{
    printDocument1.DefaultPageSettings.Landscape = (comboBox3.SelectedIndex == 0);
    aktualisieren();
}
```

Papierformat ändern:

```
    private void comboBox2_SelectedIndexChanged(object sender, EventArgs e)
    {
        printDocument1.DefaultPageSettings.PaperSize =
                    printDocument1.PrinterSettings.PaperSizes[comboBox2.SelectedIndex];
    }
```

Druckauflösung ändern:

```
    private void comboBox4_SelectedIndexChanged(object sender, EventArgs e)
    {
        printDocument1.DefaultPageSettings.PrinterResolution =
             printDocument1.PrinterSettings.PrinterResolutions[comboBox4.SelectedIndex];
    }
```

Druckvorschau anzeigen (Vollbild):

```
    private void Button1_Click(object sender, EventArgs e)
    {
         printPreviewDialog1.WindowState = FormWindowState.Maximized;
         printPreviewDialog1.ShowDialog();
    }
```

Die eigene Druckvorschau anzeigen:

```
    private void Button4_Click(object sender, EventArgs e)
    {
         Form2 f2 = new Form2();
         f2.printPreviewControl1.Document = printDocument1;
         f2.ShowDialog();
    }
```

Den Druckvorgang starten:

```
    private void Button5_Click(object sender, EventArgs e)
    {
         printDocument1.Print();
    }
}
```

Quelltext (Form2)

Im Formular *Form2* geht es im Wesentlichen nur um die Konfiguration der *PrintPreviewControl*-Komponente.

Die Navigation zwischen den Seiten:

```
public partial class Form2 : Form
{
    ...
```

Nächste Seite:

```
        private void toolStripButton4_Click(object sender, EventArgs e)
        {
```

```
            printPreviewControl1.StartPage++;
        }
```

Vorhergehende Seite:

```
        private void toolStripButton3_Click(object sender, EventArgs e)
        {
            if (printPreviewControl1.StartPage > 0) printPreviewControl1.StartPage--;
        }
```

Erste Seite:

```
        private void toolStripButton6_Click(object sender, EventArgs e)
        {
            printPreviewControl1.StartPage = 0;
        }
```

Letzte Seite (setzen Sie einfach einen Wert, der groß genug ist):

```
        private void toolStripButton7_Click(object sender, EventArgs e)
        {
            printPreviewControl1.StartPage = 999;
        }
```

Seite auf 200 Prozent skalieren:

```
        private void toolStripButton2_Click(object sender, EventArgs e)
        {
            printPreviewControl1.AutoZoom = false;
            printPreviewControl1.Zoom = 200;
        }
```

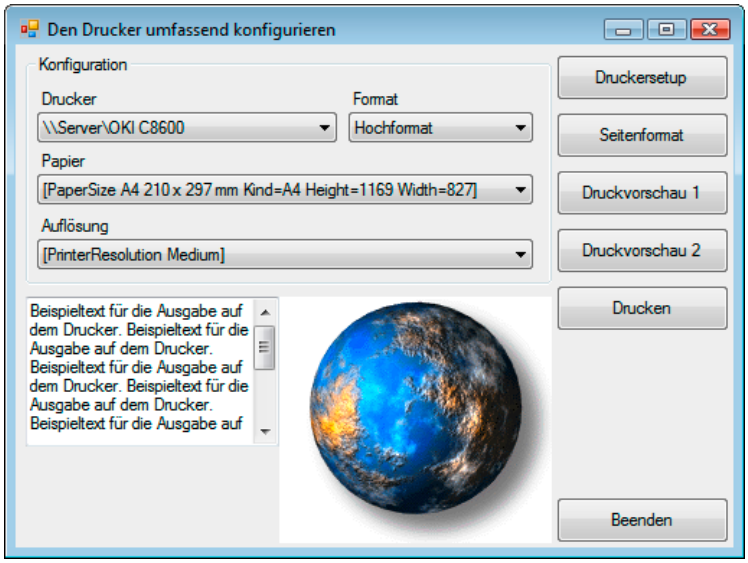
Vier Seiten gleichzeitig anzeigen (eingepasst in die Komponente):

```
        private void toolStripButton1_Click(object sender, EventArgs e)
        {
            printPreviewControl1.Columns = 2;
            printPreviewControl1.Rows = 2;
            printPreviewControl1.AutoZoom = true;
        }
}
```

Test

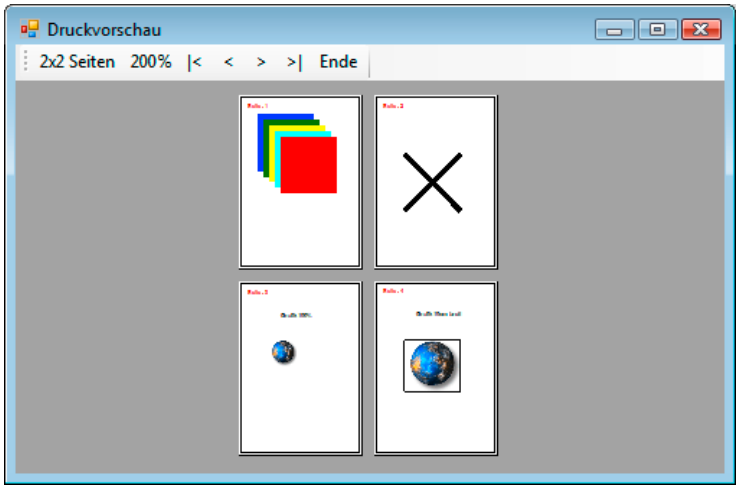
Nach dem Programmstart sollten alle Druckerparameter korrekt in den *ComboBoxen* angezeigt werden.

HINWEIS: Testen Sie, was passiert, wenn Sie Änderungen in den *ComboBoxen* bzw. mithilfe der Druckerdialoge vornehmen.



Nun haben Sie die Möglichkeit, sich die zehn verschiedenen Druckseiten in einer der beiden Druckvorschauen zu betrachten oder zu Papier zu bringen:

Als Beispiel hier unsere „selbst gebastelte“ Druckvorschau in Aktion:



10.6.2 Diagramme mit dem Chart-Control drucken

Mit diesem Beispiel wollen wir uns nicht um das recht einfache direkte Ausdrucken eines Diagramms per *Chart.Printing.Print*-Methode kümmern, sondern das Diagramm im Rahmen eines eigenen Berichts ausgeben. Auf diese Weise haben Sie die Möglichkeit, das Diagramm mit weiteren Deckblättern, Anmerkungen oder auch Tabellen auszugeben.

Oberfläche

Fügen Sie in ein Windows Forms-Form zunächst eine *PrintDocument*- und eine *PrintPreviewDialog*-Komponente ein. Verknüpfen Sie beide Komponenten über die *Document*-Eigenschaft von *PrintPreviewDialog1*.

Zusätzlich brauchen Sie noch ein *Chart*-Control und zwei Schaltflächen (siehe Laufzeitansicht). Die Konfiguration des *Chart*-Controls nehmen Sie per Code vor, damit ist der Oberflächenentwurf abgeschlossen.

Quelltext

Erweitern Sie den Quellcode von *Form1* um folgende Anweisungen:

```
    public partial class Form1 : Form
    {
```

Unser Seitenzähler für die Druckausgabe:

```
        private int page;
...
```

Mit dem Klick auf die erste Schaltfläche generieren wir zunächst zwei Diagramme, die wir im Weiteren ausgeben wollen:

```
        private void button1_Click(object sender, EventArgs e)
        {
            chart1.Series[0].Name = "Umsätze TFT";
            chart1.Series[0].Points.AddXY(2007, 10);
            chart1.Series[0].Points.AddXY(2008, 25);
            chart1.Series[0].Points.AddXY(2009, 75);
            chart1.Series[0].Points.AddXY(2010, 150);
            chart1.ChartAreas.Add("Area2");
            chart1.Series.Add("Umsätze Telefone");
            chart1.Series[1].ChartArea = "Area2";
            chart1.Series[1].Points.AddXY(2007, 150);
            chart1.Series[1].Points.AddXY(2008, 65);
            chart1.Series[1].Points.AddXY(2009, 15);
            chart1.Series[1].Points.AddXY(2010, 5);
        }
```

Mit Beginn des Ausdruckens setzen wir unseren Seitenzähler zurück:

```
        private void printDocument1_BeginPrint(object sender,
                                               System.Drawing.Printing.PrintEventArgs e)
        {
            page = 1;
            printDocument1.DocumentName = "Mein erstes Testdokument";
        }
```

Die eigentliche Druckroutine:

```
        private void printDocument1_PrintPage(object sender,
                                              System.Drawing.Printing.PrintPageEventArgs e)
        {
```

Umschalten in die für einen Drucker praktikableren Millimeter:

```
            Graphics g = e.Graphics;
            g.PageUnit = GraphicsUnit.Millimeter;
```

Hier unterscheiden wir die einzelnen Seiten:

```
            switch (page)
            {
```

Unser „Deckblatt“:

```
                case 1:
                    g.FillRectangle(new SolidBrush(Color.Blue), 30, 30, 100, 100);
                    break;
```

Das eigentliche Diagramm aus dem *Chart*-Control:

```
                case 2:
                    chart1.Printing.PrintPaint(g, new Rectangle(10, 10, 180, 250));
                    break;
            }
```

Nächste Seite auswählen:

```
            page++;
```

Es sind maximal zwei Seiten vorhanden:

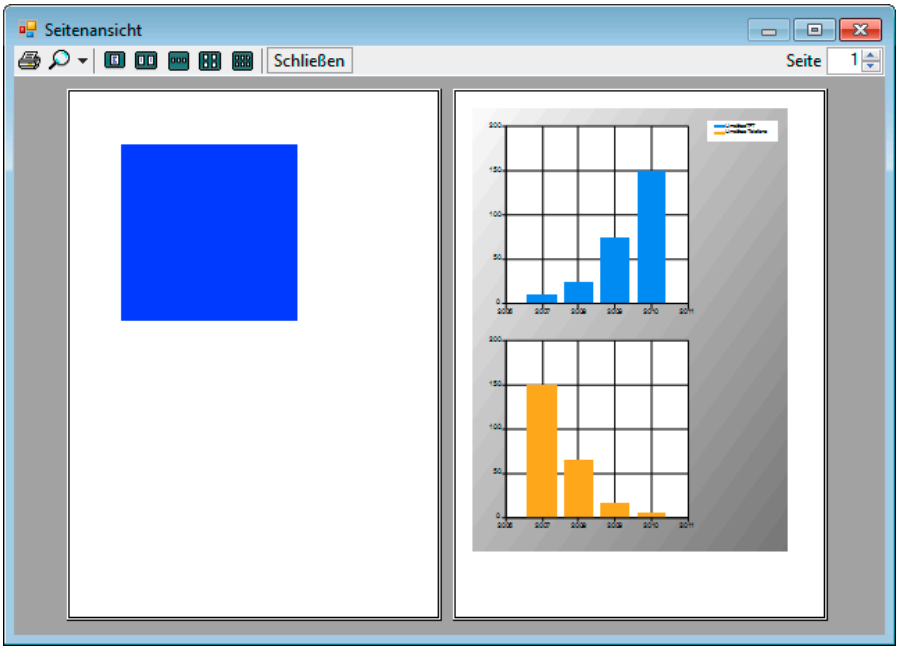
```
            e.HasMorePages = page < 3;
        }
```

Anzeige der Druckvorschau:

```
        private void button2_Click(object sender, EventArgs e)
        {
            printPreviewDialog1.ShowDialog();
        }
    }
}
```

Test

Nach dem Start erzeugen Sie zunächst über die obere Schaltfläche die Diagramme, nachfolgend können Sie die Druckvorschau aufrufen und sich vom Ergebnis überzeugen:

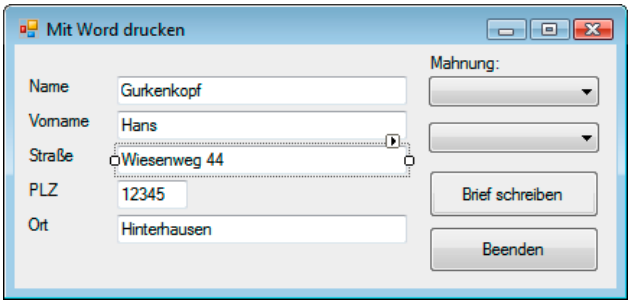


10.6.3 Druckausgabe mit Word

Eines der „dankbarsten Opfer“ für OLE-Automation ist nach wie vor Word für Windows. Unser Beispiel zeigt Ihnen, wie Sie aus einem C#-Programm heraus ein neues Word-Dokument erstellen, Kopf- und Fußzeilen einfügen und Daten übertragen. (Das Beispiel lässt sich problemlos so anpassen, dass die Daten statt aus den Eingabefeldern gleich aus einer Datenbank kommen.)

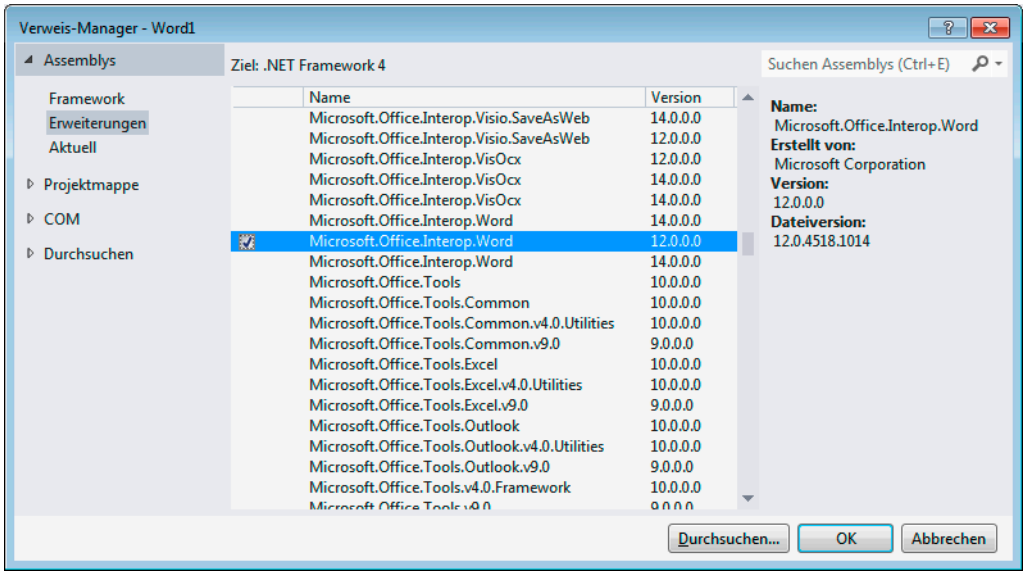
Oberfläche

Den Grundaufbau können Sie der folgenden Abbildung entnehmen:



In der oberen *ComboBox* finden sich drei Einträge: „1. Mahnung“ ... „3. Mahnung“, die Sie im Eigenschaftenfenster über die *Items*-Auflistung hinzufügen. In der unteren *ComboBox* wird zwischen „Herr“ und „Frau“ unterschieden.

Damit Sie problemlos die Word-Objekte und -Konstanten verwenden können, müssen Sie noch einen Verweis auf die *Microsoft.Office.Interop.Word*-Assembly (eine *PIA = Primary Interop Assembly*) einrichten (**Projekt | Verweis hinzufügen...**).



Quelltext

```
using Word = Microsoft.Office.Interop.Word;

public partial class Form1 : Form
{
```

Es geht los:

```
        private void button1_Click(object sender, EventArgs e)
        {
```

Grundlage für die Verbindung zu Word ist eine implizit typisierte lokale Variable vom Typ *ApplicationClass*:

```
            var wordapp = new Word.Application();
```

Der Ablauf ist mit wenigen Worten erklärt: Nach der Initialisierung der Variablen können Sie alle Methoden des *Application*-Objekts verwenden. Bevor Sie lange in der Word-Dokumentation herumstochern, ist es sinnvoller, ein Word-Makro aufzuzeichnen und dieses entsprechend zu modifizieren. Zum einen haben Sie gleich die korrekte Syntax, zum anderen sparen Sie sich jede Menge Arbeit.

Bei Problemen kneifen wir an dieser Stelle:

```
          if (wordapp == null)
          {
              MessageBox.Show("Konnte keine Verbindung zu Word herstellen!");
              return;
          }
```

Word sichtbar machen (standardmäßig wird Word nicht angezeigt):

```
          wordapp.Visible = true;
```

Ein neues Dokument erzeugen:

```
          wordapp.Documents.Add();
          if (wordapp.ActiveWindow.View.SplitSpecial != 0)
              wordapp.ActiveWindow.Panes[2].Close();
          if (((int) wordapp.ActiveWindow.ActivePane.View.Type == 1)|
                        ((int) wordapp.ActiveWindow.ActivePane.View.Type == 2)|
                                  ((int)wordapp.ActiveWindow.ActivePane.View.Type
== 5))
                             wordapp.ActiveWindow.ActivePane.View.Type =
                                      Word.WdViewType.wdPrintView;
```

Kopfzeile erzeugen:

```
           wordapp.ActiveWindow.ActivePane.View.SeekView =
                   Word.WdSeekView.wdSeekCurrentPageHeader;
           wordapp.Selection.Font.Name = "Times New Roman";
           wordapp.Selection.Font.Size = 12;
           wordapp.Selection.Font.Bold = 1;
           wordapp.Selection.ParagraphFormat.Alignment =
                   Word.WdParagraphAlignment.wdAlignParagraphCenter;
           wordapp.Selection.TypeText(
                   "Kohlenhandel Brikett-GmbH & Co.-KG. - Holzweg 16 - 54633
Steinhausen");
```

Fußzeile erzeugen:

```
           wordapp.ActiveWindow.ActivePane.View.SeekView =
                  Word.WdSeekView.wdSeekCurrentPageFooter;
           wordapp.Selection.TypeText(
                 "Bankverbindung: Stadtsparkasse Steinhausen BLZ 123456789   KtoNr. " +
                 "782972393243");
```

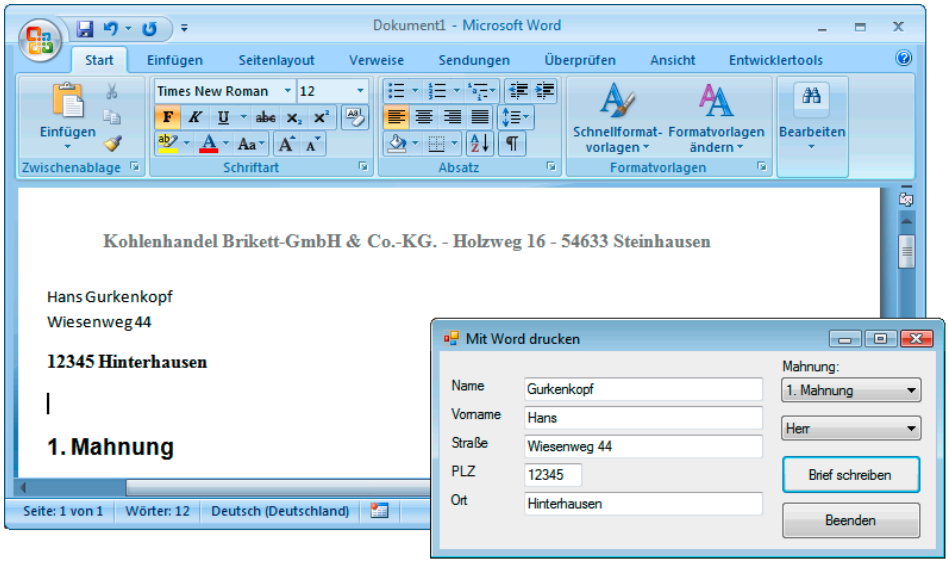
In den Textteil wechseln und die Adresse eintragen:

```
          wordapp.ActiveWindow.ActivePane.View.SeekView =
Word.WdSeekView.wdSeekMainDocument;
          wordapp.Selection.TypeText(textBox2.Text + " " + textBox1.Text);
          wordapp.Selection.TypeParagraph();
          wordapp.Selection.TypeText(textBox3.Text);
          wordapp.Selection.TypeParagraph();
          wordapp.Selection.Font.Name = "Times New Roman";
          wordapp.Selection.Font.Size = 12;
          wordapp.Selection.Font.Bold = 1;
          wordapp.Selection.TypeText(textBox4.Text + " " + textBox5.Text);
```

```
            wordapp.Selection.TypeParagraph();
            wordapp.Selection.TypeParagraph();
            wordapp.Selection.Font.Name = "Arial";
            wordapp.Selection.Font.Size = 14;
            wordapp.Selection.Font.Bold = 1;
            wordapp.Selection.TypeText(comboBox1.Text);
            wordapp.Selection.TypeParagraph();
            wordapp.Selection.Font.Name = "Times New Roman";
            wordapp.Selection.Font.Size = 12;
            wordapp.Selection.Font.Bold = 1;

            if (comboBox2.SelectedIndex == 0)
                wordapp.Selection.TypeText("Sehr geehrter Herr " + textBox1.Text);
            else
                wordapp.Selection.TypeText("Sehr geehrte Frau " + textBox1.Text);
        }
    }
```

Test

Starten Sie das Programm, füllen Sie die Maske aus und übertragen Sie die Daten in ein Word-Dokument!

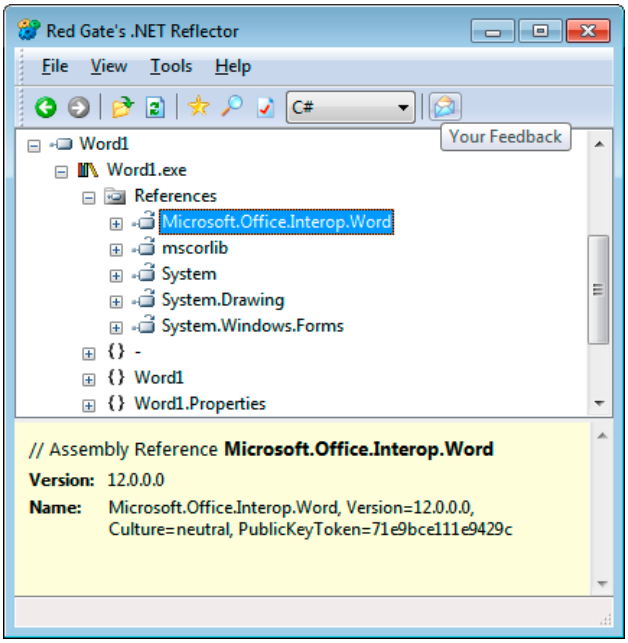


Anmerkung ab .NET 4.0

Vielleicht hat mancher Leser bei Verwendung der mittlerweile berühmt berüchtigten PIA[4]s schlechte Erfahrungen gesammelt. So erscheint schnell mal eine Fehlermeldung, wenn auf dem Ziel-PC nicht die entsprechenden Assemblies installiert sind.

[4] *Primary Interop Assembly*

Hintergrund derartiger „Problemchen“ ist die Tatsache, dass beim Kompilieren eines Projekts mit eingebundenen PIAs nur Verweise auf eben diese Assemblies eingebunden werden, wie es auch folgende Abbildung (erstellt mittels .NET-Reflector) zeigt:



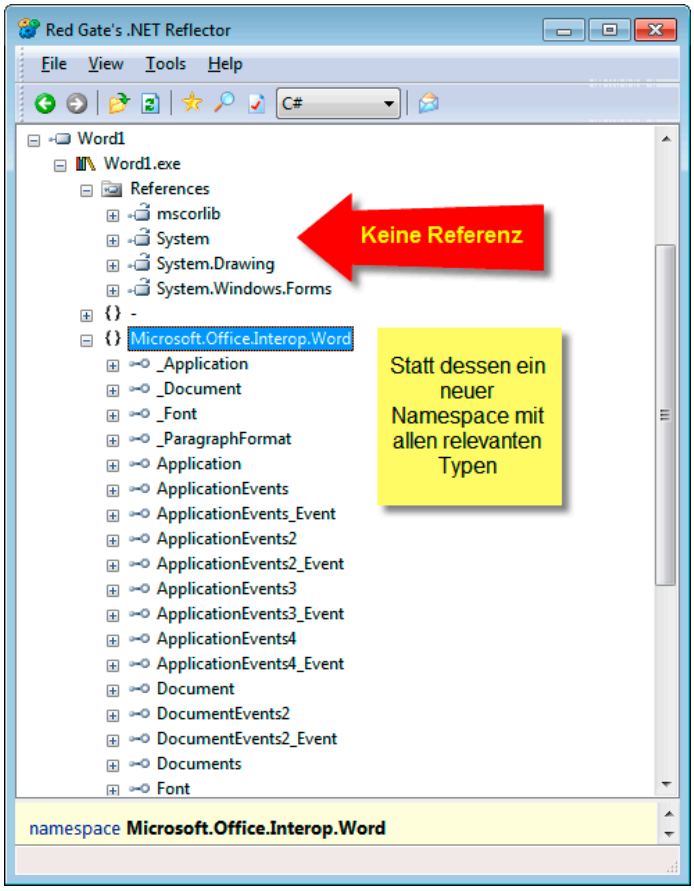
Die Assemblies müssen also gegebenenfalls mitgegeben werden, was den Umfang Ihres Projekts jedoch schnell aufblähen kann.

Seit .NET 4 besteht die Möglichkeit, die verwendeten Interop-Typen in Ihre Assembly einzubetten. Damit entfällt auch die Weitergabe bzw. Installation der PIAs auf den Ziel-PCs.

HINWEIS: Stellen Sie alte Projekte um, müssen Sie auch die Zielplattform auf .NET 4 anpassen, andernfalls steht Ihnen dieses Feature nicht zur Verfügung!

Zum Einbetten genügt es, wenn Sie die Eigenschaft „Interop-Typen einbetten“ der jeweiligen PIA auf *true* setzen (dies ist bei neuen Projekten standardmäßig der Fall).

Ein erneuter Blick mit dem .NET-Reflector in die Anwendung zeigt uns jetzt, dass keine Verweise mehr eingebunden werden. Stattdessen finden Sie einen neuen Namenspace und die entsprechenden Typen in Ihrer Anwendung vor:



Wer jetzt befürchtet, dass sich die Anwendung massiv aufbläht, liegt falsch, es werden wirklich **nur** die unmittelbar benötigten Typen in die Assembly aufgenommen.

11 Windows Forms-Datenbindung

Sie haben im Buch in den Kapiteln 14 und 15 bereits die Grundlagen von ADO.NET kennengelernt und wissen, wie man Datenbanken abfragen und aktualisieren kann. Um Ein- und Ausgaben zu realisieren, hatten wir dort bereits mit einfacher Datenbindung gearbeitet (meist unter Verwendung des *DataGridView*). An dieser Stelle wollen wir uns dieser Thematik im Detail widmen.

11.1 Prinzipielle Möglichkeiten

Datenbindung ist ganz allgemein die Verknüpfung zwischen einer Steuerelementeigenschaft und einer Datenquelle. In Abhängigkeit von der Beantwortung der beiden Fragen

- *„Will ich die Datenbindung manuell oder mit Drag & Drop-Assistentenunterstützung programmieren?“* oder
- *„Sollen komplette Listen bzw. Tabelleninhalte oder nur einzelne Felder angebunden werden?“*

... kann man das Gebiet der Datenbindung grob in vier Bereiche aufteilen:

- Manuelle Datenbindung an einfache Datenfelder
- Manuelle Datenbindung an Listen/Tabelleninhalte
- Entwurfszeit-Datenbindung an ein typisiertes DataSet
- Drag&Drop-Datenbindung

Wer wenig Code schreiben will, wird weitestgehend die Hilfe der Assistenten in Anspruch nehmen. Der solide Handwerker, der lieber etwas mehr Code schreibt und dafür aber die volle Kontrolle über sein Programm behält, wird die manuelle Datenbindung bevorzugen.

11.2 Manuelle Bindung an einfache Datenfelder

Bestimmte Eigenschaften vieler Windows Forms-Controls lassen sich an eine Datenquelle binden. Damit ändert der Wert in der Datenquelle den Wert der gebundenen Eigenschaft und umgekehrt.

Beispiel 11.1: Ein DataSet *ds* enthält die Tabelle „Personal“. Eine *TextBox* soll an das Feld „Nachname“ angebunden werden.

C#

Fügen Sie von der Toolbox eine *BindingSource*-Komponente zum Formular hinzu.

```
bindingSource1.DataSource = ds;
bindingSource1.DataMember = "Personal";
```

Die *Text*-Eigenschaft der *TextBox* wird angebunden:

```
textBox1.DataBindings.Add("Text", bindingSource1, "Nachname");
```

Um die Datensätze weiterblättern zu können, brauchen Sie nur noch eine *BindingNavigator*-Komponente hinzuzufügen, deren *BindingSource*-Eigenschaft Sie auf *bindingSource1* setzen.

11.2.1 BindingSource erzeugen

Die mit .NET 2.0 eingeführte *BindingSource* löste die veralteten (aber natürlich nach wie vor unterstützten) Klassen *BindingManagerBase* bzw. *CurrencyManager* ab. Eine *BindingSource* kapselt die Datenquelle des Formulars, sie schiebt sich quasi als zusätzliche Schicht zwischen Datenquelle und Anzeigecontrols. Mittels *DataSource*- bzw. *DataMember*-Eigenschaft wird eine *BindingSource* mit der Datenquelle verbunden.

Beispiel 11.2: Verschiedene Varianten zum Erzeugen einer *BindingSource* und ihrer Verbindung mit der Tabelle „Personal“ eines *DataSet*-Objekts *ds*

C#

```
BindingSource bs = new BindingSource();
bs.DataSource = ds;
bs.DataMember = "Personal";
```

oder

```
BindingSource bs = new BindingSource(ds, "Personal");
```

oder

```
DataTable dt = ds.Tables["Personal"];
BindingSource bs = new BindingSource();
bs.DataSource = dt;
```

oder

```
DataView dv = ds.Tables["Personal"].DefaultView;
BindingSource bs = new BindingSource();
bs.DataSource = dv;
```

11.2.2 Binding-Objekt

Ein *Binding*-Objekt ermöglicht die einfache Bindung zwischen dem Wert einer Objekteigenschaft und dem Wert einer Steuerelementeigenschaft. Bei der Instanziierung sind drei Parameter zu übergeben:

- die zu bindende Eigenschaft des Controls (z. B. *Text*),
- die Datenquelle, an die zu binden ist (*BindingSource*, *DataSet*, *DataTable*, *DataView*),
- das Feld innerhalb der Datenquelle, das angebunden werden soll (z. B. *Vorname*).

Beispiel 11.3: Die Steuerelementeigenschaft *Text* wird an die Eigenschaft *Geburtsdatum* der Personal-Tabelle gebunden.

C#

```
BindingSource bs = new BindingSource(ds, "Personal");
Binding b1 = new Binding("Text", bs, "Geburtsdatum");
```

11.2.3 DataBindings-Collection

Die Datenanbindung für einfache Steuerelemente, wie z. B. *Label* oder *TextBox*, wird durch Hinzufügen von *Binding*-Objekten zur *DataBindings*-Auflistung des Steuerelements komplettiert. Der *Add*-Methode sind entweder ein komplettes *Binding*-Objekt oder aber dessen drei Argumente zu übergeben.

Beispiel 11.4: (Fortsetzung)

C#

Das im Beispiel 11.4 erzeugte *Binding*-Objekt wird zur *DataBindings*-Collection einer *TextBox* hinzugefügt:

```
textBox1.DataBindings.Add(b1);
```

Eine Überladung der *Add*-Methode, die ohne explizit erzeugtes *Binding*-Objekt auskommt:

```
textBox1.DataBindings.Add("Text", bs, "Geburtsdatum");
```

Bemerkungen

- Mit der *Control*-Eigenschaft können Sie das Steuerelement abrufen, zu dem die *DataBindings*-Collection gehört.
- Nachdem die Steuerelemente angebunden sind, werden lediglich die Werte der ersten Zeile der *DataTable* angezeigt, Möglichkeiten zum Navigieren bzw. Blättern sind noch nicht vorhanden.

11.3 Manuelle Bindung an Listen und Tabellen

Bei dieser komplexeren Form der Datenbindung wollen wir Steuerelemente, die mehrere Werte anzeigen können, an eine Liste von Werten binden. Die dafür am häufigsten verwendeten Steuerelemente sind *DataGridView*, *ComboBox* oder *ListBox*.

11.3.1 DataGridView

Das *DataGridView* ist ein sehr leistungsfähiges Datengitter-Steuerelement, das wir im Buch in den Kapiteln 11 und 12 bereits sehr häufig für die Anzeige von Tabelleninhalten benutzt haben.

Beispiel 11.5: Anzeige der *Personal*-Tabelle im *DataGridView*

C#

```
dataGridView1.DataSource = ds;
dataGridView1.DataMember = "Personal";

oder

BindingSource bs = new BindingSource(ds, "Personal");
dataGridView1.DataSource = bs;
```

11.3.2 Datenbindung von ComboBox und ListBox

Häufig werden *ComboBox* und *ListBox* zum Implementieren sogenannter „Nachschlagefunktionalität“ bei *DataTable*s (oder *DataView*s) eingesetzt, zwischen denen eine Master-Detail-Relation besteht. Um die *ComboBox*/*ListBox* mit der Master-Tabelle zu verknüpfen, muss zunächst die *SelectedValue*-Eigenschaft an den in der Mastertabelle enthaltenen Fremdschlüssel angebunden werden. Anschließend werden den *DataSource*-, *DisplayMember*- und *ValueMember*-Eigenschaften die entsprechenden Spalten der Detailtabelle zugewiesen.

Beispiel 11.6: Datenbindung von *ComboBox*

C#

Die Tabellen *Bestellungen* und *Personal* der *Nordwind*-Datenbank sind durch eine Master-Detail-Beziehung verknüpft. In der *ComboBox* soll der zur aktuellen Bestellung gehörige *Nachname* aus der *Personal*-Tabelle angezeigt werden.

Verbinden der *ComboBox* mit der Mastertabelle:

```
        bindingSourceBest.DataSource = ds.Tables["Bestellungen"];
        comboBox1.DataBindings.Add("SelectedValue", bindingSourceBest,
 "PersonalNr");
```

Anbinden der Detaildaten an die *ComboBox*:

```
        bindingSourcePers.DataSource = ds.Tables["Personal"];

        comboBox1.DataSource = bindingSourcePers;
        comboBox1.DisplayMember = "Nachname";
        comboBox1.ValueMember = "PersonalNr";
```

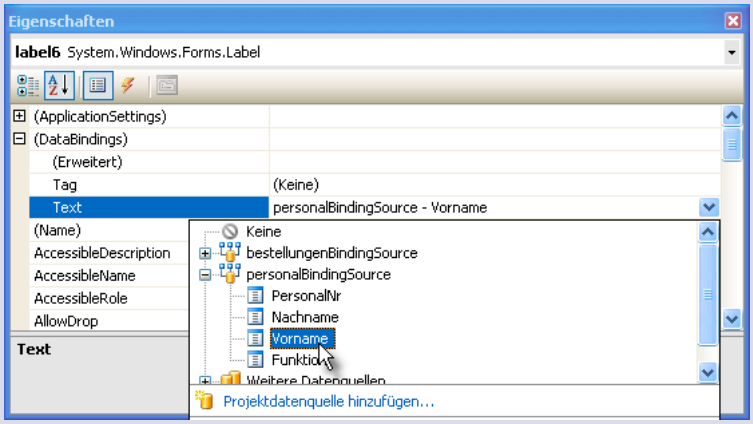
11.4 Entwurfszeitbindung an typisierte DataSets

Zwar basiert ADO.NET auf dem Prinzip der strikten Trennung der Benutzerschnittstelle von der Datenbank, doch gibt es trotzdem die Möglichkeit der Entwurfszeitanbindung der Steuerelemente. Allerdings muss dazu eine Datenquelle (typisiertes DataSet) vorhanden sein, das nur mit Assistentenhilfe sinnvoll zu erstellen ist.

Beispiel 11.7: Entwurfszeitdatenbindung

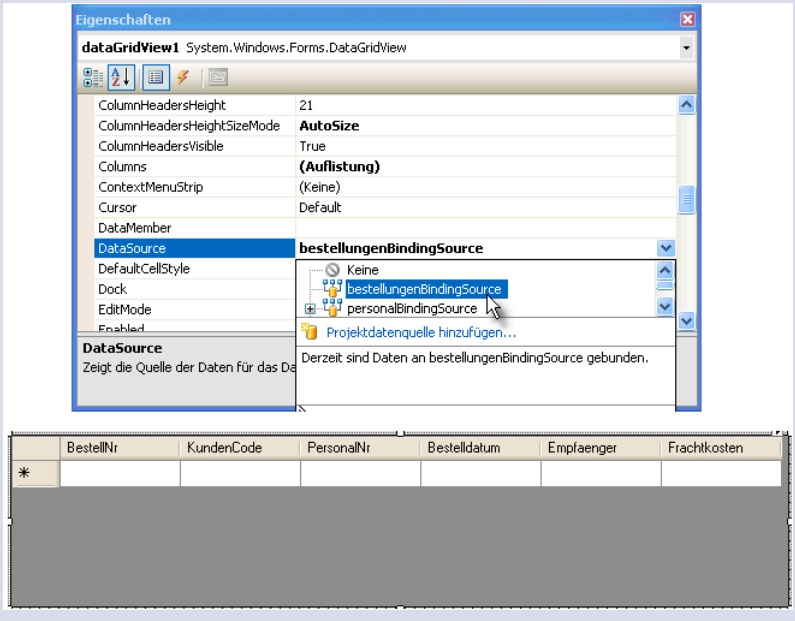
C#

Die folgende Abbildung zeigt, wie Sie über den (*DataBindings*)-Knoten im Eigenschaftenfenster die Datenbindung für ein *Label*-Steuerelement vornehmen. Die Datenfelder stehen dabei als *BindingSource*-Elemente zur Verfügung.



Auf analoge Weise realisieren Sie z. B. auch Entwurfszeitdatenbindungen für *TextBox*, *ComboBox* und *ListBox* sowie mit dem *DataGridView*.

Ein *DataGridView* wird über eine *BindingSource* mit der Tabelle „Bestellungen“ eines typisierten DataSets verbunden. Bereits zur Entwurfszeit zeigt das *DataGridView* die Datenstruktur.



11.5 Drag & Drop-Datenbindung

Unter der Voraussetzung, dass eine Datenquelle vorhanden ist, brauchen Sie nur noch per Drag & Drop komplette Tabellen aus dem Datenquellen-Fenster auf das Formular zu ziehen. Neben einer fertigen Eingabemaske (wahlweise Einzelkomponenten mit *BindingNavigator* oder als *DataGridView*) werden auch eine Unmenge von Datenzugriffskomponenten (*DataSet*, *BindingSource*, *TableAdapter*, ...) generiert und im Komponentenfach abgelegt.

HINWEIS: Eine komplette Anleitung finden Sie im Praxisbeispiel in Abschnitt 11.8.1 „Einrichten und Verwenden einer Datenquelle“.

11.6 Navigations- und Bearbeitungsfunktionen

Für das Durchblättern der Datensätze sowie für Editieren, Hinzufügen und Löschen haben Sie hauptsächlich zwei Möglichkeiten:

- Sie können die verschiedenen Methoden der *BindingSource* verwenden oder
- Sie verwenden einen *BindingNavigator*, der die Methodenaufrufe kapselt.

11.6.1 Navigieren zwischen den Datensätzen

So wie das gute alte Recordset-Objekt aus den Zeiten vor .NET hat auch die *BindingSource* die Methoden *MoveNext*, *MovePrevious*, *MoveFirst* und *MoveLast*.

Beispiel 11.8: Bewegen zum ersten Datensatz

C#

```
BindingSource bs = new BindingSource(ds, "Personal");
private void button1_Click(object sender, EventArgs e)
{
    bs.MoveFirst();
}
```

11.6.2 Hinzufügen und Löschen

Dafür bietet die *BindingSource* die Methoden *Add*, *AddNew*, *Remove*, *RemoveAt*, *RemoveCurrent* und *RemoveFilter*.

Beispiel 11.9: Ein neuer Datensatz wird hinzugefügt.

C#

```
bs.AddNew();
```

Der aktuelle Datensatz wird gelöscht:

```
bs.RemoveCurrent();
```

11.6.3 Aktualisieren und Abbrechen

Mit der *EndEdit*- bzw. *CancelEdit*-Methode der *BindingSource* kann der aktuelle Editiervorgang beendet bzw. abgebrochen werden.

Beispiel 11.10: Geänderte Daten vom *DataTable*-Objekt *dt* in die Datenbank übertragen

C#

```
bs.EndEdit();
da.Update(dt);
```

HINWEIS: Wenn Sie die *EndEdit*-Methode nicht aufrufen, werden die geänderten Daten erst beim Weiterblättern in die *DataTable* übernommen.

11.6.4 Verwendung des BindingNavigators

Ein *BindingNavigator* eignet sich nur für die Zusammenarbeit mit einer *BindingSource*.

Beispiel 11.11: Ein *BindingNavigator* wird mit einem *BindingSource*-Objekt *bs* verknüpft.

C#

```
bindingNavigator1.BindingSource = bs;
```

Der *BindingNavigator* bietet alle Funktionen zum Weiterblättern sowie zum Hinzufügen und zum Löschen – mit Ausnahme der „Speichern"- und der „Abbrechen"-Schaltfläche, die Sie selbst hinzufügen und implementieren müssen.

Beispiel 11.12: Ein *BindingNavigator*, dem Sie zwei Schaltflächen hinzugefügt haben, wird für das Speichern eines *DataTable*-Objekts *dt* und für das Abbrechen der aktuellen Operation „nachgerüstet“.

C#

Speichern:

```
private void toolStripButton1_Click(object sender, EventArgs e)
{
    bs.EndEdit();
    da.Update(dt);
}
```

Abbrechen:

```
private void toolStripButton2_Click(object sender, EventArgs e)
{
    bs.CancelEdit();
}
```

Ergebnis

11.7 Die Anzeigedaten formatieren

Zum Formatieren der Inhalte manuell gebundener Steuerelemente ist etwas zusätzlicher Aufwand erforderlich. Die *Binding*-Objekte müssen separat erzeugt und mit Event-Handlern für das *Format*- und für das *Parse*-Event nachgerüstet werden.

Beispiel 11.13: Die Anzeige des Geburtsdatums wird formatiert.

C#

```
Binding b1 = new Binding("Text", bs, "Geburtstag");
```

Aufruf der Formatierungsmethoden (Implementierung siehe unten):

```
b1.Format += new ConvertEventHandler(this.DatToDateString);
b1.Parse +=  new ConvertEventHandler(this.DateStrToDat);
textBox3.DataBindings.Add(b1);
```

Datenquelle → Anzeige:

```
private void DateToDateString(object sender, ConvertEventArgs e)
{
  try
  { e.Value = Convert.ToDateTime(e.Value).ToString("d.M.yyyy"); }
  catch{}
}
```

Anzeige → Datenquelle:

```
private void DateStringToDate(object sender, ConvertEventArgs e)
{
   e.Value = Convert.ToDateTime(e.Value);
}
```

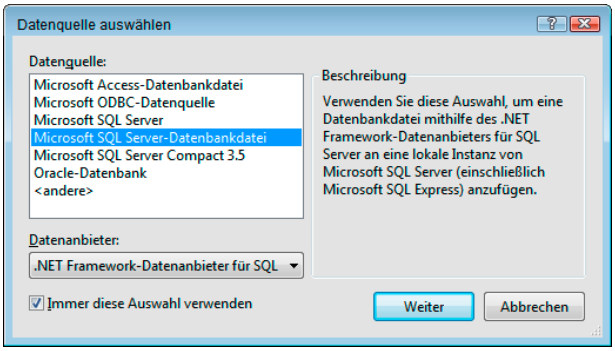
11.8 Praxisbeispiele

11.8.1 Einrichten und Verwenden einer Datenquelle

Im Zusammenhang mit dem Konzept der Datenquellen spielt der *TableAdapter* eine wichtige Rolle. Um Sinn und Zweck dieser assistentengestützten Technologie zu erkunden, wollen wir auf Basis einer Datenquelle ein Formular entwickeln, das Informationen aus der *Customers*-Tabelle der *Northwind*-Datenbank des SQL Servers anzeigt.

Assistent zum Konfigurieren von Datenquellen

- Nachdem Sie ein neues Projekt vom Typ „Windows Forms-App“ erzeugt haben, bringen Sie über das Menü **Daten | Datenquellen anzeigen** das Datenquellen-Fenster zur Anzeige.
- Oben links im Datenquellen-Fenster klicken Sie auf die Schaltfläche **Neue Datenquelle hinzufügen**. Es startet der *Assistent zum Konfigurieren von Datenquellen*.
- Klicken Sie auf das „Datenbank“-Symbol und dann auf „Weiter“.
- Im folgenden Dialogfenster wählen Sie die Schaltfläche **Neue Verbindung...** und es erscheint das Dialogfenster „Datenquelle auswählen“.
- Hier klicken Sie auf den Eintrag „Microsoft SQL Server-Datenbankdatei“. Genauso gut hätten Sie aber auch „Microsoft SQL Server“ wählen können. Wir aber gehen diesmal davon aus, dass die Datenbank nicht auf dem SQL Server installiert ist, sondern als separate Datei *Northwind.mdf* zur Verfügung steht[1].

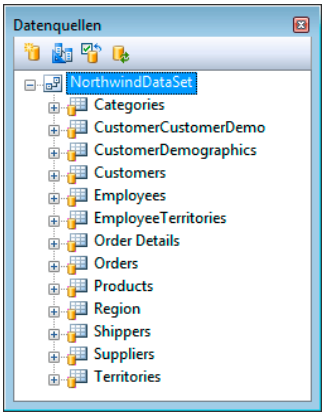


[1] Die ca. 3,7 MB große Datei *Northwind.mdf* finden Sie in den Buchbeispielen.

- Nach erfolgreichem Verbindungstest steht die Datenverbindung nun unter dem Namen *Northwind.mdf* zur Auswahl bereit.
- Anschließend werden Sie von einem Meldungsfenster befragt, ob Sie diese Datei in das Projekt kopieren wollen. Bestätigen Sie mit „Ja“, denn Sie sparen sich damit eine Menge Ärger beim Pflegen der Anwendung bzw. bei deren späterer Weitergabe.
- Im nun folgenden Dialog können Sie guten Gewissens das Häkchen setzen, damit die Verbindungszeichenfolge als „NorthwindConnectionString“ in der Anwendungskonfigurationsdatei gespeichert wird (ein Blick in die Datei *app.config* bestätigt Ihnen, dass der Eintrag in der *connectionStrings*-Sektion gelandet ist). Wenn Sie wollen, können Sie später per Code wie folgt darauf zugreifen:

```
...
 string connStr =
WindowsApplication1.Properties.Settings.Default.NorthwindConnectionString;
...
```

- Schließlich offeriert Ihnen der Assistent, nachdem er die Datenbankinformationen abgerufen hat, das Dialogfenster „Datenbankobjekte auswählen“. Hier können Sie die Tabellen, Ansichten, gespeicherten Prozeduren oder Funktionen auswählen, die Sie für Ihre konkrete Anwendung brauchen.
- Die Datenquelle steht Ihnen jetzt im *Datenquellen*-Fenster zur freien Verfügung.

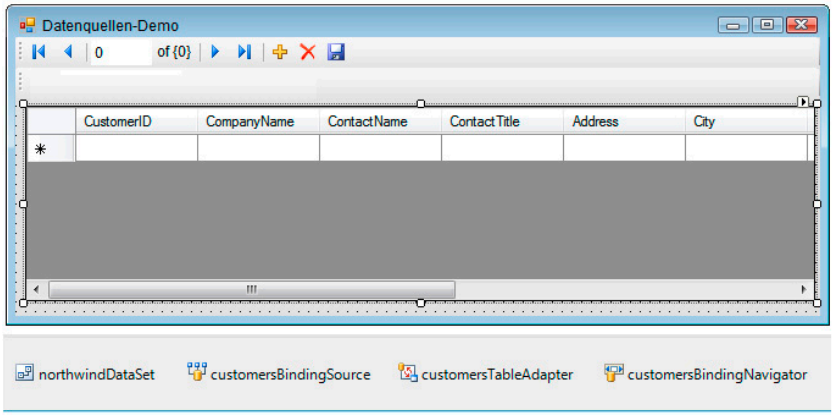


Verwenden der Datenquelle

Nachdem Sie per Drag & Drop die *Customers*-Tabelle vom *Datenquellen*-Fenster auf *Form1* gezogen haben, geschehen wundersame Dinge: Wie von Geisterhand entfaltet sich ein *DataGridView*-Datengitter, dessen Spalten bereits beschriftet sind, auf dem Formular. Am oberen Rand hat ein *BindingNavigator* Platz genommen. Im vollen Komponentenfach tummeln sich folgende Objekte:

- *northwindDataSet*
  eine Instanz des typisierten DataSets
- *customersTableAdapter*
  ein typisierter *DataAdapter* für die *Customers*-Tabelle

- *customersBindingSource*
  die Datenanbindung des Formulars
- *customersBindingNavigator*
  navigiert *customersBindingSource*



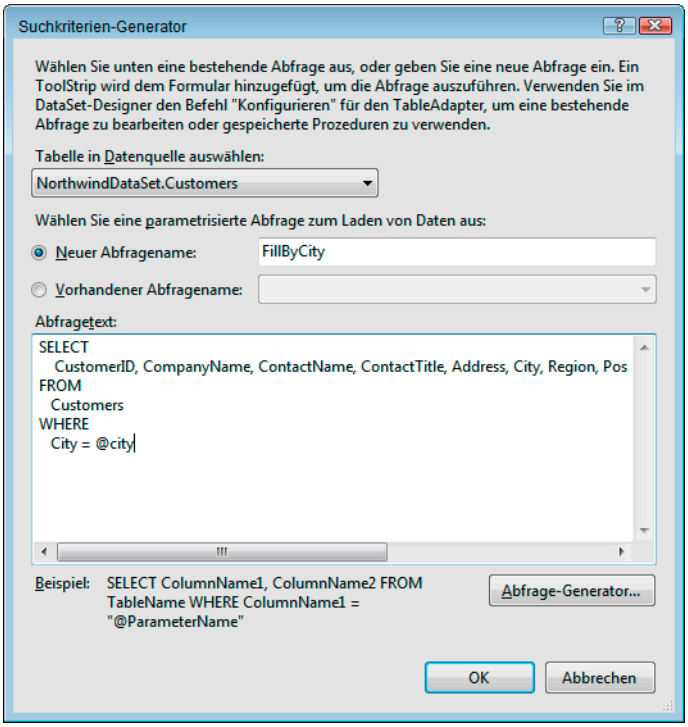
HINWEIS: Die Komponenten *northwindDataSet* und *customersTableAdapter* lassen sich im komfortablen DataSet-Designer weiterbearbeiten (Aufruf über Kontextmenü).

Test

Ohne dass Sie eine einzige Zeile Code geschrieben haben, liegt bereits eine voll funktionsfähige Anwendung vor, in der Sie nicht nur durch die Datensätze blättern können. Auch Editieren, Hinzufügen und (sofern die referenzielle Integrität nicht verletzt wird) Löschen sind möglich.

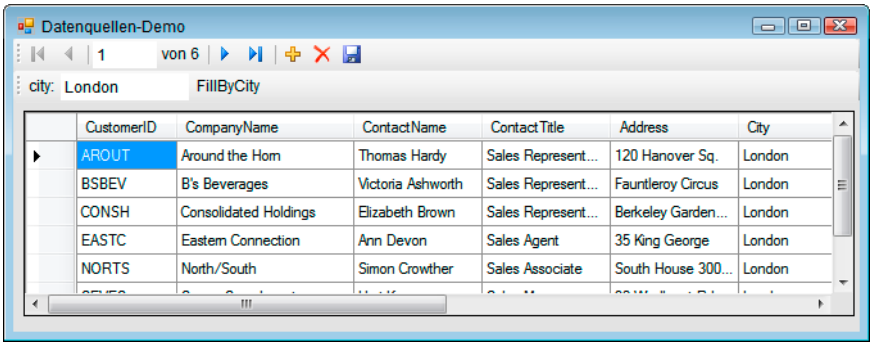
Abfragemethoden hinzufügen

Wir haben bis jetzt nur die Spitze des Eisbergs gesehen. Das Kontextmenü des *customersTableAdapter* bietet zum Beispiel auch einen Eintrag *Abfrage hinzufügen...*



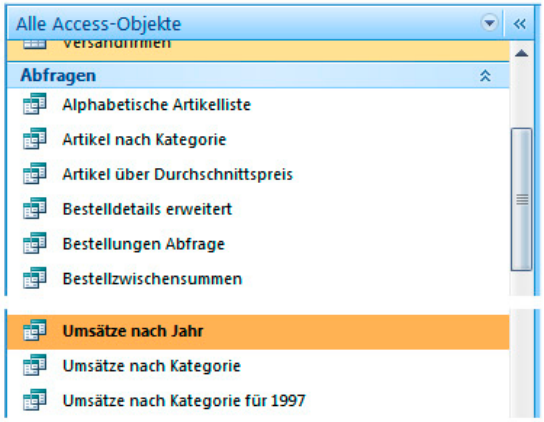
Im nachfolgenden Dialog vergeben Sie z. B. den Abfragenamen *FillByCity* und als Abfragetext den in obiger Abbildung angegebenen SQL-Befehl.

Nach dem OK wird abermals gezaubert: Unterhalb der Navigatorleiste erscheint ein automatisch generierter *ToolStrip* mit einer *TextBox* und einem *Button*. Nach Eingabe der gewünschten Stadt und Klick auf den Button „FillByCity“ sehen Sie im *DataGridView* bereits das Abfrageergebnis.

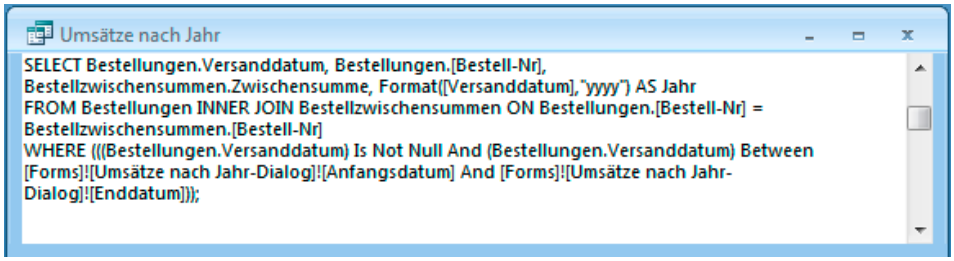


11.8.2 Eine Auswahlabfrage im DataGridView anzeigen

Die unter Microsoft Access gespeicherten Auswahlabfragen kann man quasi als Pendant zu den Stored Procedures des Microsoft SQL Servers betrachten. Öffnen Sie das Datenbankfenster von *Nordwind.mdb* und Sie sehen das vielfältige Angebot an vorbereiteten Abfragen, die Sie natürlich auch selbst um weitere ergänzen können:



Hinter jeder Auswahlabfrage verbirgt sich in der Regel eine parametrisierte SQL-SELECT-Anweisung, die Sie sich im Access-Datenbankprogramm durch Öffnen der Entwurfsansicht über den Kontextmenübefehl *SQL-Ansicht* anschauen können. Dabei finden Sie auch die zu übergebenden Parameter und deren Datentypen leicht heraus:



Oberfläche

Ein *DataGridView*, zwei *TextBox*en und ein *Button* sollen für unseren Test genügen (siehe Laufzeitansicht am Schluss).

Quellcode

```
using System.Data.OleDb;
public partial class Form1 : Form
{
...
```

```
private void button1_Click(object sender, System.EventArgs e)
{
    string connStr = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;";
    OleDbConnection conn = new OleDbConnection(connStr);

    OleDbCommand cmd = new OleDbCommand("[Umsätze nach Jahr]", conn);
    cmd.CommandType = CommandType.StoredProcedure;
```

Die Definition der beiden Parameter und das Hinzufügen zur *Parameters*-Auflistung des *Command*-Objekts:

```
    OleDbParameter parm1 = new OleDbParameter("@Anfangsdatum", OleDbType.DBDate);
    parm1.Direction = ParameterDirection.Input;
    parm1.Value = Convert.ToDateTime(textBox1.Text);
    cmd.Parameters.Add(parm1);

    OleDbParameter parm2 = new OleDbParameter("@EndDatum", OleDbType.DBDate);
    parm2.Direction = ParameterDirection.Input;
    parm2.Value = Convert.ToDateTime(textBox2.Text);
    cmd.Parameters.Add(parm2);
```

Das *Command*-Objekt wird dem Konstruktor des *DataAdapter*s übergeben. Nach dem Öffnen der *Connection* wird die Abfrage ausgeführt. Die zurückgegebenen Datensätze werden in einer im *DataSet* neu angelegten Tabelle mit einem von uns frei bestimmten Namen *Jahresumsätze* gespeichert:

```
    OleDbDataAdapter da = new OleDbDataAdapter(cmd);
    DataSet ds = new DataSet();
    try
    {
        conn.Open();
        da.Fill(ds, "Jahresumsätze");
        conn.Close();
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
```

Die Anzeige:

```
    dataGridView1.DataSource = ds;
    dataGridView1.DataMember = "Jahresumsätze";
```

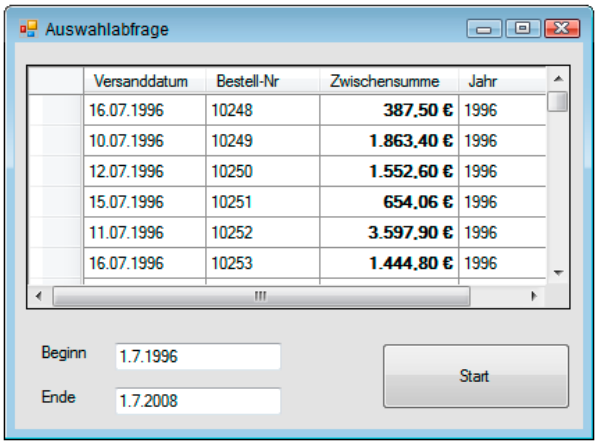
Wenigstens die Währungsspalte sollte eine ordentliche Formatierung erhalten (bei den übrigen Spalten belassen wir es bei den Standardeinstellungen):

```
  dataGridView1.Columns.Remove("Zwischensumme");
  DataGridViewTextBoxColumn tbc = new DataGridViewTextBoxColumn();
  tbc.DataPropertyName = "Zwischensumme";
  tbc.HeaderText = "Zwischensumme";
  tbc.Width = 80;
  tbc.DefaultCellStyle.Format = "c";
  tbc.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
  tbc.DefaultCellStyle.Font = new Font(dataGridView1.Font, FontStyle.Bold);
  tbc.DisplayIndex = 2;
```

```
      dataGridView1.Columns.Add(tbc);
   }
}
```

Test

Nach Eingabe sinnvoller Datumswerte dürfte sich Ihnen der folgende Anblick bieten:



11.8.3 Master-Detailbeziehungen im DataGrid anzeigen

Das „gute alte“ *DataGrid* kann mehrere Tabellen gleichzeitig verwalten, dies ist fast der einzige (wenn auch nicht unbedeutende) Vorteil gegenüber dem strahlenden Nachfolger *DataGridView*. Im vorliegenden Beispiel zeigen wir, wie man ohne viel Mehraufwand eine Darstellung von zwei verknüpften Tabellen (*Kunden* und *Bestellungen* aus der *Nordwind*-Datenbank) erreichen kann. Dabei lernen wir, wie man eine *DataRelation* erstellt und anwendet.

Oberfläche

Ein *DataGrid* und ein *Button* genügen für einen kleinen Test. Da Visual Studio das *DataGrid* aus dem Werkzeugkasten vertrieben hat, müssen wir es aus der „Mottenkiste“ wieder herausholen (Kontextmenü *Elemente auswählen...* und unter „.NET Framework-Komponenten“ suchen).

Quellcode

```
using System.Data.OleDb;

public partial class Form1 : Form
{
   ...
```

Einrichten der Verbindung zur Datenbank:

```
    private void button1_Click(object sender, System.EventArgs e)
    {
        string connStr = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;" ;
        OleDbConnection conn = new OleDbConnection(connStr);
```

Die Tabelle *Kunden* wird in das *DataSet* geladen:

```
        string selStr = "SELECT KundenCode, Firma, Kontaktperson, Telefon FROM Kunden";
        OleDbDataAdapter da = new OleDbDataAdapter(selStr, conn);
        DataSet ds = new DataSet();
        conn.Open();
        da.Fill(ds, "Kunden");
```

Die Tabelle *Bestellungen* wird geladen:

```
        selStr = "SELECT Bestellungen.BestellNr, Bestellungen.KundenCode," +
                 " Bestellungen.Bestelldatum, Bestellungen.Versanddatum" +
                 " FROM Kunden, Bestellungen WHERE (Kunden.KundenCode =
                                                    Bestellungen.KundenCode)";
        da = new OleDbDataAdapter(selStr, conn);
        da.Fill(ds, "Bestellungen");
        conn.Close();
```

Die *DataRelation* wird zum *DataSet* hinzugefügt:

```
        ds.Relations.Add("KundenBestellungen", ds.Tables["Kunden"].Columns["KundenCode"],
                                               ds.Tables["Bestellungen"].Columns["KundenCode"]);
```

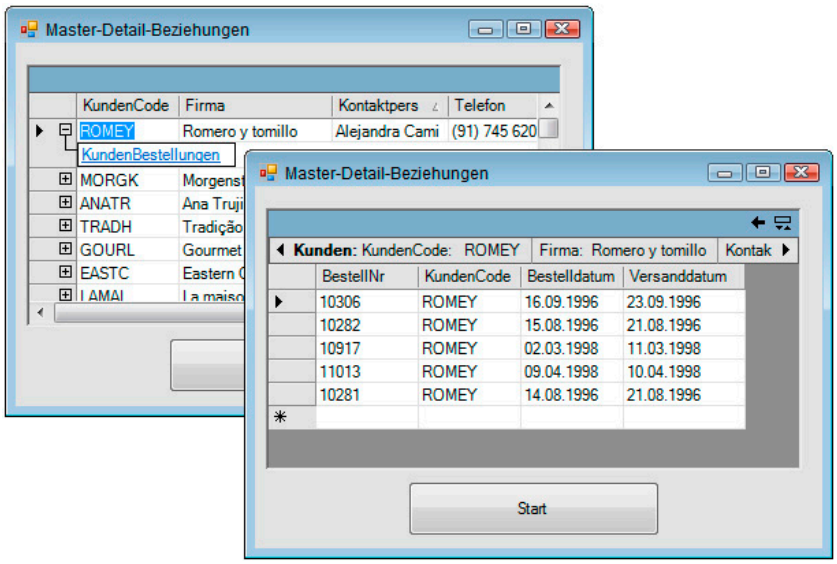
Anbinden des *DataGrid*:

```
        dataGrid1.SetDataBinding(ds, "Kunden");
    }
}
```

Test

Das *DataGrid* zeigt zunächst eine scheinbar normale Darstellung der Kundenliste. Nach dem Klick auf das Kreuzchen in der ersten Tabellenspalte können Sie die Darstellung expandieren. Nachdem Sie auf den Hotspot „KundenBestellungen“ geklickt haben, erscheinen im *DataGrid* die gewünschten Detaildatensätze.

HINWEIS: Um zur Master-Tabelle zurückzukehren, klicken Sie auf den kleinen Pfeil rechts oben in der Titelleiste der Detailansicht.



11.8.4 Datenbindung Chart-Control

Selbstverständlich beherrscht auch das neue *Chart*-Control alle Möglichkeiten der Datenbindung. Die Sorge, die Daten einzeln übergeben zu müssen, können Sie also gleich wieder vergessen.

HINWEIS: Grundsätzlich sollten Sie sich jedoch darüber im Klaren sein, welche Datenmengen Sie im Chart noch **effektiv** anzeigen können. Meist ist es sinnvoll, Daten zum Beispiel mit der bekannten SELECT TOP ...-Klausel zu filtern bzw. einzuschränken.

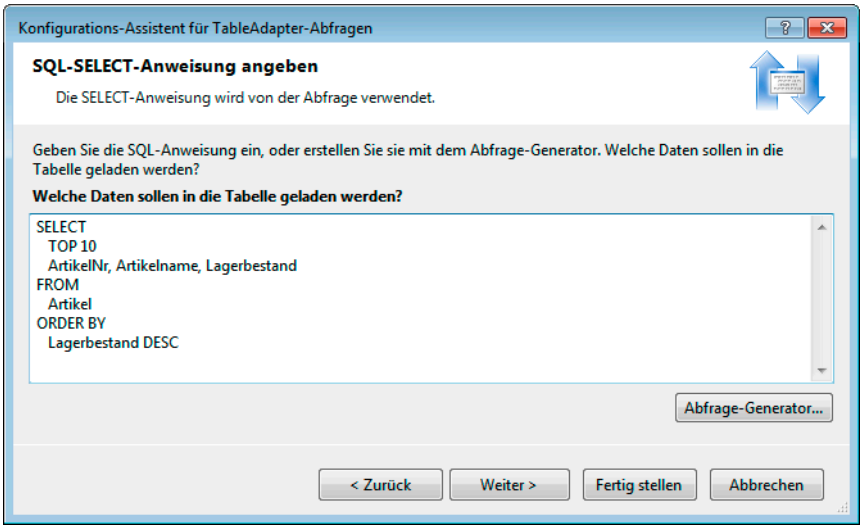
Unser kleines Beispielprogramm soll den Lagerbestand der Artikel aus der Access-Datenbank *Nordwind.mdb* in einem übersichtlichen Diagramm anzeigen. Allerdings werden Sie schnell feststellen, dass es kaum sinnvoll ist, mehrere hundert Artikel in einem Diagramm darzustellen. Aus diesem Grund beschränken wir uns auf die zehn häufigsten Artikel (Lagerbestand).

Oberfläche

Entwerfen Sie eine Windows Forms-Anwendung, der Sie zunächst die Datenbank *Nordwind.mdb* hinzufügen. Nachfolgend können Sie sich um das Erstellen einer neuen Datenquelle, basierend auf einem *DataSet*, kümmern.

HINWEIS: Für das Einbinden der Datenbank und das Erstellen des *DataSet*s sowie der übrigen datengebundenen Komponenten verweisen wir Sie an das Praxisbeispiel in Abschnitt 11.8.1, „Einrichten und Verwenden einer Datenquelle“.

Da wir uns auf die zehn häufigsten Artikel beschränken wollten, müssen wir entweder die bereits automatisch erzeugte *Fill*-Methode des erzeugten *TableAdapter*s (*ArtikelTableAdapter*) anpassen oder wir erstellen eine neue *Fill*-Methode (*FillbyTop10*) mit angepasstem SQL-Abfragestring (öffnen Sie dazu das neu erstellte Dataset):



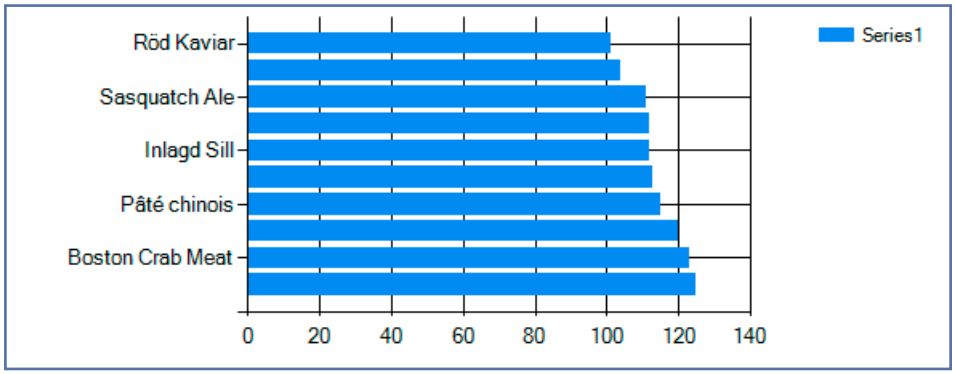
Mittels „TOP 10“ beschränken wir uns auf zehn Datensätze. Gleichzeitig beeinflussen wir mit „ORDER BY...“ die Sortierfolge, um die zehn häufigsten Artikel abzurufen.

In *Form1* finden Sie jetzt neben dem *nordwindDataSet* auch noch die *artikelBindingSource* und den *artikelTableAdapter* vor. Fügen Sie abschließend ein *Chart*-Control hinzu, um dessen Konfiguration wir uns im Folgenden kümmern wollen.

Legen Sie zunächst die *DataSource*-Eigenschaft von *Chart1* auf *artikelBindingSource* fest. Nachfolgend können wir uns um die Eigenschaft *Series* kümmern, die für die Anzeige der einzelnen Datenreihen verantwortlich ist:

- Ändern Sie zunächst mittels Assistent den Diagrammtyp (*ChartType*). Wir wählen ein horizontales Balkendiagramm (*Bar*), so können später die Artikelbeschriftungen voll ausgeschrieben werden (Y-Achse).
- Über die Eigenschaften *XValueMember* und *YValueMember* können Sie die Tabellendaten der X- bzw. Y-Achse zuordnen. Doch Vorsicht: Beim vorliegenden Diagrammtyp müssen Sie die *Artikelnamen* der Eigenschaft *XValueMember* und den *Artikelbestand* der Eigenschaft *YValueMember* zuweisen.

Mit den bereits vorgenommenen Einstellungen würde zur Laufzeit folgendes Diagramm erzeugt werden:



Wie Sie sehen, werden nicht alle Y-Werte korrekt beschriftet, ansonsten sieht das Ergebnis doch schon recht ansprechend aus.

Um die ordentliche Beschriftung unserer zehn Artikel sicherzustellen, öffnen Sie zunächst den Assistenten für die *ChartAreas*-Eigenschaft. Suchen Sie hier die Eigenschaft *Axis* und öffnen Sie den nächsten Assistenten. In der Liste der Achsen wählen Sie „X axis“ und legen für diese die Eigenschaft *Intervall* auf „1“ fest. Damit werden jetzt alle Artikel ordentlich beschriftet.

Was jetzt noch bleibt, ist das optische Aufbessern des Diagramms:

- Den Diagrammtitel können Sie über die *Titles*-Collection bestimmen. Fügen Sie mittels Assistent einen neuen Member hinzu und legen Sie dessen *Text*-Eigenschaft wie gewünscht fest (z. B. „Lagerbestand“).
- Größe, Farbe, Position können Sie über die weiteren Eigenschaften des *Titles*-Members bestimmen.
- Die Legende (aktuell „Series 1“) können Sie über die *Name*-Eigenschaft der jeweiligen Serie anpassen.
- Für mehr Eindruck sorgen Sie über die 3D-Darstellung (Collection *ChartAreas*, *ChartArea1*, Eigenschaft *Enable3D*). Die Ansicht können Sie nachfolgend über *Rotation* und *PointDepth* noch wie gewünscht anpassen.
- Weitere optische „Spielereien“ wollen wir an dieser Stelle nicht besprechen, wir wollen ja Ihre Fähigkeiten nicht unterschätzen :-).

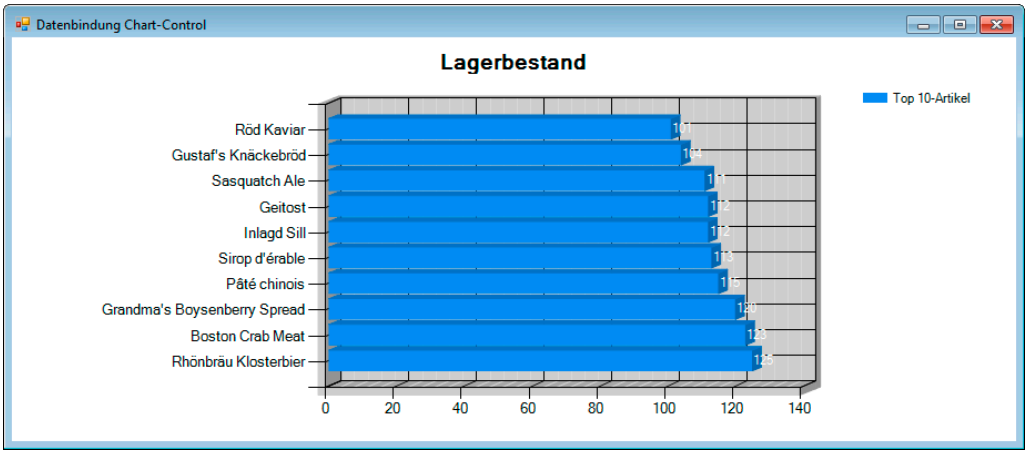
Quellcode

Der Datenquellen-Assistent hatte schon eine erste Quellcodezeile im *Form_Load*-Event generiert, mit der die Daten per *TableAdapter* geladen werden. Ändern Sie diese Anweisung und ersetzen Sie die *Fill*- mit der *FillByTop10*-Methode, damit auch nur zehn und nicht alle Datensätze geladen werden:

```
private void Form1_Load(object sender, EventArgs e)
{
    this.artikelTableAdapter.FillByTop10(this.nordwindDataSet.Artikel);
}
```

Test

Nach dem Start präsentiert sich Ihnen bereits das fertige Diagramm.



12 Erweiterte Grafikausgabe

Wie es die Kapitelüberschrift schon verrät, werden wir Sie im Folgenden mit den **erweiterten** Möglichkeiten der Grafikausgabe unter .NET traktieren. Grundkenntnisse der Grafikprogrammierung, siehe Kapitel 9, sind für die Lektüre dieses Kapitels unerlässlich.

Die wichtigsten Themen im Überblick:

- die GDI+-Transformationen
- Low-Level-Grafikzugriff
- erweiterte Techniken (Alpha-Blending, Animationen ...)
- 3D-Grafik unter .NET
- die Verwendung von GDI-Funktionen unter .NET

12.1 Transformieren mit der Matrix-Klasse

Die bereits in Abschnitt 9.3 über die Seitenkoordinaten vorgestellten Transformationen lassen sich auch mithilfe einer Transformationsmatrix realisieren. Die Mathematiker unter den Lesern werden das zu schätzen wissen. Dass es nicht bei den beschriebenen Transformationen bleiben muss und dass vieles auch einfacher realisierbar ist, zeigt der folgende Abschnitt.

12.1.1 Übersicht

Ausgangspunkt aller Überlegungen ist zunächst die Klasse *Matrix*, die vom Namespace *System.Drawing.Drawing2D* bereitgestellt wird. Hierbei handelt es sich intern um eine 3x3 Matrix, auch wenn Sie nur Zugriff auf die beiden ersten Spalten haben (die anderen Werte sind konstant).

$$\begin{vmatrix} m11 & m12 & 0 \\ m21 & m22 & 0 \\ dx & dy & 1 \end{vmatrix}$$

Das Füllen der Matrix können Sie bereits mithilfe des Konstruktors bewerkstelligen, übergeben Sie einfach die Werte *m11* bis *dy*.

Beispiel 12.1: Erzeugen einer neuen Matrix

C#

```
Matrix m = new Matrix(1, 0,
                      0, 1,
                      0, 0);
```

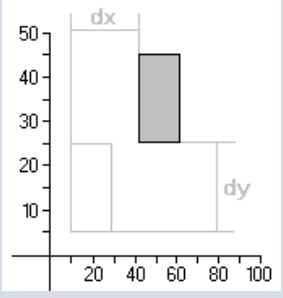
HINWEIS: Zur besseren Übersicht haben wir im Listing noch Zeilenumbrüche hinzugefügt.

Alternativ können Sie über die *Elements*-Auflistung auf die einzelnen Elements der Matrix zugreifen, es handelt sich jedoch **nicht** um ein zweidimensionales Array.

Die Elemente *dx* und *dy* stehen über die Eigenschaften *OffsetX* und *OffsetY* zur Verfügung.

12.1.2 Translation

Eine Verschiebung bzw. Translation realisieren Sie durch eine Änderung der *dx*-, *dy*-Werte.



$$x' = x + dx$$
$$y' = y + dy$$

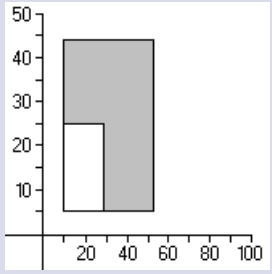
... als Transformationsmatrix:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx & dy & 1 \end{vmatrix}$$

Alternativ können Sie die Methode *Translate* auf eine initialisierte Matrix anwenden. Übergeben Sie dazu die Werte für *dx* und *dy*.

12.1.3 Skalierung

Eine Größenänderung bzw. Skalierung erreichen Sie über die Werte *m11* bzw. *m22*:



$$x' = x \cdot m11$$
$$y' = y \cdot m22$$

... als Transformationsmatrix:

$$\begin{vmatrix} m11 & 0 & 0 \\ 0 & m22 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Gleiches erreichen Sie über die Methode *Scale*.

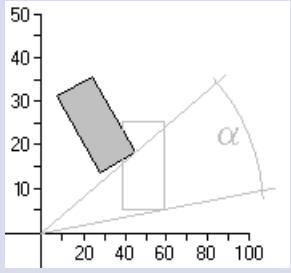
Beispiel 12.2: Verwendung von *Scale*

C#

```
Matrix m = new Matrix(1, 0, 0, 1, 0, 0);
m.Scale(2, 0);
```

12.1.4 Rotation

Auch an das Drehen des Koordinatensystems haben die .NET-Entwickler gedacht, allerdings genügt nicht das einfache Einsetzen des gewünschten Drehwinkels, sondern Sie müssen schon etwas rechnen:



$$x' = (x \cdot \cos\alpha) - (y \cdot \sin\alpha)$$
$$y' = (y \cdot \sin\alpha) - (y \cdot \cos\alpha)$$

... als Transformationsmatrix:

$$\begin{vmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Sie vermuten es sicher schon, mithilfe der Methode *Rotate* bietet sich ein wesentlich einfacherer Weg zum Drehen des Koordinatensystems.

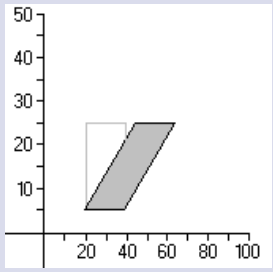
Beispiel 12.3: Verwendung von *Rotate*

C#

```
Matrix m = new Matrix(1, 0, 0, 1, 0, 0);
m.Rotate(45);       // Gradangabe
```

12.1.5 Scherung

Eine Scherung in x- bzw. y-Richtung erreichen Sie über die folgende Transformationsmatrix:



$$x' = x + (m12 \cdot y)$$
$$y' = y + (m21 \cdot x)$$

... als Transformationsmatrix:

$$\begin{vmatrix} 1 & m12 & 0 \\ m21 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Beispiel 12.4: Auch hier sind Sie mit der Methode *Shear* besser bedient.

C#

```
Matrix m = new Matrix(1, 0, 0, 1, 0, 0);
m.Shear(0.1f, 0);
```

12.1.6 Zuweisen der Matrix

Die Bearbeitung der Transformationsmatrix ist kein Selbstzweck, über die Eigenschaft *Transform* können Sie die mit obigen Methoden bearbeitete Transformationsmatrix einem *Graphics*-Objekt zuweisen. Nachfolgende Zeichenoperationen werden nun mit dieser Matrix verarbeitet.

Beispiel 12.5: Verschieben (20,10) und Skalieren (2,2) der Ausgabe

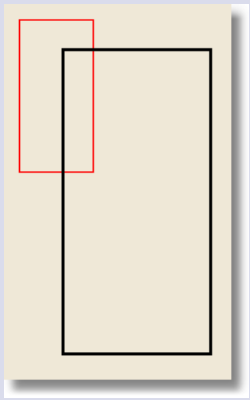
C#

```
Graphics g = this.CreateGraphics();
Matrix   m = new Matrix();
m.Translate(20, 10);
m.Scale(2, 2);
g.DrawRectangle(Pens.Red, 10, 10, 50, 100);
g.Transform = m;
g.DrawRectangle(Pens.Black, 10, 10, 50, 100);
```

oder bei direktem Bearbeiten der Matrix:

```
Graphics g = this.CreateGraphics();
Matrix m = new Matrix(2, 0, 0, 2, 20, 10);
g.DrawRectangle(Pens.Red, 10, 10, 50, 100);
g.Transform = m;
g.DrawRectangle(Pens.Black, 10, 10, 50, 100);
```

Ergebnis



12.2 Low-Level-Grafikmanipulationen

Möchten Sie Grafiken nicht nur anzeigen, sondern auch verändern, bieten sich auf den ersten Blick die *GetPixel-/SetPixel*-Methoden an. Diese ermöglichen den Zugriff auf die einzelnen Bildpunkte einer **Bitmap**[1], es ist demnach kein Problem, zum Beispiel eine Farbe durch eine andere auszutauschen.

Doch wer bereits erste Schritte auf diesem Gebiet unternommen hat, wird schnell enttäuscht sein. Das Auslesen und Setzen der einzelnen Punkte verbraucht bei komplexeren Aufgaben so viel Zeit, dass dies wohl kaum einem Programmnutzer zumutbar ist, es sei denn, man möchte ihn ärgern.

[1] Dies ist wichtig, Sie müssen eine Pixelgrafik in das *Image* geladen haben.

Beispiel 12.6: Drehen einer Bitmap[2] (in *picturebox1.Image*) um 90°

C#

```
private void button1_Click(object sender, EventArgs e)
{
   int x, y;
   Bitmap b1, b2;
   Color mypix;

   b1 = (Bitmap) pictureBox1.Image;
   b2 = new Bitmap(b1.Height, b1.Width);
   for(x = 0; x < b1.Width; x++)
      for (y = 0; y < b1.Height; y++)
      {
        mypix = b1.GetPixel(x, y);
        b2.SetPixel(b1.Height - y - 1, x, mypix);
      }
    pictureBox1.Image = b2;
    pictureBox1.Refresh();
}
```

Als Alternative bietet sich dem begeisterten GDI-Programmierer die Arbeit mit den *Device Independent Bitmaps* (DIB) an. Doch die Verwendung der entsprechenden Funktionen erfordert nicht nur eine genaue Kenntnis des GDI, sondern auch unnötigen Schreibaufwand, muss doch die DIB wieder in das Bitmap-Format zurückkonvertiert werden. Vom zusätzlichen Speicherverbrauch wollen wir an dieser Stelle gar nicht erst sprechen.

Die Zauberworte für unsere Probleme heißen *LockBits* und *Scan0*. Dabei handelt es sich um Eigenschaften des *Bitmap*-Objekts. Mit *LockBits* wird eine Bitmap im Arbeitsspeicher für uns gesperrt, gleichzeitig können wir die Bitmap in einen für uns günstigen Datentyp umwandeln. Der eigentliche Clou ist *Scan0*, ein Pointer auf das erste Bitmap-Byte.

HINWEIS: Auch wenn es auf den ersten Blick so scheinen mag, *Scan0* ist kein direkter Ersatz für *GetPixel* und *SetPixel*. Wie die Bitmap-Daten aufgebaut sind, wie viele Spalten und Zeilen es gibt, bleibt unberücksichtigt. Ein Fehler bei der Arbeit mit diesem Pointer führt meist zu einem Programmabsturz.

12.2.1 Worauf zeigt Scan0?

Die allgemein Antwort lautet: auf das erste Byte der Bitmap. Wie die folgenden Daten aufgebaut sind und aus wie vielen Bytes sie bestehen, wird durch das Bitmap-Format bestimmt. Dieses können Sie mit *LockBits* direkt angeben:

Syntax:

```
BitmapData LockBits(Rectangle rect, ImageLockMode flags, PixelFormat format);
```

[2] Ja, ja, es gibt dafür eine extra Methode, an dieser Stelle geht es aber nur um das Grundprinzip!

Übergeben werden der gewünschte Ausschnitt der Bitmap (im Zweifel alles), der Lockmode (meist *ReadWrite*) und das gewünschte Bitmap-Format.

C# unterstützt unter anderem folgende wichtige Bitmap-Formate:

Format	Beschreibung
Format1bppIndexed	Schwarz-Weiß-Bilder, bei denen jedes Pixel durch ein Bit dargestellt wird
Format4bppIndexed	Bilder mit 16 Farben, ein Byte stellt somit die Informationen für zwei benachbarte Pixel zur Verfügung.
Format8bppIndexed	Bilder dieses Typs können 256 Farben darstellen. Damit entspricht ein Byte auch einem Pixel. Doch freuen Sie sich nicht zu früh, es handelt sich nicht um den direkten Farbwert, sondern nur um den Index in einer getrennt gespeicherten Farbpalette.
Format16bppRgb555, *Format16bppRgb565*	Bei diesem Format werden mit jeweils 5 bzw. 6 Bit für die drei Grundfarben die Farbwerte dargestellt bzw. abgespeichert. Das interne Format: ▪ pf15bit:0rrrrrgggggbbbbb ▪ pf16bit: rrrrrgggggbbbbb Dass die Arbeit mit derart verschachtelten Daten nicht unbedingt einfach ist, dürfte schnell ersichtlich sein.
Format24bppRgb	Das Wunschformat jedes Grafikprogrammierers: Jeder Punkt wird mit drei Bytes (RGB) zu je 8 Bit dargestellt. Der Zugriff auf derartige Bitmaps ist relativ problemlos realisierbar, beachten Sie jedoch, dass die Bildzeilen immer auf Vielfache von vier aufgerundet werden.
Format32bppArgb	Noch etwas schneller lassen sich 32-Bit-Bilder bearbeiten. Dies wird durch die bessere Speicherausrichtung (4 Byte) erreicht. Das vierte Byte hat normalerweise keine Bedeutung, kann aber für Transparenzinformationen genutzt werden. Der zusätzlich benötigte Speicher spielt heute wohl kaum noch eine Rolle.

12.2.2 Anzahl der Spalten bestimmen

Eigentlich müsste die Frage anders gestellt werden, da meist nicht die Anzahl der Bildpunkte, sondern die Anzahl der nötigen Bytes von Bedeutung ist. Die Pixelanzahl können Sie mit *Bitmap.Width* bestimmen, die Byte-Anzahl berechnet sich aus den jeweiligen Bildformaten, wie sie im vorhergehenden Abschnitt vorgestellt wurden.

32-Bit-Bild, Bitmap.Width = 300

*4 * Bitmap.Width = 1200 Bytes/Zeile*

Doch Vorsicht: Ein 24-Bit-Bild mit einer Breite von 299 Pixeln hat nicht etwa

*3 * Bitmap.Width = 897 Bytes/Zeile*

sondern

*((3 * Bitmap.Width + 3) DIV 4) * 4 = 900 Bytes/Zeile*

HINWEIS: Die Bitmaps werden in jeder Zeile auf das Vielfache von 4 Bytes aufgefüllt!

Aus diesem Grund finden Sie auch eine zusätzliche Eigenschaft *Stride*, die uns die tatsächliche Länge einer Bitmap-Zeile in Pixeln liefert.

Beispiel 12.7: Bestimmen der Anzahl von Füll-Bytes (24-Bit-Bitmap)

C#

```
using System.Drawing.Imaging;
...
    Bitmap b;
    BitmapData bmpData;
    int lineoffs;

    b = (Bitmap) pictureBox1.Image;
    bmpData = b.LockBits(new Rectangle(0, 0, b.Width, b.Height),
ImageLockMode.ReadWrite,
                         PixelFormat.Format24bppRgb);
    lineoffs = bmpData.Stride - b.Width * 3;
```

HINWEIS: Dieser Offset muss nach der Abarbeitung einer Zeile zum Pointer hinzuaddiert werden, um wieder auf das erste Byte der folgenden Zeile zu zeigen.

12.2.3 Anzahl der Zeilen bestimmen

Diese Antwort ist schnell gegeben. Über die Eigenschaft *Bitmap.Height* steht Ihnen direkt der Wert zur Verfügung.

12.2.4 Zugriff im Detail (erster Versuch)

Folgende Reihenfolge müssen Sie beim direkten Zugriff auf die einzelnen Bitmap-Bytes beachten:

- Bitmap mit *LockBits* sperren,
- mit *Scan0* einen Pointer auf das erste Byte ermitteln,
- über *Marshal.Read...* die gewünschten Bytes/Integers etc. lesen,
- über *Marshal.Write...* die gewünschten Bytes/Integers schreiben,
- die Bitmap mit *UnlockBits* freigeben und
- eventuell die zugehörige *PictureBox* mit *Refresh* aktualisieren.

Den wohl wichtigsten Punkt dürfen wir natürlich auch nicht vergessen:

HINWEIS: Auch wenn die Konstanten *Format24bppRgb* oder *Format32bppArgb* heißen, lassen Sie sich nicht ins Boxhorn jagen! Die Bytes liegen immer in der Reihenfolge Blau-Grün-Rot bzw. Blau-Grün-Rot-Alpha im Speicher!

Beispiel 12.8: Alle Pixel der Grafik sollen auf Schwarz gesetzt werden (24-Bit-Bitmap).

C#

```
using System.Drawing.Imaging;
using System.Runtime.InteropServices;
...

    private void Button4_Click(object sender, EventArgs e)
    {
      Bitmap b = PictureBox1.Image as Bitmap;
      int x, y, offset;
      BitmapData bmpData;
      Byte p;
      IntPtr ptr;
```

Sperren der Bitmap:

```
      bmpData = b.LockBits(new Rectangle(0, 0, b.Width, b.Height),
                           ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);
```

Pointer ermitteln:

```
      ptr = bmpData.Scan0;
      offset = 0;
      for (y=0; y < b.Height -1; y++ )
      {
        for (x=0; x < b.Width *3 -1; x++ )
          {
           p = 0;
```

Schreiben in den gewünschten Speicherbereich (alle Farbwerte = 0):

```
           Marshal.WriteByte(ptr, offset, p);
```

Offset für *Marshal.Write* setzen:

```
           offset += 1;
          }
      }
```

Freigabe der Bitmap und *PictureBox* aktualisieren:

```
      b.UnlockBits(bmpData);
      PictureBox1.Refresh();
    }
```

Haben Sie sich zu einem Test hinreißen lassen, wird es Ihnen sicher nicht anders als den Autoren ergangen sein. Nach quälenden Sekunden ist endlich das Bild schwarz. Grafikprogrammierer der ersten Stunde (Visual Studio 2002/2003) werden sich jetzt verwundert die Augen reiben, war doch dort dasselbe Beispiel ausreichend schnell (im Millisekunden-Bereich).

Ursache sind die mittlerweile quälend langsamen Zugriffe per *Marshal*-Objekt.

12.2.5 Zugriff im Detail (zweiter Versuch)

Nach unserem desaströsen ersten Versuch wollen wir es jetzt besser machen. Dazu bietet sich in C# die Verwendung von „unsicherem" Code an. Hier können wir mit Pointern schalten und walten, wie wir wollen, und natürlich auch reichlich Fehler produzieren.

HINWEIS: Gekennzeichnet werden unsichere Codeabschnitte mit dem *unsafe*-Bezeichner.

Beispiel 12.9: Wir versuchen uns mit dem gleichen Beispiel wie im vorherigen Abschnitt.

C#

```
            int x, y;
            BitmapData bmpData;

            bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                                   ImageLockMode.ReadWrite,
 PixelFormat.Format24bppRgb);
            unsafe
            {
```

Hier ermitteln wir einen Byte-Pointer auf das erste Byte der Bitmap:

```
                byte* ptr = (byte*)bmpData.Scan0;
                int lineoffs = bmpData.Stride - bmp.Width * 3;
```

Für alle Zeilen und Spalten:

```
                for (y = 0; y < bmp.Height - 1; y++)
                {
                    for (x = 0; x < bmp.Width * 3 - 1; x++)
                    {
```

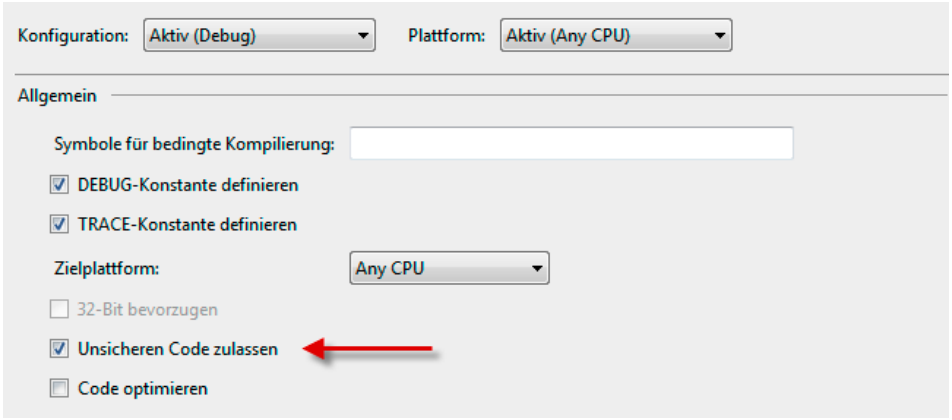
Wert setzen und Pointer inkrementieren:

```
                        ptr[0] = 0;
                        ptr++;
                    }
```

Am Zeilenende den Offset addieren:

```
                ptr += lineoffs;
            }
        }
        bmp.UnlockBits(bmpData);
```

HINWEIS: Damit Sie das Beispiel kompilieren können, muss in den Projektoptionen auch das Kompilieren von unsicherem Code erlaubt werden.



Ein zweiter Test unseres optimierten Codes dürfte zwar schon etwas besser verlaufen, aber rund acht Sekunden sind trotzdem keine berauschende Zeit für die paar Pixel.

Weiter optimieren

Ein Blick auf unseren Code zeigt, dass wir eigentlich viel zu viele Berechnungen in den Schleifen ausführen (x, y berechnen, Offset addieren). Diesem Missstand wollen wir nun abhelfen. Um die unselige Offset-Rechnerei zu vermeiden, erzeugen wir einfach eine 32-Bit-Grafik. Damit entfällt auch die Notwendigkeit, zwischen Zeilen und Spalten zu unterscheiden, wir können die geschachtelten Schleifen auflösen und der Pointer kann über den gesamten Speicherbereich iterieren:

```
BitmapData bmpData;
bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                       ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
unsafe
{
   byte* ptr = (byte*)bmpData.Scan0;
   int size = bmp.Height * bmp.Width * 4;  // 4 Bytes pro Pixel
   for (int i = 0; i < size; i++)
       ptr[i] = 0;
}
bmp.UnlockBits(bmpData);
```

Endlich dürfte die Ausführungsgeschwindigkeit auch den letzten Nörgler zufriedenstellen, 16 Millisekunden für die Beispielgrafik sind doch schon ganz gut – oder?

HINWEIS: Wer noch mehr mit der Zeit geizt, der kann auch einen Integer-Pointer inkrementieren (drei Zugriffe weniger pro Pixel).

Damit wird uns dieser Lösungsansatz auch für die weiteren Beispiele als Vorlage dienen.

Beispiel 12.10: Nur die Farbe Blau eines Pixels soll bearbeitet werden.

C#

```
using System.Drawing.Imaging;
using System.Runtime.InteropServices;
...
        public static void NurBlau(Bitmap bmp)
        {
            BitmapData bmpData;
            Byte blau;
            bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                        ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
            unsafe
            {
                byte* ptr = (byte*)bmpData.Scan0;
                int size = (bmp.Height) * (bmp.Width);
                for (int i = 0; i < size; i++)
                {
```

Achtung, der Blauwert ist immer das erste der vier Bytes:

```
                    blau = ptr[0];  // [1] = grün; [2] = rot
                    ptr[0] = 0;
                    ptr += 4;
                }
            }
            bmp.UnlockBits(bmpData);
        }
```

Das soll für einen ersten Eindruck genügen, die folgenden Beispiele zeigen an verschiedenen Aufgabenstellungen die Möglichkeiten, die Ihnen mit *Scan0* offen stehen.

HINWEIS: Ein Testprogramm für diese Funktionen finden Sie in den Beispieldaten zum Buch!

Im Folgenden wollen wir uns damit beschäftigen, das Aussehen der Bilder, das heißt die einzelnen Farbwerte, zu verändern.

12.2.6 Invertieren

Eine der einfachsten Operationen ist das Invertieren einer Bitmap, die einzelnen RGB-Werte brauchen nur negiert zu werden, das heißt, der Farbwert ist von 255 abzuziehen.

Beispiel 12.11: Eine Bitmap wird invertiert.

C#

```
        public static void Invert(Bitmap bmp)
        {
            BitmapData bmpData;
            bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                                   ImageLockMode.ReadWrite,
PixelFormat.Format32bppRgb);
            unsafe
            {
                byte* ptr = (byte*)bmpData.Scan0;
                int size = bmp.Height * bmp.Width * 4;
                for (int i = 0; i < size; i++)
```

HINWEIS: Wir verwenden *xor 255*, das ist noch mal etwas schneller als die Subtraktion.

```
                    ptr[i] = (byte)(ptr[i]^255);
                //    oder auch
                //    ptr[i] = (Byte)(255 - ptr[i]);
            }
            bmp.UnlockBits(bmpData);
        }
```

Ergebnis

12.2.7 In Graustufen umwandeln

Beim Umwandeln einer Farbgrafik in ein Graustufenbild werden die einzelnen Farben entsprechend ihrer Leuchtkraft bewertet und daraus ein Graustufenwert (8 Bit) berechnet. Dieser Wert wird nachfolgend allen drei Farbkanälen zugewiesen.

Beispiel 12.12: Eine farbige Bitmap wird in ein Graustufenbild transformiert.

C#

```
        public static void Grey(Bitmap bmp)
        {
            BitmapData bmpData;

            bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                                   ImageLockMode.ReadWrite,
 PixelFormat.Format32bppRgb);
            unsafe
            {
                byte* ptr = (byte*)bmpData.Scan0;
                int size = bmp.Height * bmp.Width;
                for (int i = 0; i < size; i++)
                {
```

Farbwerte auslesen:

```
                    Byte blau = ptr[0];
                    Byte grün = ptr[1];
                    Byte rot = ptr[2];
```

Grauwert berechnen:

```
                    Byte grau = (Byte) ((77 * blau + 151 * grün + 28 * rot) / 256);
```

Grauwert in die RGB-Bytes eintragen:

```
                    ptr[0] = ptr[1] = ptr[2] = grau;
```

Und Sprung auf das nächste Pixel:

```
                    ptr += 4;
                }
            }
            bmp.UnlockBits(bmpData);
        }
```

Alternativ können Sie auch diese Anweisung verwenden:

```
 ptr[0] = ptr[1] = ptr[2] = (Byte)((77 * ptr[0] + 151 * ptr[1] + 28 * ptr[2]) / 256);
```

12.2.8 Heller/Dunkler

Um ein Bild aufzuhellen oder dunkler zu machen, genügt es, dass zu jedem Wert eine Konstante addiert wird. Um Werteüberläufe zu verhindern, müssten wir entweder bei jedem Wert abfragen, ob das Berechnungsergebnis den Wertebereich (255) überschreitet, oder wir legen gleich ein Array an, in dem für jeden der möglichen 256 Werte der neue Wert gespeichert ist. Insbesondere bei großen Bildern können Sie so wertvolle Sekunden sparen, da nur noch der Wert aus dem Array ausgelesen werden muss (LUT = Look-Up-Table).

Beispiel 12.13: Die Helligkeit einer Bitmap wird geändert.

C#

Normieren auf den Bereich 0... 255:

```
private static byte normiere(int Value)
      {
          if (Value < 0) return 0;
          if (Value > 255) return 255;
          return (byte)Value;
        }
```

Die Konvertierungsfunktion (*Value* = Änderung der Helligkeit):

```
        public static void Brightness(Bitmap bmp, short Value)
        {
```

Zunächst die LUT berechnen:

```
            Byte[] ar = new Byte[256];
            for (int i = 0; i < 256; i++)
                ar[i] = normiere(i + Value);
            BitmapData bmpData;
            bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                                   ImageLockMode.ReadWrite,
PixelFormat.Format32bppRgb);
            unsafe
            {
                byte* ptr = (byte*)bmpData.Scan0;
                int size = bmp.Height * bmp.Width;
                for (int i = 0; i < size; i++)
                {
```

Hier lesen wir einfach die Werte aus der LUT aus:

```
                    ptr[0] = ar[ptr[0]];
                    ptr[1] = ar[ptr[1]];
                    ptr[2] = ar[ptr[2]];
                    ptr += 4;
                }
            }
            bmp.UnlockBits(bmpData);
        }
```

Ergebnis

Übergeben Sie der Funktion einen positiven oder negativen Wert, um das Bild aufzuhellen oder abzudunkeln.

12.2.9 Kontrast

Um den Kontrast eines Bilds zu erhöhen, normieren wir zunächst die Farbwerte, indem wir diese in einen Integerwert umwandeln und 128 abziehen. Den resultierenden Wert multiplizieren wir mit einem konstanten Faktor, nachfolgend wird die Normierung durch Addition von 128 wieder aufgehoben. Da wir die Gleitkomma-Operationen nicht für jeden Pixel ausführen möchten (Performance!), verwenden wir wieder ein Array (LUT), in welchem wir die Farbwerte vorberechnen.

Beispiel 12.14: Kontrast einer Bitmap verändern

C#

```
        public static void Contrast(Bitmap bmp, Single Value)
        {
            Byte[] ar = new Byte[256];
            Value = 1 + Value / 100;
```

Zunächst die LUT berechnen:

```
            for (int i = 0; i < 256; i++)
                ar[i] = normiere((int)((i - 128) * Value) + 128);

            BitmapData bmpData;
            bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                                   ImageLockMode.ReadWrite,
PixelFormat.Format32bppRgb);
            unsafe
            {
                byte* ptr = (byte*) bmpData.Scan0;
                int size = bmp.Height * bmp.Width;
```

Werte aus der LUT auslesen und zuweisen:

```
                for (int i = 0; i < size; i++)
                {
                    ptr[0] = ar[ptr[0]]; ptr[1] = ar[ptr[1]]; ptr[2] = ar[ptr[2]];
                    ptr += 4;
                }
            }
            bmp.UnlockBits(bmpData);
        }
```

Ergebnis

12.2.10 Gamma-Wert

Möchten Sie den Gamma-Wert eines Bilds anpassen, ist die Rechnerei schon etwas aufwendiger. Auch hier hilft nur eine LUT weiter, sonst ist die Laufzeit nicht zu verantworten.

Beispiel 12.15: Der Gamma-Wert einer Bitmap wird angepasst.

C#

```
        public static void Gamma(Bitmap bmp, double Value)
        {
            Byte[] ar = new Byte[256];
```

Zunächst die LUT berechnen:

```
            for (int i = 0; i < 256; i++)
                ar[i] = (byte)Math.Min(255, (int)((255.0 *
                          Math.Pow(i / 255.0, 1.0 / Value)) + 0.5));
            BitmapData bmpData;
            bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                      ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
            unsafe
            {
                byte* ptr = (byte*)bmpData.Scan0;
                int size = bmp.Height * bmp.Width;
```

Auf alle drei Farbebenen anwenden:

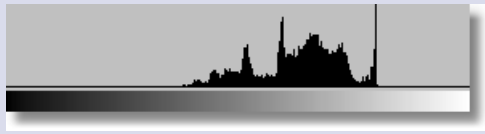
```
                for (int i = 0; i < size; i++)
                {
                    ptr[0] = ar[ptr[0]];
                    ptr[1] = ar[ptr[1]];
                    ptr[2] = ar[ptr[2]];
                    ptr += 4;
                }
            }
            bmp.UnlockBits(bmpData);
        }
```

12.2.11 Histogramm spreizen

Haben Sie Fotos mit falscher Belichtung aufgenommen, wird der verfügbare Dynamikumfang (0...255) für die einzelnen Pixel nicht voll ausgeschöpft. Die Folge sind laue Farben, graue Bereiche, kein echtes Weiß und kein echtes Schwarz.

Beispiel 12.16: Bild mit zugehörigem Histogramm

Ergebnis



Wie Sie sehen, ist links und rechts im Histogramm noch reichlich „Platz“, der nicht genutzt wird.

HINWEIS: Durch das Spreizen des Histogramms wird der Dynamikumfang wieder erweitert, das „Grau in Grau“ verschwindet.

Beispiel 12.17: Obiges Bild nach dem Spreizen des Histogramms

Ergebnis

Beispiel 12.18: Histogramm spreizen

C#

Die Umsetzung wird diesmal etwas aufwendiger:

```
public static void AutoAdjust(Bitmap bmp)
{
```

Unsere LUT wird für alle drei Farbebenen benötigt, deshalb die zweite Dimension:

```
    int[,] ar = new int[256, 3];
```

Beachten Sie auch den Wertebereich, diesen brauchen wir, wenn in der LUT zunächst das Histogramm abgespeichert wird.

```
    BitmapData bmpData;
    bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                           ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
    unsafe
    {
        byte* ptr = (byte*)bmpData.Scan0;
        int size = bmp.Height * bmp.Width;
```

Erster Schritt ist das Erzeugen eines Histogramms, das heißt, wir zählen die Häufigkeit bestimmter Farbwerte. Dazu nutzen wir gleich die LUT, die Größe stimmt ja überein:

```
        for (int i = 0; i < size; i++)
        {
            ar[ptr[0], 0]++;  // blau
            ar[ptr[1], 1]++;  // grün
            ar[ptr[2], 2]++;  // rot
            ptr += 4;
        }
```

Wir bearbeiten alle Farbkanäle:

```
                for (int c = 0; c < 3; c++)
                {
```

Das Minimum (kleinster verwendeter Farbwert) im Histogramm suchen:

```
                    int i = 0;
                    while (ar[i, c] == 0) i++;
                    int min = i;
```

Das Maximum (größter verwendeter Farbwert) suchen:

```
                    i = 255;
                    while (ar[i, c] == 0) i--;
                    int max = i;
```

Die nötige Spreizung berechnen:

```
                    Single scale = 255 / (Single)(max - min);
```

Die passende LUT für den Farbkanal ermitteln:

```
                    for (int j = 0; j < 256; j++)
                        ar[j, c] = normiere((int)((j - min) * scale));
                }
```

Zeiger auf das erste Byte setzen:

```
                ptr = (byte*)bmpData.Scan0;
```

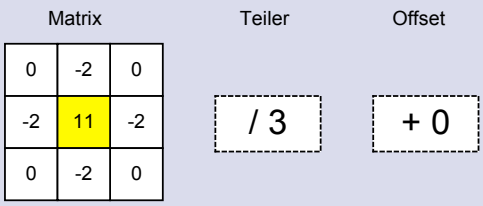
Alle Farbkanäle mit der LUT verarbeiten:

```
                for (int i = 0; i < size; i++)
                {
                    ptr[0] = (byte)ar[ptr[0], 0];  // blau
                    ptr[1] = (byte)ar[ptr[1], 1];  // grün
                    ptr[2] = (byte)ar[ptr[2], 2];  // rot
                    ptr += 4;
                }
            }
            bmp.UnlockBits(bmpData);
        }
```

12.2.12 Ein universeller Grafikfilter

Das Grundprinzip derartiger Verfahren ist die Verarbeitung der Farbwerte aus den umliegenden Pixeln als gewichtete Summe der Einzelwerte (Multiplizieren der Farbwerte mit dem jeweiligen Filtermatrixwert und nachfolgendes Addieren), das nachfolgende Teilen sowie das Addieren eines Offsets.

Beispiel 12.19: Ein Schärfefilter



Beispiel: Eine universelle Filter-Klasse entwickeln

Wie Sie der obigen Beschreibung bzw. der Skizze entnehmen können, brauchen wir für die einzelnen Grafikoperationen nicht jedes Mal „das Rad neu zu erfinden“. Alle Berechnungen laufen immer nach dem gleichen Schema ab, eine universelle Klasse kann uns also die meiste Arbeit abnehmen.

Wir beginnen mit dem Einbinden der Namespaces:

```
...
using System.Text;
using System.Drawing;
using System.Drawing.Imaging;

namespace GDI_Filter
{
    public class GrafikFilter
    {
```

In dieser Eigenschaft speichern wir die Koeffizienten ab:

```
        public int[ , ] Matrix = {{0,0,0},{0,0,0},{0,0,0}};
```

Der Teiler und der Offset:

```
        public int Teiler = 0;
        public int Offset = 0;
```

Eine schon bekannte Funktion zum Normieren von Bytewerten:

```
        public byte normiere(int Value)
        {
            if (Value < 0) return 0;
            if (Value > 255) return 255;
            return (byte)Value;
        }
```

An die Methode *Execute* übergeben Sie später die zu bearbeitende Bitmap:

```
        public void Execute(Bitmap bmp)
        {
```

Wir benötigen zwischenzeitlich eine Kopie der Bitmap, da wir sonst mit schon bearbeiteten Farbwerten rechnen würden:

```
            Bitmap bmpQuelle = (Bitmap)bmp.Clone();
```

32-Bit-RGB-Bitmaps erzeugen:

```
            BitmapData dataQuelle = bmpQuelle.LockBits(new Rectangle(0, 0, bmp.Width,
                                                bmp.Height), ImageLockMode.ReadWrite,
                                                PixelFormat.Format32bppRgb);
            BitmapData dataZiel = bmp.LockBits(new Rectangle(0, 0, bmp.Width,
bmp.Height),
                                                ImageLockMode.ReadWrite,
                                                PixelFormat.Format32bppRgb);
```

Um nicht umständlich in mehreren Schleifendurchläufen die Matrixwerte abzufragen, erzeugen wir gleich drei Pointer für die jeweils abzufragenden Zeilen:

```
            unsafe
            {
                byte* pByte = (byte*)dataZiel.Scan0 + bmp.Width * 4 + 4;
                byte* pZeile0 = (byte*)dataQuelle.Scan0;
                byte* pZeile1 = (byte*)dataQuelle.Scan0 + bmp.Width * 4;
                byte* pZeile2 = (byte*)dataQuelle.Scan0 + bmp.Width * 8;
                int size = (bmp.Width - 2) * (bmp.Height - 2);
                for (int i = 0; i < size; i++)
                {
```

Was hier so kompliziert aussieht, ist nichts anders als das Wichten mit den Matrix-Werten, das Addieren, nachfolgende Teilen und das Zuweisen des Offsets für die drei Farbwerte:

```
                    pByte[0] = normiere((((pZeile0[0] * Matrix[0, 0]) + (pZeile0[4] *
                                        Matrix[1, 0]) + (pZeile0[8] * Matrix[2, 0]) +
                                        (pZeile1[0] * Matrix[0, 1]) + (pZeile1[4] *
                                        Matrix[1,1]) + (pZeile1[8] * Matrix[2, 1]) +
                                        (pZeile2[0] * Matrix[0, 2]) + (pZeile2[4] *
                                        Matrix[1,2]) + (pZeile2[8] * Matrix[2, 2])) /
Teiler)
                                         + Offset);

                    pByte[1] = normiere((((pZeile0[1] * Matrix[0, 0]) + (pZeile0[5] *
                                        Matrix[1, 0]) + (pZeile0[9] * Matrix[2, 0]) +
                                        (pZeile1[1] * Matrix[0, 1]) + (pZeile1[5] *
                                        Matrix[1, 1]) + (pZeile1[9] * Matrix[2, 1]) +
                                        (pZeile2[1] * Matrix[0, 2]) + (pZeile2[5] *
                                        Matrix[1, 2]) + (pZeile2[9] * Matrix[2, 2]))
                                        / Teiler) + Offset);

                    pByte[2] = normiere((((pZeile0[2] * Matrix[0, 0]) + (pZeile0[6] *
                                        Matrix[1, 0]) + (pZeile0[10] * Matrix[2, 0]) +
                                        (pZeile1[2] * Matrix[0, 1]) + (pZeile1[6] *
                                         Matrix[1, 1]) + (pZeile1[10] * Matrix[2, 1]) +
                                        (pZeile2[2] * Matrix[0, 2]) + (pZeile2[6] *
                                        Matrix[1, 2]) + (pZeile2[10] * Matrix[2, 2]))
                                        / Teiler) + Offset);
```

Pointer um 4 Byte weiterbewegen (32 Bit RGB):

```
                    pByte += 4;
                    pZeile0 += 4;
                    pZeile1 += 4;
                    pZeile2 += 4;
                }
            }
            bmp.UnlockBits(dataZiel);
            bmpQuelle.UnlockBits(dataQuelle);
            bmpQuelle.Dispose();
        }
    }
}
```

HINWEIS: Wer mag, kann sich an dieser Stelle ja versuchen und eine Parallelisierung durchführen, was auf Mehrprozessorsystemen für einen weiteren Geschwindigkeitsschub sorgen dürfte.

In der Klasse *GrafikFilter* sind eigentlich alle Berechnungen gekapselt, was uns noch fehlt, sind die Matrixwerte für die einzelnen Filter.

Die Klasse GrafikFilter im Einsatz

Am Beispiel von vier Filteralgorithmen wollen wir kurz die Verwendung der Klasse demonstrieren.

Beispiel 12.20: Bild schärfen

C#

```
        public static void Sharpen(Bitmap bmp)
        {
```

Klasse instanziieren:

```
            GrafikFilter gf = new GrafikFilter();
```

Werte festlegen:

```
            gf.Teiler = 3;
            gf.Matrix[0, 0] = 0; gf.Matrix[1, 0] = -2; gf.Matrix[2, 0] = 0;
            gf.Matrix[0, 1] = -2; gf.Matrix[1, 1] = 11; gf.Matrix[2, 1] = -2;
            gf.Matrix[0, 2] = 0; gf.Matrix[1, 2] = -2; gf.Matrix[2, 2] = 0;
```

Grafik bearbeiten:

```
            gf.Execute(bmp);
        }
```

Beispiel 12.21: Weichzeichnen

C#

```
public static void Smoothing(Bitmap bmp)
{
    GrafikFilter gf = new GrafikFilter();
    gf.Teiler = 8;
    gf.Matrix[0, 0] = 1; gf.Matrix[1, 0] = 1; gf.Matrix[2, 0] = 1;
    gf.Matrix[0, 1] = 1; gf.Matrix[1, 1] = 1; gf.Matrix[2, 1] = 1;
    gf.Matrix[0, 2] = 1; gf.Matrix[1, 2] = 1; gf.Matrix[2, 2] = 1;
    gf.Execute(bmp);
}
```

Beispiel 12.22: Emboss-Effekt

C#

```
public static void Emboss(Bitmap bmp)
{
    GrafikFilter gf = new GrafikFilter();
    gf.Teiler = 1;
    gf.Offset = 127;
    gf.Matrix[0, 0] = -1; gf.Matrix[1, 0] = 0; gf.Matrix[2, 0] = -1;
    gf.Matrix[0, 1] = 0; gf.Matrix[1, 1] = 4; gf.Matrix[2, 1] = 0;
    gf.Matrix[0, 2] = -1; gf.Matrix[1, 2] = 0; gf.Matrix[2, 2] = -1;
    gf.Execute(bmp);
}
```

Beispiel 12.23: Gaußscher Weichzeichner

C#

```
public static void Gauss(Bitmap bmp)
{
    GrafikFilter gf = new GrafikFilter();
    gf.Teiler = 16;
    gf.Matrix[0, 0] = 1; gf.Matrix[1, 0] = 2; gf.Matrix[2, 0] = 1;
    gf.Matrix[0, 1] = 2; gf.Matrix[1, 1] = 4; gf.Matrix[2, 1] = 2;
    gf.Matrix[0, 2] = 1; gf.Matrix[1, 2] = 2; gf.Matrix[2, 2] = 1;
    gf.Execute(bmp);
}
```

HINWEIS: Von der Funktionsfähigkeit des Grafikfilters können Sie sich anhand des Beispielprogramms überzeugen!

12.3 Fortgeschrittene Techniken

Im Folgenden möchten wir noch auf einige Techniken eingehen, mit denen Sie die Ausgabeergebnisse bezüglich Geschwindigkeit und/oder Qualität verbessern können.

12.3.1 Flackerfrei dank Double Buffering

Sicher haben Sie auch schon vor dem Problem gestanden, dass Sie umfangreiche Grafiken ausgeben wollten, die Darstellung auf dem Bildschirm aber während dieser Zeit unerträglich flackert. Ein Beispiel zeigt die Problematik und im Anschluss auch den Lösungsansatz.

Beispiel 12.24: Ausgabe einiger Linien

C#

```
double von = System.Environment.TickCount;
Graphics g = this.CreateGraphics();
for (int i = 0; i < 800; i++)
{
    g.DrawLine(Pens.Red, 0, i, 400, i);
    g.DrawLine(Pens.Blue, 0, 0, 400, i);
    g.DrawLine(Pens.Green, 400, 0, 0, i);
}
g.Dispose();
double bis = System.Environment.TickCount;
label1.Text = ((bis - von) / 1000).ToString() + " s";
```

Nach unerträglichen 0,45 Sekunden und viel Flackerei ist die Grafik endlich aufgebaut. Zwischenzeitlich sind auch Zeichenoperationen zu sehen, die wieder überdeckt werden.

Eine Puffer-Bitmap erzeugen

Mithilfe einer zusätzlichen Speicher-Bitmap können wir die Ausgaben zunächst unabhängig von der Oberfläche realisieren und nachfolgend ausgeben.

Beispiel 12.25: Fortführung des obigen Beispiels

C#

```
double von = System.Environment.TickCount;
```

Bitmap erzeugen (entsprechend der Fenstergröße):

```
Bitmap bmp = new Bitmap(ClientRectangle.Width, ClientRectangle.Height);
```

Grafikausgaben in der Bitmap vornehmen:

```
Graphics g = Graphics.FromImage(bmp);
for (int i = 0; i < 800; i++)
{
```

```
                g.DrawLine(Pens.Red, 0, i, 400, i);
                g.DrawLine(Pens.Blue, 0, 0, 400, i);
                g.DrawLine(Pens.Green, 400, 0, 0, i);
            }
            g.Dispose();
```

Die Bitmap im Fenster wiedergeben:

```
            g = this.CreateGraphics();
            g.DrawImage(bmp, 0, 0);
            g.Dispose();
            bmp.Dispose();
            double bis = System.Environment.TickCount;
            label1.Text = ((bis - von) / 1000).ToString() + " s";
```

Mit 0,078 Sekunden Zeitbedarf können wir schon eine wesentliche Verbesserung feststellen, zusätzlich ist die Ausgabe flackerfrei und – last but not least – können wir die Bitmap mit jeder Paint-Operation ausgeben, ohne sie erneut aufbauen zu müssen.

Und was ist mit der PictureBox?

Hier haben wir es mit einem Sonderfall zu tun, die *PictureBox* beherrscht bereits „ab Werk" Double Buffering. Doch Achtung:

HINWEIS: Sie müssen mit der **enthaltenen** Grafik arbeiten, nicht mit der Bildschirmdarstellung.

Falsch:

```
Graphics g = pictureBox1.CreateGraphics();
...
```

Richtig:

```
Graphics g = Graphics.FromImage(pictureBox1.Image);
...
g.Dispose();
pictureBox1.Invalidate();
```

HINWEIS: Wichtig ist das *Invalidate*, sonst passiert auf dem Bildschirm nichts, bis die Grafik – zum Beispiel nach einem Verdecken – neu gezeichnet werden muss.

Sollten Sie keine Grafik in die *PictureBox* geladen haben, erzeugen Sie einfach eine entsprechende Grafik:

```
Bitmap bmp = new Bitmap(ClientRectangle.Width, ClientRectangle.Height);
pictureBox1.Image = bmp;
```

12.3.2 Animationen

Welcher Programmierer wird nicht ab und zu vom Spieltrieb übermannt? Auch die Autoren bilden hier keine Ausnahme. Zu jedem Spiel gehört auch etwas Action und damit sind wir schon mitten im Thema angelangt. Wie können wir in .NET ein paar Bitmaps möglichst flackerfrei über den Bildschirm bewegen, ohne gleich auf DirectX zurückgreifen zu müssen?

Vorbereiten des Ausgabeobjekts

Im vorhergehenden Abschnitt war ja bereits die Rede von Double Buffering, eine Technik, die wir auch hier einsetzen wollen.

Die naheliegende Lösung dürfte also das Erzeugen einer Hintergrundbitmap sein, auf der wir die einzelnen Bitmaps verschieben. Mit einem *Timer* blenden wir diese Bitmap zyklisch in den Vordergrund ein.

So weit, so gut, das Flackern beim Bildaufbau können wir auf diese Weise vermeiden, allerdings macht uns Windows hier einen Strich durch die Rechnung. Es flackert trotzdem und zwar mit der Frequenz des Timers. Die Ursache findet sich in der Messagebehandlung für das Aktualisieren des Fensterhintergrunds (gilt auch für ein Control). Windows löscht bei jedem Refresh zunächst den Hintergrund mit der entsprechenden Hintergrundfarbe.

Lange Rede kurzer Sinn, mithilfe veränderter *ControlStyles* können wir Einfluss auf die Messagebehandlung durch Windows nehmen und stattdessen selbst für das Neuzeichnen des Controls/Fensters sorgen.

Beispiel 12.26: Ändern der Messagebehandlung im Formular-Konstruktor

C#

```
public Form1()
{
    InitializeComponent();
    SetStyle(ControlStyles.UserPaint, true);
    SetStyle(ControlStyles.AllPaintingInWmPaint, true);
    SetStyle(ControlStyles.DoubleBuffer, true);
}
```

Alternativ auch:

```
    SetStyle(ControlStyles.UserPaint | ControlStyles.AllPaintingInWmPaint |
             ControlStyles.DoubleBuffer , true);
```

Die Änderungen im Einzelnen:

- *UserPaint* – das Control zeichnet sich selbst, Windows bleibt außen vor.
- *AllPaintingInWmPaint* – die Message WM_ERASEBKGND wird ignoriert.
- *DoubleBuffer* – alle Zeichenoperationen werden zunächst im Hintergrund durchgeführt.

Nur alle drei Änderungen zugleich bewirken das gewünschte Ergebnis, nämlich kein Eingriff von Windows während unserer Zeichenaktivitäten.

Styles bei Controls setzen

Leider beschränkt sich dieses einfache Vorgehen zunächst auf das Formular, in allen anderen Controls sind die entsprechenden Styles nicht erreichbar (geschützte Methode).

Es hindert Sie aber nichts daran, einfach eine Ableitung der *Panel*-Komponente zu erzeugen und hier die gewünschten Styles einzutragen.

Beispiel 12.27: Styles bei Controls setzen

C#

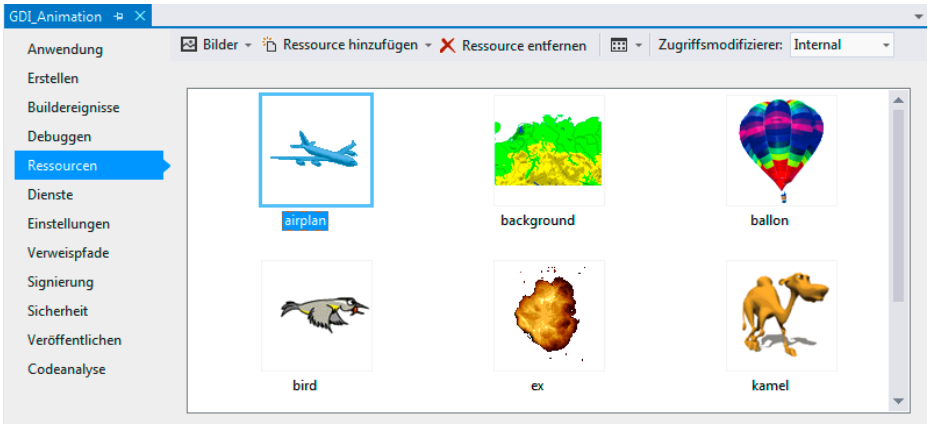
```
public partial class GrafikPanel : Panel
{
    public GrafikPanel()
    {
        InitializeComponent();
        SetStyle(ControlStyles.UserPaint | ControlStyles.AllPaintingInWmPaint |
                 ControlStyles.DoubleBuffer, true);
    }

    public GrafikPanel(IContainer container)
    {
       ...
        SetStyle(ControlStyles.UserPaint | ControlStyles.AllPaintingInWmPaint |
                 ControlStyles.DoubleBuffer, true);
    }
}
```

Die eigentliche Grafikausgabe

Bisher hatten wir uns nur damit beschäftigt, wie das Fenster für die Ausgabe vorbereitet werden muss. Doch wie bringen wir unsere Grafik auf den Bildschirm?

Einige Ausschnitte aus dem Beispielprogramm zeigen die Vorgehensweise.

- Betten Sie zunächst die gewünschten Grafiken als Ressourcen in die Anwendung ein. Nutzen Sie dafür den Ressourcen-Editor von Visual Studio:



- Erzeugen Sie für alle benötigten Grafiken globale Variablen und laden Sie die Grafiken beim Initialisieren aus den Ressourcen nach:

```
public partial class Form1 : Form
{
    Bitmap bmp1 = GDI_Animation.Properties.Resources.bird;
    Bitmap bmp2 = GDI_Animation.Properties.Resources.ballon;
    Bitmap bmp3 = GDI_Animation.Properties.Resources.airplan;
    Bitmap bckbmp = GDI_Animation.Properties.Resources.background;
 ...
```

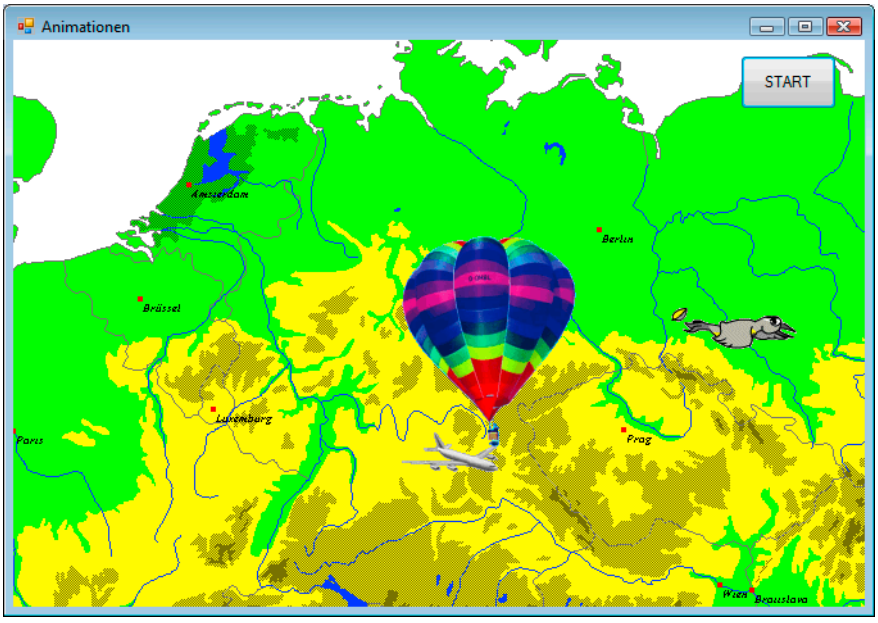
- Überschreiben Sie die *OnPaint*-Methode des Formulars und führen Sie hier Ihre Grafikoperationen aus:

```
    protected override void OnPaint(PaintEventArgs e)
    {
        e.Graphics.DrawImage(bckbmp, 0, 0);
        e.Graphics.DrawImage(bmp3, pos, pos);
        e.Graphics.DrawImage(bmp2, pos, 450 - pos);
        e.Graphics.DrawImage(bmp1, 2 * pos - 100, 200);
    }
```

- Nutzen Sie einen *Timer*, um zyklisch die *Invalidate*-Methode aufzurufen:

```
    private void timer1_Tick(object sender, EventArgs e)
    {
        pos++;
        if (pos > 500) pos = 0;
        this.Invalidate();
    }
```

Im Beispielprogramm verschieben wir drei Sprites über einer Hintergrundbitmap, ein Ruckeln werden Sie trotz des recht einfachen Verfahrens nicht feststellen:



Bemerkungen

- Beachten Sie, dass im Beispielprojekt für die Sprites GIF-Grafiken verwendet wurden. Diese ermöglichen es, Transparenz bereits in der Grafik festzulegen. Wir müssen die Objekte also nicht erst freistellen (Maskieren).
- Die Animation des Vogels, das heißt dessen Flügelbewegung, steht im Mittelpunkt des folgenden Abschnitts.

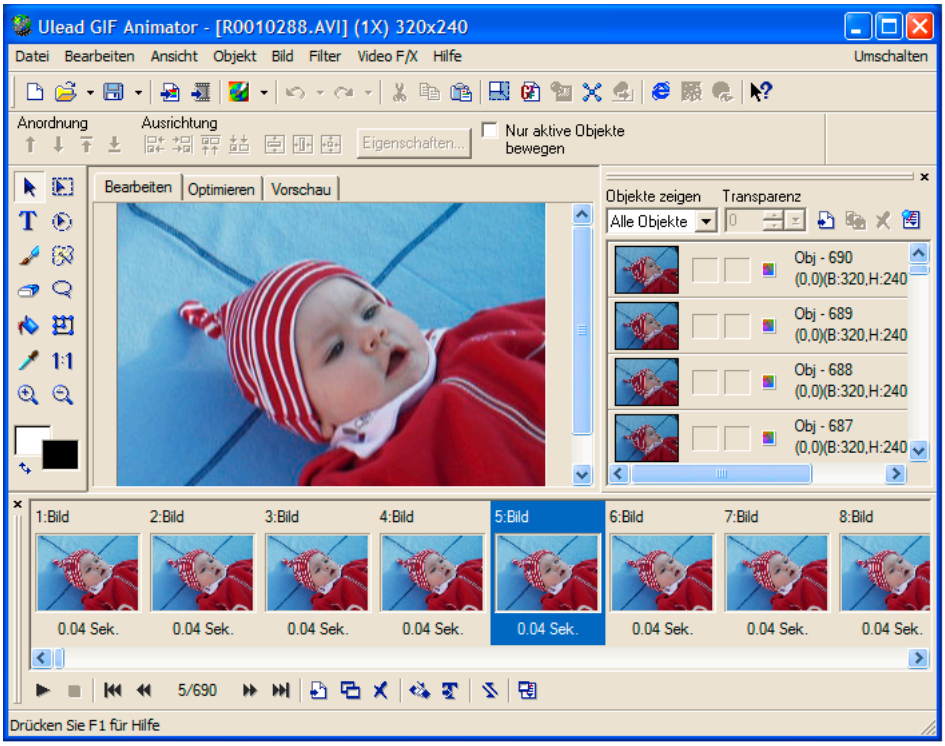
12.3.3 Animated GIFs

Soll der Eindruck von Bewegung entstehen, genügt es meist nicht (wie im vorhergehenden Abschnitt beschrieben) ein Sprite einfach über den Bildschirm zu schieben. Das funktioniert zwar ganz gut, ein echtes „Kino-Feeling“ wird so aber nicht aufkommen. Besser funktioniert es mithilfe von animierten GIF-Grafiken (das gute alte „Daumenkino“ lässt grüßen).

Beispiel 12.28: Einzelbilder aus einer animierten GIF-Datei

Wie Sie sehen, wird hier das Prinzip des Filmstreifens auf recht einfache Weise imitiert, nur die Anzahl der Bilder pro Sekunde ist wesentlich geringer, was aber in den meisten Fällen vollkommen genügt.

HINWEIS: Derartige Grafiken finden Sie zu Tausenden im Internet, oder Sie investieren selbst etwas Arbeit und erzeugen diese aus Videos oder Einzelgrafiken (zum Beispiel mithilfe des *Ulead GIF-Animator*s).



Wie lernen die Bilder das Laufen?

Auch für diese Aufgabe stellt das .NET-Framework die nötige Infrastruktur in Gestalt der *ImageAnimator*-Klasse zur Verfügung. Diese hat bereits einen integrierten Timer, der automatisch zwischen den einzelnen Bildern umschaltet.

Beispiel 12.29: Animierte GIFs wiedergeben

C#

Fügen Sie die Animated GIFs als Ressourcen in Ihre Anwendung ein und laden Sie diese zur Laufzeit in eine Bitmap.

```
Bitmap bmp2 = GDI_AnimGIF.Properties.Resources.dino3;
Bitmap bmp1 = GDI_AnimGIF.Properties.Resources.dino6;
```

Wer jetzt schon neugierig ist, wird leider enttäuscht. Die Grafik zeigt nur das erste Bild aus der GIF-Sequenz.

Melden Sie nachfolgend die Bitmap beim *ImageAnimator* an:

```
if (ImageAnimator.CanAnimate(bmp1))
    ImageAnimator.Animate(bmp1, this.OnNextFrame);
if (ImageAnimator.CanAnimate(bmp2))
    ImageAnimator.Animate(bmp2, this.OnNextFrame);
```

Verwenden Sie dazu die *Animate*-Methode, der Sie neben der jeweiligen Grafik auch einen Eventhandler übergeben können. Immer wenn ein neues Bild fällig ist, wird das Ereignis ausgelöst.

Erstellen Sie den Eventhandler, der für die Ausgabe verantwortlich ist:

```
private void OnNextFrame(object o, EventArgs e)
{
   this.Invalidate();
}
```

Geben Sie die Grafiken aus und schalten Sie mit *UpdateFrames* auf das jeweils nächste Bild in der Sequenz um:

```
protected override void OnPaint(PaintEventArgs e)
{
   e.Graphics.DrawImage(bmp2, 10, 10);
   e.Graphics.DrawImage(bmp1, 200, 10);
   ImageAnimator.UpdateFrames();
}
```

Ergebnis

12.3.4 Auf einzelne GIF-Frames zugreifen

Im vorhergehenden Abschnitt haben wir ja bereits eine Möglichkeit aufgezeigt, wie Sie als Programmierer die einzelnen Frames einer animierten GIF-Datei auslesen können, um zum Beispiel eine Animation zu realisieren. Allerdings haben Sie mit den oben genannten Mitteln keinen Zugriff auf einen beliebigen Frame und Sie können auch nicht die Anzahl der Frames bestimmen.

Verantwortlich für diese Aufgaben ist ein *FrameDimension*-Objekt. Dem Konstruktor übergeben Sie die GUID der *FrameDimensionsList* des gewählten Bilds.

Beispiel 12.30: Abrufen eines FrameDimension-Objekts

C#

```
FrameDimension fdim = new FrameDimension(bmp.FrameDimensionsList[0]);
```

HINWEIS: Über die *FrameDimensionsList* werden die einzelnen Frames bzw. verschiedenen Auflösungen des Bilds von .NET verwaltet.

Wiedergabe einzelner Frames

Möchten Sie einzelne Frames wiedergeben (zum Beispiel durch Verschieben eines *TrackBars*), brauchen Sie neben der Anzahl der Frames auch eine Möglichkeit, den aktuellen Frame zu setzen. In beiden Fällen hilft Ihnen das oben genannte *FrameDimension*-Objekt weiter.

Beispiel 12.31: Frameauswahl per *TrackBar*

C#

```
        Bitmap bmp = Properties.Resources.dino6;
        FrameDimension fdim;
...
        private void Form1_Load(object sender, EventArgs e)
        {
           fdim = new FrameDimension(bmp.FrameDimensionsList[0]);
           trackBar1.Maximum = bmp.GetFrameCount(fdim)-1;
        }

        private void trackBar1_Scroll(object sender, EventArgs e)
        {
            bmp.SelectActiveFrame(fdim, trackBar1.Value);
            Graphics g = this.CreateGraphics();
            g.Clear(this.BackColor);
            g.DrawImage(bmp,0,0);
        }
```

GetFrameCount liefert die Anzahl der verfügbaren Frames für die gewählte Bildabmessung. *SelectActiveFrame* setzt den aktiven Frame, der zum Beispiel beim Kopieren mittels *DrawImage* genutzt wird.

Erzeugen eines Bitmap-Strips

Sollen alle Bilder aus einer animierten GIF extrahiert und zum Beispiel als fortlaufender Streifen in einer Bitmap gesichert werden, können Sie sich am folgenden Code orientieren:

Beispiel 12.32: Extrahieren und Sichern aller Bilder einer animierten GIF-Bitmap

C#

```
        Bitmap bmp = Properties.Resources.dino6;
        FrameDimension fdim;
        fdim = new FrameDimension(bmp.FrameDimensionsList[0]);
```

Hilfsbitmap erzeugen, mit der Breite=Frameanzahl*Framebreite:

```
        Bitmap bmp2 = new Bitmap((bmp.GetFrameCount(fdim) - 1) * bmp.Width,
 bmp.Height);
        Graphics g = Graphics.FromImage(bmp2);
```

Nacheinander die Frames kopieren:

```
for (int i = 0; i < bmp.GetFrameCount(fdim); i++)
{
    bmp.SelectActiveFrame(fdim, i);
    g.DrawImageUnscaled(bmp, i * bmp.Width, 0);
}
```

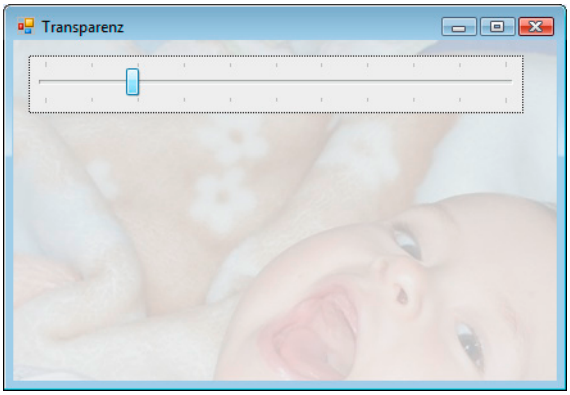
Die Hilfsbitmap können Sie gegebenenfalls einer *PictureBox* zuweisen:

```
pictureBox1.Image = bmp2;
g.Dispose();
```

Ergebnis

12.3.5 Transparenz realisieren

Sollen Bilder ein- bzw. ausgeblendet werden, müssen Sie das Rad nicht neu erfinden. Hier hilft Ihnen GDI+ mit seiner bereits eingebauten Fähigkeit, die Transparenz mithilfe des Alpha-Kanals zu steuern.



Beispiel 12.33: Ein Bild mit einer Transparenz zwischen 0 und 100 % darstellen

C#

```
private Bitmap bmp;
public Form1()
{
    InitializeComponent();
```

Das Flackern unterbinden:

```
            SetStyle(ControlStyles.UserPaint, true);
            SetStyle(ControlStyles.AllPaintingInWmPaint, true);
            SetStyle(ControlStyles.DoubleBuffer, true);
```

Bild aus den Ressourcen laden:

```
            bmp = GDI_Samples.Properties.Resources._0000003550;
        }
```

Bild zeichnen:

```
        protected override void OnPaint(PaintEventArgs e)
        {
```

Der Bitmap werden neue Attribute (eine *ColorMatrix*) zugewiesen:

```
            ImageAttributes iattr = new ImageAttributes();
            ColorMatrix m = new ColorMatrix();
```

Hier wird die Transparenz festgelegt:

```
            m.Matrix33 = trackBar1.Value / 100f;
            iattr.SetColorMatrix(m);
```

Leider ist die erforderliche Überladung der *DrawImage*-Methode etwas umfangreich:

```
            e.Graphics.DrawImage(bmp, new Rectangle(0, 0, bmp.Width, bmp.Height),
 0, 0,
                                 bmp.Width, bmp.Height, GraphicsUnit.Pixel, iattr);
        }
```

Zu guter Letzt müssen wir noch die Bildaktualisierung erzwingen:

```
        private void trackBar1_Scroll(object sender, EventArgs e)
        {
           this.Invalidate();
        }
```

12.3.6 Eine Grafik maskieren

Für die Wiedergabe von Animationen benötigen Sie meist keine rechteckigen Objekte (Sprites), sondern bereits freigestellte Grafiken, das heißt, der Hintergrund ist transparent.

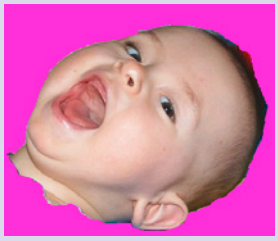
Zwei Verfahren bieten sich an:

- Sie verwenden das GIF- oder PNG-Format und definieren bereits hier (im Zeichenprogramm) den transparenten Hintergrund. Lesen Sie eine derartige Datei mittels GDI+ ein, wird auch die Transparenz bei Ausgaben berücksichtigt.
- Sie füllen den Hintergrund mit einer definierten Farbe (z. B. Violett) und stellen die Grafik erst zur Laufzeit frei.

HINWEIS: Verwenden Sie ein verlustloses Komprimierverfahren (GIF, PNG, BMP), andernfalls werden durch die Kompressionsartefakte die Masken unscharf wiedergegeben.

Variante 1 unterscheidet sich nicht von der normalen Wiedergabe mittels *DrawImage* etc., wir gehen deshalb nicht weiter darauf ein. Variante 2 erfordert etwas Vorarbeit.

Beispiel 12.34: Freigestellte Grafik, der Hintergrund wurde mit Violett gefüllt.



Im Programm selbst müssen wird zunächst die Grafik laden und können dann mithilfe der Methode *MakeTransparent* den „Hintergrund“ entfernen. Dazu lesen wir die Farbe eines der violetten Pixel aus:

```
Bitmap bmp1 = global::GDI_Maskieren.Properties.Resources._1;
Graphics g = this.CreateGraphics();
bmp1.MakeTransparent(bmp1.GetPixel(2, 2));
g.DrawImage(bmp1, 0, 0);
g.Dispose();
```

Nachfolgend lässt sich das Bild problemlos über einem anderen Hintergrund einblenden:

12.3.7 JPEG-Qualität beim Sichern bestimmen

Sicher hat es Sie auch schon gestört, dass Sie Ihre Bilder zwar problemlos im JPEG-Format sichern können, es aber zunächst keine Möglichkeit zu geben scheint, die Kompressionsrate einzustellen.

Beispiel 12.35: Sichern einer Grafik im JPEG-Format

C#

```
pictureBox1.Image.Save("Bild.jpg", ImageFormat.Jpeg);
```

Wer genauer hinschaut, der wird unter den zahlreichen Überladungen der *Save*-Methode einen Kandidaten für unsere Aufgabenstellung finden:

Syntax:

```
Image.Save (String, ImageCodecInfo, EncoderParameters)
```

Doch woher nehmen wir *ImageCodecInfo* und *EncoderParameters*?

Dank der universellen Architektur ist die Verwendung nicht ganz trivial.

Beispiel 12.36: JPEG-Qualität beim Sichern bestimmen

C#

Zunächst müssen wir die *JPEG-Codec* aus der Liste der möglichen Codes ermitteln:

```
using System.Drawing.Imaging;
...
ImageCodecInfo myImageCodecInfo = null;
foreach (ImageCodecInfo codec in ImageCodecInfo.GetImageEncoders())
   if (codec.MimeType == "image/jpeg") myImageCodecInfo = codec;
```

Nachfolgend müssen wir noch einen Parameter erzeugen

```
EncoderParameters myParameter = new EncoderParameters();
```

und parametrieren (hier 25%):

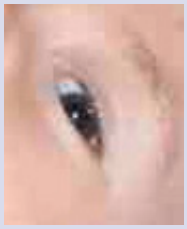
```
myParameter.Param[0] = new
EncoderParameter(System.Drawing.Imaging.Encoder.Quality, 25);
```

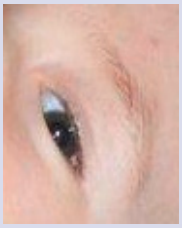
Zu guter Letzt speichern wir mithilfe des Codes und des Parameters unsere JPEG-Grafik ab:

```
pictureBox1.Image.Save("Bild.jpg", myImageCodecInfo, myParameter);
```

Ergebnis

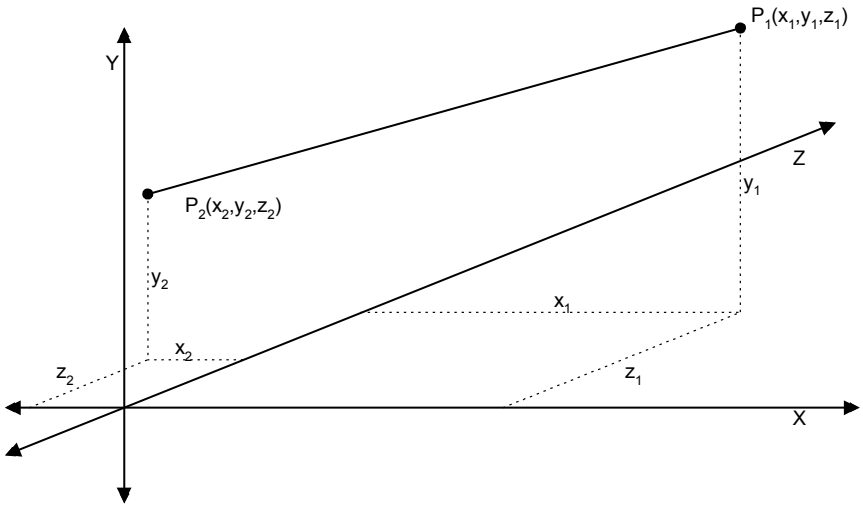
Die folgenden Abbildungen zeigt einen Vergleich zwischen der Qualität 25 % (deutliche Artefakte, kaum noch Details) und der Qualität von 75 %.





12.4 Grundlagen der 3D-Vektorgrafik

Prinzipiell gelten für 3D-Vektorgrafiken auch die allgemeinen Ausführungen zur 2D-Grafik, wie sie auch in GDI+ zum Einsatz kommt. Um aber bei der Menge der Koordinaten (jede Linie wird nicht durch vier, sondern durch sechs Werte repräsentiert) noch halbwegs den Überblick zu behalten, müssen wir das 2D-Konzept erweitern.



Zusätzlich zur x- und y-Koordinate müssen wir die z-Achse für die räumliche Tiefenwirkung verwenden.

Wir wollen im Folgenden davon ausgehen, dass der darzustellende Körper nur aus Linien besteht (sogenanntes Drahtmodell), die Erweiterung der Algorithmen auf n-Ecke bzw. Kreise ist jederzeit möglich, würde jedoch den Rahmen dieses Kapitels sprengen. Ein Kreis

muss durch seinen Mittelpunkt und mindestens einen Raumpunkt, der auf dem Umfang liegt, definiert sein. Übrigens lässt sich auch ein Kreis als Vieleck (Polygon) auffassen, die Rechenzeit verkürzt sich dadurch erheblich!

Ausgangspunkt für die weitere Arbeit ist die Suche nach einer sinnvollen Möglichkeit für das Speichern eines Körpers.

12.4.1 Datentypen für die Verwaltung

In C# bietet sich für diesen Zweck ein generisches *List*-Objekt an, das zum einen typsicher ist, zum anderen können beliebig viele Objekte gespeichert werden. Auch das spätere Abarbeiten der Liste ist mittels *foreach*-Schleife recht einfach realisierbar.

Grundlage des Körpers sind Linien, die durch Raumpunkte dargestellt werden. Jeder Raumpunkt besteht wiederum aus drei Koordinaten.

Beispiel 12.37: Die Umsetzung eines Raumpunkts in C#

C#

```
struct punkt3D
{
    public Double x, y, z;
    public PointF OnScreen;
}
```

In *OnScreen* speichern wir gleich die 2D-Ausgabekoordinaten für jeden 3D-Punkt, so bleibt alles zusammen.

Diese 3D-Punkte verpacken wir wiederum in universellen *Objects3D*-Objekten:

```
abstract class Objects3D
{
    public punkt3D[] points;
    public String Name;
    abstract public void Draw(Graphics g, Pen p);
}
```

HINWEIS: Bei der Klasse *Objects3D* handelt es sich um einen abstrakten Vorfahrtyp für unsere eigentlichen 3D-Objekte. Allen gemein ist ein Array von Punkten, mit dem das 3D-Objekt beschrieben wird, ein Name (z. B. für das Debugging) sowie eine *Draw*-Methode, mit der sich das Objekt selbst zeichnet. Auf die Bedeutung der einzelnen Parameter kommen wir später zurück.

12.4.2 Eine universelle 3D-Grafik-Klasse

Die Liste zum Speichern eines 3D-Objekts verpacken wir in eine Klasse *c3D*:

```
class c3D
{
```

Zunächst allgemeine Optionen:

```
        public double Abstand = 1000;      // Betrachterabstand für die 2D-Darstellung
        public Pen Stift1 = Pens.Black;    // der Zeichenstift
```

Einige Hilfsvariablen und zwei kleine Arrays (LUT), in denen wir Werte vorberechnen:

```
        private double r =  Math.PI / 180;
        private Double[] sint = new Double[3];
        private Double[] cost = new Double[3];
```

Und hier die (generische) Liste der 3D-Objekte:

```
        private List<Objects3D> objectList = new List<Objects3D>();
```

Der Konstruktor:

```
        public c3D()
        {
            for (int i = -1; i < 2; i++)
            {
                sint[i + 1] = Math.Sin(i * r * 2);
                cost[i + 1] = Math.Cos(i * r * 2);
            }
        }
```

Neue Objekte fügen Sie in die Liste mit der Methode *AddObject* ein:

```
        public void AddObject(Objects3D o)
        {   objectList.Add(o);   }
```

Die weiteren Methoden der Klasse *c3D* (die eigentlichen Algorithmen stellen wir Ihnen erst in den folgenden Abschnitten vor):

```
        public void Scale(double f)  // Skalieren aller Objekte
        { ... }

        public void Translate(double x, double y, double z)  // Verschieben
        { ... }

        public void Rotate(int alpha, int beta, int gamma)  // Drehen
        { ... }
```

Hier nutzen wir die Vorteile der Vererbung, nach dem Zeichnen eines Koordinatensystems werden alle Objekte der Liste „aufgefordert“, sich selbst per *Draw*-Methode zu zeichnen:

```
        public void Draw(Graphics g)
        {
```

```
            g.DrawLine(Pens.LightYellow, 200, 10, 200, 400);
            g.DrawLine(Pens.LightYellow, 10, 200, 400, 200);
            foreach (Objects3D o3d in objectList)
            {
```

Hier steht später die Umwandlung der 2D- in 3D-Koordinaten:

```
               // Erklärung folgt ...
               o3d.Draw(g, Stift1);
            }
        }
```

Übergeben werden lediglich das Ziel-*Graphics*-Objekt und der Zeichenstift.

12.4.3 Grundlegende Betrachtungen

Vor der weiteren Betrachtung stehen zwei Fragen:

- Wie sollen die Drehwinkel übergeben werden (relativ oder absolut)?
- Welche Darstellungsform auf dem Bildschirm soll gewählt werden?

Drehwinkel

Um den Aufwand möglichst niedrig zu halten, werden wir die Drehwinkel als relative Koordinaten bezüglich des letzten Drehpunkts interpretieren[3]. Dieses Vorgehen bietet mehrere Vorteile:

- Wir können die gleichen Algorithmen für die Rotation verwenden wie für die 2D-Grafik.
- Die Algorithmen sind nicht so komplex wie Algorithmen für die absolute Berechnung.
- Wir benötigen weniger Arbeitsspeicher.

Die letzte Aussage hört sich im Windows-Zeitalter etwas weltfremd an, um aber auch das Letzte an Geschwindigkeit aus unserem C#-Programm „herauszukitzeln“, müssen wir statt der laufenden Berechnung von *Cos()* und *Sin()* auf vorbereitete Tabellen zugreifen. Da trigonometrische Funktionen nicht gerade superschnell sind, lässt sich auf diese Weise einiges an Rechenzeit einsparen.

Zurück zum Thema Speicherplatz: Für die absolute Positionierung auf 1/10° genau brauchen wir allein für die Sinus-Tabelle 360 * 10 Werte, und das im *double*-Format. Für die inkrementelle Methode sieht das ganz anders aus. Wir können einfach ein paar sinnvolle Werte speichern, die häufig gebraucht werden, z. B. ±1/10°, ±1°, ±10°, ±90° etc. Sie sehen, der Aufwand ist bedeutend geringer!

Wo viel Licht ist, da ist natürlich auch Schatten. Die inkrementelle Methode hat auch ihre Nachteile. Ein Problem ist beispielsweise, dass sich Berechnungsfehler immer weiter aufsummieren können. Nach genügend vielen Rotationen/Skalierungen wird es Ihnen kaum gelingen, den Ursprungskörper wiederherzustellen. Dem kann jedoch mit dem *double*-Format entgegengewirkt werden. Demgegenüber kommt die absolute Methode eventuell sogar

[3] Im Programm besteht die Möglichkeit, den Körper um jeweils ± 1 ° zu drehen.

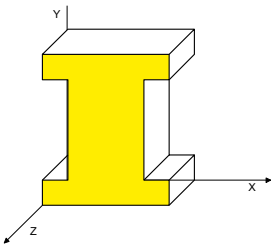
mit Integer-Arithmetik aus. Wie gesagt, die Wahl der Methode sollten Sie von Ihrer jeweiligen Anwendung abhängig machen.

Bleibt noch das Problem der Darstellung auf dem Bildschirm.

Darstellungsmöglichkeiten

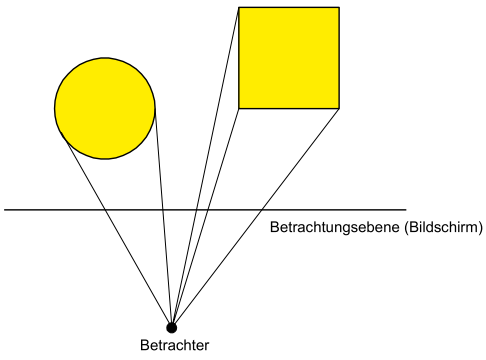
Für die Bildschirmdarstellung der 3D-Koordinaten können wir zwei verschiedene Verfahren verwenden:

- **Parallelprojektion**
  Bei der Parallelprojektion, in diesem Fall der sogenannten Kavalliersperspektive, bleibt die x-y-Ebene erhalten, z-Koordinaten werden in bestimmten Teilungsverhältnissen in der x-y-Ebene abgebildet.

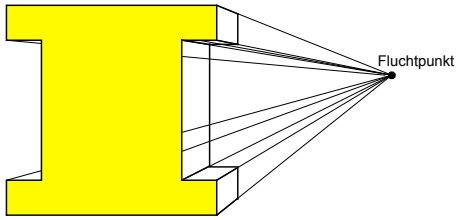


  Diese Darstellungsform verzerrt zwar den Körper, da aber die Strecken-Teilungsverhältnisse erhalten bleiben, wird die Parallelperspektive in der Praxis bevorzugt. Für ein Programm bedeutet das, dass x- und y-Koordinaten unverändert in die Betrachtung eingehen. Um die Darstellung nicht allzu sehr zu verzerren, sollten Sie den Neigungswinkel auf 18° reduzieren (Kavalliersperspektive 45°). Die z-Achse ist um 18° geneigt, zur x-Koordinate kommt also z/2 hinzu, zur y-Koordinate z/10.

- **Zentralprojektion** (Fluchtpunktperspektive)
  Die Zentralprojektion geht von einem Betrachterstandpunkt aus. Die Fluchtlinien Körperkante/Betrachter werden in der Betrachtungsebene abgebildet:

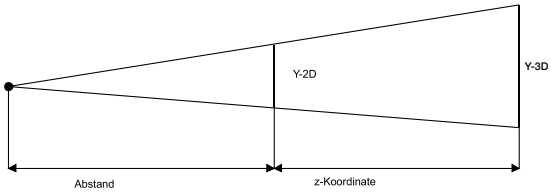


  Durch diese Art der Darstellung entsteht ein Fluchtpunkt, der vom Betrachterstandpunkt abhängig ist.



Sie können diesen Effekt im Beispielprogramm nachvollziehen, indem Sie den Körper nach rechts/links verschieben und vergrößern/verkleinern.

Damit sind wir auch schon bei der Umsetzung angelangt. Für die Zentralprojektion benötigen wir zusätzlich den Abstand des Betrachters. Nach dem Strahlensatz gilt:



$$Y_{2D} = \frac{Y_{3D} \cdot Abstand}{Abstand + Z}$$

Entsprechendes ist natürlich auch für die x-Koordinate zutreffend.

HINWEIS: Bei ungünstigem Betrachterstandpunkt kann ein Laufzeitfehler „Division durch null“ auftreten! Fangen Sie diesen gegebenenfalls mit *try-catch* ab.

Da in unserem Beispiel die Klasse *c3D* für die Konvertierung der 3D- in 2D-Koordinaten verantwortlich ist, können wir die Berechnung in die Methode *Draw* verlegen:

```
        public void Draw(Graphics g)
        {
            ...
            foreach (Objects3D o3d in objectList)
            {
               for (int i = 0; i < o3d.points.Length; i++)
                   o3d.points[i].OnScreen = new PointF(Convert.ToSingle(o3d.points[i].x *
                                            Abstand / (Abstand + o3d.points[i].z) +
links),
                                            Convert.ToSingle(-o3d.points[i].y * Abstand /
                                            (Abstand + o3d.points[i].z) + oben));
               o3d.Draw(g, Stift1);
            }
        }
```

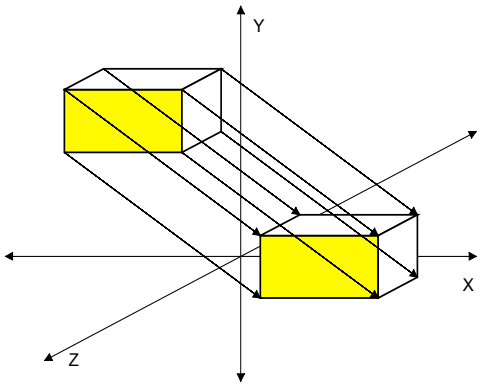
Wenden wir uns nun den eigentlichen Routinen für Translation, Streckung und Rotation zu.

12.4.4 Translation

Die Translation oder Verschiebung ist nichts anderes als die Addition von Punktvektor und Verschiebungsvektor. Was sich so kompliziert anhört, ist relativ trivial.

Jede Komponente des Punktvektors wird zu ihrem Verschiebungsvektor addiert.

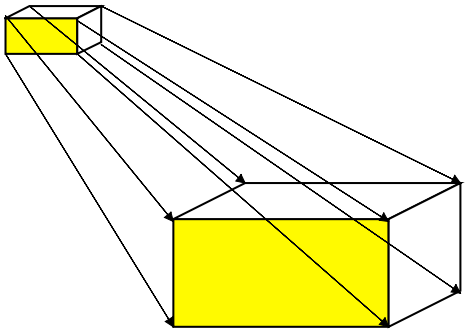
```
public void Translate(double x, double y, double z)
{
    foreach (Objects3D o3d in objectList)
    {
        for (int i = 0; i < o3d.points.Length; i++)
        {
            o3d.points[i].x += x;
            o3d.points[i].y += y;
            o3d.points[i].z += z;
        }
    }
}
```



HINWEIS: Um den gesamten Körper zu bewegen, müssen natürlich alle Punkte verschoben werden, wir kommen später darauf zurück.

12.4.5 Streckung/Skalierung

Neben der Verschiebung kann ein Körper auch vergrößert, das heißt skaliert werden. Die Skalierung wird durch Multiplikation des Punktvektors mit einer Konstanten (dem Skalierungsfaktor) realisiert.



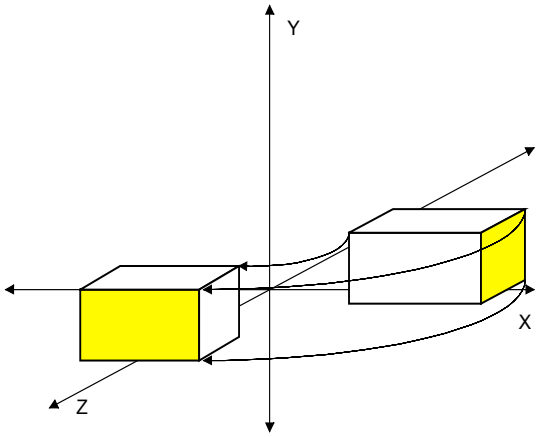
```
public void Scale(double f)
{
    foreach (Objects3D o3d in objectList)
    {
        for (int i = 0; i < o3d.points.Length; i++)
        {
           o3d.points[i].x *= f;
           o3d.points[i].y *= f;
           o3d.points[i].z *= f;
        }
    }
}
```

Je nachdem, ob der Skalierungsfaktor größer oder kleiner 1 ist, wird der Körper vergrößert oder verkleinert.

12.4.6 Rotation

Die Programmierung eines Rotationsalgorithmus gestaltet sich naturgemäß schwieriger als für die Translation bzw. Skalierung. Um den Überblick nicht zu verlieren, führen wir folgende Notation ein:

- Rotation um die x-Achse **α** (alpha),
- Rotation um die y-Achse **β** (beta),
- Rotation um die z-Achse **γ** (gamma).



Für die Drehung um die einzelnen Achsen sind Transformationsmatrizen R_i gegeben:

$$R_X = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

$$R_Y = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix}$$

$$R_Z = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Der neue Vektor bildet sich durch Multiplikation mit der entsprechenden Rotationsmatrix. Um unnötige Operationen zu vermeiden, werden die Matrizen aufgelöst und (soweit sinnvoll) berechnet (keine Multiplikationen mit null!).

Die Methode *Rotate* erwartet als Parameter die drei oben genannten Rotationswinkel. Im Programm sind nur Inkrementswinkel von -1°, 0°, +1° zulässig. Sie können die Anzahl der Winkelinkremente jedoch jederzeit in der *Init*-Prozedur ändern.

```
public void Rotate(int alpha, int beta, int gamma)
{
    punkt3D ph;
    if (alpha != 0)
    {
        alpha++;
        foreach (Objects3D o3d in objectList)
        {
            for (int i = 0; i < o3d.points.Length; i++)
            {
                ph = o3d.points[i];
                o3d.points[i].y = o3d.points[i].y * cost[alpha] -
                                  o3d.points[i].z * sint[alpha];
```

```
                        o3d.points[i].z = ph.y * sint[alpha] + o3d.points[i].z *
cost[alpha];
                    }
                }
            }
            if (beta != 0)
            {
                beta++;
                foreach (Objects3D o3d in objectList)
                {
                    for (int i = 0; i < o3d.points.Length; i++)
                    {
                        ph = o3d.points[i];
                        o3d.points[i].z = o3d.points[i].z * cost[beta] -
                                          o3d.points[i].x * sint[beta];
                        o3d.points[i].x = ph.z * sint[beta] + o3d.points[i].x *
cost[beta];
                    }
                }
            }
            if (gamma != 0)
            {
                gamma++;
                foreach (Objects3D o3d in objectList)
                {
                    for (int i = 0; i < o3d.points.Length; i++)
                    {
                        ph = o3d.points[i];
                        o3d.points[i].x = o3d.points[i].x * cost[gamma] -
                                          o3d.points[i].y * sint[gamma];
                        o3d.points[i].y = ph.x * sint[gamma] + o3d.points[i].y *
cost[gamma];
                    }
                }
            }
```

HINWEIS: Die Rotation wird nur ausgeführt, wenn ein Inkrement ungleich null vorliegt. Auf diese Weise sollen unnötige Berechnungen vermieden werden.

12.4.7 Die eigentlichen Zeichenroutinen

An dieser Stelle werden Sie sicher enttäuscht sein. Wer hier noch irgendwelche aufwendigen Berechnungen sucht, hat die bisherigen Ausführungen nicht richtig gelesen. In den Elementen *punkt3D.OnScreen* befinden sich bereits vor dem Aufruf der Objekt-*Draw*-Methoden die 2D-Koordinaten, wie Sie für die GDI+-Ausgaberoutinen erforderlich sind. Deshalb reicht es jetzt, die Punktkoordinaten den GDI-Anweisungen zuzuweisen.

Ausgabe einer Linie

Definition einer neuen Klasse, abgeleitet von *Objects3D*:

```
class line3D : Objects3D
    {
```

Im Konstruktor erwarten wir die Ursprungskoordinaten:

```
        public line3D(double x1, double y1, double z1, double x2, double y2, double z2)
        {
```

Abspeichern in *points*:

```
            points = new punkt3D[2];
            points[0].x = x1; points[0].y = y1; points[0].z = z1;
            points[1].x = x2; points[1].y = y2; points[1].z = z2;
            Name = "line3D";
        }
```

Und hier die Zeichenroutine, die die abstrakte *Draw*-Methode von *Objects3D* ersetzt:

```
        override public void Draw(Graphics g, Pen p)
        {  g.DrawLine(p, points[0].OnScreen, points[1].OnScreen);  }
    }
```

Die Verwendung dürfte recht einfach sein, dank intensiver Vorarbeit durch die Klasse *c3D*.

Ausgabe eines Koordinatensystems

Auch hier leiten wir von *Objects3D* ab:

```
class coordinateSystem3D : Objects3D
{
```

Der Konstruktor erwartet nur die Achslänge, den Rest berechnen wir selbst:

```
        public coordinateSystem3D(double size)
        {
            points = new punkt3D[6];
```

X-Achse:

```
            points[0].x = -size;  points[0].y = 0;     points[0].z = 0;
            points[1].x = size;   points[1].y = 0;     points[1].z = 0;
```

Y-Achse:

```
            points[2].x = 0;      points[2].y = -size; points[2].z = 0;
            points[3].x = 0;      points[3].y = size;  points[3].z = 0;
```

Z-Achse:

```
            points[4].x = 0;      points[4].y = 0;     points[4].z = -size;
            points[5].x = 0;      points[5].y = 0;     points[5].z = size;
```

```
            Name = "coordinateSystem3D";
        }
```

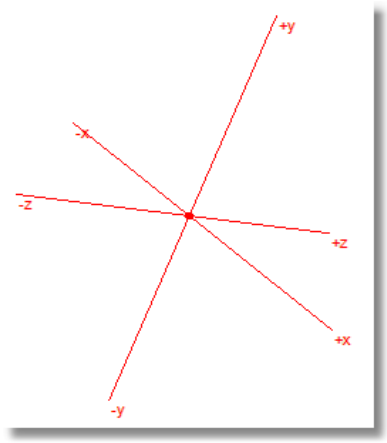
In der *Draw*-Methode müssen wir jetzt schon etwas mehr tippen:

```
        override public void Draw(Graphics g, Pen p)
        {
```

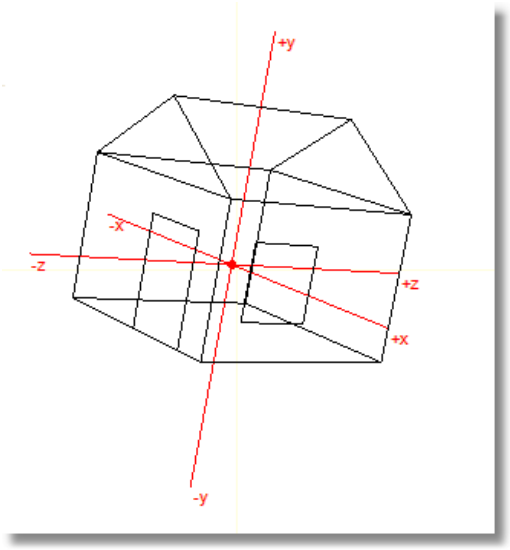
Ausgabe der Achsen und der Koordinatenangaben:

```
            g.DrawLine(Pens.Red, points[0].OnScreen, points[1].OnScreen);
            g.DrawString("+x", new Font("Arial", 8), Brushes.Red, points[1].OnScreen);
            g.DrawString("-x", new Font("Arial", 8), Brushes.Red, points[0].OnScreen);
            g.DrawLine(Pens.Red, points[2].OnScreen, points[3].OnScreen);
            g.DrawString("+y", new Font("Arial", 8), Brushes.Red, points[3].OnScreen);
            g.DrawString("-y", new Font("Arial", 8), Brushes.Red, points[2].OnScreen);
            g.DrawLine(Pens.Red, points[4].OnScreen, points[5].OnScreen);
            g.DrawString("+z", new Font("Arial", 8), Brushes.Red, points[5].OnScreen);
            g.DrawString("-z", new Font("Arial", 8), Brushes.Red, points[4].OnScreen);
        }
    }
```

Mit den obigen Anweisungen wird später das folgende Koordinatensystem gezeichnet:



Nach der Definition einiger Linien entsteht zum Beispiel das folgende 3D-Objekt:



HINWEIS: Das komplette 3D-Grafikbeispiel finden Sie im Praxisbeispiel in Abschnitt 12.6.2.

12.5 Und doch wieder GDI-Funktionen ...

Wer geglaubt hätte, GDI+ sei so etwas wie der komplette Ersatz von GDI, sieht sich in vielen Fällen getäuscht. Abgesehen davon, dass selbst mit GDI die Programmierung wesentlich einfacher als mit GDI+ ist, fehlen schlicht und einfach einige Funktionen. Andere sind so langsam, dass man glaubt, der alte 486er aus dem Keller sei wiederauferstanden.

Aus diesem Grund haben wir uns entschlossen, GDI in das vorliegende Buch aufzunehmen, auch wenn Sie damit **keinen managed Code** produzieren.

Da Sie nur schwer GDI+ und GDI-Funktionen „mischen“ können (das *Graphics*-Objekt ist während der Arbeit mit den GDI-Funktionen gesperrt), bleibt Ihnen bei der Verwendung von GDI-Funktionen meist nur der komplette Umstieg während einer bestimmten Grafikoperation (z. B. Zeichnen eines Diagramms). Aus diesem Grund stellen wir Ihnen GDI noch einmal im Zusammenhang vor, auch wenn Ihnen einige Funktionen aus dem bisherigen Kapitel bekannt vorkommen dürften.

12.5.1 Am Anfang war das Handle ...

Wenn Sie bereits in der Online-Hilfe bzw. in der MSDN-Library geschmökert haben, werden Sie im Zusammenhang mit GDI-Funktionen dort auch auf den Begriff *Handle* gestoßen sein. Was also ist ein Handle?

Mit Handle wird eine Nummer (in .NET vom Typ *IntPtr*) bezeichnet, mit der ein Fenster, Steuerelement etc. eindeutig gekennzeichnet ist. Jedes Fenster bzw. jede Komponente in C# verfügt über diese Eigenschaft.

Neben dem Handle finden Sie noch die Bezeichnungen *Gerätekontext* (*DC*) bzw. *Handle auf den Gerätekontext* (*HDC*). Dieser Gerätekontext ist die Schnittstelle zwischen Ihrem Anwendungsprogramm und den eigentlichen Ausgabegeräten wie Bildschirm, Fenster, Drucker etc. Der Kontext eines Geräts beinhaltet nicht nur alle Informationen, Eigenschaften und Attribute, sondern auch die Funktionen, um dieses Gerät anzusprechen.

Daraus folgt auch der prinzipielle Ablauf beim Arbeiten mit den GDI-Funktionen:

- Anlegen eines Gerätekontextes,
- Anlegen von Objekten,
- Zuweisen der neuen Objekte,
- Grafikausgaben (mit obigen Objekten),
- Wiederherstellen der Ursprungsobjekte,
- Zerstören der neuen Objekte,
- Freigabe bzw. Schließen des Gerätekontexts.

HINWEIS: Die Verwendung aller GDI-Funktionen setzt voraus, dass Sie diese vorher als Klassenmethoden (*DllImport*-Attribut) in Ihr C#-Programm eingebunden haben. Das betrifft ebenfalls alle verwendeten Konstanten.

Doch genug der Theorie, ein praktisches Beispiel wird Ihnen die Augen öffnen.

Beispiel 12.38: Grafikausgabe über einen Gerätekontext (Screen)

C#

1. Binden Sie zunächst die nötigen GDI-Funktionen in das Hauptformular ein:

```
using System.Runtime.InteropServices;
...
    [DllImport("user32.dll")]
    private  static extern int  GetDC(int  hwnd);

    [DllImport("user32.dll")]
    private  static extern int  ReleaseDC(int  hwnd,int  hdc);

    [DllImport("gdi32.dll")]
    private  static extern int  CreatePen(int  nPenStyle,int  nWidth,int  crColor);
```

```
[DllImport("gdi32.dll")]
private  static extern int  SelectObject(int  hdc,int  hObject);

[DllImport("gdi32.dll")]
private  static extern int  SetPixel(int  hdc,int  x,int  y,int  crColor);

[DllImport("gdi32.dll")]
private  static extern int  LineTo(int  hdc,int  x,int  y);
[DllImport("gdi32.dll")]
private  static extern int  DeleteObject(int  hObject);
```

2. Anlegen eines Gerätekontexts, der Rückgabewert ist ein Handle auf den DC:

```
int dc , hp, ho, dx, dy, i, l;

Random rnd = new Random();
dc = GetDC(0);
dx = 1000;
dy = 500;
```

3. Erstellen eines Stifts (Objekt), der Rückgabewert ist ein Handle:

```
hp = CreatePen(0, 10, 0);
```

4. Auswahl des neuen Stifts über das Handle, der Rückgabewert ist das Handle auf den vorherigen, nun ungültigen Stift:

```
ho = SelectObject(dc, hp);
```

5. Diverse Grafikausgaben vornehmen:

```
for (i = 0; i <50; i++)
{
  l = SetPixel(dc, rnd.Next(dx) , rnd.Next(dy), 0);
  l = LineTo(dc, rnd.Next(dx), rnd.Next(dy));
}
```

6. Wiederherstellen der alten Objekte (dazu haben wir uns den Wert gemerkt!) und Löschen der erstellten Objekte:

```
DeleteObject(SelectObject(dc, ho));
```

7. Löschen des Gerätekontexts:

```
ReleaseDC(0,dc);
```

Was mit obigem Beispiel möglich ist, das direkte Zeichnen auf dem Desktop, werden Sie nicht mit GDI+ realisieren können.

HINWEIS: Generell gilt für alle GDI-Aufrufe die Maßeinheit Pixel!

12.5.2 Gerätekontext (Device Context Types)

In C# führen zwei verschiedene Wege zum HDC. Zum einen können Sie einen vom *Graphics*-Objekt bereitgestellten DC verwenden (Methode *GetHDc*), zum anderen können Sie auch einen eigenen DC erzeugen. Welche Variante zu bevorzugen ist, lässt sich nicht eindeutig beantworten.

Einerseits ist die Arbeit mit C#-DCs recht komfortabel und sicher (Sie vergessen nie das Löschen des DC, da Sie erst wieder den DC mit *ReleaseHdc* freigeben müssen!). Andererseits kommen Sie in bestimmten Fällen nicht um die Definition eines eigenen DCs herum, z. B. wenn Sie einen DC brauchen, den C# nicht zur Verfügung stellt (Screen), oder wenn Sie mit einigen bestimmten GDI-Funktionen arbeiten.

Windows unterstützt vier verschiedene Typen von Gerätekontexten:

Gerätekontext	Beschreibung
Display	Ausgaben auf dem Bildschirm, das heißt Fenster, Controls oder Screen ▪ Erzeugen mit *GetDC*, *GetDCEx* oder *BeginPaint* ▪ Löschen mit *ReleaseDC* oder *EndPaint*
Printer	Drucker/Plotter der verschiedenen Typen ▪ Erzeugen mit *CreateDC* ▪ Löschen mit *DeleteDC*
Memory	Speicherabbilder von Bitmaps. Diese sind nicht sichtbar, können aber das Ziel von Grafikausgaben sein. ▪ Erzeugen mit *CreateCompatibleDC* Löschen mit *DeleteDC*
Information	... dient lediglich der Informationsgewinnung (z. B. im Zusammenhang mit dem Drucker), es können keine Grafikausgaben vorgenommen werden (*GetDeviceCaps*). ▪ Erzeugen mit *CreateIC* ▪ Löschen mit *DeleteDC*

Formular-DC über Graphics-Objekt erzeugen

Wie Sie beispielsweise den DC des Screens mit *GetDC* ermitteln, wurde bereits im vorhergehenden Abschnitt gezeigt. Wie Sie mit *GetHDc* den DC des Formulars abfragen, zeigt das folgende Beispiel.

Beispiel 12.39: Verwendung von *Graphics.GetHDc*

C#

```
using System.Runtime.InteropServices;
...
    [DllImport("gdi32.dll")]
    private  static extern int LineTo(IntPtr  hdc, int  x,int  y);
```

Achten Sie auf die obige DC-Deklaration als *IntPtr*, nur so können Sie problemlos den C#-DC verwenden.

```
...
    private void button1_Click(object sender, EventArgs e)
    {
      Graphics g;
      IntPtr mydc;
      g = this.CreateGraphics();     // Graphics-Objekt erzeugen
      mydc = g.GetHdc();             // DC abfragen
      LineTo(mydc, 0, 0);            // GDI-Funktionen verwenden ...
      LineTo(mydc, 100, 100);
      g.ReleaseHdc(mydc);            // DC freigeben
    }
```

Sie können natürlich auch gleich das im *Paint*-Ereignis bereitgestellte Objekt *e.Graphics* verwenden.

Formular-DC mit GetDC erzeugen

Über die Funktion *GetDC*, die ein Fenster-Handle als Parameter erwartet, können Sie ebenfalls einen DC erzeugen.

Beispiel 12.40: Verwendung von *GetDC*

C#

```
using System.Runtime.InteropServices;
...
    [DllImport("gdi32.dll")]
    private  static extern int LineTo(IntPtr  hdc, int  x, int  y);

    [DllImport("user32.dll")]
    private  static extern IntPtr GetDC(IntPtr  hwnd);

    [DllImport("user32.dll")]
    private  static extern int ReleaseDC(IntPtr  hwnd, IntPtr  hdc);
...
    private void button1_Click(object sender, EventArgs e)
    {
      IntPtr mydc;
      mydc = GetDC(this.Handle);
      LineTo(mydc, 0, 0);            // GDI-Funktionen verwenden ...
      LineTo(mydc, 100, 100);
      ReleaseDC(this.Handle, mydc);
    }
```

HINWEIS: Die gleiche Vorgehensweise ist bei allen Steuerelementen möglich, die ein Fenster-Handle bereitstellen.

12.5.3 Koordinatensysteme und Abbildungsmodi

Bisher sind Sie wahrscheinlich nur mit dem C#-eigenen Koordinatensystem in Berührung gekommen, dessen Nullpunkt sich zunächst in der linken oberen Ecke befindet. Ähnlich wie bei GDI+ können auch bei GDI verschiedene Skalierungsmodi sowie Koordinatensysteme gewählt werden. Zum Einsatz kommt die Funktion *SetMapMode*, mit der Sie einem Ausgabegerät (bzw. dessen DC) ein neues Koordinatensystem bzw. einen anderen Abbildungsmodus zuweisen können.

Für die Druckausgabe ist insbesondere der Abbildungsmodus *MM_LOMETRIC* interessant, in dem Sie unabhängig von der Druckerauflösung mit Millimetern arbeiten können.

Beispiel 12.41: Drucken eines 10 cm großen Rechtecks

C#

```
using System.Runtime.InteropServices;
...
    [DllImport("gdi32.dll")]
    private  static extern int  SetMapMode(IntPtr  hdc,int  nMapMode);

    private const int MM_LOMETRIC = 2;

    [DllImport("gdi32.dll")]
    private  static extern int  Rectangle(IntPtr  hdc,int  X1,int  Y1,int  X2,int
Y2);
...
    private void printDocument1_PrintPage(object sender,
Printing.PrintPageEventArgs e)
    {
      IntPtr mydc;
      mydc = e.Graphics.GetHdc();
      SetMapMode(mydc, MM_LOMETRIC);
      Rectangle(mydc, 100, -100, 1100, -1100);
      e.Graphics.ReleaseHdc(mydc);
    }
```

HINWEIS: Beachten Sie, dass die y-Koordinaten negativ angegeben werden müssen!

Die folgende Tabelle listet die zulässigen Konstanten auf.

Typ	Beschreibung
MM_ANISOTROPIC	Virtuelle Einheiten werden in willkürliche Einheiten mit willkürlich skalierten Achsen umgewandelt. Das Setzen des Abbildungsmodus verändert die aktuellen Fenster- und Grafikfenster-Einstellungen nicht. Zum Ändern der Einheiten für Orientierung und Skalierung sollte Ihre Anwendung die Funktionen *SetWindowExtEx* und *SetViewportExtEx* verwenden.
MM_HIENGLISH	Jede virtuelle Einheit wird in 0.001 Zoll umgewandelt. Positive x sind rechts, positive y oben. Beachten Sie, dass es bei diesem Abbildungsmodus sehr schnell zu Wertebereichsüberschreitungen kommen kann.
MM_HIMETRIC	Eine virtuelle Einheit wird in 0.01 Millimeter umgewandelt. Positive x sind rechts, positive y oben.
MM_ISOTROPIC	Virtuelle Einheiten werden in willkürliche Einheiten mit gleich skalierten Achsen umgewandelt, das heißt, eine Einheit auf der x-Achse entspricht einer Einheit auf der y-Achse. Um die gewünschten Einheiten und Achsenorientierungen zu bestimmen, müssen Sie die Funktionen *SetWindowExtEx* und *SetViewportExtEx* verwenden.
MM_LOENGLISH	Jede virtuelle Einheit wird in 0.01 Zoll umgewandelt. Positive x sind rechts, positive y oben.
MM_LOMETRIC	Eine virtuelle Einheit wird in 0.1 Millimeter umgewandelt. Positive x sind rechts, positive y oben.
MM_TEXT	Eine virtuelle Einheit wird in einen (1) Geräte-Bildpunkt gewandelt. Positive x sind rechts, positive y unten.
MM_TWIPS	Eine virtuelle Einheit wird in 1/20 Punkt umgewandelt. Da 1 Punkt gleich 1/72 Zoll ist, entspricht ein Twip 1/1440 Zoll. Positive x sind rechts, positive y oben.

Für die Ausgabe auf dem Bildschirm sind die Modi *MM_ANISOTROPIC* und *MM_ISOTROPIC* interessant, können Sie doch damit beliebige Koordinatensysteme realisieren. Allerdings ist der Modus *MM_ISOTROPIC* dahingehend eingeschränkt, dass x- und y-Achse im gleichen Maßstab skaliert werden, die Abbildung also proportional vergrößert bzw. verkleinert wird.

Die Funktionen *SetWindowExtEx* und *SetViewportExtEx* brauchen Sie für die Auswahl der Skalierungsfaktoren.

Beispiel 12.42: Eine technische Zeichnung mit den virtuellen Abmessungen 4000 x 4000 soll in den Clientbereich eines Formulars skaliert werden.

C#

```
using System.Runtime.InteropServices;
...
    private const int MM_ISOTROPIC = 7;

    [DllImport("gdi32.dll")]
    private  static extern int  SetMapMode(IntPtr  hdc,int  nMapMode);
```

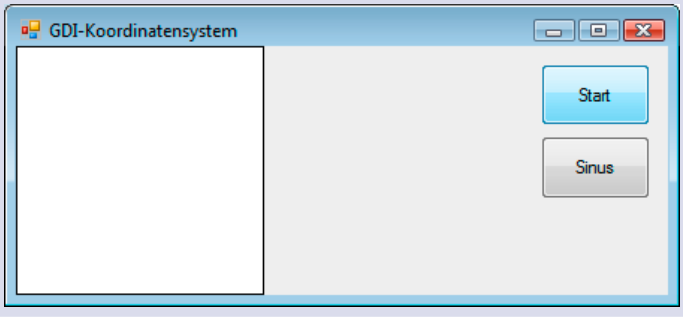
```
    [DllImport("gdi32.dll")]
    private  static extern int  SetWindowExtEx(IntPtr  hdc,int  nX,int  nY, int
lpSize);

    [DllImport("gdi32.dll")]
    private  static extern int  SetViewportExtEx(IntPtr  hdc,int  nX,int  nY, int
lpSize);

    [DllImport("gdi32.dll")]
    private  static extern int  Rectangle(IntPtr  hdc,int  X1,int  Y1,int  X2,int
Y2);

    [DllImport("user32.dll")]
    private  static extern IntPtr  GetDC(IntPtr  hwnd);

    [DllImport("user32.dll")]
    private  static extern int  ReleaseDC(IntPtr  hwnd,IntPtr  hdc);

    private void button1_Click(object sender, EventArgs e) {
      IntPtr mydc;
      mydc = GetDC(this.Handle);
      SetMapMode(mydc, MM_ISOTROPIC);
      SetWindowExtEx(mydc, 4000, 4000, 0);
      SetViewportExtEx(mydc, this.ClientSize.Width, this.ClientSize.Height, 0);
      Rectangle(mydc, 0, 0, 4000, 4000);    // ... zeigt die Größe der Zeichnung
      ReleaseDC(this.Handle, mydc);
    }
```

Nach der Auswahl des Abbildungsmodus (natürlich verwenden wir *MM_ISOTROPIC*, die Zeichnung soll ja nicht verzerrt werden) übergeben wir der Funktion *SetWindowExtEx* die virtuellen Maximalabmessungen unseres neuen Fensters. Die Größe des Viewports entspricht den realen Innenabmessungen des Formulars (*ScaleWidth*, *ScaleHeight*).

Alle weiteren Grafikausgaben (Text, Linien etc.) werden jetzt so skaliert, als ob das Fenster eine Innenabmessung von 4000 x 4000 Pixel hätte.

Ergebnis

Wie Sie der Abbildung entnehmen können, wird das Rechteck dank *MM_ISOTROPIC* unverzerrt dargestellt:



Möchten Sie positive y-Werte nach oben abtragen, können Sie auch folgende Zuweisung für *SetViewportExtEx* verwenden:

```
SetViewportExtEx(mydc, this.ClientSize.Width, -this.ClientSize.Height, 0);
```

Mithilfe der Funktion *SetWindowOrgEx* ist es weiterhin möglich, den Nullpunkt des Koordinatensystems beliebig zu verschieben. Sie brauchen bei späteren Grafikausgaben also nicht extra einen Offset zu addieren.

Beispiel 12.43: Zeichnen einer Sinuskurve im Bereich 0 ... 2π

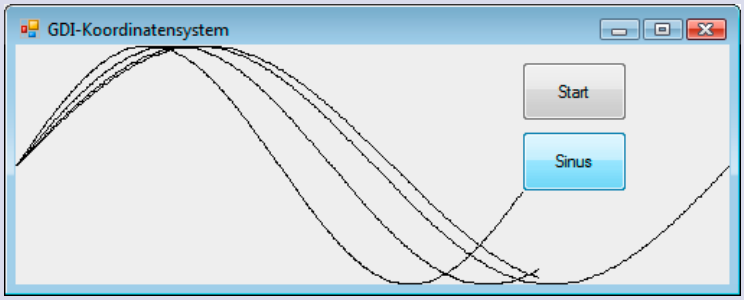
C#

Die Funktion soll so skaliert werden, dass der Clientbereich unabhängig von der Formulargröße voll ausgenutzt wird.

```
using System.Runtime.InteropServices;
...
    [StructLayout(LayoutKind.Sequential)]
      private  struct POINTAPI {
     public  int  x;
     public  int  y;  }

    private const int MM_ANISOTROPIC = 8;
    [DllImport("gdi32.dll")]
    private  static extern int  SetMapMode(IntPtr  hdc,int  nMapMode);

    [DllImport("gdi32.dll")]
    private  static extern int  SetWindowExtEx(IntPtr  hdc,int  nX,int  nY, int
lpSize);

    [DllImport("gdi32.dll")]
    private  static extern int  SetViewportExtEx(IntPtr  hdc,int  nX,int  nY, int
lpSize);
    [DllImport("user32.dll")]
    private  static extern IntPtr  GetDC(IntPtr  hwnd);

    [DllImport("user32.dll")]
    private  static extern int  ReleaseDC(IntPtr  hwnd,IntPtr  hdc);
    [DllImport("gdi32.dll")]
    private  static extern int  LineTo(IntPtr  hdc,int  x,int  y);

    [DllImport("gdi32.dll")]
    private  static extern int  SetWindowOrgEx(IntPtr  hdc,int  nX,int  nY,
             [MarshalAs(UnmanagedType.Struct)] ref  POINTAPI lpPoint);

    [DllImport("gdi32.dll")]
    private  static extern int  MoveToEx(IntPtr  hdc,int  x,int  y,
             [MarshalAs(UnmanagedType.Struct)] ref  POINTAPI lpPoint);
...
private void button1_Click(object sender, EventArgs e)
 {
  IntPtr mydc;
  POINTAPI p;
  int i;
  Single max;
```

```
    p.x = 0; p.y = 0;
    mydc = GetDC(this.Handle);
    max = (int) (2 * 3.1415926 * 100);
    SetMapMode(mydc, MM_ANISOTROPIC);
    SetWindowExtEx(mydc, (int) max, 200, 0);
    SetViewportExtEx(mydc, this.ClientSize.Width, - this.ClientSize.Height, 0);
    SetWindowOrgEx(mydc, 0, 100, ref p);   // Verschiebung Nullpunkt
    MoveToEx(mydc, 0, 0, ref p);
    for (i = 0; i <= max; i++)
      LineTo(mydc, i, (int) (100 * Math.Sin( i / 100f )));
    ReleaseDC(this.Handle, mydc);
  }
```

Ergebnis



Mit den oben genannten Funktionen ist lediglich eine Verschiebung bzw. Skalierung möglich. Wem das nicht reicht, der findet eine weitere GDI-Funktion, die alle mathematisch relevanten Transformationen vornehmen kann. Die Rede ist von *SetWorldTransform*, mit der Sie Verschiebungen, Skalierungen, Rotationen, Spiegelungen und Scherungen realisieren können.

HINWEIS: Das Gleiche gibt es auch in GDI+, siehe Abschnitt 12.1!

12.5.4 Zeichenwerkzeuge/Objekte

Für die Ausgabe von Grafiken (dazu zählt auch Text!) unterstützt Windows-GDI verschiedene grafische Objekte:

- Stifte (Pen)
- Pinsel (Brush)
- Bitmaps
- Paletten
- Schriftarten (Font)
- Regionen (Region)
- Paths

In den folgenden Abschnitten wollen wir kurz auf die wichtigsten Möglichkeiten und Funktionen eingehen.

Erzeugen und Parametrieren der Objekte

Bis auf wenige Ausnahmen müssen Sie alle oben genannten Objekte erst erzeugen (*CreatePen*, *CreateBrush* etc.), bevor Sie diese nutzen können. Neben dem Erstellen des neuen Objekts müssen Sie sich auch noch um das Sichern der bisherigen Objekte kümmern.

HINWEIS: Die neuen Objekte müssen wieder gelöscht werden, sonst werden wertvolle Systemressourcen verschwendet!

Beispiel 12.44: Erzeugen eines neuen Schrifttyps (Font-Objekt)

C#

```
using System.Runtime.InteropServices;
...
    [DllImport("user32.dll")]
    private  static extern IntPtr  GetDC(IntPtr  hwnd);

    [DllImport("user32.dll")]
    private  static extern int  ReleaseDC(IntPtr  hwnd,IntPtr  hdc);

    [DllImport("gdi32.dll")]
    private  static extern IntPtr  CreateFont(int  H,int  W,int  E,int  O,int
w,int  I,
             int  u,int  S,int  C,int  OP,int  CP,int  Q,int  PAF,string  F);

    [DllImport("gdi32.dll")]
    private  static extern IntPtr  SelectObject(IntPtr  hdc, IntPtr  hObject);

    [DllImport("gdi32.dll")]
    private  static extern IntPtr  DeleteObject(IntPtr  hObject);

    [DllImport("gdi32.dll")]
    private  static extern int  TextOut(IntPtr  hdc,int  x,int  y,string lpstring,
                                        int  nCount);
...
    private void button1_Click(object sender, EventArgs e)
    {
      IntPtr mydc, hFont, fontOld;
      mydc = (IntPtr) GetDC(this.Handle);
```

Neues Objekt erzeugen:

```
   hFont = CreateFont(-40, 0, 45 * 10, 0, 400, 0, 0, 0, 1, 4, 0x10, 2, 4, "Arial");
```

Altes Objekt sichern und neues Objekt setzen:

```
   fontOld = SelectObject(mydc, hFont);
```

GDI-Grafikausgaben:

```
   TextOut(mydc, 100, 100, "Hallo", 5);
```

Altes Objekt wiederherstellen:

```
SelectObject(mydc, fontOld);
```

Neues Objekt löschen:

```
DeleteObject(hFont);
ReleaseDC(this.Handle, mydc);
}
```

Das Zurücksetzen und Löschen des Objekts können Sie auch mit einer Anweisung realisieren

```
DeleteObject(SelectObject(mydc, fontOld));
```

... da der Rückgabewert der Funktion *SelectObject* der bisher gültige Handle ist.

HINWEIS: Eine weitere Variante, Objekte zu erzeugen, bietet sich mit der Funktion *GetStockObject*, über die bereits vordefinierte Systemobjekte abgerufen werden können. Diese Objekte brauchen nicht gelöscht zu werden.

12.5.5 Bitmaps

Eine Bitmap besteht neben dem Speicherabbild (Array aller Pixel) aus einem Header, der Größe und Auflösung beschreibt, sowie einer Palette. Windows unterstützt zwei Arten von Bitmaps:

- geräteabhängige (DDBs[4]),
- geräteunabhängige (DIBs[5]).

Bei geräteabhängigen Bitmaps wird die Grafik so im Arbeitsspeicher abgelegt, wie es das Ausgabegerät erfordert. Im Unterschied dazu werden geräteunabhängige Bitmaps in einem fest definierten Format abgelegt, also unabhängig vom Ausgabegerät.

An dieser Stelle wollen wir nicht weiter auf die Arbeit mit den Systemfunktionen eingehen, bietet doch C# im Zusammenhang mit Bitmaps fast schon alles, was der Programmierer so braucht. Bis auf eine Ausnahme: Wollen Sie Screenshots von einzelnen Fenstern bzw. vom gesamten Screen erzeugen, müssen Sie sich zwangsläufig mit einigen GDI-Funktionen beschäftigen.

[4] Device-Dependent Bitmap
[5] Device-Independent Bitmap

Beispiel 12.45: Einen Screenshot realisieren

C#

```
using System.Runtime.InteropServices;
...
private int SRCCOPY = 0xCC0020;

    [DllImport("gdi32.dll")]
    private  static extern int  BitBlt(IntPtr  hDestDC,int  x,int  y,int  nWidth,
             int  nHeight,IntPtr  hSrcDC,int  xSrc,int  ySrc,int  dwRop);

    [DllImport("user32.dll")]
    private  static extern IntPtr  GetDC(int  hwnd);
    [DllImport("user32.dll")]
    private  static extern int  ReleaseDC(int  hwnd,IntPtr  hdc);
...
      Graphics g1;
      IntPtr dc1, dc2;
      Image img;
```

Zunächst ein passendes Bitmap-Objekt in C# erstellen (Screengröße):

```
    img = new Bitmap(Screen.PrimaryScreen.WorkingArea.Width,
                     Screen.PrimaryScreen.WorkingArea.Height);
```

Ein *Graphics*-Objekt zuordnen:

```
   g1 = Graphics.FromImage(img);
```

Den DC des Desktops ermitteln:

```
   dc1 = GetDC(0);
```

Den DC unserer (leeren) Bitmap ermitteln:

```
   dc2 = g1.GetHdc();
```

Mittels *BitBlt* die Bitmap-Inhalte des Desktops in unserer C#-Bitmap kopieren:

```
   BitBlt(dc2, 0, 0, Screen.PrimaryScreen.WorkingArea.Width,
          Screen.PrimaryScreen.WorkingArea.Height, dc1, 0, 0, SRCCOPY);
```

Desktop-DC freigeben:

```
   ReleaseDC(0, dc1);
```

Bitmap-DC freigeben:

```
   g1.ReleaseHdc(dc1);
```

Im Weiteren können Sie jetzt alle C#-Möglichkeiten nutzen, beispielsweise kann die Bitmap im PNG-Format gespeichert werden:

```
   img.Save("c:\\Screen.png", Drawing.Imaging.ImageFormat.Png);
   ...
```

Einzelheiten zu den verschiedenen DCs sind in folgender Tabelle zusammengefasst:

Gerätekontext	Quellcode
DC gesamtes Fenster	*dc* = GetWindowDC(hWnd); *ReleaseDC(0, dc);*
DC Clientbereich des Fensters	*dc* = GetDC(hWnd); *ReleaseDC(hWnd, dc);*
DC Bildschirm	*dc = GetDC(0);* *ReleaseDC(0, dc);*

Bitmaps kopieren

Zum Kopieren von Bitmaps verwenden Sie die *BitBlt*-Funktion. Ziel und Quelle können identisch sein.

Syntax:

```
[DllImport("gdi32.dll")]
          private  static extern int BitBlt(IntPtr hDestDC,int x,int y,int nWidth,
int nHeight,
                                            IntPtr hSrcDC,int xSrc,int ySrc, int dwRop);
```

Der wichtigste Parameter ist *dwRop*, mit dem Sie die Art der Verknüpfung von Quelle und Ziel festlegen:

Parameter	Beschreibung
BLACKNESS	... schaltet alle Ausgaben auf Schwarz
DSTINVERT	... invertiert die Ziel-Bitmap
MERGECOPY	... verknüpft das Muster mit der Quell-Bitmap unter Verwendung der AND-Operation
MERGEPAINT	... verknüpft die invertierte Quell-Bitmap mit der Ziel-Bitmap unter Verwendung der OR-Operation
NOTSRCCOPY	... kopiert die invertierte Quell-Bitmap in die Ziel-Bitmap
NOTSRCERASE	... invertiert das Ergebnis der OR-Verknüpfung von Quell- und Ziel-Bitmap
PATCOPY	... kopiert das Muster in die Ziel-Bitmap
PATINVERT	... verknüpft die Ziel-Bitmap mit dem Muster unter Verwendung der XOR-Operation
PATPAINT	... verknüpft die invertierte Quell-Bitmap mit dem Muster unter Verwendung der OR-Bedingung, verknüpft das Ergebnis dieses Vorgangs mit der Ziel-Bitmap unter Verwendung der OR-Operation
SRCAND	... verknüpft Pixel der Quell- und Ziel-Bitmaps unter Verwendung der AND-Operation
SRCCOPY	... kopiert die Quell-Bitmap auf die Ziel-Bitmap
SRCERASE	... invertiert die Ziel-Bitmap und verknüpft das Ergebnis mit der Quell-Bitmap unter Verwendung der AND-Operation

(Fortsetzung nächste Seite)

(Fortsetzung)

Parameter	Beschreibung
SRCINVERT	... verknüpft Pixel von Quell- und Ziel-Bitmaps unter Verwendung der XOR-Operation
SRCPAINT	... verknüpft Pixel von Quell- und Ziel-Bitmaps unter Verwendung der OR-Operation
WHITENESS	... schaltet alle Ausgaben auf Weiß

HINWEIS: Die Werte für die oben genannten Parameter können Sie auch im Zusammenhang mit der *StretchBlt*-Funktion verwenden.

Natürlich sind die meisten der Optionen nur mit zwei verschiedenen Bitmaps sinnvoll anwendbar.

Bitmaps skalieren

Obwohl das Skalieren von Bitmaps nicht allzu sinnvoll ist (Ausgabequalität!), finden sich doch einige Anwendungsgebiete. Möchten Sie beispielsweise eine Bitmap in bestimmter Größe auf dem Drucker ausgeben (10 x 10 cm), bleibt Ihnen meist nichts anderes übrig als die Skalierung der Bitmap. Das GDI bietet dazu die Funktion *StretchBlt*:

Syntax:

```
[DllImport("gdi32.dll")]
          private static extern int StretchBlt(IntPtr hDestDC,int x,int y,
                                               int nWidth, int nHeight,
                                               IntPtr hSrcDC,int xSrc,
                                               int ySrc,int nSrcWidth,
                                               int nSrcHeight,int dwRop);
```

Beispiel 12.46: Durch geschickte Wahl der Koordinaten lässt sich eine Bitmap in horizontaler und vertikaler Richtung spiegeln.

C#

Spiegeln vertikal:

```
StretchBlt(myDC1, 0, 0, Breite, Höhe, myDC2, 0, Höhe, Breite, Höhe, SRCCOPY);
```

Spiegeln horizontal:

```
Stretchblt(myDC1, 0, 0, Breite, Höhe, myDC2, Breite, 0, -Breite, Höhe, SRCCOPY);
```

HINWEIS: Sind Quell- und Ziel-Bitmap gleich groß, sollten Sie zum Kopieren besser die *BitBlt*-Funktion verwenden, da diese schneller ist.

Damit wollen wir die Ausführungen zu GDI beenden, auch wenn noch weitere interessante Themen wie XOR-Zeichenmodus etc. damit möglich sind. Die grundsätzliche Vorgehensweise dürfte dennoch klar geworden sein.

12.6 Praxisbeispiele

12.6.1 Die Transformationsmatrix verstehen

Wem die Ausführungen in Abschnitt 12.1 zu theoretisch waren, hier ist ein praktisches Beispiel zum Experimentieren.

Über sechs *TextBox*en erhalten Sie die Möglichkeit, die Matrix-Elemente *m11* bis *dy* zur Laufzeit frei zu verändern. Einige weitere, bereits vorkonfigurierte Beispiele können Sie über eine *ComboBox* abrufen.

Oberfläche

Siehe Laufzeitansicht am Schluss des Beispiels. In die *ComboBox* tragen Sie folgende Werte ein:

```
-
Reset
Rotate(15)
Rotate(45)
Shear(0.1,0)
Shear(0,0.1)
Translate(5,0)
Translate(0,5)
Scale(2,0)
Scale(0,2)
```

Quelltext

Binden Sie zunächst den Namespace *System.Drawing.Drawing2D* ein.

Das Button-Klick-Ereignis:

```
Matrix m = new Matrix(Convert.ToSingle(textBox1.Text), Convert.ToSingle(textBox2.Text),
                      Convert.ToSingle(textBox3.Text), Convert.ToSingle(textBox4.Text),
                      Convert.ToSingle(textBox5.Text), Convert.ToSingle(textBox6.Text));
```

Wie Sie sehen, füllen wir die Matrix gleich beim Erstellen. Auf eine Fehlerprüfung haben wir verzichtet.

```
Graphics g = this.CreateGraphics();
```

Ausgabe löschen und zunächst ein Rechteck **ohne** Transformation zeichnen:

```
g.Clear(this.BackColor);
g.DrawRectangle(Pens.Red, 10, 10, 50, 100);
```

Transformationsmatrix anwenden:

```
g.Transform = m;
```

Die eigentlichen Zeichenfunktionen realisieren:

```
g.DrawLine(Pens.Red, 0, 0, 200, 0);
g.DrawLine(Pens.Red, 0, 0, 0, 200);
g.DrawRectangle(Pens.Black, 10, 10, 50, 100);
```

Die über die *ComboBox* abrufbaren Beispiele:

```
        private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
        {
            Matrix m;
```

Erster Eintrag (ohne Funktion):

```
            if (comboBox1.SelectedIndex == 0) return;
```

Zweiter Eintrag (Zurücksetzen der Matrix):

```
            if (comboBox1.SelectedIndex == 1)
            {
                m = new Matrix(1, 0, 0, 1, 0, 0);
            }
```

Auslesen der *TextBox*en:

```
            else
            {
                m = new Matrix(Convert.ToSingle(textBox1.Text),
                               Convert.ToSingle(textBox2.Text),
                               Convert.ToSingle(textBox3.Text),
                               Convert.ToSingle(textBox4.Text),
                               Convert.ToSingle(textBox5.Text),
                               Convert.ToSingle(textBox6.Text));
            }
            Graphics g = this.CreateGraphics();
            g.Clear(this.BackColor);
            g.DrawRectangle(Pens.Red, 10, 10, 50, 100);
```

Direktes Verändern der Matrix durch Methodenaufrufe:

```
            switch (comboBox1.SelectedIndex)
            {
              case 2:   // Rot 15
                  m.Rotate(15);
                  break;
              case 3:   // Rot 45
                  m.Rotate(45);
                  break;
              case 4:
                  m.Shear(0.1f, 0);
                  break;
              case 5:
                  m.Shear(0, 0.1f);
                  break;
              case 6:
                  m.Translate(5, 0);
                  break;
```

```
            case 7:
                m.Translate(0, 5);
                break;
            case 8:
                m.Scale(2, 0);
                break;
            case 9:
                m.Scale(0, 2);
                break;
            default:
                  break;
        }
```

Die neuen Werte in die *TextBox*en eintragen:

```
        textBox1.Text = m.Elements[0].ToString();
        textBox2.Text = m.Elements[1].ToString();
        textBox3.Text = m.Elements[2].ToString();
        textBox4.Text = m.Elements[3].ToString();
        textBox5.Text = m.Elements[4].ToString();
        textBox6.Text = m.Elements[5].ToString();
```

Transformationsmatrix zuordnen:

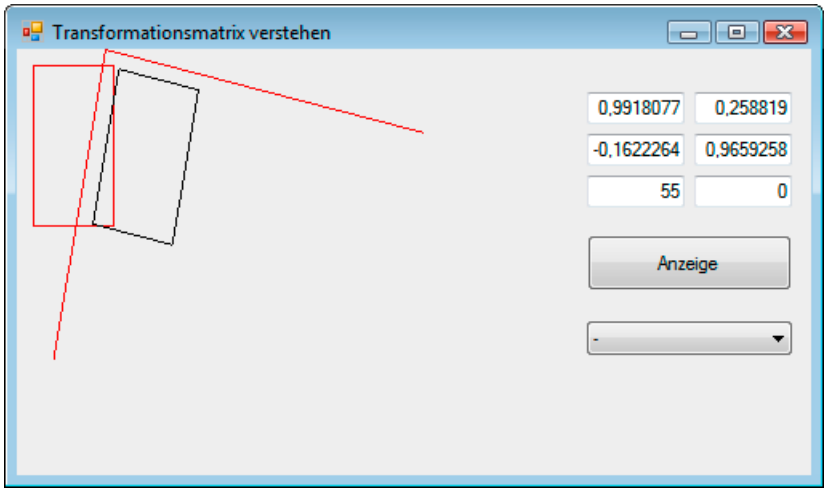
```
        g.Transform = m;
```

Grafikausgabe:

```
        g.DrawLine(Pens.Red, 0, 0, 200, 0);
        g.DrawLine(Pens.Red, 0, 0, 0, 200);
        g.DrawRectangle(Pens.Black, 10, 10, 50, 100);
        comboBox1.SelectedIndex = 0;
    }
```

Test

Starten Sie die Anwendung und versuchen Sie sich zunächst an Skalierungen und Translationen (siehe Abschnitt 12.1). Wer zunächst einen Überblick bekommen möchte, setzt die Matrix über die *ComboBox* zurück und wählt dann eine Transformation per *ComboBox*. Die Änderung der Werte in den *TextBox*en dürften schnell für mehr Klarheit sorgen:



12.6.2 Eine 3D-Grafikausgabe in Aktion

Nachdem wir Sie im Abschnitt 12.4 mit endlosen Ausführungen zur 3D-Grafik traktiert haben, möchten Sie die Algorithmen sicher auch in Aktion erleben.

Oberfläche

Fügen Sie entsprechend der Abbildung am Ende des Beispiels Schaltflächen für die Skalierung, die Translation und die Rotation ein:

Der *TrackBar* bestimmt die Refresh-Geschwindigkeit (ein *Timer* ist auch noch einzufügen).

Quelltext

Auf die Algorithmen und die Klasse *c3D* sind wir ja bereits im Abschnitt 12.4 eingegangen. An dieser Stelle beschränken wir uns auf das Anzeigeprogramm.

```
public partial class Form1 : Form
{
```

Eine Instanz unserer 3D-Klasse bilden:

```
    c3D my3DSystem = new c3D();
```

In dieser Variablen speichern wir die angeklickte Schaltfläche (wird im *Timer* ausgewertet):

```
    object cmd;
```

Im Konstruktor kümmern wir uns zunächst um die Anzeigequalität (Flimmern verringern):

```
    public Form1()
    {
        InitializeComponent();
```

```
            SetStyle(ControlStyles.UserPaint, true);
            SetStyle(ControlStyles.AllPaintingInWmPaint, true);
            SetStyle(ControlStyles.DoubleBuffer, true);
        }
```

Die eigentliche Anzeigeroutine fällt recht kurz aus:

```
        protected override void OnPaint(PaintEventArgs e)
        {
            my3DSystem.Draw(e.Graphics);
        }
```

Wird auf eine der Tasten geklickt, starten wir den *Timer*:

```
        private void button1_MouseDown(object sender, MouseEventArgs e)
        {
            cmd = sender;
            timer1.Start();
        }
```

Beim Loslassen halten wir den *Timer* an:

```
        private void button1_MouseUp(object sender, MouseEventArgs e)
        {
            timer1.Stop();
        }
```

Die Refresh-Geschwindigkeit verändern:

```
        private void trackBar1_Scroll(object sender, EventArgs e)
        {
            timer1.Interval = trackBar1.Value;
        }
```

Ach ja, die zu manipulierenden Objekte müssen wir ja auch noch definieren:

```
        private void Form1_Load(object sender, EventArgs e)
        {
            my3DSystem.AddObject(new point3D(0, 0, 0));
            my3DSystem.AddObject(new coordinateSystem3D(3));
            my3DSystem.AddObject(new line3D(-2, 1, 2, 1, 1, 2));
            my3DSystem.AddObject(new line3D( -2, 1, 2, -2, -1, 2));
...
            my3DSystem.AddObject(new line3D(0.5f, 0.5f, -1, 0.5f, -1, -1));
            my3DSystem.Scale(50);
        }
```

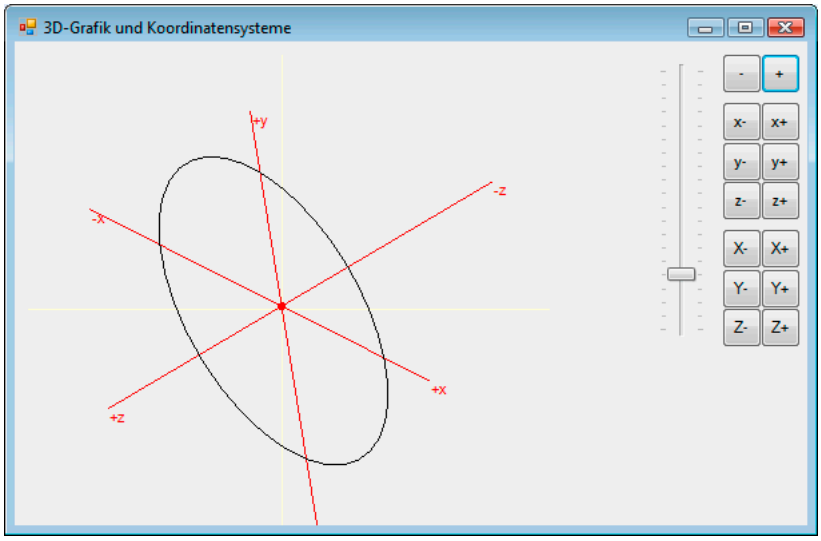
Der *Timer* kümmert sich um die Transformationen:

```
        private void timer1_Tick(object sender, EventArgs e)
        {
            switch ( (cmd as Button).Text)
            {
                case "+":
                    my3DSystem.Scale(1.1f);
                    break;
...
```

```
                case "Z+":
                    my3DSystem.Rotate(0, 0, 1);
                    break;
                default:
                    break;
            }
            this.Invalidate();
        }
```

Test/Bemerkungen

Natürlich können Sie das Programm zunächst so testen, wie es ist. Besser ist es jedoch, wenn Sie sich auch mal daran versuchen, ein anderes Objekt in unserem 3D-Testlabor anzuzeigen.



Die Lösung:

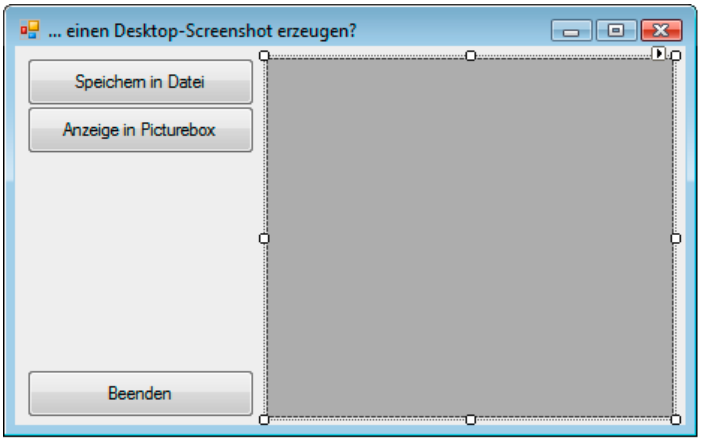
```
            Single xold, yold, xnew, ynew, r;

            r = 2;
            xold = r;
            yold = 0;
            for (int i = 0; i <= 360; i += 10)
            {
                xnew = Convert.ToSingle(r * Math.Cos(i * Math.PI / 180));
                ynew = Convert.ToSingle(r * Math.Sin(i * Math.PI / 180));
                my3DSystem.AddObject(new line3D(xold, yold, 0, xnew, ynew, 0));
                xold = xnew;
                yold = ynew;
            }
```

12.6.3 Einen Fenster-Screenshot erzeugen

Geht es darum, einen Screenshot vom aktuellen Fenster zu erzeugen, kommen Sie um ein wenig GDI-Programmierung nicht herum.

Oberfläche

Erstellen Sie zunächst eine Oberfläche entsprechend folgender Abbildung (*PictureBox*, drei *Button*s):



Quelltext

```
using System.Runtime.InteropServices;

public partial class Form1 : Form
{
    ...
```

Binden Sie nachfolgende Konstante sowie die GDI-Funktion *BitBlt* ein:

```
    private const int SRCCOPY = 0xCC0020;

    [DllImport("gdi32.dll")]
    private  static extern int  BitBlt(IntPtr  hDestDC,int  x,int  y,int  nWidth,
                                       int  nHeight, IntPtr  hSrcDC,int  xSrc,
                                       int  ySrc,int  dwRop);
    ...
```

Die Routine zum Speichern des Screenshots:

```
    private void Button1_Click(object sender, EventArgs e)
    {
        Graphics g1, g2;
        IntPtr dc1, dc2;
        Image img;
```

Erzeugen einer neuen Bitmap mit den Maßen und der Farbtiefe des aktuellen Fensters:

```
        g1 = this.CreateGraphics();
        img = new Bitmap(this.ClientRectangle.Width, this.ClientRectangle.Height, g1);
        g2 = Graphics.FromImage(img);
```

Kopieren der Fenster-Bitmap in die eigene Bitmap:

```
        dc1 = g1.GetHdc();
        dc2 = g2.GetHdc();
        BitBlt(dc2, 0, 0, this.ClientRectangle.Width,
               this.ClientRectangle.Height, dc1, 0, 0, 13369376);
        g1.ReleaseHdc(dc1);
        g2.ReleaseHdc(dc2);
```

Speichern der Daten im PNG-Format:

```
        img.Save("c:\\Form1.png", System.Drawing.Imaging.ImageFormat.Png);
        MessageBox.Show("Fenster-Screenshot gesichert", "Info");
    }
```

Ähnlich gestaltet sich die Routine zur Anzeige in der *PictureBox*:

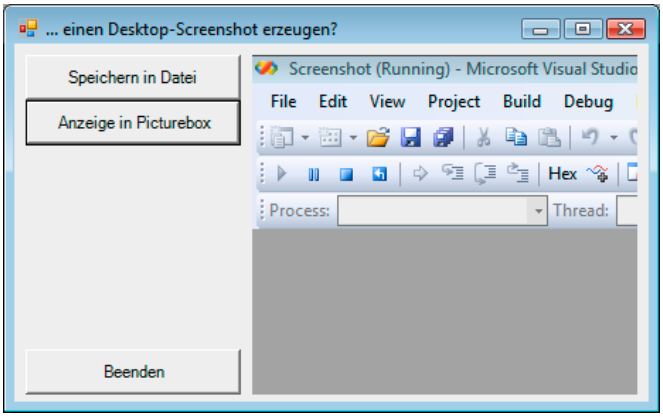
```
    private void Button2_Click(object sender, EventArgs e)
    {
       Graphics g1, g2;
       IntPtr dc1, dc2;
       Image img;

       g1 = this.CreateGraphics();
       img = new Bitmap(this.ClientRectangle.Width, this.ClientRectangle.Height, g1);
       g2 = Graphics.FromImage(img);

       dc1 = g1.GetHdc();
       dc2 = g2.GetHdc();
       BitBlt(dc2, 0, 0, this.ClientRectangle.Width, this.ClientRectangle.Height,
              dc1, 0, 0, 13369376);
       g1.ReleaseHdc(dc1);
       g2.ReleaseHdc(dc2);
```

Wir weisen das *Image* der *PictureBox* zu:

```
        pictureBox1.Image = img;
    }
    ...
}
```

Test

Nach dem Programmstart und dem Klick auf den Button „Anzeige in Picturebox“ sollte sich Ihnen der folgende Anblick bieten:



Die erzeugte Datei *Form1.png* findet sich im Verzeichnis „C:\“. Sie können diese zum Beispiel mit dem Internet Explorer anzeigen oder aber auch mit der in Windows integrierten Bild- und Faxanzeige.

13 Ressourcen/ Lokalisierung

Über Ressourcen können Sie externe Informationen in Ihr Programm aufnehmen. Das betrifft Texte, Grafiken und andere Elemente, die sich nicht ohne Weiteres per Code darstellen lassen. Das .NET-Framework kennt prinzipiell zwei Typen von Ressourcen:

- Manifestressourcen und
- typisierte Ressourcen.

Mit beiden Ressourcentypen und ihren Ableitungen (streng typisierte Ressourcen) werden wir uns in diesem Kapitel auseinandersetzen.

Eine besondere Bedeutung haben typisierte Ressourcen vor allem im Zusammenhang mit der Lokalisierung von Anwendungen (siehe Abschnitt 13.4).

13.1 Manifestressourcen

Manifestressourcen können nahezu beliebige Dateien sein, die zur Entwurfszeit (als Ergebnis des Build-Prozesses) in die Assembly integriert werden. Eine Assembly kann mehrere Manifestressourcen enthalten. „Manifestressource“ heißt es deshalb, weil die Namen der Ressourcen im Manifest der Assembly abgelegt sind. Zur Laufzeit kann jede Manifestressource über ihren Namen als Stream ausgelesen werden.

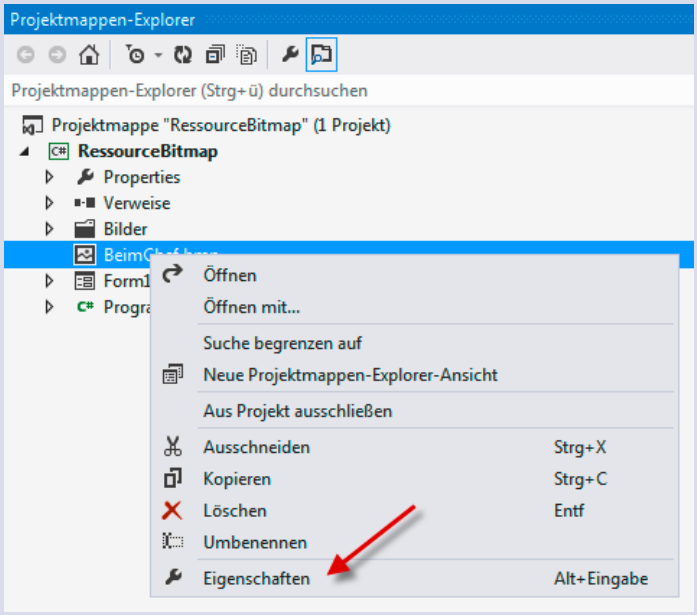
13.1.1 Erstellen von Manifestressourcen

Normalerweise verwendet man zum Anlegen einer Manifestressource den Assembly Linker (*al.exe*). Besitzer von Visual Studio können aber auf dieses Tool locker verzichten, da sie die entsprechenden Dateien lediglich zum Projekt hinzufügen und die *Buildvorgang*-Eigenschaft auf *Eingebettete Ressource* setzen müssen.

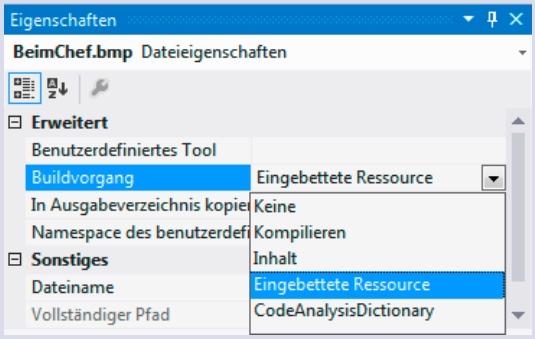
Beispiel 13.1: Manifestressource erstellen

C#

Um die Bilddatei *Bild1.jpg* als Manifestressource anzulegen, wählen Sie im Kontextmenü des Projektmappen-Explorers **Hinzufügen | Vorhandenes Element...** (oder Hauptmenü **Projekt | Vorhandenes Element hinzufügen...**) und selektieren die Datei *Bild1.jpg*.
Im Kontextmenü von *Bild1.jpg* klicken Sie auf **Eigenschaften**:



Im Eigenschaftendialog stellen Sie die Eigenschaft *Buildvorgang* auf *Eingebettete Ressource*:



Nach dem Kompilieren der Anwendung befindet sich die Bilddatei in der erzeugten Assembly, Sie brauchen also keine weitere Datei mitzugeben.

13.1.2 Zugriff auf Manifestressourcen

Der übliche Weg zu eingebetteten Ressourcen führt über die Methoden der *Assembly*-Klasse. So kann man die *GetManifestResourceNames*- bzw. *GetManifestResourceStream*-Methode verwenden, um alle eingebetteten Ressourcen einer bestimmten Assembly aufzulisten bzw. zu laden. Bevor wir aber zu Einzelheiten kommen, wollen wir erläutern, nach welchem Muster die Namensvergabe erfolgt.

Namensgebung eingebetteter Ressourcen

Allgemein entspricht der Name einer eingebetteten Ressource folgendem Muster:

Syntax:

```
<DefaultNamespace>.<Unterverzeichnisse>.Dateiname
```

Der *DefaultNamespace* wird in den Projekteigenschaften festgelegt und ist meist identisch mit dem Namen der Assembly (z. B. *WindowsApplication1*). Falls sich – wie in unserem Fall – die Ressource im Rootverzeichnis des Projekts befindet, entfallen die *Unterverzeichnisse* und der Name ist *<DefaultNamespace>.Dateiname*. Mehrere Unterverzeichnisse sind nicht durch Schrägstrich, sondern durch Punkte voneinander zu trennen. Es ist möglich, dass Sie spezielle Verzeichnisse innerhalb Ihres Projekts erzeugen, um dort die Ressource(n) abzulegen, z. B. ein Verzeichnis *\Bilder*. Dann wäre der Name der Ressource z. B. *WindowsApplication1.Bilder.Bild1.jpg*.

HINWEIS: Ist der Code einmal generiert, so können Sie zwar den Namespace nachträglich ändern, nicht aber die Unterverzeichnisse, die zur Benennung einer eingebetteten Ressource benutzt wurden!

Auflisten aller eingebetteten Ressourcen

Für diesen Zweck kommt die *GetManifestResourceNames*-Methode der *Assembly*-Klasse zur Anwendung.

Beispiel 13.2: Auflisten aller eingebetteten Ressourcen

C#

Über einen Dateidialog wird eine Assembly geladen. Die Namen aller darin enthaltenen Ressourcen werden in einer *ListBox* angezeigt.

```
using System.Reflection;
using System.IO;
...

Assembly  ass = Assembly.LoadFrom(openFileDialog1.FileName);
string [] resNames = ass.GetManifestResourceNames();
listBox1.Items.Clear();
if( resNames.Length > 0 )
{
```

```
    listBox1.BeginUpdate();
    foreach(string resName in resNames)
    {
      listBox1.Items.Add(resName);
    }
    listBox1.EndUpdate();
}
```

Die Inhalte eingebetteter Ressourcen auslesen

Hierfür verwenden Sie die *GetManifesResourceStream*-Methode der *Assembly*-Klasse:

Beispiel 13.3: Eine in der eigenen Assembly eingebettete Bildressource wird angezeigt.

C#

```
...
using System.Reflection;
using System.IO;
...
Assembly ass = Assembly.GetExecutingAssembly();
if (pictureBox1.Image != null)
   pictureBox1.Image.Dispose();
Stream strm = ass.GetManifestResourceStream("RessourceBitmap.Bild1.jpg");
pictureBox1.Image = new Bitmap(strm);
```

Die *GetManifestResourceStream*-Methode hat zwei Überladungen. Die erste erwartet den Namen der Ressource, die zweite erwartet stattdessen einen Typ und einen String. Das vereinfacht die Namensgewinnung für die Ressource: Intern wird der Namespace des Typs genommen und mit dem String wird der Name vervollständigt.

Einige Klassen des .NET Framework verwenden dieses Verfahren ebenfalls, aber anstatt *Stream*-Objekte zurückzugeben, erzeugen sie damit ein bestimmtes Objekt. So hat die *Bitmap*-Klasse einen Konstruktor, der eine eingebettete Ressource in ein *Bitmap*-Objekt laden kann.

Beispiel 13.4: Eine deutlich kürzere Version des obigen Beispiels

C#

```
if (pictureBox1.Image != null)
      pictureBox1.Image.Dispose();
pictureBox1.Image = new Bitmap(typeof(Form1), "Bild1.jpg");
```

Dem *Bitmap*-Konstruktor wird auf diese Weise mitgeteilt, eine Ressource zu finden, deren *Namespace* dem von *Form1* entspricht (weil *Form1* im Rootverzeichnis des Projekts liegt).

Eine weitere Variante der zweiten Zeile:

```
pictureBox1.Image = new Bitmap(this.GetType(), "Bild1.jpg");
```

13.2 Typisierte Ressourcen

Typisierte Ressourcen bauen auf einfachen Manifestressourcen auf. Es handelt sich hierbei um Zusammenstellungen von Schlüssel-Wert-Paaren, wobei der Schlüssel ein eindeutiger String und der Wert ein beliebiges Objekt ist. Im Gegensatz zu den Manifestressourcen sind diese Ressourcen typisiert und wesentlich effizienter, da sie nicht streambasiert arbeiten. Die Bereitstellung erfolgt in der Regel als Datei mit der Extension *.resources*.

13.2.1 Erzeugen von .resources-Dateien

Dazu benötigen Sie einen *ResourceWriter* aus dem Namespace *System.Resources*.

Beispiel 13.5: Erzeugen einer *.resources*-Datei

C#

Die Methode *createResources* erzeugt eine Ressourcendatei, die Texte für beliebige Meldungen enthält.

```
        static void createResources()
        {
            ResourceWriter rw = new
 System.Resources.ResourceWriter("Messages.resources");
```

Die String-Ressourcen als Schlüssel-Wert-Paar hinzufügen:

```
            rw.AddResource("1", "Wahrscheinlich haben Sie recht.");
            rw.AddResource("2", "Ja, natürlich.");
            rw.AddResource("3", "Versuchen Sie es später noch einmal!");
            rw.AddResource("4", "Bitte keine Ablenkungsmanöver!");
            rw.AddResource("5", "Leider nein.");
            rw.AddResource("6", "Wie ich sehe - ja!");
```

Die Datei *Messages.resources* wird erzeugt und geschlossen:

```
            rw.Generate();
            rw.Close();
        }
```

Nach Aufruf von *createResources()* werden Sie im Unterverzeichnis *\bin\Debug* die Datei *Messages.resources* vorfinden.

13.2.2 Hinzufügen der .resources-Datei zum Projekt

Die erzeugte Ressourcendatei *Messages.resources* fügen Sie – genauso wie oben für eine Manifestressource beschrieben – dem Projekt hinzu. Wählen Sie also den Menüpunkt **Projekt | Vorhandenes Element hinzufügen...** und setzen Sie die *Buildvorgang*-Eigenschaft auf

Eingebettete Ressource. Alternativ können Sie auch das *Hinzufügen*-Kontextmenü im Projektmappen-Explorer verwenden.

HINWEIS: Da die Datei *Messages.ressources* nach dem Kompilieren in der Assembly *ResourcesTest.exe* eingebettet ist, können Sie sie auch aus dem Verzeichnis *\bin\Debug* löschen.

13.2.3 Zugriff auf die Inhalte von .resources-Dateien

Für den Zugriff spielt die Klasse *ResourceManager* (Namespace *System.Resources*) eine wichtige Rolle. Sie benötigt zur Instanziierung zwei Parameter: einen Verweis auf die *resources*-Datei, die als Manifestressource in der Assembly abgelegt ist, sowie einen Verweis auf die Assembly selbst.

Der *ResourceManager* stellt für den Zugriff auf Ressourcenelemente die Methoden *GetString*, *GetObject* und *GetStream* bereit. Anzugeben ist der Name des Ressourcenelements unter Beachtung der Groß-/Kleinschreibung. Die Bedeutung der Groß-/Kleinschreibung kann aber mittels *IgnoreCase*-Eigenschaft deaktiviert werden.

Beispiel 13.6: Zugriff auf die Inhalte von .resources-Dateien

C#

Der Zugriff auf die im Projekt *ResourcesTest* eingebettete Datei *Messages.resources* erfolgt über die *GetString*-Methode des *ResourceManager*, wobei der Methode der Schlüssel als Parameter übergeben wird. Um einen einfachen Test zu ermöglichen, erzeugen wir den Schlüssel mit einem Zufallsgenerator, der zugehörige Wert wird in einem *Label* angezeigt.

```
using System.Resources;
...
private ResourceManager rm =  new ResourceManager("ResourcesTest.Messages",

System.Reflection.Assembly.GetExecutingAssembly());
System.Random z = new System.Random();
string num = z.Next(1, 41).ToString();
label1.Text = rm.GetString(num);
```

Bemerkungen zum Zugriff auf .resources-Dateien

- Die Extension *resources* ist **kein** Bestandteil des Namens, der dem Konstruktor als Argument übergeben wird.
- Das Objekt für die Assembly, in der sich der laufende Code befindet, erhält man über die Methode *System.Reflection.Assembly.GetExecutingAssembly*.
- Die *GetString*-Methode des *ResourceManager* ist eine typsichere Alternative zur *GetObject*-Methode.

13.2.4 ResourceManager einer .resources-Datei erzeugen

Wollen Sie die Ressourcen nachträglich manipulieren, so ist die gezeigte Vorgehensweise nicht geeignet, da die *.resources*-Datei unmittelbar nach ihrem Erzeugen erst per Hand in das Projekt „eingebettet" werden muss und nach dem Kompilieren – wie jede andere Manifestressource auch – nicht nachträglich manipuliert werden kann.

Ein Ausweg bietet sich, wenn Sie die Datei nicht in die Assembly einbetten, sondern als separate Datei mitführen. Dann können Sie einen passenden *ResourceManager* mithilfe der statischen Methode *CreateFilebasedResourceManager* erzeugen.

Beispiel 13.7: Erzeugen eines *ResourceManager* aus der Datei *Messages.resources*.

C#

```
private ResourceManager rm = ResourceManager.CreateFileBasedResourceManager(
                                  "Messages", Application.StartupPath, null);
```

In diesem Fall befindet sich die Datei *Messages.resources* im selben Verzeichnis wie die Assembly.

Bemerkungen

- Das Pendant zum *ResourceWriter* ist der *ResourceReader*, mit dem sich – als Alternative zum gezielten Zugriff per *ResourceManager* – komplette Ressourcendateien verarbeiten lassen.
- Bequemer als per Code können Ressourcen auch mit dem in Visual Studio integrierten Ressourceneditor angelegt werden (Menü **Projekt | Neues Element hinzufügen... | Ressourcendatei**) .

13.2.5 Was sind .resx-Dateien?

Ressourcen liegen in *resources*-Dateien in einem binären Format vor. Anstatt, wie oben gezeigt, mittels *RessourceWriter* lassen sie sich aber auch aus **.resx*-Dateien erzeugen.

HINWEIS: *resx*-Dateien sind XML-Dokumente!

Zur Umwandlung einer binären *resources*- in eine XML-basierte *resx*-Datei können Sie das Tool *resgen.exe* (*Resource File Generator*) einsetzen.

Visual Studio bietet einen komfortablen Editor für *resx*-Dateien. Der Aufruf erfolgt über den Menüpunkt **Projekt | Neues Element hinzufügen... | Ressourcendatei** oder einfach durch Doppelklick auf eine vorhandene *resx*-Datei im Projektmappen-Explorer.

HINWEIS: *resx*-Ressourcendateien haben besondere Bedeutung im Zusammenhang mit der Lokalisierung von Anwendungen. Dieses Thema wird in Abschnitt 13.4 behandelt.

13.3 Streng typisierte Ressourcen

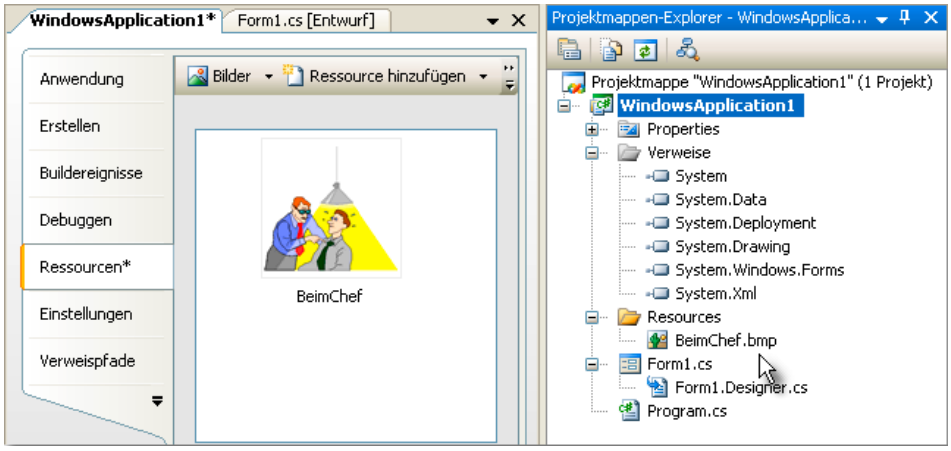
Beim Arbeiten mit dem *ResourceManager*-Objekt kann es häufig zu Fehlern kommen, da die Namen der Ressourcenelemente als Zeichenketten vorliegen und Tippfehler dazu führen, dass das Element nicht gefunden wird. Aus diesem Grund werden im .NET-Framework auch streng typisierte Ressourcen (*strongly-typed resources*) unterstützt.

So bietet das Tool *resgen.exe* die Option, eine Wrapper-Klasse für eine beliebige Ressourcendatei zu generieren, sodass der Programmierer mit *early binding* auf die Ressourcennamen zugreifen kann. Laufzeitfehler aufgrund falscher Ressourcennamen lassen sich so vermeiden.

13.3.1 Erzeugen streng typisierter Ressourcen

Visual Studio stellt einen Designer bereit, der unter anderem auch das automatische Generieren von Wrapper-Klassen für Ressourcen übernimmt. Die Bedienung ist sehr einfach:

Über das Menü **Projekt | ...-Eigenschaften...** (oder durch Doppelklick auf den *Properties*-Eintrag im Projektmappen-Explorer) öffnen Sie die Seite „Ressourcen“ des Projektdesigners.



Im Ergebnis wurde innerhalb des Projektverzeichnisses ein neues Unterverzeichnis namens */Resources* angelegt, in dem die Bitmap gespeichert ist.

13.3.2 Verwenden streng typisierter Ressourcen

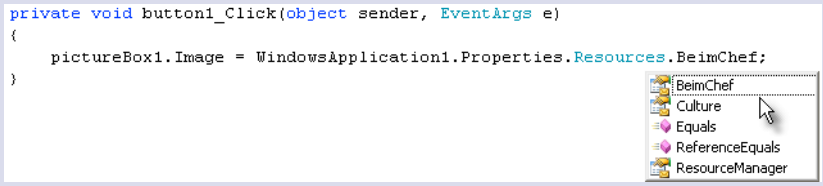
Die Verwendung der streng typisierten Ressourcen ist recht simpel, der Zugriff erfolgt über das *Properties.Resources*-Objekt.

Beispiel 13.8: Anzeige einer Bitmap

C#

```
pictureBox1.Image = WindowsApplication1.Properties.Resources.BeimChef;
```

Da auch eine Unterstützung per IntelliSense erfolgt, wird die Fehleranfälligkeit deutlich verringert:



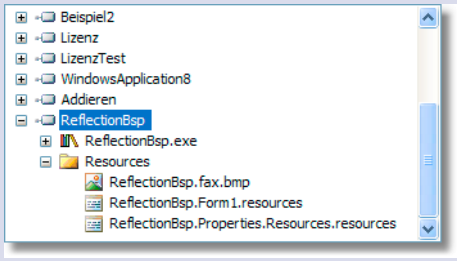
13.3.3 Streng typisierte Ressourcen per Reflection auslesen

Für den Zugriff auf Ressourcen wie Grafiken, Videos, Sound etc. bietet sich die *GetManifest-ResourceStream*-Methode an.

Beispiel 13.9: Ressourcen in der Assembly

C#

Ab .NET 2.x (das Verhalten für eingelagerte Ressourcen hat sich zwar nicht geändert, Ressourcen, die über die Projekt-Eigenschaften hinzugefügt wurden, sind aber jetzt im Stream *...Properties.Resources.resources* gespeichert):



Beispiel 13.10: Alle Ressourcen ermitteln

C#

```
...
using System.Reflection;
using System.IO;
using System.Resources;
using System.Collections;
```

Assembly laden:

```
myass = Assembly.GetExecutingAssembly();
foreach (String s in myass.GetManifestResourceNames())
{
```

Hier bestimmen wir zunächst die einzelnen Ressource-Streams:

```
    listBox1.Items.Add(s);
    if (s.ToLower().EndsWith(".resources"))
```

Wenn in diesem *Stream* weitere Ressourcen enthalten sind:

```
    {
      Stream stream = myass.GetManifestResourceStream(s);
      ResourceReader Reader = new ResourceReader(stream);
      IDictionaryEnumerator id = Reader.GetEnumerator();
      while (id.MoveNext())
      {
```

ID.key bezeichnet die gleiche Ressource, die Sie auch mit *Properties.Resources.xyz* auslesen können:

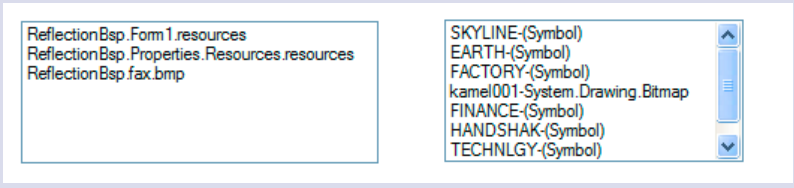
```
        listBox2.Items.Add(id.Key + "-" + id.Value);
```

Über *id.Value* können wir direkt auf die einzelnen Bitmaps zugreifen:

```
        if (id.Value is Bitmap)
        {
            Bitmap bmp = (Bitmap) (id.Value as Bitmap).Clone();
            bmp.Save(id.Key + ".bmp");
        }
      }
      Reader.Close();
    }
}
```

Ergebnis

Die Anzeige in *listBox1* und *listBox2*:



HINWEIS: Nach dem Ausführen des obigen Beispiels (siehe Buch-Beispieldaten) werden alle Bitmaps aus der Assembly extrahiert.

Beispiel 13.11: Auslesen der eingebetteten Ressource *fax.bmp* und Speichern in einer externen Datei bei Doppelklick auf den entsprechenden Eintrag in *listBox1*

C#

```
private void listBox1_MouseDoubleClick(object sender, MouseEventArgs e)
{
    Bitmap  bmp = new Bitmap(
    Assembly.GetExecutingAssembly().GetManifestResourceStream((String)
                                    listBox1.SelectedItem));
    bmp.Save((string) listBox1.SelectedItem);
}
```

HINWEIS: Achten Sie beim Zugriff auf die Ressourcen peinlichst auf die korrekte Schreibweise (Groß-/Kleinschreibung!).

13.4 Anwendungen lokalisieren

Das .NET Framework ermöglicht die komfortable Lokalisierung von Anwendungen. Texte und andere sprachabhängige Informationen befinden sich nicht mehr im eigentlichen Quellcode, sondern werden in eigenen Assemblies (sogenannte Satelliten-Assemblies) bereitgestellt, die parallel zur eigentlichen Assembly existieren, darüber hinaus jedoch keinen Code beinhalten.

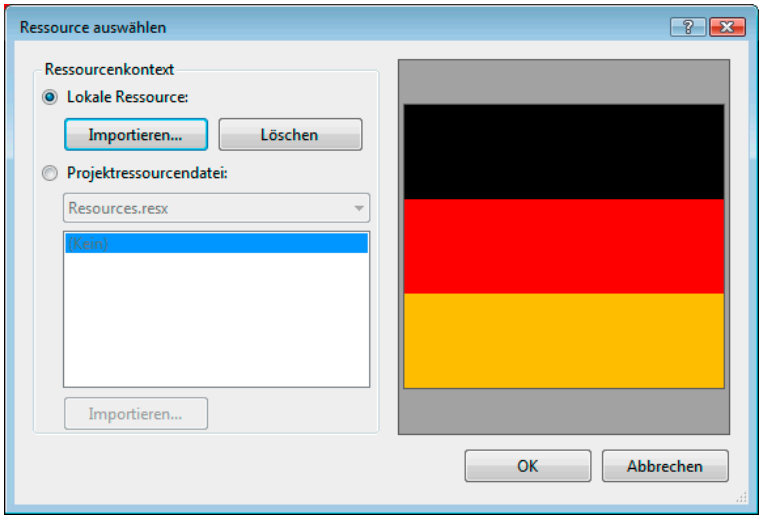
13.4.1 Localizable und Language

Visual Studio kann automatisch die Ressourcendateien für die zu lokalisierenden Elemente der Benutzerschnittstelle, wie z. B. Beschriftungen der Steuerelemente eines Formulars, erzeugen. Von Bedeutung sind dabei die Formulareigenschaften *Localizable* und *Language*.

HINWEIS: *Localizable* und *Language* sind „künstliche" Formulareigenschaften. Sie sorgen lediglich dafür, dass der Designer bei der Code-Generierung die zu lokalisierenden Informationen in der richtigen *.resx*-Datei ablegt.

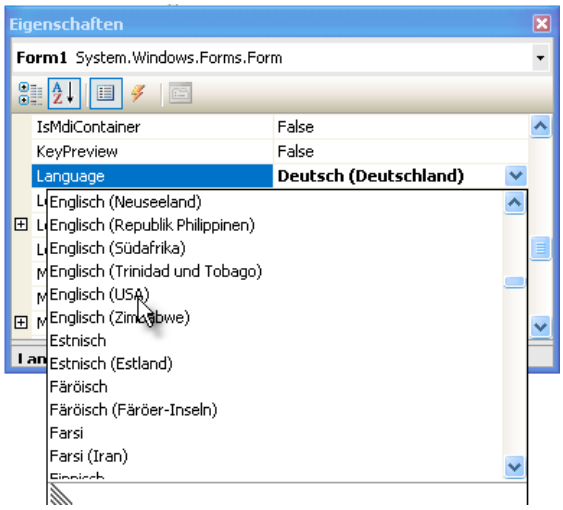
13.4.2 Beispiel „Landesfahnen"

In einer neuen Windows-Anwendung setzen wir die *Localizable*-Eigenschaft von *Form1* auf *True*, die *Language*-Eigenschaft verbleibt zunächst auf ihrem Standardwert (*Default*). Auf *Form1* befinden sich ein *Label*, eine *PictureBox* und ein *Button*. Der *Image*-Eigenschaft der *PictureBox* weisen wir per Dialog eine Bitmap mit der deutschen Flagge zu:



Die Beschriftung erfolgt in deutscher Sprache, sodass die Standardversion etwa so aussieht:

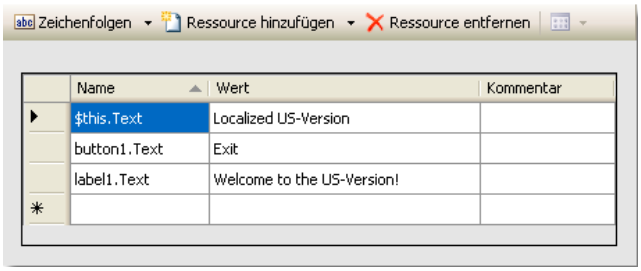
Nun stellen wir im Eigenschaftenfenster von *Form1* die *Language*-Eigenschaft auf *Deutsch* ein. Dazu selektieren wir unser Land aus einer schier endlos langen Liste, in der (fast) jedes Land der Erde vertreten ist.



Wählen Sie im Anschluss *Englisch (USA)* als neue *Language* und gestalten Sie auf analoge Weise die US-Version von *Form1*:

Ein Blick in den Projektmappen-Explorer zeigt, dass die Ressourcendateien *Form1.de.resx* und *Form1.en-US.resx*, welche die sprachabhängigen Informationen kapseln, hinzugekommen sind.

Wie z. B. ein Doppelklick auf *Form1.en-US.resx* zeigt, sind die Text-Ressourcen als Schlüssel-Wert-Paar hinterlegt:



Nach dem Kompilieren werden wir zunächst nur die deutsche Version des Programms zu Gesicht bekommen, es sei denn, wir ändern die Spracheinstellung des aktuellen Threads. Dazu fügen wir die fett gedruckten Programmzeilen hinzu:

HINWEIS: Nach wie vor haben wir es mit einem einzigen Formular (*Form1*) zu tun, es ist deshalb völlig egal, von welcher Formularansicht aus wir das Quellcodefenster öffnen.

```
using System.Globalization;   // !
using System.Threading;       // !
...
        public Form1()
        {
```

Wichtig ist, dass die Einstellung der *CurrentUICulture* **vor** dem Aufruf von *InitializeComponent()* erfolgt:

```
            // Thread.CurrentThread.CurrentUICulture = new CultureInfo("de-DE");
             Thread.CurrentThread.CurrentUICulture = new CultureInfo("en-US");
             InitializeComponent();
        }
    }
}
```

13.4.3 Einstellen der aktuellen Kultur zur Laufzeit

Damit zum Ändern der *CurrentUICulture* nicht immer das Programm neu kompiliert werden muss, empfiehlt sich für Demozwecke das Vorschalten eines weiteren Formulars *Form2*, in dem die gewünschte Sprache zur Laufzeit gewechselt werden kann:

Der Code des „Vorschaltformulars“ *Form2*:

```
private void button1_Click(object sender, EventArgs e)
{
    string ci = "de-DE";
    if (radioButton2.Checked) ci = "en-US";
    Form1 frm = new Form1(ci);
    frm.Show();
}
```

Den Konstruktor von *Form1* ändern wir wie folgt:

```
public Form1(string ci)
{
     Thread.CurrentThread.CurrentUICulture = new CultureInfo(ci);
     InitializeComponent();
}
```

Schließlich muss das Startformular der Anwendung im Hauptprogramm *Program.cs* auf *Form2* gesetzt werden:

```
static class Program
{
...
        Application.EnableVisualStyles();
        Application.Run(new Form2());
     }
}
```

Bemerkungen

- Neben der Hauptassembly wurden zwei neue Ordner (*de* und *en-US*) erzeugt, die jeweils eine Satelliten-Assembly **.resources.dll* enthalten (Satelliten-Assemblies bestehen nur aus Ressourcen).
- Die Original-Bilddateien der deutschen und der amerikanischen Flagge sind nur für den Entwurf des Projekts, nicht aber für die Weitergabe der kompilierten Anwendung erforderlich, da auch die kompletten Bildressourcen in den Satelliten-Assemblies eingelagert sind.
- Die Lokalisierung von .NET-Anwendungen beschränkt sich nicht nur auf das Anlegen von Text-Ressourcen. Nahezu jede Eigenschaft, wie z. B. die Höhe oder Breite eines jeden Formulars oder Steuerelements, lässt sich landesspezifisch anpassen.

- Zwar ist es auch möglich, Grafiken und andere Dateien direkt als Ressourcen in eine Assembly einzubetten oder mit der Assembly zu verlinken, allerdings unterstützen diese Ressourcen nicht die Lokalisierung, denn dazu müssen Grafiken etc. in eine *.resx*-Datei integriert werden.
- Die Spracheinstellung des aktuellen Threads richtet sich beim Programmstart nach den Spracheinstellungen von Windows. Zur Laufzeit können Sie die Sprache wechseln, z. B. mit:

```
Thread.CurrentThread.CurrentUICulture = new System.Globalization.CultureInfo("en-GB");
```

- Die aktuelle Sprache ermitteln Sie mit

```
System.Threading.Thread.CurrentThread.CurrentUICulture.Name;
```

oder

```
System.Threading.Thread.CurrentThread.CurrentUICulture.NativName;
```

Name liefert den englischen Sprachnamen, *NativName* den Namen der Sprache in der aktuellen Sprache.

14 Komponenten-entwicklung

Die komponentenbasierte Entwicklung gehört mit zu den Grundpfeilern der .NET-Philosophie. Visual Studio bietet vielfältige Features, die dem Programmierer die Entwicklung eigener Steuerelemente (Komponenten) erleichtern. Was Sie dabei erleben, ist OOP pur, und wir setzen für die Lektüre dieses Kapitels voraus, dass Sie einigermaßen sattelfest in Begriffen wie Klassen, Vererbung, Konstruktor usw. sind (siehe Kapitel 3 in 978-3-446-45359-3: Visual C# 2017).

14.1 Überblick

Bevor Sie sich auf die Komponentenprogrammierung stürzen, sollten Sie sich gut überlegen, welchen Komponententyp Sie wirklich benötigen, bietet Ihnen doch Visual Studio gleich ein ganzes Arsenal von Möglichkeiten an:

Typ	Bemerkung/Verwendung
Benutzersteuerelement	Sie möchten ein oder mehrere Controls in einem Container zusammenfassen und mit einer neuen Schnittstelle/Funktionalität ausstatten. Ableitung von *UserControl*.
Benutzerdefiniertes Steuerelement	Sie wollen ein neues Control erstellen oder ein vorhandenes um zusätzliche Funktionen erweitern. Ableitung vom Urtyp *Control* bzw. von vorhandenen Steuerelementen.
Geerbtes Benutzersteuerelement	Sie möchten ein Benutzersteuerelement aus Ihrem oder aus anderen Projekten um weitere Funktionen/Controls erweitern.
Komponentenklasse	Sie wollen ganz weit unten anfangen und sich sowohl um Schnittstelle als auch Funktionalität komplett selbst kümmern. Ableitung von *Component*.

Unter ähnlichen und leicht verwechselbaren Namen (dem Übersetzer sei Dank!) verbergen sich teilweise grundverschiedene Ansätze. Im Folgenden wollen wir Ihnen deshalb zunächst die Grundkonzepte und Unterschiede an kleineren Beispielen vorstellen, bevor wir die Gemeinsamkeiten bei der Definition von Eigenschaften und Methoden bzw. beim Auslösen von Ereignissen besprechen.

Im Anschluss beschäftigen wir uns noch mit einigen interessanten Fragen im Zusammenhang mit der Komponentenentwicklung.

14.2 Benutzersteuerelement

Hierbei handelt es sich quasi um einen Container für beliebige Steuerelemente, die damit zu einer Einheit verschmolzen werden können. Die Entwicklung erfolgt (wie bei einer normalen Windows-Anwendung) rein visuell, Sie platzieren die konstituierenden Steuerelemente im Container und legen deren Eigenschaften fest. In einem weiteren Schritt können Sie Ihrem Benutzersteuerelement ein eigenes Interface mit Eigenschaften, Methoden und Ereignissen geben.

Damit dürften sich auch schon die Vor- und Nachteile dieses Typs klar herausstellen:

- Die Entwicklung von Benutzersteuerelementen ist, dank visueller Unterstützung, recht einfach und schnell möglich.
- Die Programmierlogik zwischen den enthaltenen Controls kann einfach realisiert werden und ist in der Gesamtkomponente gekapselt.
- Eigenschaften und Methoden der enthaltenen Komponenten können sicher vor dem Anwender des Benutzersteuerelements ausgeblendet werden.
- Nachteilig ist der teilweise erhebliche Aufwand für das Erstellen einer sinnvollen Programmierschnittstelle (Eigenschaften/Methoden/Ereignisse).

Damit dürfte sich dieser Komponententyp hauptsächlich für Routine-Programmieraufgaben anbieten, bei denen eine öfter wiederkehrende Logik in einer Komponente gekapselt werden soll.

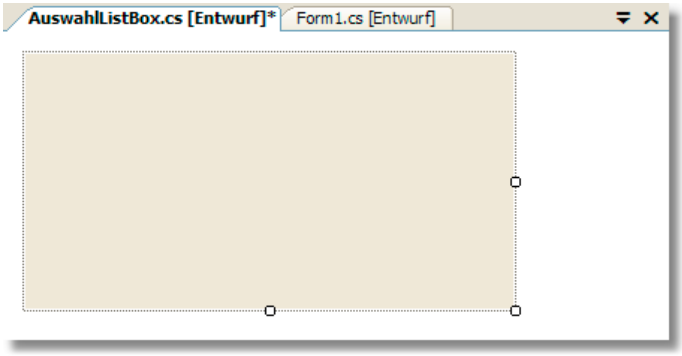
14.2.1 Entwickeln einer Auswahl-ListBox

HINWEIS: Um Ihnen den Test der neuen Komponenten möglichst einfach zu machen, integrieren wir die Komponente in ein normales Windows-Projekt.
Im Normalfall werden Sie die Komponente sicher in einer „Windows-Steuerelemente-Bibliothek“ unterbringen, da nur so eine einfache Wiederverwendbarkeit gegeben ist.

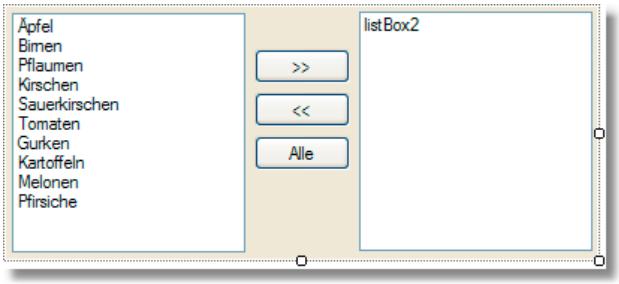
Erstellen Sie zunächst eine neue Windows Forms-Anwendung und fügen Sie über den Menüpunkt **Projekt | Benutzersteuerelement hinzufügen** ein neues Benutzersteuerelement unter dem Namen *AuswahlListBox.cs* hinzu.

Oberflächendesign

Im Designer finden Sie jetzt bereits den „Container" für die zu platzierenden Steuerelemente vor:



Die Optik und das Handling dürften Ihnen vom normalen Formularentwurf her bereits bekannt vorkommen, platzieren Sie einfach zwei *ListBox*en und drei *Button*s innerhalb des obigen Steuerelements:



Der linken *ListBox* (*listBox1*) fügen Sie im Eigenschafteneditor einige Einträge (über die *Items*-Eigenschaft) hinzu.

Implementieren der Programmlogik

Jetzt noch schnell etwas Code hinzufügen und fertig ist die neue Komponente.

Löschen der bisherigen Einträge und Kopieren **aller** Einträge in *listBox2*:

```
private void button3_Click(object sender, EventArgs e)
{
    listBox2.Items.Clear();
    listBox2.Items.AddRange(listBox1.Items);
}
```

Verschieben eines Eintrags von *listBox2* in *listBox1*:

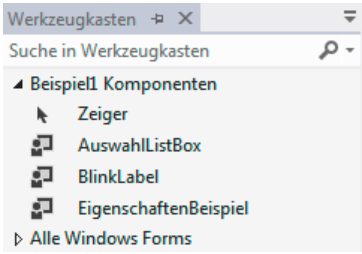
```
private void button2_Click(object sender, EventArgs e)
{
    try
    {
        listBox1.Items.Add(listBox2.Items[listBox2.SelectedIndex]);
        listBox2.Items.RemoveAt(listBox2.SelectedIndex);
    }
    catch (Exception)
    {
        MessageBox.Show("Keine Auswahl getroffen!");
    }
}
```

Verschieben eines Eintrags von *listBox1* in *listBox2*:

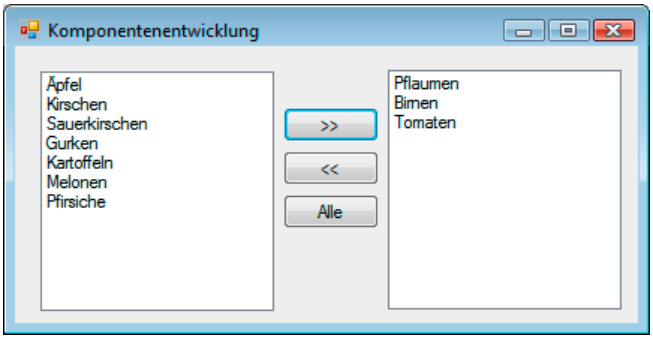
```
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        listBox2.Items.Add(listBox1.Items[listBox1.SelectedIndex]);
        listBox1.Items.RemoveAt(listBox1.SelectedIndex);
    }
    catch (Exception)
    {
        MessageBox.Show("Keine Auswahl getroffen!");
    }
}
```

14.2.2 Komponente verwenden

Zunächst kompilieren Sie die aktuelle Anwendung. Ein Blick in den Werkzeugkasten sollte im Erfolgsfall bereits die neue Komponente zeigen:



Mit dem bereits im Projekt vorhandenen Formular können wir uns jetzt an einen ersten Test wagen. Fügen Sie die neue Komponente ein und starten Sie das Programm. Bereits jetzt verfügt Ihre Komponente über jede Menge Funktionalität. Was fehlt, ist jedoch eine Interaktion mit dem eigentlichen Programm. Zu diesem Zweck können Sie weitere Eigenschaften einführen, um zum Beispiel die Einträge in *listBox2* abzufragen oder die Einträge von *listBox1* vorzudefinieren. Doch dazu später mehr.



14.3 Benutzerdefiniertes Steuerelement

Ein benutzerdefiniertes Steuerelement verwenden Sie, wenn Sie von vorhandenen Controls (z. B. *TextBox*, *Label*, *Timer* etc.) bzw. vom Urtyp *Control* eine neue Komponente **ableiten** wollen. Ihr Steuerelement erbt zunächst alle Eigenschaften, Ereignisse und Methoden des Vorgängers.

Sie können darauf aufbauend

- eigene Eigenschaften,
- Methoden und
- Ereignisse implementieren sowie
- Methoden und Ereignisse ausblenden.

HINWEIS: Die äußeren Abmessungen der Komponente sind zunächst durch den Vorfahren bestimmt. Um ein Zeichnen der Komponente mittels *Paint* brauchen Sie sich nicht zu kümmern, solange Sie nicht *Control* als Vorfahren verwenden.

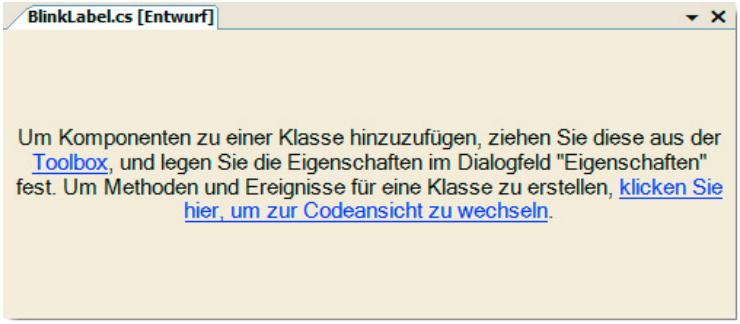
14.3.1 Entwickeln eines BlinkLabels

HINWEIS: Um den Test der neuen Komponenten möglichst einfach zu gestalten, integrieren wir die Komponente in ein normales Windows-Projekt. Im Normalfall werden Sie die Komponente in einer „Windows-Steuerelemente-Bibliothek“ unterbringen, da nur so eine einfache Wiederverwendbarkeit gegeben ist.

Wie beim Benutzersteuerelement erstellen wir zunächst eine neue Windows Forms-Anwendung und fügen über den Menüpunkt **Projekt | Neues Element hinzufügen** ein neues *Benutzerdefiniertes Steuerelement* unter dem Namen *BlinkLabel.cs* hinzu.

Oberflächendesign

Zum Projekt wurde automatisch die folgende Ansicht hinzugefügt:



Nicht sehr viel Ähnlichkeit mit einem *Label*, werden Sie sicher bemerken, aber hier handelt es sich lediglich um einen Dummy, der für alle Klassen bzw. Vorfahren gleich aussieht.

Festlegen des Typs des Vorfahrens

Ein Blick in die Liste der Eigenschaften zeigt bereits jetzt jede Menge Properties. Doch ach, bisher wird *System.Windows.Forms.Control* als Klassentyp angezeigt, was auch richtig ist, da wir noch keinen eigenen Vorfahrtyp bestimmt haben. Das wollen wir nun nachholen, indem wir in die Code-Ansicht wechseln (Doppelklick auf den Dummy).

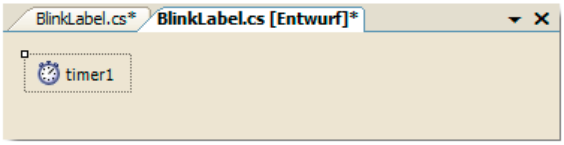
Hier suchen Sie die Klassendeklaration

```
using System;
...

namespace Beispiel1
{
    public partial class BlinkLabel : Control
    {
...
```

... und ersetzen sie durch die folgende Zeile:

```
...
    public partial class BlinkLabel : Label
...
```

Bei einem Blick in die Eigenschaftenliste werden Sie alle Label-spezifischen Eigenschaften und Ereignisse vorfinden. Wechseln Sie nun wieder zurück zum Dummy und fügen Sie einen *Timer* ein. Das Ganze sieht im Moment zwar etwas merkwürdig aus, aber es funktioniert.



HINWEIS: Sie können den *Timer* natürlich auch per Code erzeugen, was sicher eleganter ist, doch wir bohren diesmal das Brett an der dünnsten Stelle.

Implementieren der Programmlogik

Nach einem Doppelklick auf den *Timer* und dem Wechsel in die Codeansicht dürfte sich Ihnen der folgende Anblick bieten:

```
using System;
...
namespace Beispiel1
{
    public partial class BlinkLabel : Label
    {
        public BlinkLabel()
        {
            InitializeComponent();
            timer1.Interval = 500;
            timer1.Start();
        }

        protected override void OnPaint(PaintEventArgs pe)
        {
            // TODO: Benutzerdefinierten Zeichnungscode hier einfügen
            // OnPaint-Basisklasse wird aufgerufen
            base.OnPaint(pe);
        }

        private void timer1_Tick(object sender, EventArgs e)
        {
            this.Visible = !this.Visible;
        }
    }
}
```

Fügen Sie lediglich die fett hervorgehobenen Codezeilen ein, um das Steuerelement über den Konstruktor zu initialisieren und mittels *Timer* ein- und auszublenden. Das war es auch schon und wir können uns nun um das Einbinden des Steuerelements kümmern.

Kompilieren Sie jedoch zunächst die Anwendung, um das neue Steuerelement in die Toolbox aufzunehmen.

14.3.2 Verwenden der Komponente

Öffnen Sie das zum Projekt gehörende Formular *Form1* und platzieren Sie das *BlinkLabel* auf dem Formular, um sich von der Funktionstüchtigkeit zu überzeugen.

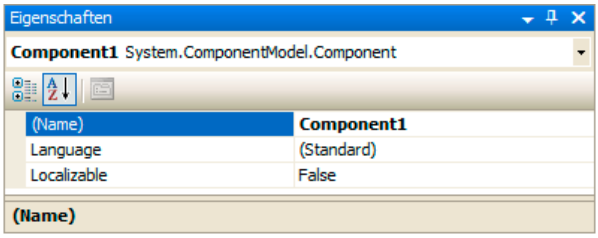
Bereits im Entwurfsmodus beginnt das Label nervend zu blinken!

HINWEIS: Natürlich handelt es sich hier nur um ein ziemlich simples Steuerelement ohne weitere Eigenschaften. Haben Sie sich aber durch dieses Kapitel durchgekämpft, sollte es Ihnen möglich sein, zum Beispiel die Blinkfrequenz über eine eigene Eigenschaft festzulegen oder weitere Ereignisse an die Komponente zu binden.

14.4 Komponentenklasse

Möchten Sie ganz unten anfangen und lediglich die rudimentärsten Funktionen übernehmen, verwenden Sie eine Komponentenklasse.

Die vorgegebene Basisklasse *System.ComponentModel.Component* können Sie übernehmen. Bei einem Blick auf die Eigenschaftenliste werden Sie aber feststellen, dass es sich um einen absoluten Basistyp handelt, der erst mit Leben befüllt werden muss:



Dieser Steuerelementetyp eignet sich zum Beispiel für den Entwurf nicht sichtbarer Komponenten (Multimedia-Timer, Datenbank-Objekte etc.).

Doch was unterscheidet eine Komponentenklasse eigentlich von einer ganz trivialen Klasse?

- Die Fähigkeit, im Designer angezeigt zu werden und Eigenschaften/Ereignisse im Eigenschaftenfenster zu präsentieren, dürfte schnell erkannt sein.
- Ein Blick in den Quellcode zeigt uns jedoch auch, dass die Komponente in der Lage ist, weitere Komponenten aufzunehmen (*IContainer*):

```
namespace Beispiel1
{
    partial class Component1
    {
        private System.ComponentModel.IContainer components = null;
...
```

  Damit können Sie auch hier per Drag&Drop zum Beispiel einen *Timer* oder ein Data Control einfügen, dessen Eigenschaften konfigurieren und Ereignisse programmieren.

Ob Sie sich nun für das Erstellen einer Klasse oder einer Komponentenklasse entscheiden, hängt nur davon ab, wie viel Komfort Sie dem Endanwender bieten wollen.

HINWEIS: Auf ein eigenes Beispiel verzichten wir an dieser Stelle, da der prinzipielle Ablauf der Vorgehensweise beim *Custom Control* entspricht.

14.5 Eigenschaften

Im Folgenden wollen wir uns mit den verschiedenen Varianten und Optionen von Eigenschaften näher befassen.

HINWEIS: Die Ausführungen lassen sich auf alle der eingangs genannten drei Steuerelementetypen anwenden, auch wenn wir uns in den folgenden Beispielen auf benutzerdefinierte Steuerelemente beschränken werden.

14.5.1 Einfache Eigenschaften

Unter dieser Rubrik wollen wir Eigenschaften verstehen, die auf einfachen Basistypen (zum Beispiel *String* oder *Integer*) basieren. Im Eigenschaftenfenster haben Sie die Möglichkeit, den Wert zu editieren (nur wenn die Eigenschaft dies auch zulässt).

Folgende Steuerungsmöglichkeiten und Optionen für die Eingabe bestehen:

- nur Lesezugriff,
- Schreib-/Lesezugriff,
- Schreibzugriff,
- Ausblenden im Eigenschaftenfenster,
- Wertebereichsbeschränkung und Fehlerprüfung,
- Hinzufügen von Beschreibungen,
- Default-Werte,
- Einfügen in Kategorien.

14.5.2 Schreib-/Lesezugriff (Get/Set)

Hierbei dürfte es sich um die wohl am häufigsten verwendete Variante bei Eigenschaften handeln. Der Nutzer des Steuerelements hat die Möglichkeit, sowohl Eigenschaftswerte zu lesen als auch zu ändern. Dazu müssen Sie als Entwickler sowohl die *get*- als auch die *set*-Option implementieren.

Beispiel 14.1: Schreib-/Lesezugriff (mit Zugriffsmethoden)

C#

Die Variable für die interne Zustandsverwaltung:

```
private int _MeineIntegerEigenschaft;
```

Die beiden Zugriffsmethoden:

```
public int MeineIntegerEigenschaft
{
    get { return _MeineIntegerEigenschaft; }
    set { _MeineIntegerEigenschaft = value; }
}
```

14.5.3 Nur-Lese-Eigenschaft (ReadOnly)

Möchten Sie dem Anwender lediglich einen Lesezugriff auf den Eigenschaftswert gestatten, lassen Sie bei der Deklaration der Eigenschaft einfach die *set*-Option weg.

Beispiel 14.2: Nur-Lese-Eigenschaft

C#

Die Variable für die interne Zustandsverwaltung:

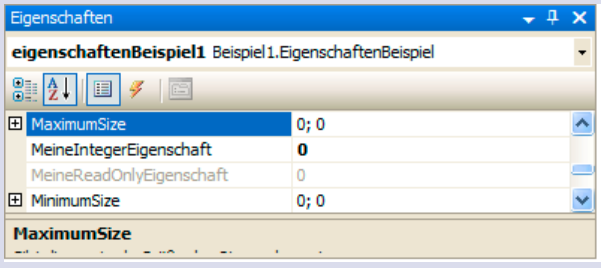
```
private int _MeineReadOnlyEigenschaft;
```

Die Zugriffsmethode:

```
public int MeineReadOnlyEigenschaft
{
    get { return _MeineReadOnlyEigenschaft; }
}
```

Ergebnis

Bei Verwendung der Komponente wird die Eigenschaft im Eigenschaftenfenster in grauer Schrift angezeigt, da sie schreibgeschützt ist.



14.5.4 Nur-Schreib-Zugriff (WriteOnly)

Bei dieser Variante wird die *get*-Option weggelassen.

```
private int _MeineWriteOnlyEigenschaft;
public int MeineWriteOnlyEigenschaft
{
    set { _MeineWriteOnlyEigenschaft = value; }
}
```

Allerdings sollten Sie für derartige Anwendungsfälle besser eine Methode verwenden, da dies den Sinn der Operation besser verdeutlicht.

HINWEIS: Die Eigenschaft wird sinnigerweise nicht im Eigenschaftenfenster angezeigt, welcher Wert sollte auch dargestellt werden?

14.5.5 Hinzufügen von Beschreibungen

Mit dem Attribut *Description* steuern Sie den Inhalt des Beschreibungsfelds im Eigenschaftenfenster.

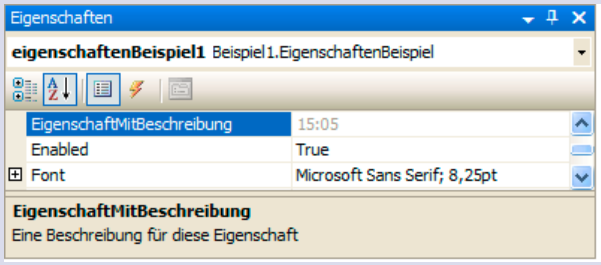
Beispiel 14.3: Eine Beschreibung festlegen

C#

```
[Description("Eine Beschreibung für diese Eigenschaft")]
public String EigenschaftMitBeschreibung
{
    get { return System.DateTime.Now.ToShortTimeString(); }
}
```

Ergebnis

Die Eigenschaft im Eigenschaftenfenster:



14.5.6 Ausblenden im Eigenschaftenfenster

Nicht in jedem Fall möchte man, dass eine Eigenschaft schon zur Entwurfszeit im Eigenschaftenfenster angezeigt wird. Über das Attribut *Browsable* haben Sie die Möglichkeit, die Sichtbarkeit der Eigenschaft zu steuern.

Beispiel 14.4: Einfügen eines Attributs

C#

```
[Browsable(false)]
public int NichtSichtbareEigenschaft
{
  get { return myValue; }
  set { myValue = value; }
}
```

14.5.7 Einfügen in Kategorien

Auch hier dient ein Attribut (*Category*) dazu, den Eigenschaften weitere Informationen für den Eigenschafteneditor mit auf den Weg zu geben.

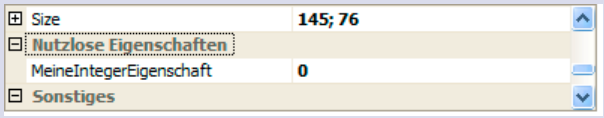
Beispiel 14.5: Einfügen von *MeineIntegerEigenschaft* in die Kategorie „Nutzlose Eigenschaften“

C#

```
[Category("Nutzlose Eigenschaften")]
public int MeineIntegerEigenschaft
{
    get { return _MeineIntegerEigenschaft; }
    set { _MeineIntegerEigenschaft = value; }
}
```

Ergebnis

Das Resultat im Eigenschaftenfenster:



HINWEIS: Dieses Attribut wirkt sich natürlich nur aus, wenn die Eigenschaften auch in Kategorien angezeigt werden.

14.5.8 Default-Wert einstellen

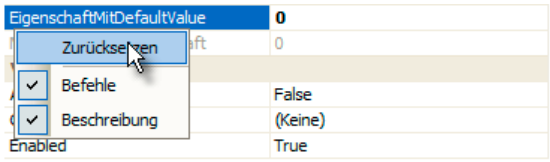
Mithilfe des Attributs *DefaultValue* können Sie dem Anwender die Möglichkeit geben, über Reset bzw. Zurücksetzen der Eigenschaft einen vorgegebenen Wert zuzuweisen.

Beispiel 14.6: Standardwert zuweisen

C#

```
        private int _myint;
        [DefaultValue(55)]
        public int EigenschaftMitDefaultValue
        {
            get { return _myint; }
            set { _myint = value; }
        }
```

Im Eigenschaftenfenster erreichen Sie die gewünschte Funktion über das Kontextmenü:



14.5.9 Standardeigenschaft (Indexer)

Da Programmierer so wenig wie möglich schreiben wollen, wurden für den Zugriff auf Array-Eigenschaften die Indexer erfunden.

Der Vorteil: Sie können auf die Angabe eines Eigenschaftennamens verzichten und direkt mit dem Objekt arbeiten.

Beispiel 14.7: Standardeigenschaft definieren

C#

```
    public partial class EigenschaftenBeispiel : Control
    {
...
        private string []mydata = {"rot","gelb","grün"};

        public string this[int index]
        {
            get { return mydata[index]; }
        }
```

Verwenden können Sie die Klasse später wie folgt:

```
        private void button1_Click(object sender, EventArgs e)
        {
            Text = eigenschaftenBeispiel1[0];
        }
```

14.5.10 Wertebereichsbeschränkung und Fehlerprüfung

Hierbei handelt es sich um eine der wohl komplexesten Aufgaben des Programmierers. Es geht darum, Fehleingaben des Anwenders zu verhindern bzw. die Eingabe auf gewünschte Werte zu beschränken. Dazu stehen dem Entwickler innerhalb der *set*-Zugriffsmethode alle Möglichkeiten offen.

Beispiel 14.8: Wertebereichsbeschränkung und Fehlerprüfung

C#

Eigenschaft, die nur Integerwerte zwischen 50 und 200 akzeptiert und gegebenenfalls eine entsprechende Anpassung vornimmt

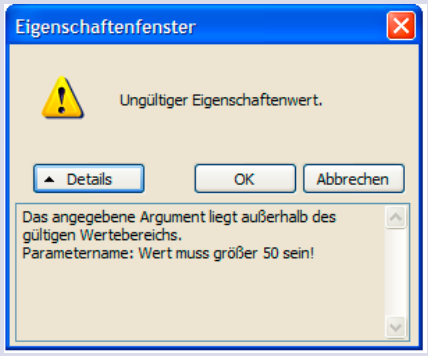
```
private int _MeineVar;
public int EigenschaftMitBereichsbeschränkung
{
    get { return _MeineVar; }
    set
    {
        if (value < 50)
            _MeineVar = 50;
        else
            if (value > 200)
                _MeineVar = 200;
            else
                _MeineVar = value;
    }
}
```

Alternativ können Sie auch den Anwender mit Fehlermeldungen zupflastern:

```
public int EigenschaftMitFehlerprüfung
{
    get { return _MeineVar; }
    set
    {
        if (value < 50)
            throw new ArgumentOutOfRangeException("Wert muss größer 50
sein!");
        else
            if (value > 200)
                throw new ArgumentOutOfRangeException("Wert muss kleiner
200 sein!");
            else
                _MeineVar = value;
    }
}
```

Ergebnis

Fehlermeldung bei Angabe eines falschen Werts:

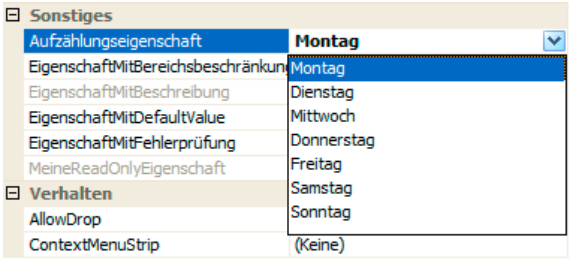


Eine weitere Möglichkeit zur Einschränkung bieten die sogenannten Aufzählungstypen (Enumerationen), die im folgenden Abschnitt besprochen werden.

14.5.11 Eigenschaften von Aufzählungstypen

Ist der Wertebereich einer Eigenschaft zur Entwicklungszeit bereits bekannt, können Sie die Fehlermöglichkeiten durch Verwendung eines Aufzählungstyps einschränken.

Im Eigenschaftenfenster wird in diesem Fall eine Liste der zulässigen Werte angezeigt:



Der Vorteil für den Endanwender liegt auf der Hand: Statt irgendwelcher numerischer Werte erscheinen aussagefähige Beschriftungen. Auf die Verwendung der Hilfefunktion kann deshalb fast immer verzichtet werden.

Für Sie als Entwickler bedeutet ein Aufzählungstyp ein wenig mehr Arbeit, da Sie zuerst einen entsprechenden Typ deklarieren müssen. Die eigentliche Umsetzung in der Komponentendefinition unterscheidet sich nicht von der einer einfachen Eigenschaft.

Beispiel 14.9: Aufzählungseigenschaft deklarieren

C#

Typ deklarieren:

```
public enum MyEnum
 {
     Montag = 0,
     Dienstag = 1,
     Mittwoch = 2,
     Donnerstag = 3,
     Freitag = 4,
     Samstag = 5,
     Sonntag = 6,
 }
```

Private Variable:

```
private MyEnum myEnum;
```

Die Eigenschaft:

```
public MyEnum Aufzählungseigenschaft
{
    get { return myEnum; }
    set { myEnum = value; }
}
```

Beispiel 14.10: Verwendung der Eigenschaft *Aufzählungseigenschaft*

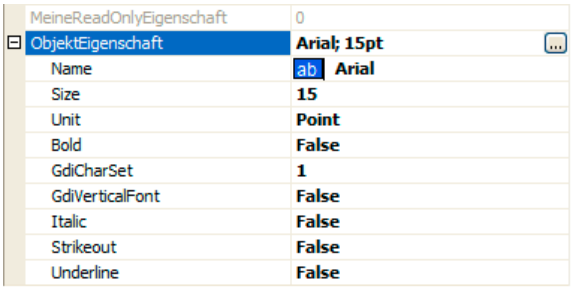
C#

```
...
eigenschaftenBeispiel1.Aufzählungseigenschaft =
                          Beispiel1.EigenschaftenBeispiel.MyEnum.Donnerstag;
```

14.5.12 Standard-Objekteigenschaften

Während Sie beim vorhergehenden Eigenschaftstyp lediglich einzelne Optionen festlegen können, bietet eine Objekeigenschaft wesentlich mehr. Das wohl prominenteste Beispiel dürfte die Eigenschaft *Font* sein, die wiederum über eigene Eigenschaften verfügt.

Im Eigenschaftenfenster können Sie entweder einen Eigenschafteneditor verwenden oder Sie expandieren den Eintrag und legen die Objekteigenschaften einzeln fest:



Beispiel 14.11: Implementieren eines *Font*-Objekts in unserer Beispielkomponente

C#

```
        private Font myFont = new Font("Arial",15);
        public Font ObjektEigenschaft
        {
            get { return myFont;  }
            set { myFont = value; }
        }
```

14.5.13 Eigene Objekteigenschaften

Komplizierter wird die ganze Geschichte, wenn Sie ein neues Objekt erstellen wollen. In diesem Fall genügt es nicht, wenn Sie einfach ein Objekt in Ihre Komponente integrieren, wie es das folgende Beispiel zeigt.

Beispiel 14.12: Unsere Komponente hat ein Objekt mit drei Eigenschaften vom Typ *Integer*.

C#

```
    public class TestKlasse : ExpandableObjectConverter
    {
        private int _Wert1;

        public int Wert1
        {
            get { return _Wert1; }
            set { _Wert1 = value; }
        }
        private int _Wert2;

        public int Wert2
        {
            get { return _Wert2; }
            set { _Wert2 = value; }
        }
        private int _Wert3;

        public int Wert3
        {
```

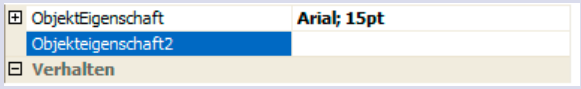
```
            get { return _Wert3; }
            set { _Wert3 = value; }
        }
    }
```

Die Eigenschaft:

```
...
        private TestKlasse _myTestklasse;

        public TestKlasse Objekteigenschaft2
        {
            get { return _myTestklasse; }
            set { _myTestklasse = value; }
        }
```

Ergebnis

Ein Test der Komponente in der IDE zeigt folgendes Ergebnis:



Das Resultat dürfte sicher nicht ganz Ihren Erwartungen entsprechen, haben wir doch einen Fehler in unserem Beispiel, wie er gern gemacht wird:

Beachten Sie bitte, dass die private Objektvariable nicht initialisiert ist.

Deshalb:

```
        private TestKlasse _myTestklasse = new TestKlasse();
```

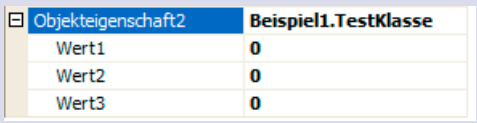
Ein erneuter Blick in das Eigenschaftenfenster zeigt keine Veränderung, aber die Komponente ist zumindest zur Laufzeit schon voll funktionstüchtig:

```
            eigenschaftenBeispiel1.Objekteigenschaft2.Wert2 = 5;
            this.Text = eigenschaftenBeispiel1.Objekteigenschaft2.Wert2.ToString();
```

Nach viel Sucherei in der Microsoft-Dokumentation und in diversen Foren stellt sich heraus, dass wir nur mit zusätzlichen Attributen, die die Anzeige im Eigenschaftenfenster steuern, weiterkommen:

```
    [TypeConverterAttribute(typeof(System.ComponentModel
.ExpandableObjectConverter))]
    public class TestKlasse
    {
```

Das Ergebnis im Eigenschaftenfenster:



HINWEIS: Wem die Anzeige von „Beispiel1.TestKlasse“ nicht gefällt, der kann sich einen eigenen *TypeConverter* von *ExpandableObjectConverter* ableiten und die Methoden *CanConvertFrom*, *ConvertFrom*, *ConvertTo*, *GetCreateInstanceSupported* sowie *Create Instance* überschreiben.

Damit dürften wir die wichtigsten Varianten von Eigenschaften berücksichtigt haben. Auf alle Möglichkeiten (Eigenschafteneditor etc.) können wir aus Platzgründen leider nicht eingehen, die gezeigten Beispiele dürften jedoch manche Unklarheit beseitigt haben.

14.6 Methoden

Hinter den Methoden von Steuerelementen verbirgt sich nichts anderes als normale Funktionen, die jedoch fest an die jeweilige Klasse gekoppelt sind. Aus der Realisierung ergibt sich auch das Einsatzgebiet: Methoden sollten, im Gegensatz zu Eigenschaften, Aktionen auslösen, die teilweise mit Rückgabewerten (Funktionen) verbunden sind. Methoden bieten sich auch dann an, wenn es darum geht, mehrere Eigenschaften gleichzeitig zu beeinflussen.

HINWEIS: Da Definition und Verwendung von Methoden zum Handwerkszeug des C#-Programmierers gehört, möchten wir im Weiteren nur noch auf einige spezielle Themen eingehen.

14.6.1 Konstruktor

Das Besondere: Diese Methode wird automatisch beim Erzeugen einer Klasseninstanz ausgeführt. Damit ist dies auch der ideale Ansatzpunkt, um

- alle internen Variablen unseres Steuerelements zu initialisieren,
- das Aussehen des Steuerelements anzupassen
- und gegebenenfalls Ereignishandler zuzuweisen.

HINWEIS: Der Konstruktor trägt immer den Namen der Klasse.

Eine Besonderheit gilt es noch zu beachten: Erzeugen Sie eine neue Komponentenklasse, legt C# automatisch zwei überladene Konstruktoren an, wie der folgende Quellcodeausschnitt zeigt:

```
namespace Beispiel2
{
```

```
    public partial class Component1 : Component
    {
        public Component1()
        {
            InitializeComponent();
        }

        public Component1(IContainer container)
        {
            container.Add(this);
            InitializeComponent();
        }
    }
}
```

Welcher Konstruktor wird nun eigentlich ausgeführt?

Die Antwort: Wenn die Komponente einem Container (*Form/Panel* etc.) zugeordnet wird, nutzt die IDE nicht den einfachen, sondern den Konstruktor mit Parameter, um die Komponente in die *IContainer*-Auflistung einzufügen.

```
    partial class Form1
    {
        private System.ComponentModel.IContainer components = null;
...
        private void InitializeComponent()
        {
            this.component11 = new Beispiel2.Component1(this.components);
```

Andernfalls wird der Standardkonstruktor verwendet:

```
    partial class Form1
    {
        private void InitializeComponent()
        {
...
            this.component11 = new Beispiel2.Component1();
```

Der Vorteil: Wird ein Formular erzeugt und klinkt sich die Komponente in die *IContainer*-Auflistung *components* ein, wird auch beim Aufruf von *Form.Dispose* die *Dispose*-Methode des Controls aufgerufen. Dazu überschreibt das Formular seine geerbte *Dispose*-Methode:

```
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
                components.Dispose();
            base.Dispose(disposing);
        }
```

Ressourcen werden so definiert freigegeben, andernfalls müsste nach dem Löschen des Formulars Ihre Komponente irgendwann vom Garbage Collector entsorgt werden.

Beispiel 14.13: Formular öffnen und wieder freigeben

C#

```
        Form2 f2 = new Form2();
        f2.ShowDialog();
        f2.Dispose();  // --> Hier wird auch Component.Dispose() aufgerufen
```

14.6.2 Class-Konstruktor

Noch ein Konstruktor? Ja! Wer schon einmal mit dem NET-Reflector in einer NET-Assembly herumgestöbert hat, wird sicher auch die Methode *.cctor* gefunden haben. Hierbei handelt es sich um den **Class-Konstruktor**, der beim ersten Zugriff auf die Klasse ausgeführt wird. Der Einwand, das tut der normale Konstruktor auch, kann so nicht stehen bleiben. Was passiert beispielsweise, wenn Sie eine statische Methode aufrufen? In diesem Fall wird vorher automatisch der Class-Konstruktor abgearbeitet (nach dem Laden der Klasse, vor dem Zugriff auf die Member). Der eigentliche Konstruktor ist zu diesem Zeitpunkt noch gar nicht in Aktion getreten.

Womit auch gleich das Anwendungsgebiet ersichtlich wird. Nutzen Sie diesen Konstruktor, um statische Eigenschaften zu initialisieren.

Beispiel 14.14: Initialisieren der Eigenschaft *Startzeit*

C#

```
    public partial class Component1 : Component
    {
        static public DateTime Startzeit;
        static Component1()   // Class-Konstruktor
        {
            Startzeit = System.DateTime.Now;
        }
...
```

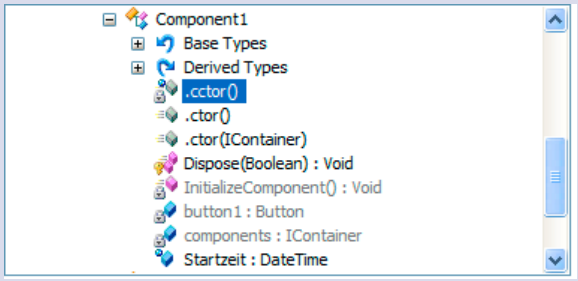
HINWEIS: Beachten Sie, dass die Eigenschaft *Startzeit* für alle späteren Instanzen der Klasse den gleichen Wert hat (es handelt sich eben um eine Klasseneigenschaft).

Beispiel 14.15: Die Verwendung im aufrufenden Programm

C#

```
        private void Form1_Load(object sender, EventArgs e)
        {
            Text = Component1.Startzeit.ToString();
        }
```

Ergebnis



14.6.3 Destruktor

Da sich in .NET bekanntlich der Garbage Collector um die endgültige Zerstörung von nicht mehr benötigten Objekten kümmert, wird ein Destruktor im herkömmlichen Sinn nicht mehr gebraucht. Möchten Sie dennoch auf das relativ unbestimmte Ende Ihres Steuerelements reagieren, können Sie eine private Methode mit dem Namen der Klasse erstellen. Zur Unterscheidung vom Konstruktor wird ein „~“-Zeichen vorangestellt.

Beispiel 14.16: Destruktor

C#

```
~MyEdit()
 {
   Debug.WriteLine(" Ich bin am Ende");
 }
```

HINWEIS: Den Destruktor selbst können Sie nicht per Code aufrufen!

14.6.4 Aufruf des Basisklassen-Konstruktors

HINWEIS: Wollen/müssen Sie einen bestimmten Konstruktor der Basisklasse aufrufen, verwenden Sie *:base(Parameterliste)*.

Beispiel 14.17: Verwendung von *base*

C#

```
public MyEdit(int i) : base(i)
{
   ...
}
```

Statt des Basisklassen-Standardkonstruktors wird im obigen Beispiel der Konstruktor mit dem Integer-Parameter verwendet.

14.6.5 Aufruf von Basisklassen-Methoden

Müssen Sie eine Methode der Basisklasse aufrufen, verwenden Sie den Bezeichner *base*.

Beispiel 14.18: Überschreiben von *OnPaint* und Aufruf der Basisklassen-Methode

C#

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    e.Graphics.DrawLine(Pens.Red, 10, 10, 100, 100);
    ...
}
```

14.7 Ereignisse (Events)

Nicht ganz so einfach wie das Programmieren von Methoden ist die Realisierung von Ereignissen bzw. Ereignisprozeduren. Events stellen einen Mechanismus dar, der ein externes Ereignis (zum Beispiel eine Windows-Botschaft) oder ein Nutzerereignis (zum Beispiel der Klick auf einen Button) mit einer eigenen Routine (Eventhandler) verbindet.

Fünf Schritte sind für das Implementieren eines Events erforderlich:

- eventuell eine neue Klasse für den *EventArgs*-Parameter (diese wird von *System.Event-Args* abgeleitet) definieren,
- eine Delegate-Deklaration (Definition des Ereignistyps),
- ein interner Delegate für den Event,
- eine Event-Deklaration,
- das eigentliche Auslösen des Ereignisses.

14.7.1 Ereignis mit Standardargument definieren

Relativ einfach ist das Implementieren eines Ereignisses, das sich mit den Standardargumenten vom Typ *System.EventArgs* zufriedengibt. Allerdings sollten Sie sich schon hier mit einigen Grundkonventionen und Eigenheiten vertraut machen.

Beispiel 14.19: Ereignis mit Standardargument definieren

C#

Anhand einer *TextBox*, die ein zusätzliches Ereignis erhält, sollen die wichtigsten Schritte erläutert werden.

```
public partial class MyEdit : TextBox
{
    public MyEdit()
    {  InitializeComponent(); }
```

Zunächst instanziieren wir das Ereignis *TextboxFull* auf Basis des *EventHandler*-Delegaten:

```
    public event EventHandler TextboxFull;
```

Die Deklaration einer Ereignismethode[1], in der das Ereignis ausgelöst wird:

```
    protected void OnTextBoxFull()
    {
```

Nur wenn dem Ereignis im Programm auch eine Ereignisprozedur zugewiesen wurde, wird auch das Ereignis ausgelöst:

```
        if (TextboxFull != null)
        {
            EventArgs args = new EventArgs();
            TextboxFull(this, args);    // Ereignis wird ausgelöst
        }
    }
```

Microsoft empfiehlt übrigens, beim Ableiten von Komponenten statt der Verwendung eines Eventhandlers besser die Ereignismethode zu überschreiben.

Hier überwachen wir die *TextBox* und rufen gegebenenfalls die Methode *OnTextBoxFull* auf:

```
    protected override void OnTextChanged(EventArgs e)
    {
        if (this.Text.Length > 8) OnTextBoxFull();
        base.OnTextChanged(e);
    }
```

Achtung: Vergessen Sie nicht *base.OnTextChanged* aufzurufen, andernfalls fällt das Ereignis *TextChanged* unter den Tisch!

[1] Entsprechend der Microsoft-Konvention sollten die Ereignismethoden immer mit *On...* beginnen.

Im Anwenderprogramm findet sich jetzt ein neues Ereignis:

```
private void myEdit1_TextboxFull(object sender, EventArgs e)
{
    MessageBox.Show("Hilfe nicht so viel Buchstaben ...");
}
```

14.7.2 Ereignis mit eigenen Argumenten

Etwas aufwendiger ist die Definition eigener Argumenttypen.

Beispiel 14.20: Der Parameter soll zusätzlich eine Message an das aufrufende Programm zurückgeben.

C#

Leiten Sie dazu eine Klasse vom Typ *System.EventArgs* ab:

```
class MyEventArgs : EventArgs
{
    private String _Message;
```

Die neue Eigenschaft für das Argument:

```
    public String Message
    {
        get { return _Message; }
        set { _Message = value; }
    }
```

Implementieren Sie gegebenenfalls einen neuen Konstruktor für das neue Argument:

```
    public MyEventArgs(String msg)
    {  _Message = msg;  }
}
```

Die eigentliche Komponente:

```
public partial class MyEditMsg : TextBox
{
```

Fügen Sie der Komponente einen Delegaten für den obigen Ereignistyp hinzu:

```
    public delegate void TextBoxFullHandler(object sender, MyEventArgs e);
```

Die Deklaration des Ereignisses:

```
    public event TextBoxFullHandler TextBoxFull;
```

Alternativ können Sie auch die folgende Deklaration nutzen:

```
    public event EventHandler<MyEventArgs> TextBoxFull;
```

Die weitere Verwendung entspricht der bisherigen Vorgehensweise, beim Aufruf des Delegaten wird jetzt jedoch ein anderer Konstruktor genutzt:

```
        protected void OnTextBoxFull()
        {
            if (TextBoxFull != null)
            {
                MyEventArgs args = new MyEventArgs("Ich bin voll!!!!!!");
                TextBoxFull(this, args);
            }
        }
        protected override void OnTextChanged(EventArgs e)
        {
            if (this.Text.Length > 8) OnTextBoxFull();
            base.OnTextChanged(e);
        }
    }
```

Das war es auch schon, im Programm können Sie den Parameter wie folgt verwenden:

```
        private void myEditMsg1_TextBoxFull(object sender, MyEventArgs e)
        {
            MessageBox.Show(e.Message);
        }
```

14.7.3 Ein Default-Ereignis festlegen

Ausnahmsweise ist diese Aufgabe mit einer einzigen Zeile „Code“ erledigt. Fügen Sie einfach vor die betreffende Klassendefinition das Attribut *[DefaultEvent]* ein.

Beispiel 14.21: Default-Ereignis definieren

C#

```
    [DefaultEvent("TextBoxFull")]
    public partial class MyEditMsg : TextBox
    {
        public delegate void TextBoxFullHandler(object sender, MyEventArgs e);
        public event TextBoxFullHandler TextBoxFull;
```

Nach einem Doppelklick auf die spätere Komponente wird jetzt automatisch eine Ereignismethode für *TextBoxFull* erzeugt.

14.7.4 Mit Ereignissen auf Windows-Messages reagieren

Ganz zum Schluss wollen wir Sie mit einem nicht minder interessanten Thema foltern. Es geht um die guten alten Windows-Ereignisse bzw. -Botschaften. Auch wenn das .NET-Framework für fast alle Eventualitäten ein(e) Methode/Ereignis im Stammbaum seiner Forms/Controls vorhält, es gibt Fälle, wo wir direkt auf derartige Ereignisse reagieren wollen.

Anlaufpunkt für den .NET-Programmierer ist die Methode *WndProc*. Da stutzt sicher jeder altgediente Win32-Programmierer. Diese Variante, in die Windows-Botschaftsbehandlung einzugreifen, dürfte vielen bekannt vorkommen. Schon damals wurde die Methode einfach überschrieben, um neue Funktionalität hinzuzufügen. Ein kleines Beispiel soll Ihnen den Übergang in die .NET-Welt demonstrieren.

Beispiel 14.22: Auf Windows-Messages reagieren

C#

Ein *TextBox*-Nachfahre soll auf das Mausrad reagieren[2]. Ein eingetragener Wert ist jeweils zu inkrementieren bzw. zu dekrementieren. Zwei neue Ereignisse sollen uns über den Mausradstatus informieren.

Unsere Komponente:

```
public partial class WheelEdit : TextBox
{
```

Die beiden Ereignisse deklarieren:

```
    public event EventHandler WheelUp;
    public event EventHandler WheelDown;
```

Und ab geht es in die gute alte Win32-Welt mit ihren Konstanten und verschachtelten Parametern:

```
    private const int WM_MOUSEWHEEL = 522;
    public WheelEdit()
    {
        InitializeComponent();
    }
```

Überschreiben der Botschaftsbehandlung:

```
    protected override void WndProc(ref Message m)
    {
```

Die eigentliche Message herausfiltern:

```
        switch (m.Msg)
        {
            case WM_MOUSEWHEEL:
```

Den Inhalt der *TextBox* ermitteln:

```
                int v = Convert.ToInt32(this.Text);
```

Über den Parameter *m.WParam* können wir die Drehrichtung bestimmen:

```
                if ((((Int32)m.WParam) >> 31) == 0)
                {
                    this.Text = (v + 1).ToString();
```

[2] Natürlich geht es auch anders, aber so haben wir ein halbwegs sinnvolles Beispiel für die Messageverarbeitung.

Auslösen der Ereignisse:

```
                        if (WheelUp != null) WheelUp(this, new EventArgs());
                    }
                    else
                    {
                        this.Text = (v - 1).ToString();
                        if (WheelDown != null) WheelDown(this, new EventArgs());
                    }
                    this.Invalidate();
                    break;
                default:
                    break;
            }
```

Das sollten Sie in keinem Fall vergessen:

```
            base.WndProc(ref m);
        }
...
```

HINWEIS: Mit obigem Codegerüst können Sie auch andere Botschaften, die von der Komponente empfangen werden, auswerten.

Damit genug zu den diversen Komponententypen, auch wenn es noch reichlich zu diesem Thema zu sagen gäbe (von datengebundenen Komponenten ganz zu schweigen). Wenden wir uns nun noch einigen spezielleren Themen zu ...

14.8 Weitere Themen

Dieser Abschnitt fasst in loser Folge einige interessante Themen rund um die Komponentenprogrammierung zusammen.

14.8.1 Wohin mit der Komponente?

In den bisherigen Beispielen haben wir es uns einfach gemacht und unsere Komponenten jeweils in das Windows-Projekt mit aufgenommen. Das ist zwar sehr praktisch, wenn Sie Komponenten testen wollen, aber im Normalfall soll die Komponente ja in mehreren Anwendungen eingesetzt werden.

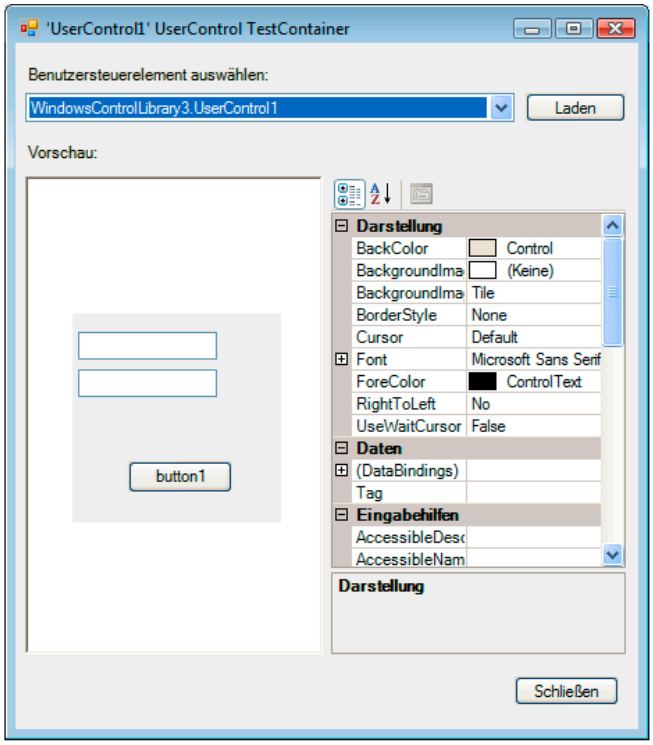
Zwei Varianten bieten sich beim Blick in den Dialog „Neues Projekt“ an:

- Klassenbibliothek,
- Windows Forms-Steuerelementebibliothek.

Worin liegt der Unterschied?

Zunächst die allgemeine Antwort: Sie können beide Projektarten für Ihre Steuerelemente verwenden. In jedem Fall wird eine Assembly mit *.dll*-Extension erzeugt.

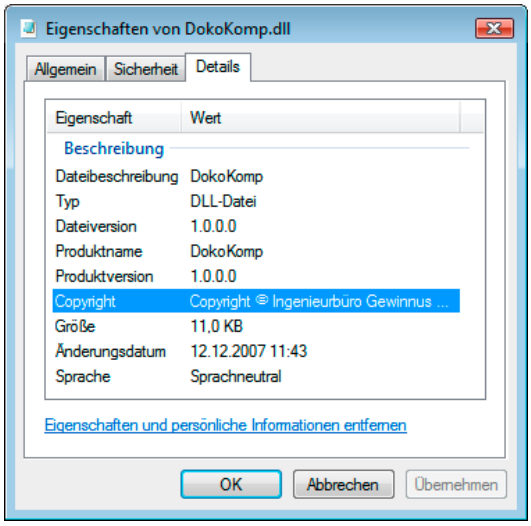
Die Unterschiede zeigen sich bei den Feinheiten. So bietet die „Windows Forms-Steuerelementebibliothek“ bereits eingebettete Ressourcen sowie einen einfachen Test-Container für Ihre UserControls. Diesen starten Sie wie gewohnt mit *F5*, ein weiterer Unterschied zu einer einfachen Klassenbibliothek, die Sie zwar erstellen, aber nicht „ausführen“ können.



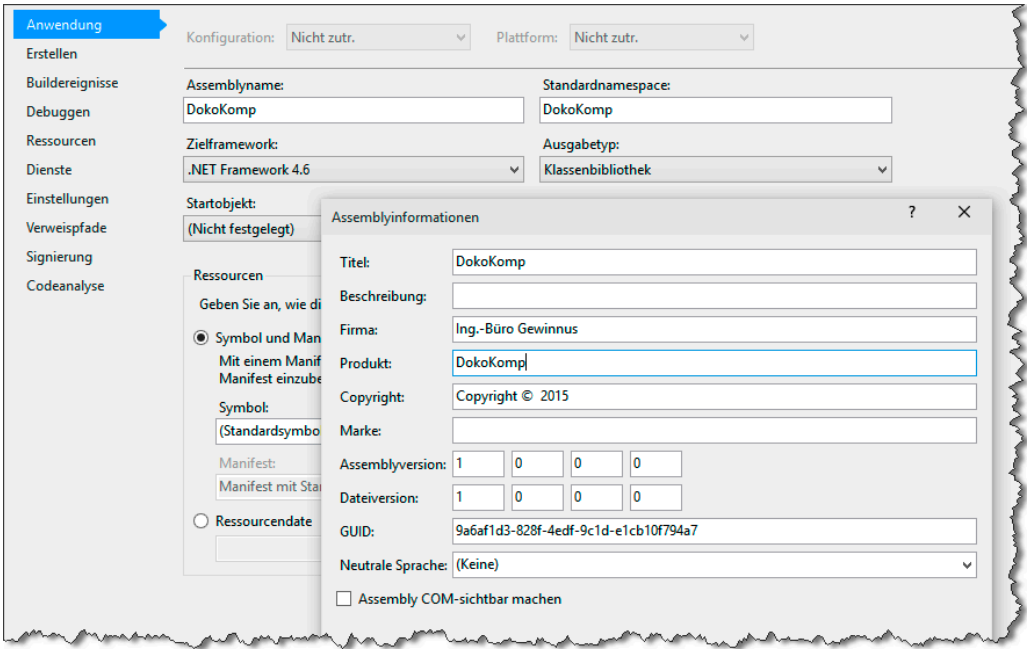
Wie Sie obiger Abbildung entnehmen, beschränkt sich der Test-Container auf die Anzeige der verfügbaren Eigenschaften. Zum Testen Ihrer Komponente ist das sicher ein nettes Feature.

14.8.2 Assembly-Informationen festlegen

In einer Assembly werden neben dem reinen Programmcode auch weitere Informationen abgelegt, die unter anderem für das Erzeugen von *Strong Names* (Starke Namen) Verwendung finden. Doch auch der normale Anwender Ihrer Assembly kann von diesen Informationen profitieren, indem er sich über das Kontextmenü (im Windows Explorer) die Eigenschaften anzeigen lässt.



Die entsprechenden Informationen können Sie in Visual Studio über einen Projekteigenschaften-Dialog festlegen:



Alternativ können Sie Assembly-Infos auch direkt in die Datei *AssemblyInfo.cs* eintragen.

Beispiel 14.23: *AssemblyInfo.cs*

C#

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// Allgemeine Informationen über eine Assembly werden über die folgenden
// Attribute gesteuert. Ändern Sie diese Attributwerte, um die Informationen zu
ändern,
// die mit einer Assembly verknüpft sind.

[assembly: AssemblyTitle("ClassLibrary1")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Ingenieurbüro Gewinnus")]
[assembly: AssemblyProduct("ClassLibrary1")]
[assembly: AssemblyCopyright("Copyright © Ingenieurbüro Gewinnus 2006")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Durch Festlegen von ComVisible auf "false" werden die Typen in dieser Assembly
unsichtbar
// für COM-Komponenten. Wenn Sie auf einen Typ in dieser Assembly von
// COM zugreifen müssen, legen Sie das ComVisible-Attribut für diesen Typ auf
"true" fest.
[assembly: ComVisible(false)]

// Die folgende GUID bestimmt die ID der Typbibliothek, wenn dieses Projekt für COM
verfügbar gemacht wird
[assembly: Guid("809314Of-abea-4e27-84a9-4763dfb2844b")]

// Versionsinformationen für eine Assembly bestehen aus den folgenden vier Werten:
//
//      Hauptversion
//      Nebenversion
//      Buildnummer
//      Revision
//
// Sie können alle Werte angeben oder die standardmäßigen Revisions- und
Buildnummern
// übernehmen, indem Sie "*" eingeben:

[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

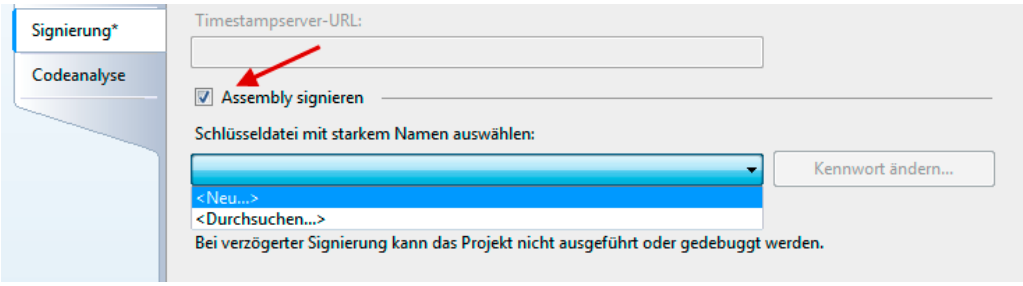
In diesem Zusammenhang ist auch die Bezeichnung *Strong Name* von Bedeutung. Dieser setzt sich aus dem Namen, der Versionsnummer, der Kulturinformation sowie einem öffentlichen Schlüssel und einer digitalen Signatur zusammen. Wie Sie diesen Schlüssel erzeugen, um einen *Strong Name* zu generieren, zeigt der folgende Abschnitt.

14.8.3 Assemblies signieren

Sicher stellt sich auch Ihnen die Frage, ob Sie Ihre Assembly signieren müssen oder nicht. Eine kurze Antwort darauf: Ihre Assembly **müssen** Sie in jedem Fall signieren, wenn Sie diese im *Global Assembly Cache* (GAC) ablegen wollen. Der erzeugte *Strong Name* stellt einen eindeutigen Bezeichner für Ihre Assembly dar.

Erstellen eines AssemblyKey-File

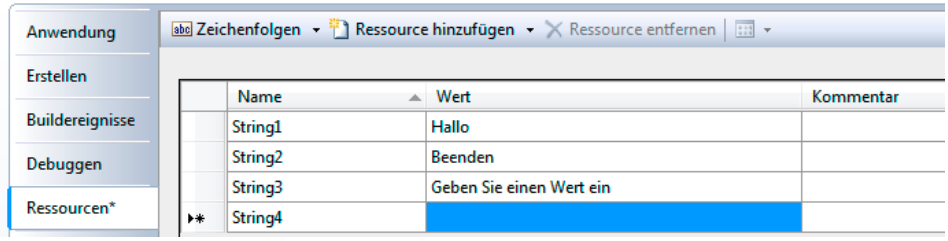
Die zur Signierung erforderliche Schlüsseldatei mit dem Schlüsselpaar wird mit dem NET-Kommandozeilentool *sn.exe* erzeugt. Seit Visual Studio 2005 ist die entsprechende Funktionalität auch über die Projekt-Eigenschaften verfügbar:



Hier erfolgt auch gleich die Verknüpfung mit dem Projekt. Sie brauchen sich also nicht mehr mit Quellcode oder der Kommandozeile herumzuplagen.

14.8.4 Komponentenressourcen einbetten

Eine Komponente besteht meist nicht nur aus Quellcode, sondern erfordert auch die Ausgabe von Grafiken und Sound bzw. diversen Zeichenfolgen etc. Alle diese „Ressourcen" können ganz normal in eine zum Projekt gehörende Ressourcendatei (*Resources.resx*) eingebettet werden und stehen über die Projekteigenschaften zur Verfügung:



Der Zugriff in der Komponente erfolgt dann wie gewohnt per *Properties.Resources....*

14.8.5 Der Komponente ein Icon zuordnen

Nachdem wir uns nun schon geraume Zeit mit der Komponentenprogrammierung herumgeschlagen haben, kommt bei den Ästheten unter den Lesern sicher bald auch der Wunsch nach einer besseren Optik für die selbst erstellten Komponenten auf. Das freudlose Icon, das standardmäßig eingeblendet wird, ist wenig informativ und wohl auch nicht jedermanns Sache.

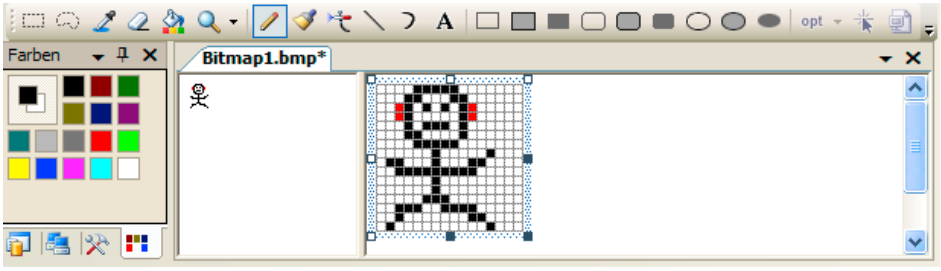
Icon erstellen

Erzeugen (oder kopieren) Sie zunächst eine 16x16 Pixel große Bitmap-Datei. Dazu können Sie zum Beispiel die Visual-Studio-Bordmittel verwenden (**Einfügen | Neues Element Bitmap**).

Speichern Sie jetzt die Datei im Projekt ab (als eingebettete Ressource). Doch schon an dieser Stelle sind einige Konventionen einzuhalten. So leitet sich der Dateiname direkt von der betreffenden Komponentenklasse ab:

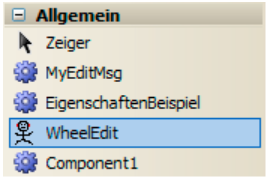
```
<Klassenname>.bmp
```

Für unser kleines Beispiel lautet der Name also *WheelEdit.bmp*.



HINWEIS: Alternativ können Sie auch das *ToolboxBitmap*-Attribut für die Komponentenklasse verwenden. In diesem Fall brauchen Sie sich nicht auf den Klassennamen zu beschränken.

Nach dem Kompilieren und Einbinden in die Visual-Studio-IDE sollte sich Ihnen etwa der folgende Anblick bieten:



14.8.6 Den Designmodus erkennen

Nicht immer läuft Ihre Komponente nur im späteren Programm des Anwenders. Auch im Visual-Studio-Form-Designer sollte die Komponente „eine gute Figur machen“. Meist stehen zu diesem Zeitpunkt nicht alle Ressourcen zur Verfügung oder das Verhalten der Komponente soll an den Designer angepasst werden. Doch wie können Sie als Entwickler den Designmodus eigentlich vom normalen Laufzeitmodus unterscheiden?

Die Antwort ist recht einfach, die Eigenschaft *DesignMode* liefert uns die gewünschte Information.

Beispiel 14.24: Unterscheidung Entwurfsmodus/Laufzeitmodus

C#

```
if (this.DesignMode)
{
    g.DrawImage(bmp, 0, 0);
}
else
{
    g.DrawImage(bmp, 0, 0);
    ImageAnimator.UpdateFrames();
}
```

14.8.7 Komponenten lizenzieren

Da mühen Sie sich mit dem Programmieren von Komponenten ab und wollen vielleicht sogar ein paar davon verkaufen. Zwangsläufig kommt dann die Frage auf, wie Sie sich davor schützen können, dass die Komponenten einfach weitergegeben werden, ohne dass der zukünftige Nutzer (Entwickler) dafür etwas bezahlt. Dass eine Firma wie Microsoft auch an diese Thematik gedacht hat, dürfte auf der Hand liegen. Das .NET Framework bringt bereits einige rudimentäre Funktionen für die Lizenzierung von Komponenten mit. Allerdings dürfte dies in den meisten Anwendungsfällen wohl eher als erste Anregung denn als Lösung verstanden werden, haben wir es doch hier mit Entwicklern zu tun, die auch mal einen Blick in den Code werfen (können).

Vier Varianten bieten sich an:

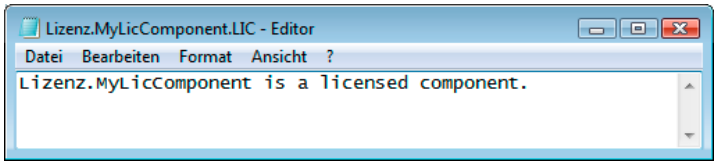
- Sie verwenden den *LicFileLicenseProvider* und eine Lizenzdatei, um minimale Lizenzierungsfunktionen zu implementieren.
- Sie leiten den *LicFileLicenseProvider* ab und implementieren einen eigenen Provider mit eigenen Prüfalgorithmen (Lizenzdatei).
- Sie erstellen sowohl einen eigenen License-Provider als auch eine eigene *License*-Klasse (freie Wahl, wo und wie der Key gespeichert wird).
- Sie vergessen die Microsoft-Klassen und kümmern sich um alles selbst.

Wir wollen im Rahmen dieses Abschnitts nur auf die ersten beiden Fälle eingehen und eine Anregung geben, wie Sie das System verbessern können.

Verwendung der LicFileLicenseProvider-Klasse

Wir bohren zunächst das Brett an der dünnsten Stelle und nutzen die von .NET bereitgestellten Möglichkeiten.

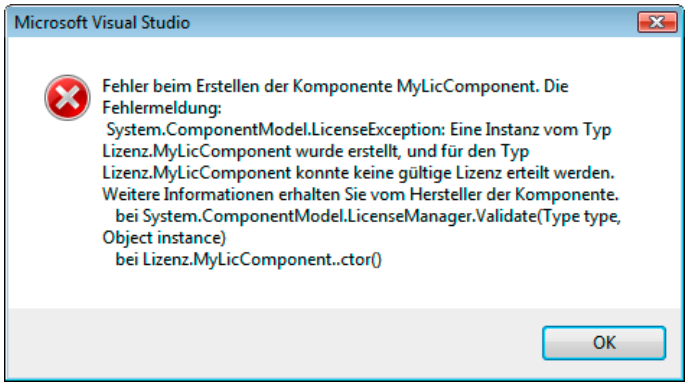
- Erstellen Sie eine neue Klassenbibliothek (z. B. mit dem Namen *Lizenz*) und Ihre Komponente (z. B. mit dem Namen *MyLicComponent*).
- Erzeugen Sie nachfolgend eine Textdatei (Notepad) mit folgendem Inhalt:



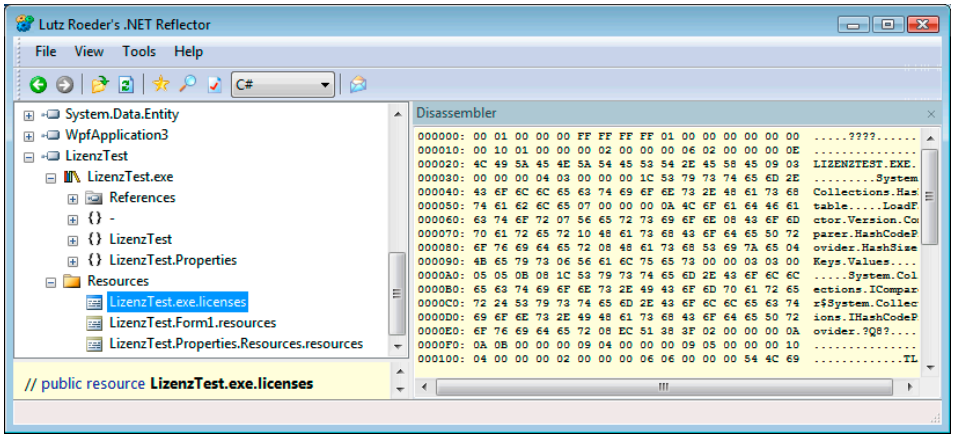
- Der Dateiname setzt sich aus dem Assembly-Namen und dem Klassennamen der Komponente zusammen.
- Erweitern Sie Ihre Komponente wie in folgendem Listing.

```
namespace Lizenz
{
// Provider zuweisen:
    [LicenseProvider(typeof(LicFileLicenseProvider))]
    public partial class MyLicComponent2 : Component
    {
// Licence-Instanz
        private License myLicence;

        public MyLicComponent2()
        {
            try
            {
                myLicence = LicenseManager.Validate(this.GetType(), this);
            }
            catch
            {
                MessageBox.Show("Fehlende Lizenz!");
                throw;
            }
            InitializeComponent();
        }
```

- Kompilieren Sie die Assembly und fügen Sie diese und die Lizenzdatei in das *bin*-Verzeichnis des finalen Projekts ein.
- Installieren Sie die Komponente in der Visual-Studio-Toolbox und versuchen Sie, die Komponente in ein Formular einzufügen. Solange die Lizenzdatei vorhanden ist, funktioniert dies auch, andernfalls erscheint unsere Fehlermeldung und dann der folgende Laufzeitfehler:



Geben Sie die kompilierte Anwendung weiter, ist keine Lizenzdatei mehr notwendig. Doch eine Unterscheidung zwischen Laufzeit- und Entwurfsmodus haben wir eigentlich gar nicht vorgenommen. Ein Blick in die Assembly zeigt, warum es mit der EXE läuft:



Ganz unbemerkt hat der Compiler die erforderlichen Lizenzen in einer Datei *licences.licx* gesammelt und als Ressource in die EXE kompiliert. Von dort werden sie dann auch geladen.

HINWEIS: Über die Sicherheit dieser Variante brauchen wir wahrscheinlich nicht zu streiten.

Ableiten der LicFileLicenseProvider-Klasse

Eine etwas sicherere Variante können Sie durch einfaches Ableiten der Klasse *LicFileLicenseProvider* erreichen. Überschreiben Sie einfach die Methode *IsKeyValid* und erstellen Sie Ihren eigenen Algorithmus:

```
public class MyPrivatLicenseProvider : LicFileLicenseProvider
{
    protected override bool IsKeyValid(string key, Type type)
    {
        if (key.Substring(2, 1) == "G")
            return true;
        else
            return false;
    }
}
```

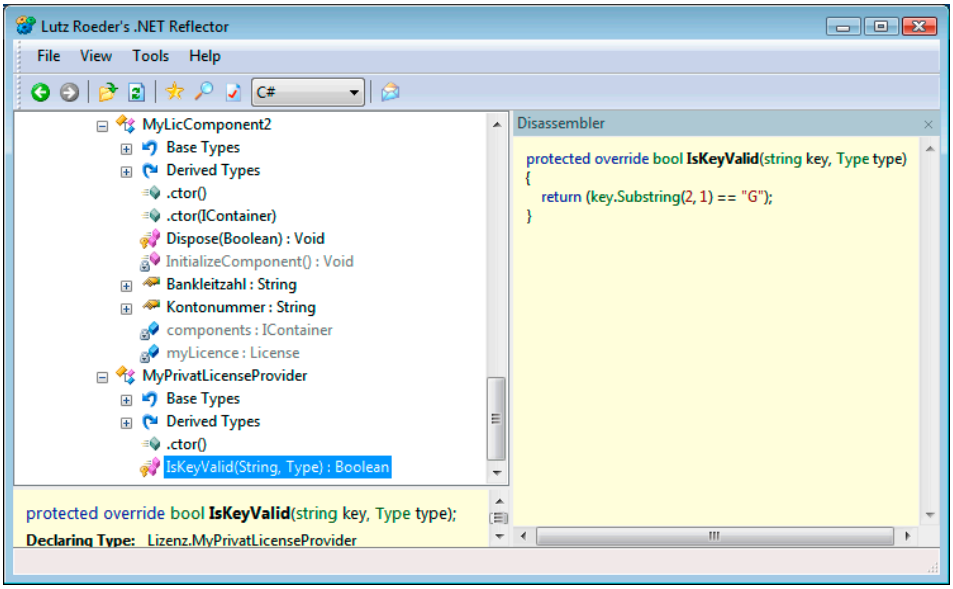
In diesem Fall muss an der dritten Stelle in der Textdatei ein großes „G“ stehen.

Für Ihre Komponente ist von Ihnen nun noch der neue *LicenceProvider* vorzugeben:

```
namespace Lizenz
{
    [LicenseProvider(typeof(MyPrivatLicenseProvider))]
    public partial class MyLicComponent2 : Component
```

Das waren bereits die Änderungen zum Standardverfahren, der Rest läuft wie im vorhergehenden Abschnitt beschrieben.

HINWEIS: Bevor Sie jetzt viel Zeit für den Algorithmus verschwenden, nutzen Sie ruhig mal den .NET-Reflector, um in die Komponenten-Assembly zu schauen. Der angezeigte Quellcode dürfte Ihnen sicher bekannt vorkommen:



Bemerkung

Wer mit den beiden genannten Varianten nicht ganz zufrieden ist, der kann sich – wenn er etwas Arbeit investiert – auch selbst helfen:

- Verkaufen Sie zusammen mit der Komponente einen Key (GUID).
- Prüfen Sie beim Erstellen der Komponenten (Konstruktor) auf den Designmodus.
- Handelt es sich um diesen, lesen Sie die Mac-Adresse der Netzwerkkarte aus (oder eine andere eindeutige ID).
- Lassen Sie per Dialogfenster die GUID eingeben und übertragen Sie diese zusammen mit der Mac-Adresse an einen Webservice.
- Im Webservice prüfen Sie zunächst, ob die GUID stimmt und speichern die Mac-Adresse ab.
- Senden Sie per Webservice einen eindeutigen Schlüssel (kann auch RSA-verschlüsselt sein) an den Client zurück und speichern Sie diesen in einer Lizenzdatei.
- Die Lizenzdatei ermöglicht dann die Prüfung der Lizenz, weitere Verbindungen zum Server sind nicht notwendig.
- Wenn jemand mehr als dreimal versucht, die Komponente auf verschiedenen Rechnern zu installieren, können Sie per GUID den Kunden ermitteln bzw. die Registrierung dauerhaft unterbinden.

HINWEIS: Vergessen Sie aber vor lauter Lizenzierung nicht die Programmierung der eigentlichen Komponenten!

14.9 Praxisbeispiele

14.9.1 AnimGif – Anzeige von Animationen

Mit dem folgenden einfachen Beispiel möchten wir Ihnen die bisherigen Ausführungen zur Komponentenentwicklung noch einmal im Zusammenhang demonstrieren.

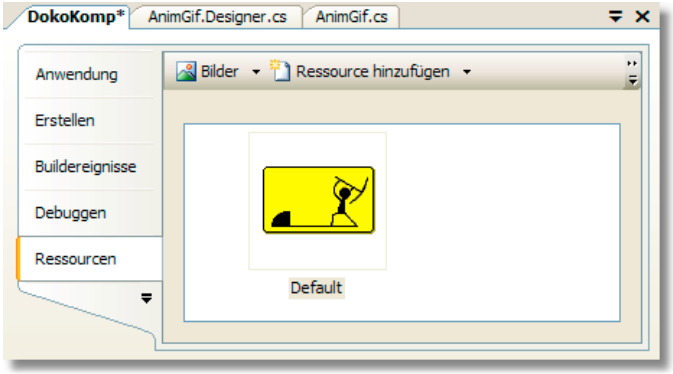
Ziel ist das Erstellen einer Komponente zur Anzeige animierter GIF-Grafiken. Neben dem standardmäßig mitgelieferten Bild soll der Anwender auch eigene Grafiken zuweisen können. Die Komponente soll sich daraufhin an die Abmessungen der Grafik anpassen. Auf diverse Extras, wie Ein-/Ausschalten etc., verzichten wir an dieser Stelle.

Oberfläche/Ressourcen

Erstellen Sie zunächst eine neue Klassenbibliothek und fügen Sie ein *Benutzerdefiniertes Steuerelement* hinzu. Speichern Sie dieses unter dem Namen *AnimGif* ab.

Nachfolgend können Sie sich im Internet auf die Suche nach einer Default-Grafik für Ihr neues Control machen. Unter den Stichworten „Animated GIF“ werden Sie ganz sicher fündig, es gibt Tausende derartiger „Nervtöter“.

Speichern Sie die Grafik auf Ihrem PC ab. Über den Menüpunkt **Projekt | Eigenschaften** können Sie die Grafik den Assembly-Ressourcen zuordnen. Vergeben Sie hier den Namen *Default*:



Damit sind die „Oberflächlichkeiten“ abgeschlossen und wir können uns den Innereien zuwenden.

Quelltext

```
namespace DokoKomp
{
```

Da der Vorfahrtyp *Control* bereits alle Eigenschaften aufweist, die für uns wichtig sind, belassen wir es bei diesem Typ:

```
    public partial class AnimGif : Control
    {
```

Für die aktuelle Grafik eine private Variable:

```
        private Bitmap bmp;
```

Der Konstruktor kümmert sich um das Auslesen der Grafik aus den Ressourcen und die Vorbereitungen für eine flackerfreie Darstellung (*Double Buffering*):

```
        public AnimGif()
        {
            InitializeComponent();
            SetStyle(ControlStyles.UserPaint, true);
            SetStyle(ControlStyles.AllPaintingInWmPaint, true);
            SetStyle(ControlStyles.DoubleBuffer, true);
            bmp = Properties.Resources.Default;
        }
```

Die neue Eigenschaft *Gif* ermöglicht es dem Anwender, eine neue Grafik zuzuordnen:

```
public Bitmap Gif
{
    get { return bmp; }
    set
    {
        bmp = value;
        if (bmp == null) bmp = Properties.Resources.Default;
```

Größenanpassung an die Grafik:

```
        this.Width = bmp.Width;
        this.Height = bmp.Height;
    }
}
```

HINWEIS: Wird die Eigenschaft zurückgesetzt, das heißt, der Anwender weist keine Grafik zu, nehmen wir automatisch die Defaultgrafik aus den Ressourcen.

Mit dem Initialisieren des Controls wird es für uns ernst:

```
protected override void InitLayout()
{
    base.InitLayout();
```

Handelt es sich nicht um den Designmodus und ist die Grafik animierbar (was für ein Wort!), dann nutzen wir die Klasse *ImageAnimator*, um ein automatisches Umschalten der Einzelbilder per integriertem Timer zu erreichen:

```
    if ((!this.DesignMode)&(ImageAnimator.CanAnimate(bmp)))
        ImageAnimator.Animate(bmp, this.OnNextFrame);
}
```

Diese Methode wird immer aufgerufen, wenn ein neuer Frame angezeigt werden soll:

```
private void OnNextFrame(object o, EventArgs e)
{
    this.Invalidate();
}
```

Die Hauptarbeit leistet die überschriebene *OnPaint*-Methode:

```
protected override void OnPaint(PaintEventArgs pe)
{
    Graphics g = pe.Graphics;
```

Im Designmodus wird immer dasselbe Bild angezeigt:

```
    if (this.DesignMode)  g.DrawImage(bmp, 0, 0);
```

Andernfalls schalten wir zum nächsten Frame um:

```
            else
            {
                g.DrawImage(bmp, 0, 0);
                ImageAnimator.UpdateFrames();
            }
        }
    }
}
```

Test

Kompilieren Sie die Klassenbibliothek und fügen Sie die Komponente in die Toolbox ein. In einem neuen Windows Forms-Projekt können Sie sich von der Funktionstüchtigkeit überzeugen.

14.9.2 Eine FontComboBox entwickeln

Zum Schluss noch ein „quick and dirty"-Beispiel.

Lassen Sie uns die *ComboBox* so modifizieren, dass später ohne zusätzlichen Programmieraufwand alle verfügbaren Schriftarten angezeigt und als fertiger Font abgerufen werden können.

Quelltext

Erster Schritt zur fertigen Komponente ist das Ableiten von einer bereits vorhandenen. Öffnen Sie dazu ein neues Projekt und fügen Sie über den Menüpunkt **Projekt | Neues Element hinzufügen | Benutzerdefiniertes Steuerelement** ein neues Steuerelement in das Projekt ein.

Das bereits bestehende Code-Gerüst ändern Sie dahingehend, dass Sie statt der Basisklasse *Control* eine *ComboBox* verwenden:

```
public partial class FontCombo : ComboBox
```

Die Klasse im Überblick:

```
    public partial class FontCombo : ComboBox
    {
```

Initialisieren:

```
        public FontCombo()
        {
            InitializeComponent();
```

Wir zeichnen die Einträge selbst:

```
            this.DrawMode = DrawMode.OwnerDrawFixed;
            this.ItemHeight = 28;
```

```
            this.DropDownStyle = ComboBoxStyle.DropDownList;
            this.DropDownWidth = 200;
            SetStyle(ControlStyles.Selectable, false);
        }
```

Zur Laufzeit werden die Fonts ermittelt und in der *Items*-Collection gespeichert:

```
        protected override void InitLayout()
        {
            base.InitLayout();
            this.Items.Clear();
            if (!this.DesignMode)
                foreach (FontFamily ff in (new InstalledFontCollection().Families))
                    try
                    {
                        Items.Add(new Font(ff, 12));
                    }
                    catch
                    { }
        }
```

Wir müssen uns um das Zeichnen eines Eintrags kümmern:

```
        protected override void OnDrawItem(DrawItemEventArgs e)
        {
```

Keine Auswahl:

```
            if (e.Index == -1)  return;
```

Hintergrund zeichnen:

```
            e.DrawBackground();
```

Auswahlrechteck zeichnen:

```
            if ((e.State & DrawItemState.Focus) != 0) e.DrawFocusRectangle();
```

String mit dem Namen der Schriftart und dem jeweiligen Font ausgeben:

```
            e.Graphics.DrawString((Items[e.Index] as Font).Name,
                                  (Items[e.Index] as Font), Brushes.Black, e.Bounds);
        }
```

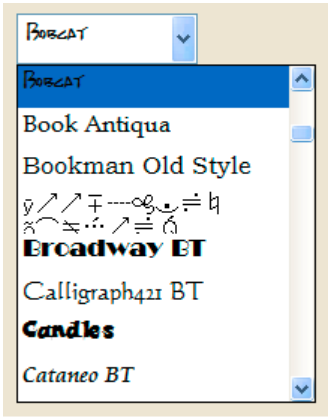
Über die Eigenschaft *SelectedFont* kann der Nutzer den ausgewählten Font abfragen und gleich nutzen:

```
        public Font SelectedFont
        {
            get
              {
                  if (this.SelectedIndex == -1)
                    return null;
                  else
                    return (this.SelectedItem as Font);
              }
```

```
            }
        }
    }
```

Test

Erstellen Sie zunächst die Assembly und binden Sie dann die Komponente in den Werkzeugkasten ein. Ein kleines Testprogramm beweist die Funktionsfähigkeit:



Noch einmal: Das war ein **einfach** gehaltenes Beispiel. Sie können natürlich auch unterschiedliche Zeilenhöhen festlegen, die Stringlänge messen usw.

14.9.3 Das PropertyGrid verwenden

Sicher haben Sie sich auch schon gefragt, wie Sie zur Laufzeit die Eigenschaften von Objekten möglichst einfach und ohne großen Aufwand verändern können. Bevor Sie jetzt lange suchen, sollten Sie einen Blick auf die recht unscheinbare *PropertyGrid*-Komponente werfen, die schon zu .NET-1.x-Zeiten ein Aschenputteldasein in der Toolbox fristete.

Für unser Beispiel werden wir eine eigene Klasse definieren, deren Eigenschaften Sie per *PropertyGrid* zur Laufzeit bearbeiten können.

Oberfläche

Fügen Sie in ein Windows-Formular eine *PropertyGrid*-Komponente ein. Das ist zunächst alles.

Quelltext

Zunächst definieren wir die gewünschte Klasse (Sie können das *PropertyGrid* aber auch an jede beliebige Komponente „klemmen“).

```
    class cMitarbeiter
    {
        public cMitarbeiter(String Name, String Vorname, DateTime Geboren)
        {
            _Name = Name;
            _Vorname = Vorname;
            _geboren = Geboren;
            _Gehalt = 0;
        }

        private String _Name;
        [Category("Personaldaten")]
        public String Name
        {
            get { return _Name; }
            set { _Name = value; }
        }

        private String _Vorname;
        [Category("Personaldaten")]
        public String Vorname
        {
            get { return _Vorname; }
            set { _Vorname = value; }
        }

        private Single _Gehalt;

        [Category("Lohndaten")]
        public Single Gehalt
        {
            get { return _Gehalt; }
            set { _Gehalt = value; }
        }

        private int _Kinder;
        [Category("Lohndaten")]
        public int Kinder
        {
            get { return _Kinder; }
            set { _Kinder = value; }
        }
...
        private DateTime _geboren;
        [Category("Personaldaten")]
        public DateTime Geboren
        {
            get { return _geboren; }
            set { _geboren = value; }
        }
    }
```

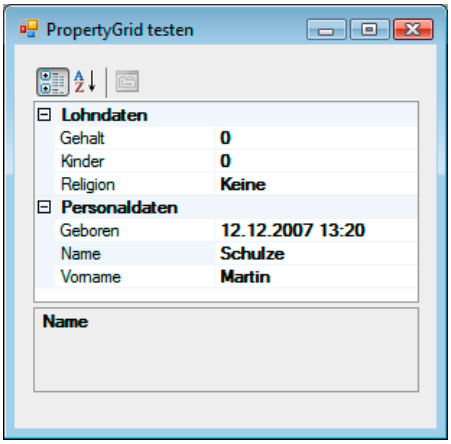
HINWEIS: Das Beispiel eignet sich auch recht gut, wenn Sie den Einfluss von Attributen testen wollen.

Die Verbindung von Grid und späterem Objekt ist absolut trivial:

```
private cMitarbeiter mitarbeiter = new cMitarbeiter("Schulze", "Martin",
System.DateTime.Now);

private void Form1_Load(object sender, EventArgs e)
{
   propertyGrid1.SelectedObject = mitarbeiter;
}
```

Test

Beim Start der Anwendung sollten auch schon alle Eigenschaften des Objekts *mitarbeiter* angezeigt werden:



Index

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z