4 Benutzerdefinierte Steuerelemente

Manche Leute sind einfach nie zufrieden, und Programmierer sind in dieser Beziehung besonders schlimm. Selbst angesichts der vielen Steuerelemente, die unter Windows Forms zur Verfügung stehen, wünscht man sich manchmal, einen Schritt (oder vielleicht mehrere) über die gewöhnlichen Steuerelemente hinauszugehen und das Reich der benutzerdefinierten Steuerelemente zu betreten.

Aus der Sicht des Programmierers ist ein benutzerdefiniertes Steuerelement (custom control) eine Klasse, die man selbst definiert und die direkt oder indirekt von `Control` abgeleitet ist. Ein benutzerdefiniertes Steuerelement kann ein vorhandenes Steuerelement erweitern oder es kann ein ganz neues Steuerelement sein. Sie können zwar weit gehende Anpassungen vornehmen, indem Sie einfach Ereignishandler für vorhandene Steuerelemente installieren und diese Ereignisse bearbeiten, aber Sie brauchen eine neue Klasse, wenn Sie die Standardereignisbearbeitung *vollständig* überschreiben wollen. Zum Beispiel können Sie das Aussehen eines `Button`-Steuerelements neu gestalten, indem Sie einen `Paint`-Ereignishandler installieren, aber Sie können nicht verhindern, dass die Schaltfläche Teile ihrer Oberfläche ebenfalls anzeigt. Dazu müssen Sie eine neue Klasse erstellen und `OnPaint` überschreiben.

Auch wenn Sie Felder oder Eigenschaften zu einem vorhandenen Steuerelement hinzufügen wollen, müssen Sie eine neue Klasse erstellen. Falls Sie allerdings nur diverse Daten mit einem Steuerelement verknüpfen wollen, reicht es oft, die Eigenschaft `Tag` zu verwenden, die für genau diesen Zweck zur Verfügung gestellt wird. Die Eigenschaft ist vom Typ `object`, daher müssen Sie beim Zugriff eine Typumwandlung vornehmen, aber sie ist ideal, um Steuerelemente auf einfache Weise zu identifizieren oder beliebige Daten mit dem Steuerelement zu verknüpfen.

Wie bei praktisch jeder Programmieraufgabe zeigt sich der eigentliche Vorteil von benutzerdefinierten Steuerelementen dann, wenn sie in mehreren Anwendungen wieder verwendet werden oder wenn sie anderen Programmierern zur Verfügung gestellt werden, entweder für Geld oder auch nur für den Ruhm.

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Erweitern vorhandener Steuerelemente

Ein Steuerelement ist im Wesentlichen ein Filter, durch den Benutzereingaben interpretiert und in Aktionen umgesetzt werden, zum Beispiel in Ereignisse. In den meisten Fällen muss ein Steuerelement drei wesentliche Aufgaben erfüllen. Erstens zeigt es etwas auf dem Bildschirm an, sodass der Benutzer es identifizieren kann. Zweitens verarbeitet es Benutzereingaben, im Allgemeinen von der Tastatur und der Maus. (Ein Steuerelement könnte auch speziell dafür programmiert sein, die Handschrift auf einem Tablet PC auszuwerten oder sogar auf Stimmeingabe zu reagieren.) Drittens löst ein Steuerelement Ereignisse aus, um die Anwendung, die das Steuerelement benutzt, über bestimmte Veränderungen zu benachrichtigen.

Weil vorhandene Steuerelemente bereits für diese drei Funktionen entworfen und getestet wurden, ist es viel einfacher, ein vorhandenes Steuerelement zu erweitern (zum Beispiel durch Ableiten einer Klasse von `Button`), statt bei null anzufangen und ein ganz neues Steuerelement von `Control` abzuleiten.

Überschreiben von Methoden

Hier ein ganz einfaches Beispiel. Eine Schaltfläche, die piepst, wenn sie mit der Maus angeklickt wird:

BeepButton.cs

```
//---------------------------------------
// BeepButton.cs (c) 2005 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Media;
using System.Windows.Forms;

class BeepButton : Button
{   protected override void OnClick(EventArgs args)
    {
        SystemSounds.Exclamation.Play();
        base.OnClick(args);
    }
}
```

`BeepButton` wird einfach von `Button` abgeleitet, die Klasse überschreibt die geschützte Methode `OnClick`. Die überschreibende Methode spielt einen Ton ab, indem sie zwei der drei Klassen im Namespace `System.Media` nutzt, der im .NET Framework 2.0 neu eingeführt wurde. Die Klasse `SystemsSounds` enthält mehrere statische Eigenschaften namens `Asterisk`, `Beep`, `Exclamation`, `Hand` und `Question`, die mit unterschiedlichen Sounds verknüpft sind. Diese Eigenschaften geben Objekte vom Typ `SystemSound` zurück. Beachten Sie den Unterschied bei den Klassennamen: Die statischen Methoden in `SystemsSounds` (Plural) geben ein Objekt vom Typ `SystemSound` (Singular) zurück. Die Klasse `SystemSound` hat eine einzige Methode namens `Play` zum Abspielen des Sounds.

Es ist *unverzichtbar*, dass die Methode `OnClick` in Ihrer neuen Klasse die Methode `OnClick` in der Basisklasse aufruft:

```
base.OnClick(args);
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Ohne diesen Aufruf hätten Programme, die das BeepButton-Steuerelement benutzen, keinen Zugriff auf das Click-Ereignis. Das hat einen simplen Grund: Button erbt das Click-Ereignis und die Methode OnClick von Control. Der Code in Control, der das Click-Ereignis definiert, sieht wahrscheinlich so ähnlich aus wie diese Zeile:

```
public event EventHandler Click;
```

Der EventHandler-Teil in dieser Anweisung legt fest, dass alle Ereignishandler, die für das Click-Ereignis installiert werden, entsprechend dem Delegaten EventHandler definiert sein müssen.

Die Klasse Control braucht sich nicht darum zu kümmern, auf welche Weise das Programm Ereignishandler installiert und wieder entfernt. Das passiert hinter den Kulissen. Aber Control ist dafür verantwortlich, das Click-Ereignis auszulösen, und das passiert in der Methode OnClick, die wahrscheinlich etwa folgendermaßen aussieht:

```
protected virtual void OnClick(EventArgs args)
{
    ...
    if (Click != null)
        Click(this, args);
    ...
}
```

Diese OnClick-Methode wird wahrscheinlich an mindestens zwei Stellen aufgerufen. Die Methode OnMouseDown des Steuerelements (wenn der Mauszeiger über das Steuerelement bewegt und eine Maustaste gedrückt wird) ruft zweifellos OnClick auf. Bei Schaltflächen wird OnClick auch aufgerufen, wenn die Schaltfläche den Eingabefokus hat und der Benutzer die LEERTASTE oder die EINGABETASTE drückt.

Ich habe mit Ellipsen angedeutet, dass OnClick möglicherweise noch andere Aufgaben ausführt. Das ist nicht sicher, aber ganz bestimmt löst sie das Click-Ereignis aus. Der Code, den ich gezeigt habe, lässt sich so beschreiben: »Falls irgendwelche Click-Ereignishandler installiert sind, rufst du diese Ereignishandler auf; dabei übergibst du das aktuelle Objekt als erstes Argument und ein EventArgs-Objekt als zweites Argument.« Falls Sie eine Klasse definieren, die direkt oder indirekt von Control abgeleitet ist, und die Methode OnClick überschreiben, ohne die entsprechende Methode in der Basisklasse aufzurufen, wird der Code, der die ganzen Click-Ereignishandler aufruft, nicht ausgeführt. (Wenn das Deaktivieren des Click-Ereignisses bei einem neuen Steuerelement Teil eines hinterhältigen Plans ist, können Sie natürlich darauf verzichten, die Methode OnClick in der Basisklasse aufzurufen.)

Codebeispiele, die zeigen, wie eine OnClick-Methode die Methode in der Basisklasse aufruft, legen diesen Aufruf normalerweise weit an den Anfang der Methode:

```
protected override void OnClick(EventArgs args)
{
    base.OnClick(args);
    ...
}
```

Manchmal ist es sinnvoll, den Code in der Methode der Basisklasse zuerst auszuführen, aber in diesem konkreten Beispiel hat es nicht richtig funktioniert (Sie werden gleich sehen, warum), daher habe ich den Aufruf von base.OnClick in ButtonBeep ans Ende von OnClick verlegt.

Hier ein einfaches Demoprogramm, das ein Objekt vom Typ BeepButton anlegt und einen Click-Ereignishandler für die Schaltfläche installiert, in dem ein Meldungsfeld geöffnet wird:

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

BeepButtonDemo.cs

```
//---------------------------------------------
// BeepButtonDemo.cs (c) 2005 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BeepButtonDemo : Form
{
    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new BeepButtonDemo());
    }
    public BeepButtonDemo()
    {
        Text = "BeepButton-Demonstration";

        BeepButton btn = new BeepButton();
        btn.Parent = this;
        btn.Location = new Point(100, 100);
        btn.AutoSize = true;
        btn.Text = "Klicken Sie auf den BeepButton";
        btn.Click += ButtonOnClick;
    }
    void ButtonOnClick(object objSrc, EventArgs args)
    {
        SilentMsgBox.Show("Der BeepButton wurde angeklickt", Text);
    }
}
```

Genau genommen konnte ich die normale Klasse `MessageBox` nicht benutzen, weil diese Klasse selbst einen Sound abspielt, wenn das Meldungsfeld angezeigt wird. Stattdessen habe ich einen Teil der Funktionalität von `MessageBox` in dieser Klasse nachgebaut:

SilentMsgBox.cs

```
//---------------------------------------------
// SilentMsgBox.cs (c) 2005 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SilentMsgBox
{
    public static DialogResult Show(string strMessage, string strCaption)
    {
        Form frm = new Form();
        frm.StartPosition = FormStartPosition.CenterScreen;
        frm.FormBorderStyle = FormBorderStyle.FixedDialog;
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        frm.MinimizeBox = frm.MaximizeBox = frm.ShowInTaskbar = false;
        frm.AutoSize = true;
        frm.AutoSizeMode = AutoSizeMode.GrowAndShrink;
        frm.Text = strCaption;

        FlowLayoutPanel pnl = new FlowLayoutPanel();
        pnl.Parent = frm;
        pnl.AutoSize = true;
        pnl.FlowDirection = FlowDirection.TopDown;
        pnl.WrapContents = false;
        pnl.Padding = new Padding(pnl.Font.Height);

        Label lbl = new Label();
        lbl.Parent = pnl;
        lbl.AutoSize = true;
        lbl.Anchor = AnchorStyles.None;
        lbl.Margin = new Padding(lbl.Font.Height);
        lbl.Text = strMessage;

        Button btn = new Button();
        btn.Parent = pnl;
        btn.AutoSize = true;
        btn.Anchor = AnchorStyles.None;
        btn.Margin = new Padding(btn.Font.Height);
        btn.Text = "OK";
        btn.DialogResult = DialogResult.OK;

        return frm.ShowDialog();
    }
}
```

Die Dateien *BeepButton.cs*, *BeepButtonDemo.cs* und *SilentMsgBox.cs* bilden zusammen das Projekt *BeepButtonDemo*.

Sie können sich überzeugen, dass alles wahr ist, was ich über den `Click`-Ereignishandler geschrieben habe: Kommentieren Sie den Aufruf der Methode `base.OnClick` in `BeepButton` aus und kompilieren Sie das Programm neu. Sie werden feststellen, dass `BeepButtonDemo` nicht mehr über das `Click`-Ereignis benachrichtigt wird.

Versuchen Sie jetzt etwas anderes: Vertauschen Sie in der Methode `OnClick` von `BeepButton` die beiden Anweisungen, sodass die Methode `OnClick` in der Basisklasse aufgerufen wird, bevor der Sound abgespielt wird:

```
protected override void OnClick(EventArgs args)
{
    base.OnClick(args);
    SystemSounds.Exclamation.Play();
}
```

Wenn Sie die Schaltfläche mit der Maus anklicken, wird diese `OnClick`-Methode aufgerufen, wahrscheinlich von der Methode `OnMouseDown` in `Control`. In diesem veränderten Code ruft `BeepButton` zuerst die Methode `OnClick` in seiner Basisklasse auf. Diese Basisklasse ist `Button`, aber die Methode `OnClick` in `Button` ruft wiederum die Methode `OnClick` in *deren* Basisklasse auf und so weiter, bis letztlich die Methode `OnClick` in `Control` aufgerufen wird. Diese `OnClick`-Methode führt

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

den Code aus, der alle Ereignishandler aufruft, die für Click installiert sind. BeepButtonDemo hat einen solchen Ereignishandler installiert, daher wird die Methode ButtonOnClick in BeepButtonDemo aufgerufen. Diese Methode ruft die statische Methode Show in SilentMsgBox auf, die (wie MessageBox) ein modales Dialogfeld anzeigt und wartet, bis der Benutzer es wieder schließt. Wenn der Benutzer das Meldungsfeld geschlossen hat, gibt der Aufruf von Show die Kontrolle wieder an ButtonOnClick zurück, die ihrerseits die Kontrolle an die Methode OnClick in BeepButton zurückreicht, wo endlich (im veränderten Code) der Sound abgespielt wird – leider lange nachdem der Benutzer die Schaltfläche angeklickt hat.

Was Sie daraus lernen sollten: Sie können wählen, wann eine On-Methode die Methode in der Basisklasse aufruft. Überlegen Sie sich diese Entscheidung genau.

Hinzufügen neuer Eigenschaften

Die Klasse BeepButton hat nicht nur demonstriert, wie Sie vorhandene Steuerelemente erweitern können, sondern auch, wie Sie zwei der drei Klassen im Namespace System.Media benutzen. Die dritte Klasse in diesem Namespace ist SoundPlayer. Das nächste Programm verwendet sie, um ein neues Steuerelement namens SoundButton zu erstellen. Das Steuerelement SoundButton ähnelt BeepButton, es gibt aber nicht einen simplen Pieps von sich, sondern spielt eine *.wav*-Datei (Microsoft Windows Waveform-Datei) ab. Dank dieses Features können Sie Ihre Schaltflächen zum Beispiel von der samtenen Stimme von Bart Simpson untermalen lassen.

Auch diese Klasse ist von Button abgeleitet. Zuerst erstellt sie ein SoundPlayer-Objekt und speichert es als Feld:

SoundButton.cs

```
//-----------------------------------------
// SoundButton.cs (c) 2005 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.IO;
using System.Media;
using System.Windows.Forms;

class SoundButton : Button
{
    SoundPlayer sndplay = new SoundPlayer();

    public string WaveFile
    {
        set
        {
            sndplay.SoundLocation = value;
            sndplay.LoadAsync();
        }
        get
        {
            return sndplay.SoundLocation;
        }
    }
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
    public Stream WaveStream
    {
        set
        {
            sndplay.Stream = value;
            sndplay.LoadAsync();
        }
        get
        {
            return sndplay.Stream;
        }
    }
    protected override void OnClick(EventArgs args)
    {
        if (sndplay.IsLoadCompleted)
            sndplay.Play();

        base.OnClick(args);
    }
}
```

Die Klasse definiert zwei neue Eigenschaften namens `WaveFile` und `WaveStream`. Diese Eigenschaften entsprechen den `SoundPlayer`-Eigenschaften `SoundLocation` und `Stream`, mit denen ein Programm für `SoundPlayer` als Quelle eine Waveform-Datei angeben kann. Die Eigenschaft `SoundLocation` von `SoundPlayer` (und die Eigenschaft `WaveFile` von `SoundButton`) ist ein String, der eine lokale Datei oder einen URL angibt. `Stream` (und `WaveStream`) ist ein Objekt vom Typ `Stream`, eine abstrakte Klasse, die im Namespace `System.IO` definiert ist. Von `Stream` sind unter anderem `FileStream` abgeleitet, die im Allgemeinen mit einer offenen Datei verknüpft ist, und `MemoryStream`, die den Zugriff auf einen Speicherblock ermöglicht, ganz so als wäre er eine Datei. Die Option `Stream` ist insbesondere nützlich, wenn Sie Waveform-Dateien in Ihre ausführbare Datei einbetten und darauf als Ressourcen zugreifen (das demonstriere ich in Kürze). `SoundPlayer` verwendet beim Zugriff auf die Waveform-Datei die Eigenschaft, der zuletzt ein Wert zugewiesen wurde.

Normalerweise lädt `SoundPlayer` die Waveform-Datei erst dann in den Arbeitsspeicher, wenn es sie abspielt. In diesem Fall ist das, während die Methode `OnClick` ausgeführt wird. Ich war etwas besorgt, dass dieser Ladevorgang die Verarbeitung von `OnClick` verzögern könnte, daher laden die beiden Eigenschaften die Waveform-Dateien sofort in einem separaten Thread, indem sie `LoadAsync` aufrufen. (Wenn Sie stattdessen die Methode `Load` von `SoundPlayer` aufrufen, wird die Waveform-Datei synchron geladen, also im selben Thread. Das könnte die Initialisierung des Programms verzögern.)

Der normale `Play`-Befehl von `SoundPlayer` arbeitet asynchron. Die Methode `OnClick` startet das Abspielen des Sounds und erledigt dann ihre anderen Aufgaben, nämlich das Aufrufen der Methode `OnClick` der Basisklasse, was wiederum andere Arbeitsschritte auslöst. (Dagegen kehrt die Methode `PlaySync` von `SoundPlayer` erst dann zurück, wenn der Sound vollständig abgespielt wurde. Die Methode `PlayLooping` ist asynchron und spielt den Sound in einer Schleife so lange ab, bis `Stop` aufgerufen wird.)

Hier ein Programm, das drei Möglichkeiten demonstriert, `SoundButton` zu benutzen: Laden einer lokalen Datei (in diesem Fall der »Ta-da«-Sound aus dem Windows-Verzeichnis *Media*), Laden einer Datei aus dem Internet (Löwengebrüll von der Website des Oakland Zoo) und Verwenden einer Ressource. Das Projekt *SoundButtonDemo* enthält außerdem die Datei *SoundButton.cs*

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

und eine Waveform-Datei namens *MakeItSo.wav*, in der Sie mich auf der Brücke meines Computers hören können.

SoundButtonDemo.cs

```
//---------------------------------------------
// SoundButtonDemo.cs (c) 2005 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class SoundButtonDemo : Form
{
    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new SoundButtonDemo());
    }
    public SoundButtonDemo()
    {
        Text = "SoundButton-Demonstration";

        SoundButton btn = new SoundButton();
        btn.Parent = this;
        btn.Location = new Point(50, 25);
        btn.AutoSize = true;
        btn.Text = "SoundButton mit Datei";
        btn.Click += ButtonOnClick;
        btn.WaveFile = Path.Combine(
            Environment.GetEnvironmentVariable("windir"),
            "Media\\tada.wav");

        btn = new SoundButton();
        btn.Parent = this;
        btn.Location = new Point(50, 125);
        btn.AutoSize = true;
        btn.Text = "SoundButton mit URI";
        btn.Click += ButtonOnClick;
        btn.WaveFile = "http://www.oaklandzoo.org/atoz/azlinsnd.wav";

        btn = new SoundButton();
        btn.Parent = this;
        btn.Location = new Point(50, 225);
        btn.AutoSize = true;
        btn.Text = "SoundButton mit Ressource";
        btn.Click += ButtonOnClick;
        btn.WaveStream = GetType().Assembly.GetManifestResourceStream(
                "SoundButtonDemo.MakeItSo.wav");
    }
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
    void ButtonOnClick(object objSrc, EventArgs args)
    {
        Button btn = objSrc as Button;
        SilentMsgBox.Show("Der SoundButton wurde angeklickt", btn.Text);
    }
}
```

Keiner dieser drei Ansätze, eine Waveform-Datei einzulesen, ist trivial. Die Datei *tada.wav* liegt im Unterverzeichnis *Media* Ihres Windows-Verzeichnisses, aber der Name dieses Verzeichnisses kann zum Beispiel *WINDOWS* oder *WINNT* sein. Das Programm ermittelt den Verzeichnisnamen mithilfe von `GetEnvironmentVariable` der Klasse `Environment` und hängt daran dann den Namen des Unterverzeichnisses *Media* und den Dateinamen *tada.wav* an.

Es ist natürlich viel einfacher, einen URL anzugeben, aber nur wenn Sie absolut sicher sind, dass sich der URL während der kommerziellen Lebensdauer Ihres Programms nicht ändert und Ihr Programm Zugriff auf eine funktionierende Internetverbindung hat.

Die narrensichere Methode, auf Binärdateien zuzugreifen, besteht darin, sie in Form einer Ressource zu einem Teil Ihrer ausführbaren Datei zu machen. Dazu fügen Sie erst die Datei zu Ihrem Projekt hinzu. Wenn Sie in Microsoft Visual Studio die Datei im Projektmappen-Explorer anklicken, öffnet sich rechts unten das Fenster *Eigenschaften*. Es ist *sehr wichtig*, dass Sie die Eigenschaft *Buildvorgang* in *Eingebettete Ressource* ändern. Andernfalls kann Ihr Programm die Ressource während der Laufzeit nicht laden und Sie machen sich verrückt, weil Sie verzweifelt nach der Ursache suchen.

Die letzte Anweisung im Konstruktor von `SoundButtonDemo` zeigt den Code, mit dem Sie diese binäre Ressource als `Stream`-Objekt in Ihr Programm laden. Dem Dateinamen müssen Sie einen »Ressourcennamespace« voranstellen, für den Visual Studio normalerweise den Namen des Projekts wählt. Sie können diesen Namen im Eigenschaftendialogfeld des Projekts ändern. Im Abschnitt *Anwendung* dieses Dialogfelds finden Sie den Namen im Feld *Standardnamespace*.

Zeichnen des Steuerelements

Ganz radikal können Sie ein vorhandenes Steuerelement verändern, wenn Sie sein gesamtes Aussehen umgestalten. Dazu müssen Sie zumindest die Methode `OnPaint` überschreiben. Falls Ihre Zukunftsvision erfordert, dass das Steuerelement eine andere Größe als im Normalfall hat, werden Sie außerdem `GetPreferredSize` und unter Umständen `OnResize` überschreiben. Wenn Sie Glück haben, können Sie die gesamte Logik zum Verarbeiten von Tastatur- und Mauseingaben unverändert lassen.

Windows Forms bietet Unterstützung für Programmierer, die ihre Logik zum Zeichnen von Steuerelementen implementieren. Bevor Sie die Schaltfläche ganz neu erfinden, sollten Sie sich `ControlPaint` genauer ansehen. Diese Klasse enthält eine Reihe von statischen Methoden, die verschiedenen Aufgaben im Zusammenhang mit dem Zeichnen von Steuerelementen erledigen und Systemfarben in helle und dunkle Varianten konvertieren. Systemfarben (system color), Stifte (pen) und Pinsel (brush) sind übrigens in drei Klassen aus dem Namespace `System.Drawing` untergebracht: `SystemColors`, `SystemPens` und `SystemBrushes`. Die Klassen `ProfessionalColors` und `ProfessionalColorTable` in `System.Windows.Forms` stellen Farben zur Verfügung, die angeblich denen in Microsoft Office ähneln sollen. `ProfessionalColors` ist eine Auflistung statischer Eigenschaften, und `ProfessionalColorTable` enthält identische Instanzeigenschaften, auf die Sie zugreifen können, nachdem Sie eine Instanz von `ProfessionalColorTable` angelegt haben.

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Ich habe mich für die Klasse `RoundButton` entschieden, die Logik zum Zeichnen weitgehend selbst zu zimmern und meine eigenen Farben zu wählen. Wie der Name andeutet, ist `RoundButton` von `Button` abgeleitet, sie erstellt aber eine runde Schaltfläche. Dieser Code demonstriert auch, wie Sie Steuerelemente erstellen, die nicht rechteckig sind.

RoundButton.cs

```
//---------------------------------------
// RoundButton.cs (c) 2005 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class RoundButton : Button
{
    public RoundButton()
    {
        SetStyle(ControlStyles.UserPaint, true);
        SetStyle(ControlStyles.AllPaintingInWmPaint, true);
    }
    public override Size GetPreferredSize(Size szProposed)
    {
        // Größe auf Basis des Textstrings festlegen, der angezeigt werden soll.
        Graphics grfx = CreateGraphics();
        SizeF szf = grfx.MeasureString(Text, Font);
        int iRadius = (int)Math.Sqrt(Math.Pow(szf.Width / 2, 2) +
                                     Math.Pow(szf.Height / 2, 2));
        return new Size(2 * iRadius, 2 * iRadius);
    }
    protected override void OnResize(EventArgs args)
    {
        base.OnResize(args);

        // Die kreisförmige Region verhindert eine Rechteckform der Schaltfläche.
        GraphicsPath path = new GraphicsPath();
        path.AddEllipse(ClientRectangle);
        Region = new Region(path);
    }
    protected override void OnPaint(PaintEventArgs args)
    {
        Graphics grfx = args.Graphics;
        grfx.SmoothingMode = SmoothingMode.AntiAlias;
        Rectangle rect = ClientRectangle;

        // Innenbereich zeichnen (dunkler, wenn angeklickt).
        bool bPressed = Capture & ((MouseButtons & MouseButtons.Left) != 0) &
            ClientRectangle.Contains(PointToClient(MousePosition));

        GraphicsPath path = new GraphicsPath();
        path.AddEllipse(rect);
        PathGradientBrush pgbr = new PathGradientBrush(path);
        int k = bPressed ? 2 : 1;
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        pgbr.CenterPoint = new PointF(k * (rect.Left + rect.Right) / 3,
                                      k * (rect.Top + rect.Bottom) / 3);
        pgbr.CenterColor = bPressed ? Color.Blue : Color.White;
        pgbr.SurroundColors = new Color[] { Color.SkyBlue };
        grfx.FillRectangle(pgbr, rect);

        // Rand zeichnen (dicker bei Standardschaltfläche)
        Brush br = new LinearGradientBrush(rect,
                       Color.FromArgb(0, 0, 255), Color.FromArgb(0, 0, 128),
                       LinearGradientMode.ForwardDiagonal);
        Pen pn = new Pen(br, (IsDefault ? 4 : 2) * grfx.DpiX / 72);
        grfx.DrawEllipse(pn, rect);

        // Text zentriert im Rechteck zeichnen (grau, falls deaktiviert).
        StringFormat strfmt = new StringFormat();
        strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;
        br = Enabled ? SystemBrushes.WindowText : SystemBrushes.GrayText;
        grfx.DrawString(Text, Font, br, rect, strfmt);

        // Gepunktete Linie um den Text zeichnen, falls
        // die Schaltfläche den Eingabefokus hat.
        if (Focused)
        {
            SizeF szf = grfx.MeasureString(Text, Font, PointF.Empty,
                                           StringFormat.GenericTypographic);
            pn = new Pen(ForeColor);
            pn.DashStyle = DashStyle.Dash;
            grfx.DrawRectangle(pn,
                rect.Left + rect.Width / 2 - szf.Width / 2,
                rect.Top + rect.Height / 2 - szf.Height / 2,
                szf.Width, szf.Height);
        }
    }
}
```

Der Konstruktor setzt zwei `ControlStyles`-Flags. Bei `Button` sind diese zwei Flags bereits in der Standardeinstellung `true`, aber das dürfen Sie nicht für alle Steuerelemente als gegeben annehmen. Wenn Sie die Flags auf `true` setzen, ist sichergestellt, dass die gesamte Zeichenlogik in `OnPaint` durchgeführt wird. Daher reicht es aus, `OnPaint` zu überschreiben, um die gesamte Zeichenlogik zu ersetzen. (Falls das Flag `AllPaintingInWmPaint` stattdessen den Wert `false` hat, können Sie den Hintergrund des Steuerelements zeichnen, indem Sie `OnPaintBackground` überschreiben.)

`GetPreferredSize` ist eine wichtige Methode in Verbindung mit der automatischen Größeneinstellung. Wenn `AutoSize` den Wert `true` hat, ruft ein Layoutmanager diese Methode auf, um zu ermitteln, welche Größe das Steuerelement gern hätte. Immer wenn etwas passiert, das die Größe beeinflussen könnte (zum Beispiel Änderungen an den Eigenschaften `Font` oder `Text` des Steuerelements), wird `GetPreferredSize` erneut aufgerufen, um die neue Größe abzufragen. Die in `RoundButton` definierte Methode `GetPreferredSize` ermittelt die Pixelabmessungen des Strings in der Eigenschaft `Text`. Dazu ruft sie `MeasureString` auf. Dann berechnet sie den Abstand von der Mitte des Rechtecks bis zu einer Ecke. Dieser Wert ist der gewünschte Radius der runden Schaltfläche.

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Die Methode OnResize wird jedes Mal aufgerufen, wenn sich die Größe des Steuerelements ändert. Das kann durch eine automatische Größenanpassung oder durch eine explizite Größenänderung ausgelöst werden. Der Code in der Methode OnResize hat außerdem die Aufgabe, dem Steuerelement ein nicht rechteckiges Aussehen zu verschaffen. Dazu verändert die Methode die Eigenschaft Region, die in der Klasse Control definiert ist. Sie weist der Eigenschaft Region ein Objekt vom Typ Region zu, eine Klasse, die in System.Drawing definiert ist. Eine grafische Region definiert einen unregelmäßigen Bereich als Abfolge von Linien. Nachdem die Eigenschaft Region verändert wurde, hat das Steuerelement weiterhin rechteckige Abmessungen, aber alle Teile des Steuerelements, die außerhalb des Bereichs liegen, der in der Region definiert ist, wird transparent – sowohl grafisch als auch im Bezug auf Mausklicks.

Falls Sie sich allein auf die Klasse Region beschränken, können Sie nur Regionen erstellen, die sich aus booleschen Kombinationen mehrerer Rechtecke zusammensetzen. Ein allgemeinerer Ansatz besteht darin, zuerst mithilfe der Klasse GraphicsPath einen Pfad zu konstruieren. Ein Pfad ist eine Auflistung von Linien und Kurven, die miteinander verbunden sein können (aber nicht müssen) und Bereiche einschließen können (aber nicht müssen). Eine vollständige Beschreibung von Pfaden und Regionen finden Sie in Kapitel 15 meines Buchs *Windows-Programmierung mit C#* (Microsoft Press, 2001). Der Pfad, den RoundButton anlegt, besteht schlicht aus einer Ellipse, die so groß ist wie der Clientbereich der Schaltfläche. Diese Ellipse (genauer gesagt: die Innenfläche der Ellipse) wird in eine Region umgewandelt, und diese Region wird der Eigenschaft Region der Schaltfläche zugewiesen.

Die einzige andere Methode in RoundButton ist OnPaint, und diese Methode ruft *nicht* die Methode OnPaint in der Basisklasse auf, weil die Basisklassenmethode nichts tun soll. OnPaint wird immer dann aufgerufen, wenn irgendein Teil der Schaltfläche neu gezeichnet werden muss.

Die Methode zeichnet zuerst den Innenbereich der Schaltfläche. Ich verwende hier einen PathGradientBrush, damit die Schaltfläche ein plastisches Aussehen bekommt. (Alle Arten von Pinseln, nicht nur Gradientenpinsel, werden in Kapitel 17 von *Windows-Programmierung mit C#* beschrieben.) Die Schaltfläche sollte aber auch eine optische Rückmeldung liefern, wenn sie mit der Maus »gedrückt« wird. Der Code wählt eine dunklere Farbe aus, falls die Eigenschaft Capture den Wert true hat (das heißt, falls die Mauseingabe an das Steuerelement weitergeleitet wird), die linke Maustaste gedrückt ist und sich der Mauszeiger über dem Steuerelement befindet.

Als Nächstes zeichnet OnPaint die Umrandung. Dazu wird ein Stift benutzt, der auf LinearGradientBrush aufbaut. Eine normale Schaltfläche zeichnet im Allgemeinen einen dickeren Rand, wenn die Schaltfläche die Standardschaltfläche ist (das heißt, wenn sie auf die EINGABETASTE reagiert). RoundButton fragt die Eigenschaft IsDefault ab und verändert abhängig vom Ergebnis die Breite der Umrandung.

Dann zeichnet OnPaint den Text. Dies gestaltet sich wiederum komplexer, als es zuerst den Anschein hat. Falls die Schaltfläche deaktiviert ist, sollte der Text eingegraut gezeichnet werden. Die Methode wählt beim Zeichnen des Textes zwischen den Systempinseln SystemBrushes.WindowText und SystemBrushes.GrayText. Das StringFormat-Objekt hilft dabei, den Text in der Mitte der Schaltfläche zu positionieren.

Falls die Schaltfläche den Eingabefokus hat, sollte eine gestrichelte Linie um den Text erscheinen. Das ist die Aufgabe des letzten Abschnitts in OnPaint. Die Methode ruft erneut MeasureString auf, um die Breite und Höhe des Textstrings zu ermitteln, und stellt dann ein Rechteck zusammen, mit dem die gestrichelte Linie angezeigt wird.

Vielleicht ist Ihnen aufgefallen, dass MeasureString in GetPreferredSize etwas anders aufgerufen wird als in OnPaint. In OnPaint wird neben dem Textstring und der Schriftart StringFormat.Gene-

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

ricTypographic als Argument an die Methode übergeben. In der Standardeinstellung gibt MeasureString Abmessungen zurück, die ein bisschen größer sind als der String. Ich dachte, dieses Verhalten wäre geeignet, um die Größe der Schaltfläche zu bestimmen, weil so ein kleiner Rand um den Text entsteht. Als ich aber denselben MeasureString-Aufruf benutzte, um die gestrichelte Linie um den Text zu zeichnen, wurden die linke und die rechte Seite des Rechtecks durch die Umrandung der Schaltfläche überdeckt. Wenn ich StringFormat.GenericTypographic an MeasureString übergebe, liegen die Abmessungen näher an der tatsächlichen Textgröße, und so ist die gestrichelte Linie enger am Text. (In Kapitel 9 von *Windows-Programmierung mit C#* finden Sie eine ausführliche Beschreibung von GenericTypographic.)

Hier ein kleines Programm zum Testen der Schaltflächen. Das Projekt *RoundButtonDemo* umfasst *RoundButton.cs* und diese Datei.

RoundButtonDemo.cs

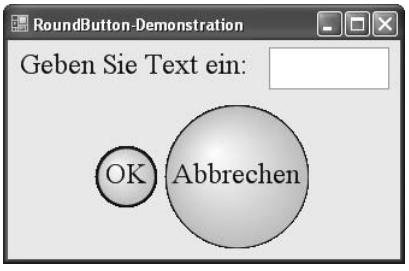
```
//---------------------------------------------
// RoundButtonDemo.cs (c) 2005 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class RoundButtonDemo : Form
{
    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new RoundButtonDemo());
    }
    public RoundButtonDemo()
    {
        Text = "RoundButton-Demonstration";
        Font = new Font("Times New Roman", 18);
        AutoSize = true;
        AutoSizeMode = AutoSizeMode.GrowAndShrink;

        FlowLayoutPanel flow = new FlowLayoutPanel();
        flow.Parent = this;
        flow.AutoSize = true;
        flow.FlowDirection = FlowDirection.TopDown;

        FlowLayoutPanel flowTop = new FlowLayoutPanel();
        flowTop.Parent = flow;
        flowTop.AutoSize = true;
        flowTop.Anchor = AnchorStyles.None;

        Label lbl = new Label();
        lbl.Parent = flowTop;
        lbl.AutoSize = true;
        lbl.Text = "Geben Sie Text ein:";
        lbl.Anchor = AnchorStyles.None;
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        TextBox txtbox = new TextBox();
        txtbox.Parent = flowTop;
        txtbox.AutoSize = true;

        FlowLayoutPanel flowBottom = new FlowLayoutPanel();
        flowBottom.Parent = flow;
        flowBottom.AutoSize = true;
        flowBottom.Anchor = AnchorStyles.None;

        RoundButton btnOk = new RoundButton();
        btnOk.Parent = flowBottom;
        btnOk.Text = "OK";
        btnOk.Anchor = AnchorStyles.None;
        btnOk.DialogResult = DialogResult.OK;
        AcceptButton = btnOk;

        RoundButton btnCancel = new RoundButton();
        btnCancel.Parent = flowBottom;
        btnCancel.AutoSize = true;
        btnCancel.Text = "Abbrechen";
        btnCancel.Anchor = AnchorStyles.None;
        btnCancel.DialogResult = DialogResult.Cancel;
        CancelButton = btnCancel;

        btnOk.Size = btnCancel.Size;
    }
}
```

Dieses Programm simuliert ein kleines Dialogfeld mit einem Beschriftungsfeld, einem `TextBox`-Steuerelement und zwei `RoundButton`-Steuerelementen für *OK* und *Abbrechen*. Drei `FlowPanel`-Steuerelemente besorgen das dynamische Layout.

Zuerst setzte ich bei beiden `RoundButton`-Steuerelementen die Eigenschaft `AutoSize` auf `true`, genau wie bei normalen Schaltflächen. Das Ergebnis sah sehr, sehr falsch aus:

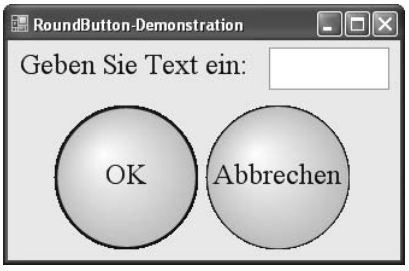


Das *musste* korrigiert werden. Ich beschloss, `AutoSize` nur bei der *Abbrechen*-Schaltfläche auf `true` zu setzen. Nachdem beide Schaltflächen erstellt wurden, bekommt die *OK*-Schaltfläche dieselbe Größe wie die *Abbrechen*-Schaltfläche:

```
btnOk.Size = btnCancel.Size;
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Das sieht schon viel besser aus:



Und ich habe den vermutlich wahren Grund entdeckt, warum Schaltflächen nicht standardmäßig rund sind.

Nehmen wir an, ein Programm will das `RoundButton`-Steuerelement benutzen und die Oberfläche noch etwas anpassen. Es installiert brav einen Ereignishandler für `Paint` und ... nichts passiert. Das Problem liegt darin, dass `RoundButton` die Methode `OnPaint` überschreibt, aber nicht die Methode `OnPaint` in der Basisklasse aufruft, weil sie ja nicht will, dass die Basisklasse irgendetwas tut. Aber die Basisklasse ist dafür zuständig, das `Paint`-Ereignis auszulösen. Und `RoundButton` kann das `Paint`-Ereignis nicht selbst auslösen, weil nur die Klasse, die das Ereignis definiert, es auch auslösen kann.

Die Lösung (sofern gewünscht) besteht darin, für `RoundButton` mithilfe des Schlüsselworts `new` ein eigenes `Paint`-Ereignis zu definieren und dieses Ereignis durch die Methode `OnPaint` in `RoundButton` auslösen zu lassen.

Kombinieren vorhandener Steuerelemente

Eine beliebte Lösung beim Erstellen neuer Steuerelemente besteht darin, vorhandene Steuerelemente zu kombinieren. Um die Unterstützung der vorhandenen Steuerelementlogik so weit wie möglich zu übernehmen, wird empfohlen, dass Sie solche Steuerelemente von der Klasse `UserControl` ableiten. Insbesondere unterstützt diese Klasse die Tastaturnavigation über mehrere Steuerelemente hinweg.

Als erstes Beispiel wollen wir ein Steuerelement erstellen, das sich gar nicht so sehr von einem beliebten vorhandenen Steuerelement unterscheidet. Dieses Steuerelement ist unter mehreren Namen bekannt: Drehfeld, Up-Down-Steuerelement, Spin-Schaltfläche. Es wird durch die Klasse `NumericUpDown` implementiert. Aber ich habe schon immer ein grundlegendes Problem mit Steuerelementen gehabt, bei denen Sie Zahlenwerte mithilfe von Schaltflächen verändern müssen, deren Pfeile nach oben und unten zeigen. Auf der einen Seite könnte der nach oben gerichtete Pfeil für »höher« stehen, das heißt für größere Werte, und der nach unten gerichtete Pfeil für »niedriger«. Aber wenn ich mir eine Liste der Zahlen aufzeichne, unter denen ich auswählen kann, bedeutet der nach oben gerichtete Pfeil offensichtlich »auf 0 zu« und der nach unten gerichtete Pfeil »auf unendlich zu«:

0
1
2
3
...

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Diese Verwechslungsgefahr ließe sich vermeiden, wenn die Bildlaufleiste horizontal wäre statt vertikal. Dann wäre klar, dass der nach links gerichtete Pfeil für kleinere Werte steht und der nach rechts gerichtete Peil für größere Werte. Das gilt zumindest in Kulturen, die von links nach rechts schreiben.

Weil ich zutiefst überzeugt bin, dass ich in dieser Frage recht habe und die übrige Welt nur mein Beispiel zu sehen braucht, um mit fliegenden Fahnen auf meine Seite überzulaufen, habe ich beschlossen, mein `NumericScan`-Steuerelement (so habe ich es genannt) in eine DLL (Dynamic-Link Library) namens *NumericScan.dll* zu packen. Ich habe auch einige Unterstützungsfunktionen hinzugefügt, sodass dieses Steuerelement sich besser in den Visual Studio-Designer einpasst.

Ob eine Sammlung von Quellcodedateien eine ausführbare Datei (*.exe*) oder eine Dynamic-Link Library (*.dll*) wird, hängt letztlich vom Schalter `/target` des C#-Compilers ab. Wenn Sie diesen Schalter auf `/target:exe` oder `/target:winexe` setzen, erstellen Sie eine ausführbare Datei, wenn Sie ihn auf `/target:library` setzen, eine DLL. In Visual Studio legen Sie in den Projekteigenschaften fest, ob Sie eine ausführbare Datei oder eine DLL erstellen.

Es ist ganz einfach, statt eines Programmprojekts ein DLL-Projekt anzulegen. Falls Sie die vordefinierten Projekttypen in Visual Studio verwenden wollen, können Sie als Vorlage entweder *Klassenbibliothek* oder *Windows-Steuerelementbibliothek* auswählen. Falls Sie die Vorlage *Leeres Projekt* bevorzugen, werden Sie nach dem Anlegen des Projekts normalerweise die Projekteigenschaften öffnen. Wählen Sie unter *Ausgabetyp* die Option *Klassenbibliothek* aus. Mit dieser Option wird statt einer ausführbaren Datei (*.exe*) eine Dynamic-Link Library (*.dll*) erstellt. Fügen Sie dann mindestens eine Quellcodedatei zum Projekt hinzu und fangen Sie an zu programmieren.

Das einzige Problem besteht darin, dass eine DLL nicht direkt ausgeführt werden kann, daher landen Sie früher oder später an dem Punkt, wo Ihr Code einwandfrei kompiliert wird, Sie aber nicht wissen, ob das Steuerelement tatsächlich funktioniert. Sie brauchen ein richtiges Programm, um das Steuerelement auszutesten.

Aus diesem Grund ist es nützlich, wenn Sie beim Erstellen einer DLL auch ein Testprogramm zur Hand haben. Und das geht am einfachsten, indem Sie das DLL-Projekt *und* das Testprogramm in dieselbe Visual Studio-Projektmappe packen. Das können Sie folgendermaßen machen.

Wählen Sie in Visual Studio den Menübefehl *Datei/Neu/Projekt*. Daraufhin öffnet sich wie üblich das Dialogfeld *Neues Projekt*. Als Projektnamen geben Sie *NumericScan* ein. Aktivieren Sie außerdem das Kontrollkästchen *Projektmappenverzeichnis erstellen*. Wenn dieses Kontrollkästchen aktiviert ist, wird ein Projektmappenverzeichnis namens *NumericScan* und innerhalb dieses Verzeichnisses ein Projektverzeichnis mit demselben Namen *NumericScan* erstellt. Wählen Sie dann erneut den Menübefehl *Datei/Neu/Projekt*. Geben Sie im Dialogfeld *Neues Projekt* als Projektnamen *TestProgram* ein und stellen Sie sicher, dass im Kombinationsfeld *Projektmappe* der Eintrag *Hinzufügen* ausgewählt ist. Jetzt enthält die Projektmappe *NumericScan* zwei Projekte namens *NumericScan* und *TestProgram*.

Stellen Sie in den Projekteigenschaften für das Projekt *NumericScan* sicher, dass der Ausgabetyp *Klassenbibliothek* lautet. Klicken Sie im Projektmappen-Explorer mit der rechten Maustaste auf *TestProgram* und wählen Sie den Befehl *Als Startprojekt festlegen*. Das bedeutet, dass Visual Studio nach dem Erstellen der gesamten Projektmappe *TestProgram* startet. Es gibt noch ein paar Kleinigkeiten zu tun, wenn Sie eine DLL erstellen und testen wollen, aber darauf komme ich später zurück.

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Das Steuerelement NumericScan, das ich hier vorstelle, umfasst ein TextBox-Steuerelement und zwei Schaltflächen. Dies sind aber keine gewöhnlichen Schaltflächen. Wenn Sie eine der Pfeilschaltflächen auf einem normalen NumericUpDown-Steuerelement anklicken und die Maustaste gedrückt halten, werden Sie feststellen, dass die Wiederholfunktion dieser Schaltflächen ganz ähnlich arbeitet wie die einer Bildlaufleiste. Sie ähnelt also der Wiederholungsfunktion der Tastatur, die auf Englisch als »Typematic« bezeichnet wird. Daher beschloss ich, eine Schaltfläche, die dasselbe Verhalten zeigt, ClickmaticButton-Steuerelement zu taufen. Dies ist die erste Datei im Projekt *NumericScan*:

ClickmaticButton.cs

```
//-----------------------------------------------
// ClickmaticButton.cs (c) 2005 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Petzold.ProgrammingWindowsForms
{
    class ClickmaticButton : Button
    {
        Timer tmr = new Timer();
        int iDelay = 250 * (1 + SystemInformation.KeyboardDelay);
        int iSpeed = 405 - 12 * SystemInformation.KeyboardSpeed;

        protected override void OnMouseDown(MouseEventArgs args)
        {
            base.OnMouseDown(args);

            if ((args.Button & MouseButtons.Left) != 0)
            {
                tmr.Interval = iDelay;
                tmr.Tick += TimerOnTick;
                tmr.Start();
            }
        }
        void TimerOnTick(object objSrc, EventArgs args)
        {
            OnClick(EventArgs.Empty);
            tmr.Interval = iSpeed;
        }
        protected override void OnMouseMove(MouseEventArgs args)
        {
            base.OnMouseMove(args);
            tmr.Enabled = Capture & ClientRectangle.Contains(args.Location);
        }
        protected override void OnMouseUp(MouseEventArgs args)
        {
            base.OnMouseUp(args);
            tmr.Stop();
        }
    }
}
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Wie Sie sehen, ist diese Klasse in einem Namespace definiert. Wenn Sie DLLs erstellen, ist es wichtig, die Klassen mit einem Namespace zu definieren, damit es in den Programmen, die diese DLL benutzen, niemals Konflikte mit Klassennamen gibt. Ich habe mich bei der Auswahl dieses Namespaces in etwa an das übliche Verfahren gehalten: zuerst der Firmenname, dann der Produktname.

Unabhängig vom Namespace ist diese konkrete Klasse von außerhalb der DLL *nicht* sichtbar, weil die Klassendefinition nicht das Schlüsselwort `public` enthält. Wir haben uns bisher kaum darum gekümmert, ob wir Klassen öffentlich machen sollen. Das wird eigentlich erst wichtig, wenn Sie Klassen in eine DLL legen.

Die Klasse ist relativ simpel: Sie ist von `Button` abgeleitet und überschreibt die Methode `OnMouseDown`, um sich über Mausklicks benachrichtigen zu lassen. Durch den Aufruf der Methode in der Basisklasse wird der normale Aufruf von `OnClick` durchgeführt, worauf das `Click`-Ereignis ausgelöst wird. `ClickmaticButton` startet in `OnMouseDown` als Nächstes einen Timer. Der Ereignishandler für den Timer ruft `OnClick` weitere Male auf, damit die `Click`-Ereignisse wiederholt ausgelöst werden. Falls die Maustaste gedrückt ist, wenn sich der Mauszeiger von der Schaltfläche herunterbewegt, sollte der Timer vorerst angehalten werden. Der Timer sollte auch angehalten werden, wenn die Maustaste losgelassen wird.

Eine Zeitlang rätselte ich, wie sinnvolle `Interval`-Einstellungen für den Timer aussehen sollten. Wenn die Schaltfläche angeklickt wird, sollte es erst eine kleine Wartezeit geben, bevor die Wiederholungsfunktion loslegt. Danach sollte der Zeitabstand kürzer sein. Glücklicherweise entdeckte ich einige Erweiterungen in der Klasse `SystemInformation`, die im .NET Framework 2.0 neu hinzugekommen sind. `SystemInformation.KeyboardDelay` ist definiert als »Die Verzögerung für die Tastaturwiederholung, von 0 (etwa 250 Millisekunden Verzögerung) bis 3 (etwa 1 Sekunde Verzögerung)«. `SystemInformation.KeyboardSpeed` ist definiert als »Die Tastaturwiederholungsgeschwindigkeit, von 0 (etwa 2,5 Wiederholungen pro Sekunde) bis 31 (etwa 30 Wiederholungen pro Sekunde)«. Diese Werte funktionierten ganz hervorragend.

Die Klasse `ArrowButton` ist von `ClickmaticButton` abgeleitet und überschreibt die Methode `OnPaint`, um mithilfe der Methode `ControlPaint.DrawScrollButton` Pfeilsymbole zu zeichnen. Die Richtung des Pfeils wird durch eine öffentliche Eigenschaft festgelegt.

ArrowButton.cs

```
//-----------------------------------------
// ArrowButton.cs (c) 2005 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Petzold.ProgrammingWindowsForms
{
    class ArrowButton : ClickmaticButton
    {
        ScrollButton scrbtn = ScrollButton.Right;

        public ArrowButton()
        {
            SetStyle(ControlStyles.Selectable, false);
        }
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        public ScrollButton ScrollButton
        {
            set
            {
                scrbtn = value;
                Invalidate();
            }
            get { return scrbtn; }
        }
        protected override void OnPaint(PaintEventArgs args)
        {
            Graphics grfx = args.Graphics;
            ControlPaint.DrawScrollButton(grfx, ClientRectangle, scrbtn,
                !Enabled ? ButtonState.Inactive :
                    (Capture & ClientRectangle.Contains(
                                    PointToClient(MousePosition))) ?
                ButtonState.Pushed : ButtonState.Normal);
        }
        protected override void OnMouseCaptureChanged(EventArgs args)
        {
            base.OnMouseCaptureChanged(args);
            Invalidate();
        }
    }
}
```

Die Klasse `ArrowButton` hat einen Konstruktor, der das Flag `ControlStyles.Selectable` auf `false` setzt. Wenn die Schaltflächen im `NumericScan`-Steuerelement benutzt werden, sollen die Schaltflächen nicht ausgewählt werden können, das heißt, sie sollen nicht in der Lage sein, den Eingabefokus zu erhalten. Der Eingabefokus sollte im Textfeld bleiben.

Hier die öffentliche Klasse `NumericScan`, die von `UserControl` abgeleitet ist. Sie erstellt ein Steuerelement aus einem `TextBox`-Steuerelement und zwei `ArrowButton`-Steuerelementen.

NumericScan.cs

```
//-----------------------------------------
// NumericScan.cs (c) 2005 by Charles Petzold
//-----------------------------------------
using System;
using System.ComponentModel;
using System.Drawing;
using System.Reflection;
using System.Windows.Forms;

[assembly: AssemblyTitle("NumericScan")]
[assembly: AssemblyDescription("NumericScan Control")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("www.charlespetzold.com")]
[assembly: AssemblyProduct("NumericScan")]
[assembly: AssemblyCopyright("(c) Charles Petzold, 2005")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyVersion("1.0.*")]
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
namespace Petzold.ProgrammingWindowsForms
{
    [DefaultEvent("ValueChanged")]
    public class NumericScan : UserControl
    {
        public event EventHandler ValueChanged;

        TextBox txtbox;
        ArrowButton btn1, btn2;

        // Diese privaten Felder haben entsprechende öffentliche Eigenschaften.
        int iDecimalPlaces = 0;
        decimal mValue = 0;
        decimal mIncrement = 1;
        decimal mMinimum = 0;
        decimal mMaximum = 100;

        public NumericScan()
        {
            txtbox = new TextBox();
            txtbox.Parent = this;
            txtbox.TextAlign = HorizontalAlignment.Right;
            txtbox.Text = ValueToText(mValue);
            txtbox.TextChanged += TextBoxOnTextChanged;
            txtbox.KeyDown += TextBoxOnKeyDown;

            btn1 = new ArrowButton();
            btn1.Parent = this;
            btn1.Text = "btn1";
            btn1.ScrollButton = ScrollButton.Left;
            btn1.Click += ButtonOnClick;

            btn2 = new ArrowButton();
            btn2.Parent = this;
            btn2.Text = "btn2";
            btn2.ScrollButton = ScrollButton.Right;
            btn2.Click += ButtonOnClick;

            Width = 4 * Font.Height;
            Height = txtbox.PreferredHeight +
                                SystemInformation.HorizontalScrollBarHeight;
        }
        string ValueToText(decimal mValue)
        {
            return mValue.ToString("F" + DecimalPlaces);
        }

        [Category("Data"), Description("Im Steuerelement angezeigter Wert")]
        public decimal Value
        {
            set
            {
                txtbox.Text = ValueToText(mValue = value);
            }
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        get
        {
            return mValue;
        }
    }

    [Category("Data"),
    Description("Inkrement- oder Dekrementwert beim Anklicken einer Schaltfläche")]
    public decimal Increment
    {
        set { mIncrement = value; }
        get { return mIncrement; }
    }

    [Category("Data"), Description("Minimalwert")]
    public decimal Minimum
    {
        set
        {
            if ((mMinimum = value) > Value)
                Value = mMinimum;
        }
        get { return mMinimum; }
    }

    [Category("Data"), Description("Maximalwert")]
    public decimal Maximum
    {
        set
        {
            if ((mMaximum = value) < Value)
                Value = mMaximum;
        }
        get { return mMaximum; }
    }

    [Category("Data"), Description("Zahl der angezeigten Nachkommastellen")]
    public int DecimalPlaces
    {
        set { iDecimalPlaces = value; }
        get { return iDecimalPlaces; }
    }
    public override Size GetPreferredSize(Size szProposed)
    {
        return new Size(4 * Font.Height, txtbox.PreferredHeight +
                            SystemInformation.HorizontalScrollBarHeight);
    }
    protected override void OnResize(EventArgs args)
    {
        base.OnResize(args);

        txtbox.Height = txtbox.PreferredHeight;
        txtbox.Width = Width;
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
            btn1.Location = new Point(0, txtbox.Height);
            btn2.Location = new Point(Width / 2, txtbox.Height);
            btn1.Size = btn2.Size = new Size(Width / 2, Height - txtbox.Height);
        }
        void TextBoxOnTextChanged(object objSrc, EventArgs args)
        {
            if (txtbox.Text.Length == 0)
                return;

            try
            {
                mValue = Decimal.Parse(txtbox.Text);
            }
            catch
            {
            }
            txtbox.Text = ValueToText(mValue);
        }
        void TextBoxOnKeyDown(object objSrc, KeyEventArgs args)
        {
            switch (args.KeyCode)
            {
                case Keys.Enter:
                    OnValueChanged(EventArgs.Empty);
                    break;
            }
        }
        void ButtonOnClick(object objSrc, EventArgs args)
        {
            ArrowButton btn = objSrc as ArrowButton;
            decimal mNewValue = Value;

            if (btn == btn1)
                if ((mNewValue -= Increment) < Minimum)
                    return;

            if (btn == btn2)
                if ((mNewValue += Increment) > Maximum)
                    return;

            Value = mNewValue;
            OnValueChanged(EventArgs.Empty);
        }
        protected override void OnLeave(EventArgs args)
        {
            base.OnLeave(args);
            OnValueChanged(EventArgs.Empty);
        }
        protected virtual void OnValueChanged(EventArgs args)
        {
            Value = Math.Max(Minimum, Value);
            Value = Math.Min(Maximum, Value);
            Value = Decimal.Round(Value, DecimalPlaces);
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
            if (ValueChanged != null)
                ValueChanged(this, args);
        }
    }
}
```

Alles, was in der Datei in eckigen Klammern steht, ist ein Attribut. Die Attribute am Anfang der Quellcodedatei habe ich in Kapitel 1 beschrieben. Die anderen sind im Namespace `System.ComponentModel` definiert. Vor jeder öffentlichen Eigenschaft stehen die Attribute `Category` und `Description`. Diese Attribute werden vom `PropertyGrid`-Steuerelement ausgewertet, um zusammengehörige Eigenschaften zusammenzufassen und mit einer kurzen Beschreibung zu versehen. Unmittelbar vor der Klassendefinition steht ein `DefaultEvent`-Attribut. Visual Studio benutzt dieses Attribut im Designer, um zu ermitteln, welches Ereignis benutzt werden soll, wenn Sie doppelt auf ein Steuerelement klicken, um einen Ereignishandler einzurichten.

Beachten Sie außerdem, dass die Klasse als `public` definiert ist. Daher ist sie von außerhalb der DLL sichtbar. Das erste Member, das in der Klasse definiert ist, ist das öffentliche Ereignis `ValueChanged`.

`NumericScan` kombiniert auf seiner Oberfläche ein `TextBox`-Steuerelement und zwei `ArrowButton`-Steuerelemente. Wie bei `NumericUpDown` stellt das Steuerelement öffentliche Eigenschaften namens `Value`, `Minimum`, `Maximum`, `Increment` und `DecimalPlaces` bereit. Es ist möglich, eine sorgfältigere Konsistenzprüfung zu implementieren, als ich hier zeige. (Das `NumericUpDown`-Steuerelement aus .NET Framework löst einige Ausnahmen auf, falls zum Beispiel ein Programm in `Value` einen Wert einträgt, der außerhalb des Bereichs von `Minimum` bis `Maximum` liegt.) Was Sie aber unbedingt vermeiden sollten, sind Konsistenzprüfungen, die fehlschlagen, falls ein Programm die Eigenschaften in einer bestimmten Reihenfolge setzt. Zum Beispiel sind die Standardwerte für die Eigenschaften `Minimum` und `Maximum` 0 beziehungsweise 100. Ein Programm kann diese beiden Eigenschaften folgendermaßen ändern:

```
numscan.Minimum = 200;
numscan.Maximum = 300;
```

Falls das Steuerelement jedes Mal eine Ausnahme auslöst, wenn `Minimum` kleiner ist als `Maximum`, würde die erste Anweisung fehlschlagen. Und das ist wirklich nicht sinnvoll.

Der Rest der Klasse bearbeitet zum größten Teil die Ereignisse der `TextBox`- und `ArrowButton`-Steuerelemente. Jedes Mal, wenn sich der Text ändert, stellt der Ereignishandler `TextBoxOnTextChanged` fest, ob es sich noch um eine Zahl handelt. Falls nicht, ändert er den Text auf den vorherigen Wert zurück. Dieser Ereignishandler versucht nicht, die Grenzen der Minimal- und Maximalwerte zu erzwingen. (Das tut auch das `NumericUpDown`-Steuerelement nicht. Sie können in diesem Steuerelement jede beliebige Zahl eintippen.)

Wie beim Steuerelement `NumericUpDown` sollte immer das Ereignis `ValueChanged` ausgelöst werden, wenn der Wert durch die Schaltflächen geändert wird, wenn der Benutzer die EINGABETASTE drückt oder wenn das Steuerelement den Eingabefokus verliert. An dieser Stelle wird sichergestellt, dass die Grenzen der Minimal- und Maximalwerte eingehalten werden.

Damit das Steuerelement erfährt, wenn die EINGABETASTE gedrückt wird, installiert das Steuerelement einen `KeyDown`-Ereignishandler für das `TextBox`-Steuerelement. In diesem Moment erkannte ich die Schwachstelle in meinem Konzept mit den nach links und rechts zeigenden Pfeilschaltflächen: Wenn der Benutzer die entsprechenden Pfeiltasten auf der Tastatur drückt, sollte dies dieselbe Wirkung haben wie das Anklicken der Schaltflächen. Aber diese Pfeiltasten

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

werden schon benutzt, um den Cursor im `TextBox`-Steuerelement zu verschieben. Vielleicht ist es doch nicht so dumm, das Drehfeld mit Schaltflächen zu implementieren, deren Pfeile nach oben beziehungsweise unten weisen!

Ganz unten in der Klasse finden Sie die Methode `OnValueChanged`. Sie ist als `virtual` definiert, sodass Programme, die eine Klasse von `NumericScan` ableiten wollen, die Methode problemlos überschreiben können. Die Methode stellt sicher, dass die Grenzen der Minimal- und Maximalwerte eingehalten werden, rundet den Wert auf die gewünschte Zahl von Nachkommastellen und löst das Ereignis `ValueChanged` aus.

Ich habe weiter oben erwähnt, dass die Projektmappe *NumericScan* zwei Projekte umfasst: *NumericScan*, das die Datei *NumericScan.dll* erstellt, und *TestProgram*, das *TestProgram.exe* aus der folgenden Quellcodedatei erstellt.

TestProgram.cs

```
//-----------------------------------------
// TestProgram.cs (c) 2005 by Charles Petzold
//-----------------------------------------
using Petzold.ProgrammingWindowsForms;
using System;
using System.Drawing;
using System.Windows.Forms;

class TestProgram : Form
{
    Label lbl;
    NumericScan numscan1, numscan2;

    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new TestProgram());
    }
    public TestProgram()
    {
        Text = "Testprogramm";

        FlowLayoutPanel pnl = new FlowLayoutPanel();
        pnl.Parent = this;
        pnl.Dock = DockStyle.Fill;

        numscan1 = new NumericScan();
        numscan1.Parent = pnl;
        numscan1.AutoSize = true;
        numscan1.ValueChanged += NumericScanOnValueChanged;

        numscan2 = new NumericScan();
        numscan2.Parent = pnl;
        numscan2.AutoSize = true;
        numscan2.ValueChanged += NumericScanOnValueChanged;
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        lbl = new Label();
        lbl.Parent = pnl;
        lbl.AutoSize = true;
    }
    void NumericScanOnValueChanged(object objSrc, EventArgs args)
    {
        lbl.Text = "Erste: " + numscan1.Value + ", Zweite: " + numscan2.Value;
    }
}
```

Das Testprogramm besteht aus zwei `NumericScan`-Steuerelementen, für die es Ereignishandler installiert, und einem `Label`-Steuerelement, das die beiden Werte anzeigt.

Wenn Sie die Verweise für das Projekt *TestProgram* definieren, brauchen Sie die üblichen DLLs *System*, *System.Drawing* und *System.Windows.Forms*, aber hier müssen Sie zusätzlich *NumericScan.dll* hinzufügen. Klicken Sie im Dialogfeld *Verweis hinzufügen* auf die Registerkarte *Projekte* und wählen Sie *NumericScan* aus. Wie Sie sehen, enthält *TestProgram.cs* auch eine `using`-Direktive für den Namespace `Petzold.ProgrammingWindowsForms`.

Sie können dieses Steuerelement zur Toolbox von Visual Studio hinzufügen, indem Sie erst mit der rechten Maustaste auf eine der Rubriken in der Toolbox klicken und dann den Befehl *Registerkarte hinzufügen* wählen. Sie können die neue Registerkarte zum Beispiel *Weitere Steuerelemente* nennen. Klicken Sie mit der rechten Maustaste auf die neue Registerkarte und wählen Sie den Befehl *Elemente auswählen*. Das Dialogfeld *Toolboxelemente auswählen* hat eine *Durchsuchen*-Schaltfläche, mit der Sie die Datei *NumericScan.dll* auswählen können.

Das folgende Programm fordert dem Steuerelement `NumericScan` etwas mehr ab. Der erste Teil des Programms ist eine Klasse, die von `TableLayoutPanel` abgeleitet ist. Sie zeigt sechs `NumericScan`-Steuerelemente an, in denen Sie die Werte für die sechs Felder eines .NET Framework-Matrizentransformationsobjekts einstellen können. Diese Klasse ähnelt dem Programm *MatrixElements* aus dem letzten Kapitel, sie ist aber allgemeiner gehalten. Das Bereichssteuerelement hat eine öffentliche Eigenschaft namens `Matrix`, über die den `NumericScan`-Steuerelementen ihr Wert zugewiesen und ihr Inhalt in Form eines `Matrix`-Objekts ausgelesen werden kann. Das Bereichssteuerelement definiert außerdem ein öffentliches Ereignis namens `Change`, das jedes Mal ausgelöst wird, wenn eines der `NumericScan`-Steuerelemente ein `ValueChanged`-Ereignis auslöst. Beachten Sie die erste `using`-Direktive für den Namespace des `NumericScan`-Steuerelements.

MatrixPanel.cs

```
//-----------------------------------------
// MatrixPanel.cs (c) 2005 by Charles Petzold
//-----------------------------------------
using Petzold.ProgrammingWindowsForms;
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
class MatrixPanel: TableLayoutPanel
{
    public event EventHandler Change;

    NumericScan[] numscan = new NumericScan[6];

    public Matrix Matrix
    {
        set
        {
            for (int i = 0; i < 6; i++)
                numscan[i].Value = (decimal)value.Elements[i];
        }
        get
        {
            return new Matrix((float)numscan[0].Value, (float)numscan[1].Value,
                              (float)numscan[2].Value, (float)numscan[3].Value,
                              (float)numscan[4].Value, (float)numscan[5].Value);
        }
    }
    public MatrixPanel()
    {
        AutoSize = true;
        Padding = new Padding(Font.Height);
        ColumnCount = 2;

        SuspendLayout();

        for (int i = 0; i < 6; i++)
        {
            Label lbl = new Label();
            lbl.Parent = this;
            lbl.AutoSize = true;
            lbl.Anchor = AnchorStyles.Left;
            lbl.Text = new string[] { "X-Streckung:", "Y-Scherung:", "X-Scherung:",
                                      "Y-Streckung:", "X-Verschiebung:",
                                      "Y-Verschiebung:" }[i];

            numscan[i] = new NumericScan();
            numscan[i].Parent = this;
            numscan[i].AutoSize = true;
            numscan[i].Anchor = AnchorStyles.Right;
            numscan[i].Minimum = -1000;
            numscan[i].Maximum = 1000;
            numscan[i].DecimalPlaces = 2;
            numscan[i].ValueChanged += NumericScanOnValueChanged;
        }
        ResumeLayout();

        Matrix = new Matrix();
    }
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
    void NumericScanOnValueChanged(object objSrc, EventArgs args)
    {
        OnChange(EventArgs.Empty);
    }
    protected virtual void OnChange(EventArgs args)
    {
        if (Change != null)
            Change(this, args);
    }
}
```

`TableLayoutPanel` ist von `Control` abgeleitet und diese neue Klasse ist wiederum von `TableLayoutPanel` abgeleitet. Ist die neue Klasse dann nicht ein benutzerdefiniertes Steuerelement? Sicher! Jede Klasse, die Sie direkt oder indirekt von `Control` ableiten, kann als benutzerdefiniertes Steuerelement behandelt werden. Wenn Sie Eigenschaften und Ereignisse zum Steuerelement hinzufügen, wird es sicherlich spezifischer und nützlicher, die Krönung ist es, wenn Sie das Steuerelement in anderen Anwendungen wieder verwenden.

Hier ein weiteres »benutzerdefiniertes Steuerelement«. Es ist von `Panel` abgeleitet. Es zeigt seine Eigenschaft `Text` an, nachdem es die Matrizentransformation eingestellt hat, die es aus seiner öffentlichen Eigenschaft `Transform` ausliest. Manche Matrizentransformationen lösen eine Ausnahme aus, wenn das `Graphics`-Objekt der Transformation zugewiesen wird. Bei einer ungültigen Matrizentransformation löst das `Graphics`-Objekt eine Ausnahme aus und das Bereichssteuerelement zeigt eine entsprechende Meldung an.

DisplayPanel.cs

```
//---------------------------------------------
// DisplayPanel.cs (c) 2005 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class DisplayPanel : Panel
{
    Matrix matx = new Matrix();

    public DisplayPanel()
    {
        ResizeRedraw = true;
    }
    public Matrix Transform
    {
        set
        {
            matx = value;
            Invalidate();
        }
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        get
        {
            return matx;
        }
    }
    protected override void OnPaint(PaintEventArgs args)
    {
        Graphics grfx = args.Graphics;
        Brush brsh = new SolidBrush(ForeColor);

        try
        {
            grfx.Transform = matx;
            grfx.DrawString(Text, Font, brsh, Point.Empty);
        }
        catch (Exception exc)
        {
            StringFormat strfmt = new StringFormat();
            strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;
            grfx.DrawString(exc.Message, Font, brsh, ClientRectangle, strfmt);
        }
        brsh.Dispose();
    }
}
```

Normalerweise zeigen Bereichssteuerelemente ihre `Text`-Eigenschaft nicht an, daher macht sich niemand die Mühe, der Eigenschaft `Text` eines Bereichssteuerelements einen Wert zuzuweisen. Ein Programm, das dieses `DisplayPanel`-Steuerelement benutzt, hat die Pflicht, seiner Eigenschaft `Text` einen gültigen Wert zuzuweisen, unter Umständen auch eine andere `Font`-Eigenschaft.

MatrixInteractive ist nicht der Name eines neuen Films (hoffe ich zumindest), sondern der eines Projekts, das *MatrixPanel.cs*, *DisplayPanel.cs* und die folgende Datei umfasst.

MatrixInteractive.cs

```
//-----------------------------------------------
// MatrixInteractive.cs (c) 2005 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class MatrixInteractive : Form
{
    MatrixPanel matxpnl;
    DisplayPanel dispppnl;

    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new MatrixInteractive());
    }
```

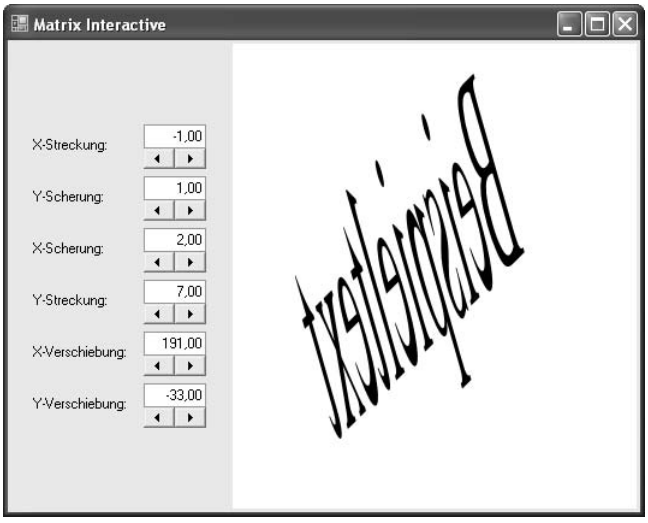

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
    public MatrixInteractive()
    {
        Text = "Matrix Interactive";

        TableLayoutPanel pnl = new TableLayoutPanel();
        pnl.Parent = this;
        pnl.Dock = DockStyle.Fill;
        pnl.ColumnCount = 2;

        matxpnl = new MatrixPanel();
        matxpnl.Parent = pnl;
        matxpnl.Anchor = AnchorStyles.Left | AnchorStyles.Right;
        matxpnl.Change += MatrixPanelOnChange;

        disppnl = new DisplayPanel();
        disppnl.Parent = pnl;
        disppnl.Dock = DockStyle.Fill;
        disppnl.BackColor = Color.White;
        disppnl.ForeColor = Color.Black;
        disppnl.Text = "Beispieltext";
        disppnl.Font = new Font(FontFamily.GenericSerif, 24);

        Width = 3 * matxpnl.Width;
        Height = 3 * matxpnl.Height / 2;
    }
    void MatrixPanelOnChange(object objSrc, EventArgs args)
    {
        disppnl.Transform = matxpnl.Matrix;
    }
}
```

Dieses Programm teilt seinen Clientbereich mithilfe eines `TableLayoutPanel`-Steuerelements in zwei Bereiche auf. Auf der linken Seite befindet sich vertikal zentriert das `MatrixPanel`-Steuerelement. Rechts ist ein `DisplayPanel`-Steuerelement, das einen weißen Hintergrund und einen schwarzen Vordergrund hat; die Eigenschaft `Text` hat den Wert »Beispieltext« und die Schrift ist 24 Punkt groß. Jedes Mal, wenn das `MatrixPanel`-Steuerelement ein `Change`-Ereignis auslöst, liest diese Klasse die Matrizentransformation aus dem `MatrixPanel`-Steuerelement und weist sie dem `DisplayPanel`-Steuerelement zu.

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Hier ein Beispiel, wie das Programm aussieht:



Über die Freuden von *AutoScroll*

Die Klasse `Form` und verschiedene `Panel`-Klassen bieten ein interessantes Feature, das nur selten benutzt wird, weil es als allgemeine Benutzeroberflächentechnik nicht recht hoffähig ist. Dieses Feature wird als »Autoscroll« (automatischer Bildlauf) bezeichnet. Sie aktivieren es, indem Sie einfach der Eigenschaft `AutoScroll` dieser Klassen den Wert `true` zuweisen. Wenn die Form oder das Bereichssteuerelement dann nicht groß genug ist, dass alle untergeordneten Steuerelemente darin Platz finden, tauchen von selbst Bildlaufleisten auf und der Benutzer kann zu den fehlenden Steuerelementen blättern. Das Autoscroll-Feature ist in `ScrollableControl` implementiert, es steht allen Steuerelementen zur Verfügung, die von `ScrollableControl` abgeleitet sind. Aber nicht jedes Steuerelement, das Bildlaufleisten hat, ist auch von `ScrollableControl` abgeleitet. `TextBox`, `ListBox` und `ScrollBar` sind *nicht* von `ScrollableControl` abgeleitet, `Form` und `Panel` schon.

Ich denke nicht, dass Autoscroll der beste Ansatz ist, eine Reihe von Steuerelementen in ein kleines Dialogfeld zu quetschen. Es kann aber eine große Hilfe für bestimmte benutzerdefinierte Steuerelemente sein. Nehmen wir zum Beispiel ein Steuerelement, das eine unendliche Zahl von Vorschaubildern (Thumbnail) für Bilddateien anzeigt. Wenn Sie diese Vorschaubilder einfach in ein Bereichssteuerelement einfüllen und die Autoscroll-Funktion aktivieren, wird die gesamte Bildlauflogik automatisch bereitgestellt.

Das nächste benutzerdefinierte Steuerelement, das ich Ihnen zeige, heißt `ImageScan`. Es zeigt für alle Bilddateien in einem bestimmten Festplattenverzeichnis Vorschaubilder an, die etwa 2,5 Zentimeter Kantenlänge haben. Alle Vorschaubilder werden in einer einzigen Zeile angezeigt, in der Sie hin und her blättern können. `ImageScan` ist von `FlowLayoutControl` abgeleitet, die Vorschaubilder sind als `PictureBox`-Steuerelement implementiert.

Nachdem ich begann, diese Steuerelemente zu kombinieren, tauchte aber ein prinzipielles Problem auf. Ich wollte mit den Tabulator- oder Pfeiltasten durch die Vorschaubilder navigieren. Das Problem war das `PictureBox`-Steuerelement selbst, das sich hartnäckig weigerte, den Eingabefokus entgegenzunehmen. Und ohne Eingabefokus kann es logischerweise auch keinerlei Rückmeldung

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

liefern, dass es den Eingabefokus erhalten hat. Der erste Schritt bestand also darin, ein Bildfeldsteuerelement zu entwickeln, das den Eingabefokus akzeptiert und eine einfache Tastaturbedienung unterstützt.

SelectablePictureBox.cs

```
//-------------------------------------------------
// SelectablePictureBox.cs (c) 2005 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SelectablePictureBox : PictureBox
{
    public SelectablePictureBox()
    {
        SetStyle(ControlStyles.Selectable, true);
        TabStop = true;
    }
    protected override void OnMouseDown(MouseEventArgs args)
    {
        base.OnMouseDown(args);
        Focus();
    }
    protected override void OnKeyPress(KeyPressEventArgs args)
    {
        if (args.KeyChar == '\r')
            OnClick(EventArgs.Empty);
        else
            base.OnKeyPress(args);
    }
    protected override void OnEnter(EventArgs args)
    {
        base.OnEnter(args);
        Invalidate();
    }
    protected override void OnLeave(EventArgs e)
    {
        base.OnLeave(e);
        Invalidate();
    }
    protected override void OnPaint(PaintEventArgs args)
    {
        base.OnPaint(args);

        if (Focused)
        {
            Graphics grfx = args.Graphics;
            grfx.DrawRectangle(new Pen(Brushes.Black, grfx.DpiX / 12),
                ClientRectangle);
        }
    }
}
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Der Konstruktor setzt das Flag `ControlStyles.Selectable` und weist der Eigenschaft `TabStop` den Wert `true` zu. Eine Überschreibung der Methode `OnPaint` lässt die Methode `OnPaint` der Basisklasse ihre Arbeit machen und gibt dann einen schwarzen Rand um das Steuerelement herum aus, falls dieses Steuerelement den Eingabefokus hat. Auch einige andere `On`-Methoden müssen erweitert werden. Wenn das Steuerelement angeklickt wird, weist sich das Steuerelement selbst den Eingabefokus zu. Wenn die EINGABETASTE gedrückt wird, simuliert das Steuerelement einen `OnClick`-Aufruf. Die Methoden `OnEnter` und `OnLeave` werden aufgerufen, wenn das Steuerelement den Eingabefokus erhält beziehungsweise verliert. Meine Überschreibungen erklären einfach die Oberfläche des Steuerelements für ungültig, sodass ein Aufruf von `OnPaint` generiert wird. So ist sichergestellt, dass das Steuerelement korrekt gezeichnet wird.

Hier das Steuerelement `ImageScan`, das von `FlowLayoutPanel` abgeleitet ist. Beachten Sie, wie der Konstruktor `WrapContents` auf `false` setzt und `AutoScroll` auf `true`.

ImageScan.cs

```
//---------------------------------------
// ImageScan.cs (c) 2005 by Charles Petzold
//---------------------------------------
using System;
using System.ComponentModel;    // für AsyncCompletedEventArgs
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class ImageScan : FlowLayoutPanel
{
    Size szImage;
    string strImageLocation;
    ToolTip tips = new ToolTip();

    public ImageScan()
    {
        FlowDirection = FlowDirection.LeftToRight;
        WrapContents = false;
        AutoScroll = true;

        // Size-Objekt mit einem Zoll Kantenlänge erstellen.
        Graphics grfx = CreateGraphics();
        szImage = new Size((int)grfx.DpiX, (int)grfx.DpiY); // 1"-Quadrat
        grfx.Dispose();

        Height = szImage.Height + Font.Height +
                        SystemInformation.HorizontalScrollBarHeight;
    }
    public string Directory
    {
        set
        {
            Controls.Clear();
            tips.RemoveAll();

            string[] astrFiles = System.IO.Directory.GetFiles(value, "*.*");
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
            foreach (string strFile in astrFiles)
            {
                PictureBox picbox = new SelectablePictureBox();
                picbox.Parent = this;
                picbox.Size = szImage;
                picbox.SizeMode = PictureBoxSizeMode.Zoom;
                picbox.Click += PictureBoxOnClick;
                picbox.LoadCompleted += PictureBoxOnLoadCompleted;
                picbox.LoadAsync(strFile);
            }
        }
    }
    public string SelectedImageFile
    {
        get
        {
            return strImageLocation;
        }
    }
    void PictureBoxOnClick(object objSrc, EventArgs args)
    {
        PictureBox picbox = objSrc as PictureBox;
        strImageLocation = picbox.ImageLocation;

        OnClick(args);
    }

    // Keine Click-Ereignisse auslösen, wenn der Benutzer den Bereich anklickt.
    protected override void OnMouseDown(MouseEventArgs args)
    {
    }
    void PictureBoxOnLoadCompleted(object objSrc, AsyncCompletedEventArgs args)
    {
        PictureBox picbox = objSrc as PictureBox;

        if (args.Error == null)
            tips.SetToolTip(picbox, Path.GetFileName(picbox.ImageLocation));
        else
            Controls.Remove(picbox);
    }
}
```

Das Steuerelement enthält im Wesentlichen eine Auflistung von `SelectablePictureBox`-Steuerelementen, die jeweils ein bestimmtes Bild aus einem Verzeichnis anzeigen. Daneben gibt es noch eine `ToolTip`-Komponente, die den Dateinamen als QuickInfo anzeigt. `ImageScan` implementiert eine öffentliche Eigenschaft namens `Directory`, die ein Festplattenverzeichnis enthält. Wenn dieser Eigenschaft ein Wert zugewiesen wird, löscht das Steuerelement erst die vorhandene Auflistung untergeordneter Steuerelemente und dann alle QuickInfos. Anschließend liest es alle Dateien im Verzeichnis aus und erstellt für jede Datei ein `SelectablePictureBox`-Objekt.

Aber halt: Dieses Steuerelement ist ausschließlich für *Bilddateien* gedacht, aber es wird für jede Datei ein `SelectablePictureBox`-Steuerelement erstellt, ob es nun eine Bilddatei ist oder nicht. Beachten Sie zwei Dinge bei diesen `SelectablePictureBox`-Steuerelementen: Erstens wird für das

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

LoadCompleted-Ereignis ein Ereignishandler installiert, und zweitens wird für die Datei die Methode LoadAsync von PictureBox aufgerufen. Diese Methode lädt die Datei in einem sekundären Thread und löst das LoadCompleted-Ereignis aus, wenn sie fertig ist. Als Argument in diesem Ereignis wird ein Objekt vom Typ AsyncCompletedEventArgs übergeben. Falls die Datei korrekt geladen werden konnte, hat die Eigenschaft Error dieses Objekts den Wert null. Konnte die Datei nicht korrekt geladen werden (und das passiert in diesem Programm oft, weil das Verzeichnis unter Umständen eine ganze Menge Dateien enthält, die keine Bilder sind), entfernt das Steuerelement die Datei aus der Auflistung untergeordneter Steuerelemente.

Und hier ein Programm, das ImageScan benutzt. Das Projekt *ImageDirectory* besteht aus *SelectablePictureBox.cs*, *ImageScan.cs* und dieser Datei.

ImageDirectory.cs

```
//-----------------------------------------------
// ImageDirectory.cs (c) 2005 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ImageDirectory: Form
{
    PictureBox picbox;
    ImageScan imgscan;
    Label lblDirectory;

    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new ImageDirectory());
    }
    public ImageDirectory()
    {
        Text = "Bilderverzeichnis";

        picbox = new PictureBox();
        picbox.Parent = this;
        picbox.Dock = DockStyle.Fill;
        picbox.SizeMode = PictureBoxSizeMode.Zoom;

        imgscan = new ImageScan();
        imgscan.Parent = this;
        imgscan.Dock = DockStyle.Top;
        imgscan.Click += ImageScanOnClick;

        FlowLayoutPanel pnl = new FlowLayoutPanel();
        pnl.Parent = this;
        pnl.AutoSize = true;
        pnl.Dock = DockStyle.Top;
```

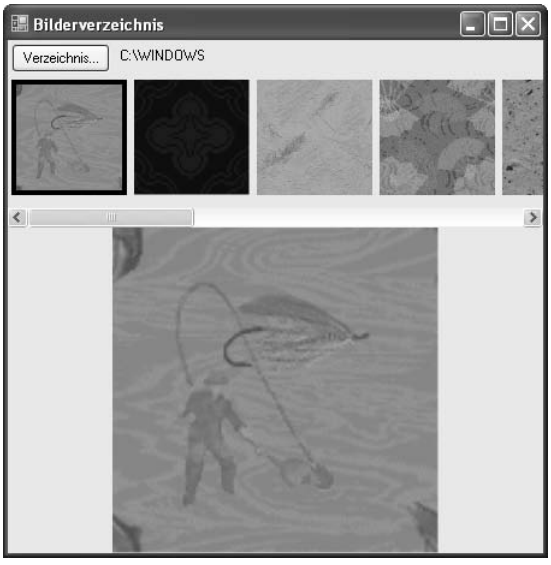

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        Button btn = new Button();
        btn.Parent = pnl;
        btn.AutoSize = true;
        btn.Anchor = AnchorStyles.Left;
        btn.Text = "Verzeichnis...";
        btn.Click += ButtonOnClick;

        lblDirectory = new Label();
        lblDirectory.Parent = pnl;
        lblDirectory.AutoSize = true;
        lblDirectory.Anchor = AnchorStyles.Right;

        // Initialisieren.
        imgscan.Directory = lblDirectory.Text =
            Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
    }
    void ButtonOnClick(object objSrc, EventArgs args)
    {
        FolderBrowserDialog dlg = new FolderBrowserDialog();
        dlg.SelectedPath = lblDirectory.Text;
        dlg.ShowNewFolderButton = false;

        if (dlg.ShowDialog() == DialogResult.OK)
            imgscan.Directory = lblDirectory.Text = dlg.SelectedPath;
    }
    void ImageScanOnClick(object objSrc, EventArgs args)
    {
        picbox.ImageLocation = imgscan.SelectedImageFile;
    }
}
```

Das Programm strukturiert eigentlich nur die Steuerelemente in Bereichssteuerelemente und verbindet sie miteinander. Ein `Button`-Steuerelement öffnet einen `FolderBrowserDialog`, in dem der Benutzer ein Verzeichnis auswählen kann. Ein `Label`-Steuerelement zeigt das Verzeichnis an. Das `ImageScan`-Steuerelement zeigt die Bilder in diesem Verzeichnis an, und das ausgewählte Bild belegt den übrigen Platz im Clientbereich des Programms.

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Das folgende Bild zeigt, wie die Form aussieht, nachdem ein Bild aus dem Verzeichnis *Windows* ausgewählt wurde:



Wenn Sie ein Verzeichnis auswählen, das eine Menge Dateien enthält, die keine Bilder sind, wird das Autoscroll-Feature von `ImageScan` praktisch sofort aktiv, weil all die `SelectablePictureBox`-Steuerelemente als untergeordnete Objekte hinzugefügt werden. Nachdem festgestellt wurde, dass diese Dateien keine Bilddateien sind, erfährt die Bildlaufleiste von diesen Änderungen und verschwindet unter Umständen wieder! Das ist ein wenig seltsam, aber es scheint zu funktionieren.

Steuerelemente von Grund auf programmieren

Es gibt natürlich Steuerelemente, die sich so stark von den Standardsteuerelementen unterscheiden, dass sie ihre Oberfläche weitgehend selbst zeichnen und auch die Benutzereingabe selbst verarbeiten müssen. Für solche Steuerelemente ist es sinnvoll, sie einfach von `Control` abzuleiten und mit dem Programmieren zu beginnen. (Am besten fangen Sie beizeiten damit an.) Als Beispiele präsentiere ich zwei Fallstudien: ein Lineal, wie es in Textverarbeitungsprogrammen wie Windows WordPad über dem Dokument angezeigt wird, und ein einfaches Farbauswahlgitter.

Ein interaktives Lineal

Werfen wir einen Blick auf Windows WordPad. WordPad ist um ein Steuerelement herum aufgebaut, das in der Windows-API den Namen `RichEdit`-Steuerelement trägt. In einem Windows Forms-Programm können Sie über die Klasse `RichTextBox` darauf zugreifen. Das `RichEdit`-Steuerelement speichert Dokumente im Rich Text Format (RTF), das es ermöglicht, Absatz- und Zeichenformatierungen auf unterschiedliche Teile des Dokuments anzuwenden.

Im Lineal (engl. ruler), das WordPad anzeigt, kann der Benutzer interaktiv den Einzug von links, den Einzug von rechts, den Einzug der ersten Zeile und die Tabulatorpositionen einstellen, die

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

für den momentan ausgewählten Absatz gültig sind. (Alternativ können Sie diese Formatierungselemente auch in den Dialogfeldern *Absatz* und *Tabstopps* ändern, die Sie über das WordPad-Menü *Format* öffnen können.)

Sehen wir und jetzt das Lineal in einem leistungsfähigeren Textverarbeitungsprogramm an. Wir wählen einfach eines aus, zum Beispiel Microsoft Office Word. Offensichtlich sieht das Lineal in Microsoft Word seinem Verwandten in WordPad recht ähnlich, aber Sie können damit die linken und rechten *Ränder* einstellen. Diese Ränder gelten für das gesamte Dokument, alternativ können Sie diese Einstellungen auch über das Dialogfeld *Seite einrichten* ändern, das Sie über das *Datei*-Menü öffnen können. Die Ränder sind eigentlich nur in Verbindung mit einer bestimmten Papierbreite sinnvoll. Die Ränder legen Bereiche auf der linken und rechten Seite des Blatts fest (normalerweise ca. 2 Zentimeter breit), in denen kein Text gedruckt wird. Die Breite des Texts auf dem Blatt ist die Seitenbreite minus linker und rechter Rand. In Word sind der linke und der rechte Einzug beim Absatz normalerweise 0, das heißt, dass die Absätze sich über die gesamte Breite zwischen dem linken und dem rechten Rand erstrecken. Der Einzug kann vergrößert werden, wenn der Absatz schmaler sein soll, oder er kann auf einen negativen Wert gesetzt werden, wenn Platz im Rand genutzt werden soll. Es ist auch möglich, für die erste Zeile in einem Absatz einen anderen Einzug festzulegen, sodass die Zeile entweder relativ zum übrigen Absatz weiter eingezogen wird oder links über den Absatz hinausragt (der so genannte »hängende« Einzug).

Es ist üblich, den linken Rand ab der linken Kante der Seite zu messen, der linke Einzug ist relativ zum linken Rand und der Einzug der ersten Zeile ist relativ zum linken Einzug: positiv für einen normalen Einzug, negativ für hängenden Einzug und 0, wenn die erste Zeile wie der übrige Absatz ausgerichtet ist. Der rechte Rand wird von der rechten Kante der Seite gemessen, und der rechte Einzug ab dem rechten Rand. Beide Zahlen sind normalerweise positiv oder 0.

Ursprünglich wollte ich ein `DocumentRuler`-Steuerelement (so habe ich es getauft) erstellen, das so ähnlich wie in Microsoft Word funktioniert, mit dem Sie Ränder und Einzüge festlegen können. Aber das `RichTextBox`-Steuerelement nutzt für Ränder ein viel primitiveres Konzept als Word. Das einzige in `RichTextBox`, was dem gewünschten Feature halbwegs nahe kommt, ist eine Eigenschaft namens `RightMargin`, mit der ein Programm die Breite des Textes, der vom Steuerelement angezeigt wird, in der Einheit Pixel angeben kann. (In der üblichen Terminologie ausgedrückt, bezieht sich diese Eigenschaft eher auf den Wert aus Seitenbreite minus linkem und rechtem Rand.) In der Standardeinstellung ist diese Eigenschaft 0, das heißt, dass die Breite des Textes, den das Steuerelement anzeigt, durch die Breite des Steuerelements selbst festgelegt wird. Eine Weile spielte ich mit der Idee, `RichTextBox` die üblichen Konzepte für Seitenbreite und Ränder aufzuzwingen, aber letztlich beschloss ich, einen simpleren Ansatz zu wählen. Mein Lineal ähnelt der WordPad-Implementierung, der Benutzer kann damit aber auch die Textbreite ändern.

`DocumentRuler` *muss* zwar nicht in Kombination mit einem `RichTextBox`-Steuerelement eingesetzt werden, es bietet aber ohnehin keine Fähigkeiten, die über `RichTextBox` hinausgehen. Sie müssten es an mehreren Stellen erweitern, wollten Sie es in einem ernsthaften Textverarbeitungsprogramm wie Word einsetzen.

Angesichts der in den USA üblichen Papierbreite von 8 1/2 Inches und Rändern von 1 1/4 Inches dachte ich mir, dass ein Standardwert von 6 Inches für die Eigenschaft `RightMargin` in etwa passen müsste. Es ist üblich, Ränder, Einzüge und Tabstopps in Einheiten wie Zentimeter oder Inch anzugeben, insbesondere wenn über dem Dokument ein Lineal angezeigt wird. Aber `RichTextBox` rechnet anders. Die Eigenschaft `RightMargin` und alle Eigenschaften für die Einzüge in `RichTextBox` haben als Maßeinheit Pixel.

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Ich beschloss, die Programmierschnittstelle des Lineals in der Einheit Inch zu entwickeln. Das Lineal bekommt also für die Einzüge eine Gruppe von Eigenschaften mit Namen wie `LeftIndent`, `RightIndent` und `FirstLineIndent`, und diese Eigenschaften sind `float`-Werte in der Maßeinheit Inch. Diese Entscheidung hatte zur Folge, dass im Linealsteuerelement und im Programm, das dieses Steuerelement benutzt, eine Menge Werte zwischen Inch und Pixel konvertiert werden mussten. Weil das Lineal auf dem Bildschirm angezeigt wird, hängt die Konvertierung zwischen Inch und Pixel natürlich von der Bildschirmauflösung des Benutzers ab, die aus den Eigenschaften `DpiX` und `DpiY` des `Graphics`-Objekts ermittelt werden kann. (Hinter den Kulissen gibt es sogar noch mehr Konvertierungen. Rich Text Format verwaltet die Werte in der Einheit »Twips«, das ist 1/20 eines Punkts beziehungsweise 1/1440 eines Inchs.)

Die Einstellungen in `DocumentRuler` basieren zwar auf der Bildschirmauflösung, ich hatte aber weniger Erfolg, als ich versuchte, andere Aspekte des Lineals wirklich geräteunabhängig zu machen. Insbesondere die kleinen Schieber, die die Einzüge darstellen, sind so klein und mit so geringen Toleranzen konstruiert, dass sie nicht gut mit Größenangaben klarkommen, die sich aus der Bildschirmauflösung ableiten.

Umwandlungen zwischen Inch und Pixel sind nicht die einzigen Konvertierungen, die bei dieser Aufgabe erforderlich waren. Die drei Eigenschaften von `RichTextBox`, die mit Einzügen zu tun haben, sind `SelectionIndent`, `SelectionRightIndent` und `SelectionHangingIndent`. Leider funktionieren die ersten beiden Eigenschaften etwas anders als die übliche Konvention bei der Absatzformatierung. `SelectionIndent` ist der Einzug der ersten Zeile von der linken Seite des Textfelds. `SelectionHangingIndent` ist der Einzug des übrigen Absatzes relativ zur ersten Zeile. Nehmen wir als Beispiel einen Absatz, der 100 Pixel von der linken Kante des Textfelds eingerückt ist und bei dem der Einzug der ersten Zeile weitere 50 Pixel beträgt: Bei diesem Absatz hat `SelectionIndent` den Wert 150 und `SelectionHangingIndent` den Wert –50. Ich beschloss, in `DocumentRuler` stattdessen die allgemein übliche Art von Einzügen zu implementieren. Ein Programm, das sowohl ein `RichTextBox`- als auch ein `DocumentRuler`-Steuerelement anlegt, muss zwischen den beiden hin und her konvertieren.

Jetzt sind wir so weit, dass wir uns etwas Code ansehen können. Das Steuerelement `DocumentRuler` implementiert nur ein einziges Ereignis, das ich schlicht `Change` genannt habe. Das Ereignis wird jedes Mal ausgelöst, wenn der Benutzer einen Rand, einen Einzug oder einen Tabstopp auf dem Lineal ändert. Ich wollte, dass das Programm, das dieses Lineal benutzt, darüber informiert wird, was sich geändert hat (zum Beispiel der linke Einzug). Daher muss das Ereignis diese Informationen liefern. Das Ereignis kann nicht auf dem Standarddelegaten `EventHandler` aufbauen, es ist ein benutzerdefinierter Delegat erforderlich. Erst wird eine Enumeration definiert, die Felder für alle Elemente enthält, die sich im Lineal einstellen lassen.

RulerProperty.cs

```
//-------------------------------------------
// RulerProperty.cs (c) 2005 by Charles Petzold
//-------------------------------------------
public enum RulerProperty
{   TextWidth,
    LeftIndent,
    RightIndent,
    FirstLineIndent,
    Tabs
}
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Im `Change`-Ereignis wird ein Objekt vom Typ `RulerEventArgs` übergeben. Diese Klasse ist von `EventArgs` abgeleitet, implementiert aber eine zusätzliche Eigenschaft vom Typ `RulerProperty`.

RulerEventArgs.cs

```
//-------------------------------------------
// RulerEventArgs.cs (c) 2005 by Charles Petzold
//-------------------------------------------
using System;

public class RulerEventArgs : EventArgs
{
    RulerProperty rlrprop;
    public RulerEventArgs(RulerProperty rlrprop)
    {
        this.rlrprop = rlrprop;
    }
    public RulerProperty RulerChange
    {
        get { return rlrprop; }
        set { rlrprop = value; }
    }
}
```

Die Klasse definiert außerdem einen Konstruktor, mit dem Sie ein neues `RulerEventArgs`-Objekt anlegen können, indem Sie ein Member von `RulerProperty` übergeben.

Wenn Sie eine neue Klasse definieren, die benutzt wird, um Informationen an einen Ereignishandler zu liefern, müssen Sie immer auch einen neuen Delegaten für den Ereignishandler definieren. Der Code ist simpel.

RulerEventHandler.cs

```
//-----------------------------------------------
// RulerEventHandler.cs (c) 2005 by Charles Petzold
//-----------------------------------------------
public delegate void RulerEventHandler(object objSrc, RulerEventArgs args);
```

Das Lineal enthält vier kleine Symbole, die linken Einzug, rechten Einzug, Einzug der ersten Zeile und Tabstopps anzeigen. Diese kleinen Elemente müssen natürlich gezeichnet werden, aber das Lineal muss auch reagieren, wenn der Benutzer eines davon mit der Maus anklickt. Ich habe diese Objekte in separaten Klassen implementiert, sie sind aber alle von einer abstrakten Klasse namens `RulerSlider` abgeleitet.

RulerSlider.cs

```
//-------------------------------------------
// RulerSlider.cs (c) 2005 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
abstract class RulerSlider
{
    // Private Felder.
    RulerProperty rlrprop;
    float fValue;
    int x, y;
    Bitmap bm;
    Region rgn;

    // Öffentliche Eigenschaften.
    public RulerProperty RulerProperty
    {
        get { return rlrprop; }
        set { rlrprop = value; }
    }
    public float Value
    {
        get { return fValue; }
        set { fValue = value; }
    }
    public int X
    {
        get { return x; }
        set { x = value; }
    }
    public virtual Rectangle Rectangle
    {
        get
        {
            return new Rectangle(X - bm.Width / 2, Y, bm.Width, bm.Height);
        }
    }

    // Geschützte Eigenschaft.
    protected int Y
    {
        get { return y; }
        set { y = value; }
    }

    // Öffentliche Methoden.
    public virtual void Draw(Graphics grfx)
    {
        grfx.DrawImage(bm, X - bm.Width / 2, Y);
    }
    public virtual bool HitTest(Point pt)
    {
        return rgn.IsVisible(pt.X - X + bm.Width / 2, pt.Y - Y);
    }
    protected void CreateBitmap(int cx, int cy, Point[] apt)
    {
        bm = new Bitmap(cx, cy);
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        GraphicsPath path = new GraphicsPath();
        path.AddLines(apt);
        rgn = new Region(path);

        Graphics grfx = Graphics.FromImage(bm);
        grfx.FillPolygon(Brushes.LightGray, apt);
        grfx.Clip = rgn;

        Shading(grfx, Pens.White, 1, apt);
        Shading(grfx, Pens.Gray, -1, apt);

        grfx.ResetClip();
        grfx.DrawPolygon(Pens.Black, apt);
        grfx.Dispose();
    }
    void Shading(Graphics grfx, Pen pn, int iOffset, Point[] apt)
    {
        grfx.TranslateTransform(iOffset, 0);
        grfx.DrawPolygon(pn, apt);
        grfx.TranslateTransform(-iOffset, iOffset);
        grfx.DrawPolygon(pn, apt);
        grfx.TranslateTransform(0, -iOffset);
    }
}
```

Wie Sie in Kürze sehen werden, weisen die verschiedenen Klassen, die von `RulerSlider` abgeleitet sind, der Eigenschaft `RulerProperty` selbst das passende Member der Enumeration `RulerProperty` zu. Sie tragen außerdem in die Eigenschaft `Y` eine feste Position der Schieber relativ zum oberen Rand des Steuerelements ein. Die Eigenschaft `X` ändert sich, wenn der Benutzer den Schieber von einer Stelle an eine andere bewegt.

Ich legte außerdem fest, dass diese Schieber eine Gleitkommaeigenschaft namens `Value` verwalten sollten, in der der aktuelle Wert in der Maßeinheit Inch gespeichert ist. Theoretisch lassen sich `Value` und `X` ineinander konvertieren, aber ich wollte den Gleitkommawert in Ruhe lassen, um Rundungsfehler während der Konvertierung zu vermeiden. Ich wollte Situationen vermeiden, in denen einer Eigenschaft wie `LeftIndent` ein Wert zugewiesen wird, die Eigenschaft dann aber einen etwas anderen Wert zurückliefert.

Die Methode `Draw` von `RulerSlider` zeichnet den Schieber an die `X`- und `Y`-Koordinaten. In `RulerSlider` ist `Draw` so implementiert, dass sie einfach eine Bitmap zeichnet. Diese Bitmap wird in der geschützten Methode `CreateBitmap` erstellt. Die Methode `CreateBitmap` bekommt die Breite und Höhe der Bitmap sowie ein Array mit `Point`-Objekten übergeben. Dieses Array definiert einen geschlossenen Bereich auf der Bitmap, der mit verschiedenen Farben und Schattierungen gefüllt wird. Dabei wird auch ein `Region`-Objekt erstellt, das in der Methode `HitTest` benutzt wird. Wenn eine Mauskoordinate an `HitTest` übergeben wird, liefert die Methode das Ergebnis `true` zurück, falls sich der Mauszeiger über dem Objekt befindet.

Die schreibgeschützte Eigenschaft `Rectangle` gibt ein Rechteck zurück, das die Bitmap umschließt, wie sie auf dem Lineal angezeigt wird. Das ist nützlich, um Bereiche des Lineals für ungültig zu erklären, wenn der Benutzer den Schieber bewegt.

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Hier die Klasse `RightIndent`, die von `RulerSlider` abgeleitet ist.

RightIndent.cs

```
//-----------------------------------------
// RightIndent.cs (c) 2005 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class RightIndent : RulerSlider
{
    public RightIndent()
    {
        RulerProperty = RulerProperty.RightIndent;
        Y = 9;
        CreateBitmap(9, 8, new Point[]
        {
            new Point(0, 7), new Point(0, 4), new Point(4, 0),
            new Point(8, 4), new Point(8, 7), new Point(0, 7)
        });
    }
}
```

Die Klasse weist lediglich den beiden Eigenschaften von `RulerSlider` Werte zu und ruft `CreateBitmap` auf, um ein Bild zu erzeugen, das einem kleinen Häuschen ähnelt. Die Klasse `LeftIndent` ist ähnlich, nur das Bild ist etwas komplizierter.

LeftIndent.cs

```
//----------------------------------------
// LeftIndent.cs (c) 2005 by Charles Petzold
//----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class LeftIndent : RulerSlider
{
    public LeftIndent()
    {
        RulerProperty = RulerProperty.LeftIndent;
        Y = 9;
        CreateBitmap(9, 14, new Point[]
        {
            new Point(0, 7), new Point(0, 4), new Point(4, 0), new Point(8, 4),
            new Point(8, 7), new Point(0, 7), new Point(0, 13), new Point(8, 13),
            new Point(8, 7), new Point(0, 7)
        });
    }
}
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Die Klasse `FirstLineIndent` ähnelt `RightIndent`, das Bild ist aber umgedreht und liegt am oberen Rand des Steuerelements.

FirstLineIndent.cs

```
//-----------------------------------------------
// FirstLineIndent.cs (c) 2005 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class FirstLineIndent : RulerSlider
{
    public FirstLineIndent()
    {
        RulerProperty = RulerProperty.FirstLineIndent;
        Y = 1;
        CreateBitmap(9, 8, new Point[]
        {
            new Point(0, 0), new Point(8, 0), new Point(8, 3),
            new Point(4, 7), new Point(0, 3), new Point(0, 0)
        });
    }
}
```

Das Tabstoppzeichen ist in der Klasse `Tab` implementiert. Da es die Form eines »L« hat, eignet sich dafür die einfache Linie am besten. Die Klasse überschreibt die Methoden `Draw` und `HitTest` sowie die Eigenschaft `Rectangle`.

Tab.cs

```
//----------------------------------
// Tab.cs (c) 2005 by Charles Petzold
//----------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class Tab : RulerSlider
{
    public Tab()
    {
        RulerProperty = RulerProperty.Tabs;
        Y = 9;
    }
    public override void Draw(Graphics grfx)
    {
        Pen pn = new Pen(Color.Black, 2);

        grfx.DrawLine(pn, X, Y, X, Y + 4);
        grfx.DrawLine(pn, X, Y + 4, X + 4, Y + 4);
    }
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
    public override bool HitTest(Point pt)
    {
        return pt.X >= X - 1 && pt.X <= X + 1 && pt.Y >= Y - 1 && pt.Y <= Y + 6;
    }
    public override Rectangle Rectangle
    {
        get
        {
            return new Rectangle(X - 1, Y - 1, 6, 6);
        }
    }
}
```

Die letzte Klasse, die von `RulerSlider` abgeleitet wird, ist `TextWidth`. Die Seitenbreite wird etwas anders geändert als die Einzüge. Die Zeichenlogik muss in der Haupt-`OnPaint`-Methode des Steuerelements abgehandelt werden, daher tut die Klasse nicht viel.

TextWidth.cs

```
//---------------------------------------
// TextWidth.cs (c) 2005 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class TextWidth : RulerSlider
{
    public TextWidth()
    {
        RulerProperty = RulerProperty.TextWidth;
    }
    public override void Draw(Graphics grfx)
    {
    }
    public override bool HitTest(Point pt)
    {
        return (pt.X >= X - 2) && (pt.X <= X + 2);
    }
    public override Rectangle Rectangle
    {
        get { return Rectangle.Empty; }
    }
}
```

Jetzt sind wir bereit für die Klasse `RulerDocument`. Damit Sie sich nicht erschlagen fühlen, habe ich die Klasse mithilfe des Schlüsselworts `partial` auf zwei Quellcodedateien verteilt. Der erste Teil enthält den Konstruktor und alle Eigenschaften.

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

DocumentRuler1.cs

```
//-------------------------------------------
// DocumentRuler1.cs (c) 2005 by Charles Petzold
//-------------------------------------------
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;

public partial class DocumentRuler : Control
{
    // Private Felder.
    int iLeftMargin;
    float fDpi;
    Control ctrlDocument;

    // RulerSlider-Objekte.
    LeftIndent rsLeftIndent = new LeftIndent();
    RightIndent rsRightIndent = new RightIndent();
    FirstLineIndent rsFirstIndent = new FirstLineIndent();
    TextWidth rsTextWidth = new TextWidth();
    List<RulerSlider> rsCollection = new List<RulerSlider>();

    // Konstruktor.
    public DocumentRuler()
    {
        Dock = DockStyle.Top;
        ResizeRedraw = true;
        TabStop = false;
        Height = 23;
        Font = new Font(Font.Name, 14, GraphicsUnit.Pixel);

        Graphics grfx = CreateGraphics();
        fDpi = grfx.DpiX;
        grfx.Dispose();

        rsCollection.Add(rsLeftIndent);
        rsCollection.Add(rsRightIndent);
        rsCollection.Add(rsFirstIndent);
        rsCollection.Add(rsTextWidth);
    }

    // Öffentliche Eigenschaften.
    public float TextWidth
    {
        get { return rsTextWidth.Value; }
        set
        {
            rsTextWidth.Value = value;
            CalculateDisplayOffsets();
        }
    }
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
    public float LeftIndent
    {
        get { return rsLeftIndent.Value; }
        set
        {
            rsLeftIndent.Value = value;
            CalculateDisplayOffsets();
        }
    }
    public float RightIndent
    {
        get { return rsRightIndent.Value; }
        set
        {
            rsRightIndent.Value = value;
            CalculateDisplayOffsets();
        }
    }
    public float FirstLineIndent
    {
        get { return rsFirstIndent.Value; }
        set
        {
            rsFirstIndent.Value = value;
            CalculateDisplayOffsets();
        }
    }
    public float[] Tabs
    {
        get
        {
            List<float> fTabs = new List<float>();

            foreach (RulerSlider rs in rsCollection)
                if (rs is Tab)
                    fTabs.Add(rs.Value);

            // RichTextBox will die Tabstopps numerisch sortiert.
            float[] afTabs = fTabs.ToArray();
            Array.Sort(afTabs);
            return afTabs;
        }
        set
        {
            // Zuerst Tabstopps löschen, die nicht im Array sind.
            List<Tab> rsTabsDelete = new List<Tab>();

            foreach (RulerSlider rs in rsCollection)
                if (rs is Tab && (Array.IndexOf(value, rs.Value) == -1))
                    rsTabsDelete.Add(rs as Tab);
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
            foreach (Tab tab in rsTabsDelete)
            {
                rsCollection.Remove(tab);
                Invalidate(tab.Rectangle);
            }

            // Dann Tabstopps hinzufügen, die nicht in rsCollection sind.
            foreach (float fTab in value)
            {
                bool bAdd = true;

                foreach (RulerSlider rs in rsCollection)
                    if (rs is Tab && rs.Value == fTab)
                        bAdd = false;

                if (bAdd)
                {
                    Tab tab = new Tab();
                    tab.Value = fTab;
                    tab.X = LeftMargin + InchesToPixels(fTab);
                    rsCollection.Add(tab);
                    Invalidate(tab.Rectangle);
                }
            }
        }
    }
    public int LeftMargin
    {
        get { return iLeftMargin; }
        set
        {
            iLeftMargin = value;
            CalculateDisplayOffsets();
        }
    }

    // Anzeigen einer Zeile, wenn die Schieber bewegt wurden.
    public Control DocumentControl
    {
        get { return ctrlDocument; }
        set { ctrlDocument = value; }
    }

    // Diese beiden Methoden berechnen X-Werte für die vier Schiebertypen
    //  (ohne Tabstopps).  Falls sich die X-Werte ändern, wird das Rechteck
    //  an der vorherigen Position und der neuen Position für ungültig erklärt.
    void CalculateDisplayOffsets()
    {
        CalculateDisplayOffsets2(rsTextWidth, LeftMargin +
                        InchesToPixels(rsTextWidth.Value));
        CalculateDisplayOffsets2(rsLeftIndent, LeftMargin +
                        InchesToPixels(rsLeftIndent.Value));
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        CalculateDisplayOffsets2(rsRightIndent, LeftMargin +
                        InchesToPixels(TextWidth - rsRightIndent.Value));
        CalculateDisplayOffsets2(rsFirstIndent, LeftMargin +
                        InchesToPixels(LeftIndent + rsFirstIndent.Value));
    }
    void CalculateDisplayOffsets2(RulerSlider rs, int xNew)
    {
        if (rs.X != xNew)
        {
            Invalidate(rs.Rectangle);
            rs.X = xNew;
            Invalidate(rs.Rectangle);
        }
    }
    float PixelsToInches(int i)
    {
        return i / fDpi;
    }
    int InchesToPixels(float f)
    {
        return (int)Math.Round(f * fDpi);
    }
}
```

Felder definieren die verschiedenen `RulerSlider`-Objekte, sodass `rsLeftIndent` ein Objekt vom Typ `LeftIndent` ist, `rsRightIndent` ein Objekt vom Typ `RightIndent` und so weiter. Außerdem wird eine `List`-Auflistung vom Typ `RulerSlider` erstellt, die den Namen `rsCollection` bekommt. Der Konstruktor macht alle diese `RulerSlider`-Objekte zu Mitgliedern der Auflistung. Die Auflistung muss schließlich auch noch mehrere Objekte vom Typ `Tab` umfassen.

Weiter unten in der Datei finden Sie eine Reihe öffentlicher Eigenschaften mit den Namen `TextWidth`, `LeftIndent` und so weiter. Diese Eigenschaften bieten Zugriff auf die Objekte `rsTextWidth`, `rsLeftIndent` und so weiter. Für jede dieser Eigenschaften berechnet ein `CalculateDisplayOffsets` die `X`-Eigenschaften, nachdem der Eigenschaft `Value` für das `RulerSlider`-Objekt ein Wert zugewiesen wurde.

Die Eigenschaft namens `Tabs` ist komplizierter als die anderen. Das hat den simplen Grund, dass sie mehrere Tabstopps verwalten muss (wie Sie schon am Plural `Tabs` erkennen können). Jeder Absatz kann seine eigenen Tabstopps haben. Wenn das Lineal Informationen zu einem anderen Teil des Dokuments anzeigt, müssen alle Tabstopps in `rsCollection` gelöscht und neue hinzugefügt werden. Mein ursprünglicher Code tat genau das. Die Folge war, dass die Symbole flackerten, weil sie ständig gelöscht und neu gezeichnet wurden, während ich im `RichTextBox`-Steuerelement Text eintippte. Der aktuelle Code vermeidet es daher, Tabstopps, deren Position sich nicht verändert hat, erst zu löschen und dann neu anzulegen.

Ihnen wird auch eine öffentliche Eigenschaft namens `LeftMargin` vom Typ `int` auffallen, die Zugriff auf das Feld `iLeftMargin` bietet. Und Sie können sehen, dass praktisch jede Berechnung im Programm diese Eigenschaft `LeftMargin` benutzt. Das Lineal braucht einen kleinen Abstand an der linken Seite, damit der Schieber für den linken Einzug vollständig angezeigt werden kann. Daher habe ich in dem Programm, das gleich die `RichTextBox`- und `DocumentRuler`-Steuerelemente kombiniert, dasselbe getan wie WordPad: Ich habe der Eigenschaft `ShowSelectionMargin` von `RichTextBox` den Wert `true` zugewiesen. Diese Eigenschaft schafft ein wenig Platz an der linken

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Seite, damit der Benutzer ganze Textzeilen auswählen kann. Aber wie viel Platz genau? Es schienen etwa 10 Pixel zu sein, daher trägt das folgende Programm diesen Wert in die Eigenschaft `LeftMargin` von `DocumentRuler` ein. Ich bin damit nicht sonderlich glücklich und fürchte, dass mich diese Entscheidung für den Rest meiner Tage verfolgen wird, aber eine bessere Lösung fiel mir nicht ein.

Die letzte öffentliche Eigenschaft von `DocumentRuler` ist `DocumentControl`. Sie verweist auf das Textverarbeitungs-Steuerelement, das mit dem `DocumentRuler`-Steuerelement verknüpft ist. `DocumentRuler` muss nur aus einem einzigen Grund auf dieses Steuerelement zugreifen: um eine senkrechte Linie durch das Steuerelement zu zeichnen, wenn der Benutzer einen der Schieber bewegt. Alle anderen Interaktionen zwischen dem `DocumentRuler`- und dem `RichTextBox`-Steuerelement werden außerhalb der beiden Steuerelemente abgewickelt.

Der erste Teil der Klasse `DocumentRuler` widmet sich der Ein- und Ausgabe für *Programme*, die das Steuerelement benutzen. Der zweite Teil der Klasse `DocumentRuler` betrifft die Ein- und Ausgabe für den *Benutzer*. Dieser Teil überschreibt die Methoden `OnPaint`, `OnMouseDown`, `OnMouseMove` und `OnMouseUp`.

DocumentRuler2.cs

```
//---------------------------------------------
// DocumentRuler2.cs (c) 2005 by Charles Petzold
//---------------------------------------------
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;

public partial class DocumentRuler : Control
{
    // Öffentliches Ereignis.
    public event RulerEventHandler Change;

    // Private Felder für das Ziehen mit der Maus.
    RulerSlider rsDragging;
    Point ptDown;
    int xOriginal;
    int xLineOverTextBox;

    // Die Methode OnPaint erledigt fast die ganze Zeichenarbeit.
    protected override void OnPaint(PaintEventArgs args)
    {
        Graphics grfx = args.Graphics;

        Rectangle rect = new Rectangle(LeftMargin, 0,
                                       rsTextWidth.X - LeftMargin, Height - 4);
        grfx.FillRectangle(Brushes.White, rect);
        ControlPaint.DrawBorder3D(grfx, rect);

        for (int i = 1; i < 8 * PixelsToInches(Width); i++)
        {
            int x = LeftMargin + InchesToPixels(i / 8f);
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
            if (i % 8 == 0)
            {
                StringFormat strfmt = new StringFormat();
                strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;
                grfx.DrawString((i / 8).ToString(), Font,
                                Brushes.Black, x, 9, strfmt);
            }
            else if (i % 4 == 0)
            {
                grfx.DrawLine(Pens.Black, x, 7, x, 10);
            }
            else
            {
                grfx.DrawLine(Pens.Black, x, 8, x, 9);
            }
        }
        // Alle Schieber anzeigen.
        foreach (RulerSlider rs in rsCollection)
            rs.Draw(grfx);
        return;
    }

    // OnMouseDown zum Bewegen der Schieber und zum Anlegen der Tabstopps.
    protected override void OnMouseDown(MouseEventArgs args)
    {
        // Ignorieren, falls es nicht die linke Taste ist.
        if ((args.Button & MouseButtons.Left) == 0)
            return;

        // In einer Schleife über alle Schieber gehen und feststellen, ob
        // einer angeklickt wurde.
        foreach (RulerSlider rs in rsCollection)
            if (rs.HitTest(args.Location))
            {
                rsDragging = rs;
                ptDown = args.Location;
                xOriginal = rsDragging.X;

                if (rsDragging is TextWidth)
                    Cursor.Current = Cursors.SizeWE;

                DrawReversibleLine(xLineOverTextBox = args.X);
                return;
            }
        // Falls kein vorhandener Tabstopp angeklickt wurde, neuen erstellen.
        rsDragging = new Tab();
        rsCollection.Add(rsDragging);
        ptDown = args.Location;
        xOriginal = rsDragging.X = ptDown.X;

        Invalidate(rsDragging.Rectangle);
        DrawReversibleLine(xLineOverTextBox = args.X);
        return;
    }
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
// OnMouseMove zum Bewegen der Schieber.
protected override void OnMouseMove(MouseEventArgs args)
{
    if (!Capture)   // Z.B. Maustaste nicht gedrückt.
    {
        // Cursor ändern, falls über dem Ende von TextWidth.
        if (!rsRightIndent.HitTest(args.Location) &&
             rsTextWidth.HitTest(args.Location))
                Cursor.Current = Cursors.SizeWE;
        return;
    }

    // Falls rsDragging ungleich null, sind wir in einer Schiebeaktion.
    if (rsDragging != null)
    {
        if (rsDragging is TextWidth)
            Cursor.Current = Cursors.SizeWE;

        int xNow = xOriginal - ptDown.X + args.X;

        // Schieber dürfen nicht über die Grenzen hinauswandern!
        if (rsDragging is Tab && (xNow < LeftMargin || xNow > rsTextWidth.X))
            return;

        if ((rsDragging == rsLeftIndent || rsDragging == rsFirstIndent) &&
                (xNow < LeftMargin || xNow > rsRightIndent.X))
            return;

        if (rsDragging == rsRightIndent && (xNow > rsTextWidth.X ||
                xNow < rsLeftIndent.X || xNow < rsFirstIndent.X))
            return;

        if (rsDragging == rsTextWidth && xNow < rsRightIndent.X)
            return;

        if (rsDragging == rsTextWidth)
        {
            Invalidate(new Rectangle(Math.Min(rsDragging.X, xOriginal) - 1, 0,
                            Math.Abs(rsDragging.X - xOriginal) + 2, Height));
            rsDragging.X = xNow;
        }
        else
        {
            // X-Eigenschaft des Schiebers aktualisieren und
            // alte und neue Position für ungültig erklären.
            Invalidate(rsDragging.Rectangle);
            rsDragging.X = xNow;
            Invalidate(rsDragging.Rectangle);
        }
        // Zeile über Textfeld verschieben.
        DrawReversibleLine(xLineOverTextBox);
        DrawReversibleLine(xLineOverTextBox = args.X);
    }
}
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
    // OnMouseUp ist die neue Position des Schiebers.
    protected override void OnMouseUp(MouseEventArgs args)
    {
        if (rsDragging != null)
        {
            // Neue Value-Eigenschaften berechnen und das Ereignis auslösen.
            if (rsDragging == rsLeftIndent || rsDragging == rsFirstIndent)
            {
                rsLeftIndent.Value = PixelsToInches(rsLeftIndent.X - LeftMargin);
                rsFirstIndent.Value =
                        PixelsToInches(rsFirstIndent.X - rsLeftIndent.X);
                OnChange(new RulerEventArgs(rsDragging.RulerProperty));
            }
            else if (rsDragging == rsRightIndent || rsDragging == rsTextWidth)
            {
                rsTextWidth.Value = PixelsToInches(rsTextWidth.X - LeftMargin);
                rsRightIndent.Value =
                        PixelsToInches(rsTextWidth.X - rsRightIndent.X);
                OnChange(new RulerEventArgs(rsTextWidth.RulerProperty));
                OnChange(new RulerEventArgs(rsRightIndent.RulerProperty));
            }

            else if (rsDragging is Tab)
            {
                rsDragging.Value = PixelsToInches(rsDragging.X - LeftMargin);
                OnChange(new RulerEventArgs(rsDragging.RulerProperty));
            }
            // Schiebeoperation beenden.
            rsDragging = null;
            DrawReversibleLine(xLineOverTextBox);
        }
    }

    // Linie in Bildschirmkoordinaten durch das Textfeld zeichnen.
    void DrawReversibleLine(int x)
    {
        if (ctrlDocument != null)
        {
            Point pt1 = ctrlDocument.PointToScreen(new Point(x, 0));
            Point pt2 = ctrlDocument.PointToScreen(
                                new Point(x, ctrlDocument.Height));
            ControlPaint.DrawReversibleLine(pt1, pt2, ctrlDocument.BackColor);
        }
    }

    // Methode OnChange löst das Change-Ereignis aus.
    protected virtual void OnChange(RulerEventArgs args)
    {
        if (Change != null)
            Change(this, args);
    }
}
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

Die Methode `OnPaint` ist erstaunlich simpel. Der Grund dafür sind zwei Codezeilen gegen Ende der Methode:

```
foreach (RulerSlider rs in rsCollection)
    rs.Draw(grfx);
```

Die `RulerSlider`-Objekte zeichnen sich selbst, daher braucht sich die Methode `OnPaint` nicht darum zu kümmern.

Auf ähnliche Weise überprüfen die `RulerSlider`-Objekte auch selbst, ob sie angeklickt wurden, und die Methode `OnMouseDown` stellt anhand dieser Methoden fest, ob der Benutzer einen der vorhandenen Schieber anklickt. Falls nicht, will der Benutzer einen neuen Tabstopp. In all diesen Fällen weist `OnMouseDown` verschiedenen privaten Feldern Werte zu, die beim Bewegen der Schieber helfen: `rsDragging` (der Schieber, der bewegt wird), `ptDown` (der Punkt, an dem die Maustaste ursprünglich gedrückt wurde) und `xOriginal` (die ursprüngliche Position des Schiebers). Die Methode `OnMouseMove` soll in erster Linie sicherstellen, dass die Schieber auf Positionen bewegt werden, die nicht erlaubt sind. Zum Beispiel darf die Eigenschaft `SelectionRightIndent` von `RichTextBox` nicht negativ sein, das heißt, dass der rechte Rand nicht links vom rechten Einzug stehen darf. `OnMouseUp` schließt die Schiebeoperation ab.

Und nun die Klasse, mit der Sie das Lineal endlich ansehen und ausprobieren können. Die Klasse `RichTextWithRuler` ist von `Form` abgeleitet und erstellt ein `RichTextBox`-Steuerelement sowie ein `DocumentRuler`-Steuerelement. Das Projekt *RichTextWithRuler* enthält diese Datei und alle anderen beschriebenen Dateien seit *RulerProperty.cs*.

RichTextWithRuler.cs

```
//-----------------------------------------------
// RichTextWithRuler.cs (c) 2005 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class RichTextWithRuler : Form
{
    DocumentRuler ruler;
    RichTextBox txtbox;
    float fDpi;

    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new RichTextWithRuler());
    }
    public RichTextWithRuler()
    {
        Text = "RichText mit Lineal";

        Graphics grfx = CreateGraphics();
        fDpi = grfx.DpiX;
        grfx.Dispose();
```

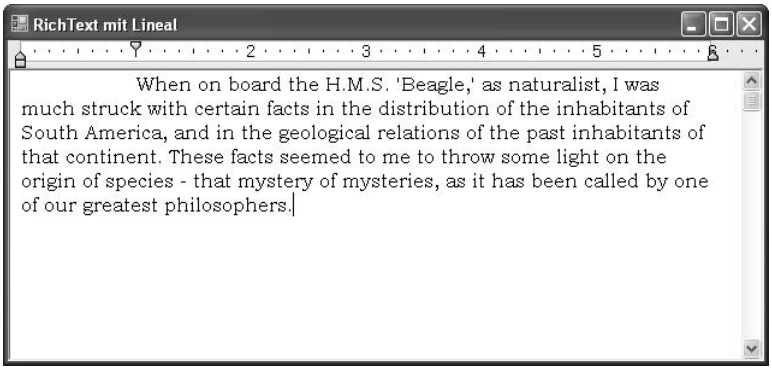

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        txtbox = new RichTextBox();
        txtbox.Parent = this;
        txtbox.AcceptsTab = true;
        txtbox.Dock = DockStyle.Fill;
        txtbox.RightMargin = InchesToPixels(6);
        txtbox.ShowSelectionMargin = true;
        txtbox.SelectionChanged += TextBoxOnSelectionChanged;

        ruler = new DocumentRuler();
        ruler.Parent = this;
        ruler.LeftMargin = 10;
        ruler.TextWidth = PixelsToInches(txtbox.RightMargin);
        ruler.DocumentControl = txtbox;
        ruler.Change += RulerOnChange;

        // Lineal mit Werten des Textfelds initialisieren.
        TextBoxOnSelectionChanged(txtbox, EventArgs.Empty);
    }
    void TextBoxOnSelectionChanged(object objSrc, EventArgs args)
    {
        ruler.LeftIndent = PixelsToInches(txtbox.SelectionIndent +
                                          txtbox.SelectionHangingIndent);
        ruler.RightIndent = PixelsToInches(txtbox.SelectionRightIndent);
        ruler.FirstLineIndent = PixelsToInches(-txtbox.SelectionHangingIndent);

        float[] fTabs = new float[txtbox.SelectionTabs.Length];

        for (int i = 0; i < txtbox.SelectionTabs.Length; i++)
            fTabs[i] = PixelsToInches(txtbox.SelectionTabs[i]);

        ruler.Tabs = fTabs;
    }
    void RulerOnChange(object objSrc, RulerEventArgs args)
    {
        switch (args.RulerChange)
        {
            case RulerProperty.TextWidth:
                txtbox.RightMargin = InchesToPixels(ruler.TextWidth);
                break;

            case RulerProperty.LeftIndent:
            case RulerProperty.FirstLineIndent:
                txtbox.SelectionIndent = InchesToPixels(ruler.LeftIndent +
                                                        ruler.FirstLineIndent);
                txtbox.SelectionHangingIndent =
                    InchesToPixels(-ruler.FirstLineIndent);
                break;

            case RulerProperty.RightIndent:
                txtbox.SelectionRightIndent = InchesToPixels(ruler.RightIndent);
                break;
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
            case RulerProperty.Tabs:
                int[] iTabs = new int[ruler.Tabs.Length];

                for (int i = 0; i < ruler.Tabs.Length; i++)
                    iTabs[i] = InchesToPixels(ruler.Tabs[i]);

                txtbox.SelectionTabs = iTabs;
                break;
        }
    }
    float PixelsToInches(int i)
    {
        return i / fDpi;
    }
    int InchesToPixels(float f)
    {
        return (int)Math.Round(f * fDpi);
    }
}
```

Wie angekündigt, enthält diese Datei eine Menge Konvertierungen zwischen Inch und Pixel. Das Programm muss die Schnittstelle zwischen `RichTextBox` und `DocumentRuler` bereitstellen, das passiert in erster Linie in Ereignishandlern für das `SelectionChanged`-Ereignis des Textfelds und das `Change`-Ereignis des Lineals. Hier das Programm mit linkem und rechtem Einzug 0 und einem Einzug für die erste Zeile von 1 Inch:



Auswählen einer Farbe

Es gibt bestimmt schon 6 Millionen Steuerelemente zum Auswählen einer Farbe, da kann eines mehr nicht schaden. `ColorGrid` zeigt ein Feld mit 40 Farben in Form eines Gitters an. Sie können natürlich eine der Farben anklicken, aber Sie können sich auch mit den Pfeiltasten der Tastatur durch die Farben bewegen. Dieses Steuerelement hat die aufwendigste Tastaturverarbeitung in diesem Kapitel.

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

ColorGrid.cs

```
//---------------------------------------
// ColorGrid.cs (c) 2005 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ColorGrid : Control
{
    // Anzahl der Farben horizontal and vertikal.
    const int xNum = 8;
    const int yNum = 5;

    // Die Farben
    Color[,] aclr = new Color[yNum, xNum]
    {
        { Color.Black, Color.Brown, Color.DarkGreen, Color.MidnightBlue,
          Color.Navy, Color.DarkBlue, Color.Indigo, Color.DimGray },

        { Color.DarkRed, Color.OrangeRed, Color.Olive, Color.Green,
          Color.Teal, Color.Blue, Color.SlateGray, Color.Gray },

        { Color.Red, Color.Orange, Color.YellowGreen, Color.SeaGreen,
          Color.Aqua, Color.LightBlue, Color.Violet, Color.DarkGray },

        { Color.Pink, Color.Gold, Color.Yellow, Color.Lime,
          Color.Turquoise, Color.SkyBlue, Color.Plum, Color.LightGray },

        { Color.LightPink, Color.Tan, Color.LightYellow, Color.LightGreen,
          Color.LightCyan, Color.LightSkyBlue, Color.Lavender, Color.White }
    };

    // Ausgewählte Farbe in privatem Feld.
    Color clrSelected = Color.Black;

    // Rechtecke zum Anzeigen der Farben und Ränder.
    Rectangle rectTotal, rectGray, rectBorder, rectColor;

    // Momentan durch Tastatur oder Maus hervorgehobene Koordinate.
    int xHighlight = -1;
    int yHighlight = -1;

    // Konstruktor.
    public ColorGrid()
    {
        AutoSize = true;

        // Bildschirmauflösung ermitteln.
        Graphics grfx = CreateGraphics();
        int xDpi = (int)grfx.DpiX;
        int yDpi = (int)grfx.DpiY;
        grfx.Dispose();
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        // Rechtecke für Farbanzeige berechnen.
        rectTotal = new Rectangle(0, 0, xDpi / 5, yDpi / 5);
        rectGray = Rectangle.Inflate(rectTotal, -xDpi / 72, -yDpi / 72);
        rectBorder = Rectangle.Inflate(rectGray, -xDpi / 48, -yDpi / 48);
        rectColor = Rectangle.Inflate(rectBorder, -xDpi / 72, -yDpi / 72);
    }

    // Eigenschaft SelectedColor: Zugriff auf das Feld clrSelected
    public Color SelectedColor
    {
        get
        {
            return clrSelected;
        }
        set
        {
            clrSelected = value;
            Invalidate();
        }
    }

    // Erforderlich für Autosizing.
    public override Size GetPreferredSize(Size sz)
    {
        return new Size(xNum * rectTotal.Width, yNum * rectTotal.Height);
    }

    // Alle Farben im Gitter zeichnen.
    protected override void OnPaint(PaintEventArgs args)
    {
        Graphics grfx = args.Graphics;

        for (int y = 0; y < yNum; y++)
            for (int x = 0; x < xNum; x++)
                DrawColor(grfx, x, y, false);
    }

    // Eine bestimmte Farbe zeichnen (grfx kann null sein).
    void DrawColor(Graphics grfx, int x, int y, bool bHighlight)
    {
        bool bDisposeGraphics = false;

        if (x < 0 || y < 0 || x >= xNum || y >= yNum)
            return;

        if (grfx == null)
        {
            grfx = CreateGraphics();
            bDisposeGraphics = true;
        }
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        // Feststellen, ob die Farbe momentan ausgewählt ist.
        bool bSelect = aclr[y, x].ToArgb() == SelectedColor.ToArgb();

        Brush br = (bHighlight | bSelect) ? SystemBrushes.HotTrack :
                        SystemBrushes.Menu;

        // Zeichnen der Rechtecke beginnen.
        Rectangle rect = rectTotal;
        rect.Offset(x * rectTotal.Width, y * rectTotal.Height);
        grfx.FillRectangle(br, rect);

        if (bHighlight || bSelect)
        {
            br = bHighlight ? SystemBrushes.ControlDark :
                              SystemBrushes.ControlLight;
            rect = rectGray;
            rect.Offset(x * rectTotal.Width, y * rectTotal.Height);
            grfx.FillRectangle(br, rect);
        }

        rect = rectBorder;
        rect.Offset(x * rectTotal.Width, y * rectTotal.Height);
        grfx.FillRectangle(SystemBrushes.ControlDark, rect);

        rect = rectColor;
        rect.Offset(x * rectTotal.Width, y * rectTotal.Height);
        grfx.FillRectangle(new SolidBrush(aclr[y, x]), rect);

        if (bDisposeGraphics)
            grfx.Dispose();
    }

    // Methoden für Mausbewegungen und -klicks.
    protected override void OnMouseEnter(EventArgs args)
    {
        xHighlight = -1;
        yHighlight = -1;
    }
    protected override void OnMouseMove(MouseEventArgs args)
    {
        int x = args.X / rectTotal.Width;
        int y = args.Y / rectTotal.Height;

        if (x != xHighlight || y != yHighlight)
        {
            DrawColor(null, xHighlight, yHighlight, false);
            DrawColor(null, x, y, true);

            xHighlight = x;
            yHighlight = y;
        }
    }
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
    protected override void OnMouseLeave(EventArgs args)
    {
        DrawColor(null, xHighlight, yHighlight, false);

        xHighlight = -1;
        yHighlight = -1;
    }
    protected override void OnMouseDown(MouseEventArgs args)
    {
        int x = args.X / rectTotal.Width;
        int y = args.Y / rectTotal.Height;
        SelectedColor = aclr[y, x];
        base.OnMouseDown(args);         // Generiert Click-Ereignis.
        Focus();
    }

    // Methoden für Tastaturbedienung.
    protected override void OnEnter(EventArgs args)
    {
        if (xHighlight < 0 || yHighlight < 0)
            for (yHighlight = 0; yHighlight < yNum; yHighlight++)
            {
                for (xHighlight = 0; xHighlight < xNum; xHighlight++)
                {
                    if (aclr[yHighlight, xHighlight].ToArgb() ==
                                    SelectedColor.ToArgb())
                        break;
                }
                if (xHighlight < xNum)
                    break;
            }

        if (xHighlight == xNum && yHighlight == yNum)
            xHighlight = yHighlight = 0;

        DrawColor(null, xHighlight, yHighlight, true);
    }
    protected override void OnLeave(EventArgs args)
    {
        DrawColor(null, xHighlight, yHighlight, false);
        xHighlight = yHighlight = -1;
    }
    protected override bool IsInputKey(Keys keyData)
    {
        return keyData == Keys.Home || keyData == Keys.End ||
               keyData == Keys.Up || keyData == Keys.Down ||
               keyData == Keys.Left || keyData == Keys.Right;
    }
    protected override void OnKeyDown(KeyEventArgs args)
    {
        DrawColor(null, xHighlight, yHighlight, false);
        int x = xHighlight, y = yHighlight;
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        switch (args.KeyCode)
        {
            case Keys.Home:
                x = y = 0;
                break;

            case Keys.End:
                x = xNum - 1;
                y = yNum - 1;
                break;

            case Keys.Right:
                if (++x == xNum)
                {
                    x = 0;
                    if (++y == yNum)
                    {
                        Parent.GetNextControl(this, true).Focus();
                    }
                }
                break;

            case Keys.Left:
                if (--x == -1)
                {
                    x = xNum - 1;
                    if (--y == -1)
                    {
                        Parent.GetNextControl(this, false).Focus();
                    }
                }
                break;

            case Keys.Down:
                if (++y == yNum)
                {
                    y = 0;
                    if (++x == xNum)
                    {
                        Parent.GetNextControl(this, true).Focus();
                    }
                }
                break;

            case Keys.Up:
                if (--y == -1)
                {
                    y = 0;
                    if (--x == -1)
                    {
                        Parent.GetNextControl(this, false).Focus();
                    }
                }
                break;
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
            case Keys.Enter:
            case Keys.Space:
                SelectedColor = aclr[y, x];
                OnClick(EventArgs.Empty);
                break;

            default:
                base.OnKeyDown(args);
                return;
        }
        DrawColor(null, x, y, true);

        xHighlight = x;
        yHighlight = y;
    }
}
```

Das Steuerelement definiert eine neue öffentliche Eigenschaft namens `SelectedColor` für die ausgewählte Farbe. Ein Programm, das dieses Steuerelement benutzt, wird über ein normales `Click`-Ereignis informiert, wenn sich `SelectedColor` geändert hat.

Der Konstruktor definiert vier Rechtecke, mit denen die einzelnen Farben im Gitter angezeigt werden. Die Abmessungen dieser Rechtecke werden allein aus der vertikalen und horizontalen Auflösung des Bildschirms berechnet. `ColorGrid` ist endlich mal ein Steuerelement, das nicht neu programmiert werden muss, wenn sich 300-dpi-Monitore am Markt verbreiten. Die Methode `OnPaint` ruft einfach `DrawColor` für jede der 40 Farben im Gitter auf. Die Methode `DrawColor` erledigt die eigentliche Arbeit. Die Zeichenlogik unterscheidet zwischen der *ausgewählten* Farbe, also der Farbe, die von der Eigenschaft `SelectedColor` geliefert wird, und der *hervorgehobenen* Farbe. Das hervorgehobene Steuerelement ändert sich, wenn der Mauszeiger über das Steuerelement bewegt wird oder wenn das Steuerelement den Eingabefokus hat und die Pfeiltasten gedrückt werden. Eine Farbe wird vom Status »hervorgehoben« in den Status »ausgewählt« verwandelt, wenn der Benutzer die Maustaste drückt oder die EINGABETASTE oder LEERTASTE betätigt.

Die hervorgehobene Farbe zu ändern, wenn der Mauszeiger bewegt wird, ist die Aufgabe der Überschreibungen `OnMouseEnter`, `OnMouseMove` und `OnMouseLeave`. Die Methode `OnMouseDown` trägt die neue ausgewählte Farbe fest und ruft die entsprechende Methode der Basisklasse auf, um ein `Click`-Ereignis auslösen zu lassen.

Die Tastaturverarbeitung ist aufwendiger. Zuerst einmal muss das Steuerelement feststellen, wann es den Eingabefokus erhält und wieder verliert. Falls sich der Mauszeiger nicht über dem Steuerelement befindet, wird eine Farbe nur dann hervorgehoben, wenn das Steuerelement den Eingabefokus hat. `ColorGrid` benutzt für diese Aufgabe die Methoden `OnEnter` und `OnLeave`.

Ein weiteres Problem: Viele der Tasten, die `ColorGrid` gern benutzen würde (insbesondere die Pfeiltasten), werden vom übergeordneten Steuerelement benutzt, um den Eingabefokus zwischen den untergeordneten Steuerelementen zu verschieben. `ColorGrid` muss `IsInputKey` überschreiben und für alle Tasten, die es allein benutzen möchte, den Wert `true` zurückgeben. Diese Tasten werden dann in der Methode `OnKeyDown` verarbeitet. Mit den Pfeiltasten kann sich der Benutzer durch die Zeilen und Spalten des Gitters bewegen, bis die Farbe in der linken oberen oder der rechten unteren Ecke erreicht ist. An dieser Stelle wird der Eingabefokus an das vorherige be-

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

ziehungsweise das nächste nebengeordnete Steuerelement übergeben. Beachten Sie auch, wie das Betätigen der EINGABE- und LEERTASTE die Auswahl ändert.

Das Projekt *ColorGridDemo* umfasst *ColorGrid.cs* und die nächste Datei.

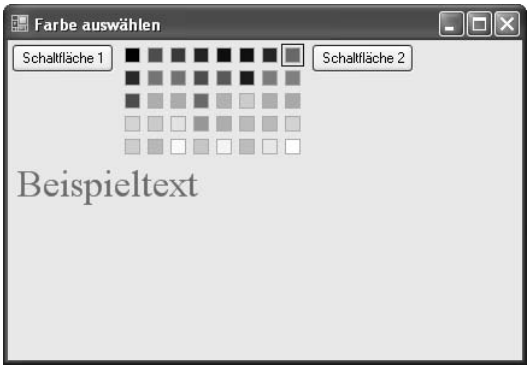
ColorGridDemo.cs

```
//---------------------------------------------
// ColorGridDemo.cs (c) 2005 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ColorGridDemo : Form
{
    Label lbl;

    [STAThread]
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new ColorGridDemo());
    }
    public ColorGridDemo()
    {
        Text = "Farbe auswählen";
        AutoSize = true;

        TableLayoutPanel table = new TableLayoutPanel();
        table.Parent = this;
        table.AutoSize = true;
        table.ColumnCount = 3;

        Button btn = new Button();
        btn.Parent = table;
        btn.AutoSize = true;
        btn.Text = "Schaltfläche 1";

        ColorGrid clrgrid = new ColorGrid();
        clrgrid.Parent = table;
        clrgrid.Click += ColorGridOnClick;

        btn = new Button();
        btn.Parent = table;
        btn.AutoSize = true;
        btn.Text = "Schaltfläche 2";

        lbl = new Label();
        lbl.Parent = table;
        lbl.AutoSize = true;
        lbl.Font = new Font("Times New Roman", 24);
        lbl.Text = "Beispieltext";
```


Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)

```
        table.SetColumnSpan(lbl, 3);
        clrgrid.SelectedColor = lbl.ForeColor;
    }
    void ColorGridOnClick(object objSrc, EventArgs args)
    {
        ColorGrid clrgrid = (ColorGrid) objSrc;
        lbl.ForeColor = clrgrid.SelectedColor;
    }
}
```

Die Klasse `ColorGridDemo` legt ein `ColorGrid`-Steuerelement an und platziert das Steuerelement zwischen zwei `Button`-Steuerelemente, damit Sie die Übergabe des Eingabefokus testen können. Wird eine neue Farbe ausgewählt, wird diese Farbe als Vordergrundfarbe für ein `Label`-Steuerelement benutzt:



Sollten Sie der Meinung sein, dass dieses Farbauswahlsteuerelement dem Steuerelement ähnelt, das Sie aus dem Menü *Format/Hintergrund* in einer Microsoft Office-Anwendung kennen, muss ich gestehen, dass dies die Quelle meiner Inspiration war. Das Steuerelement in ein Menü und eine Symbolleiste einzubauen, wird eine der Herausforderungen sein, der wir uns im nächsten Kapitel stellen.

Charles Petzold: Windows Forms-Programmierung mit Visual C# 2005. Microsoft Press 2006 (ISBN 3-86063-985-4)