Schritt- für- Schritt Anleitung MS 2010 C#

Prof. Dr. Bayerlein WS 2012/13 Version 4.6 vom 15. Nov. 2012

Inhalt

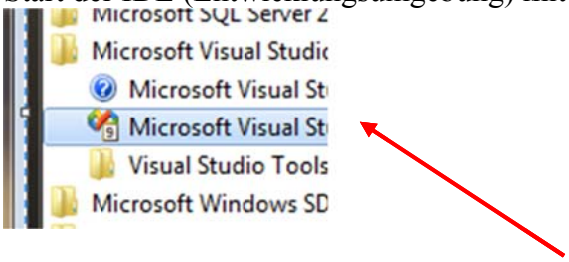
Literatur

[1]. Andreas Kühnel. Visual C# 2008 oder neuer. Galileo Computing.
[2]. Jürgen Bayer. Das C# 2008 Codebook oder neuer. Addison Wesley.
[3]. Frank Eller. Visual C# 2008 oder neuer. Addison Wesley.
[4]. Bernhard Volz. Einstieg in Visual C# 2008 oder neuer. Galileo Computing.

Kurs Intro

Alle Screenshots sind mit dem MS Studio 2008 IDE Visual C#, Prof. Version in deutscher Menüversion auf einem Vista- Laptop aufgenommen worden. Aktuell benutzte Version .NET 3.5 !!

Start der IDE (Entwicklungsumgebung) mit



Dann in Datei → Neu → Projekt im folgenden Bild

Windows Forms-Anwendung wählen.

Um ein angefangenes und abgespeichertes Projekt zu laden, muss man die Datei *.sln („Solution“ = Projektmappe“) doppelklicken. Dann wird Studio 2008 geöffnet. Nur dann sieht man noch keine Datei. Um den Form und den Code – Editor zu öffnen, öffne man mit Menu „Ansicht“ den „Projektmappen - Explorer“. Darin kann man dann entweder die Form1.cs anwählen durch Doppelklick. Dann sieht man das Formular. Mit F7 öffnet man dann den Code-Editor.

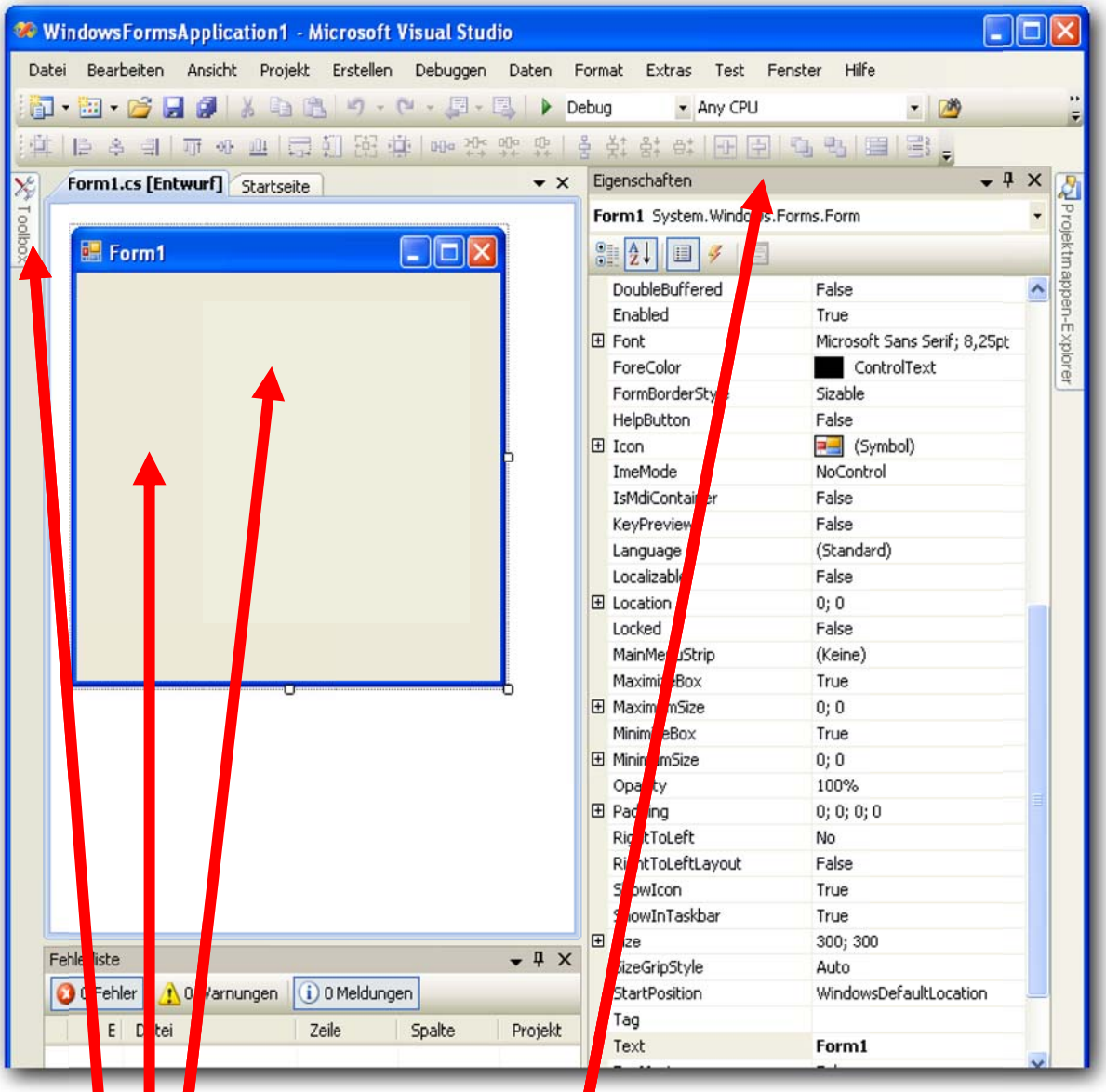
Noch ein Hinweis zum Editor: er unterstützt die Blockbildung von Strukturblöcken z.B. von einer if oder for- Anweisung. Das Einrücken wird automatisch durchgeführt, so dass durch diese Schreibweise das Programm besser lesbar wird.

```
if ...condition...)
{
    ...statement...
}
```

Setzt man den Cursor **vor** eine geschweifte Klammer „{„ oder **hinter** eine geschlossene Klammer „}“, so werden die beiden zu einem Block gehörigen Klammern grau eingefärbt und stehen direkt untereinander. Die folgende oft gesehene Syntax- Form **lehne ich ab**:

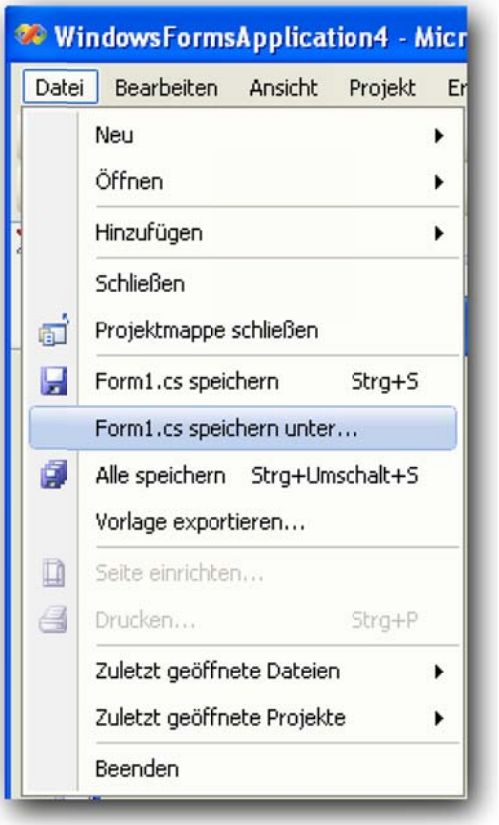
```
if ...condition...){
                ...statement...
}
```

Erste Schritte Programm „Hello World“



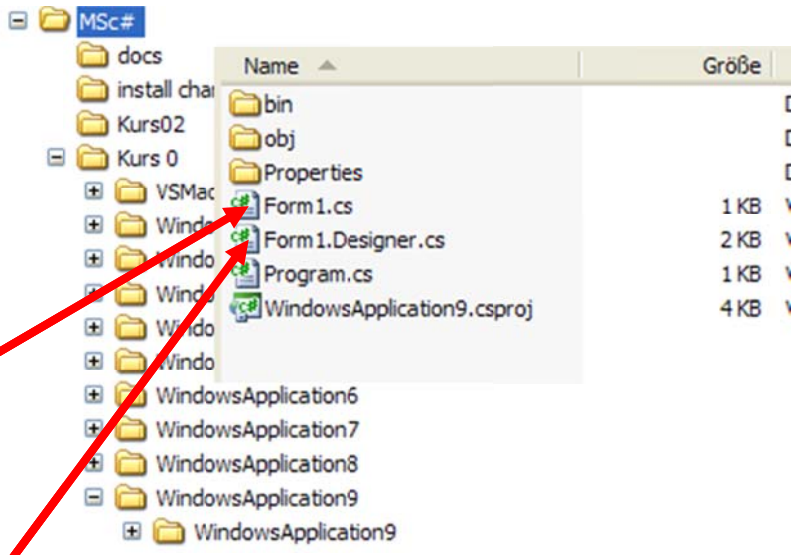
Man sieht dann folgendes Bild:
Komponentenpalette oder Toolbox
Eigenschaften Fenster zur Veränderung von Eigenschaften / Eigenschaften von Komponenten
Leeres Formular.

Vorbereitungen



Zu Beginn sollte das Projekt gespeichert und ein Pfad festgelegt werden. Der Name des Projektes wird anschließend der Programmname der exe-Datei.
Bitte nicht auf ein über Netz angeschlossenes Laufwerk, sondern immer auf lokaler Festplatte arbeiten, da z.T. doch große und viele Dateien hin und her geschoben werden.

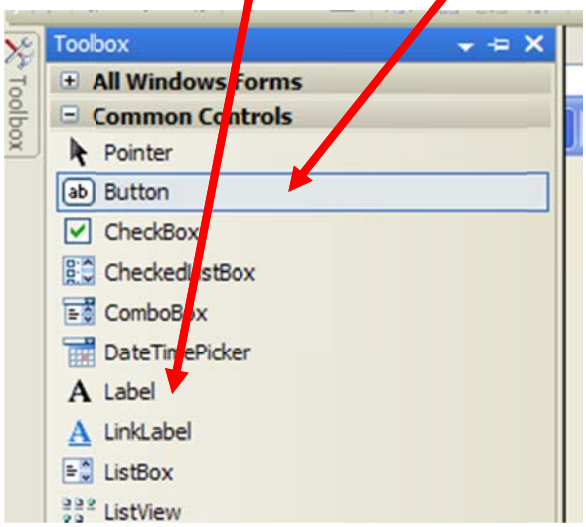
Bei mir wird in einem Pfad *D:MSC#/Kurs 0/* gespeichert, der automatische Name ist *WindowsApplication9*.



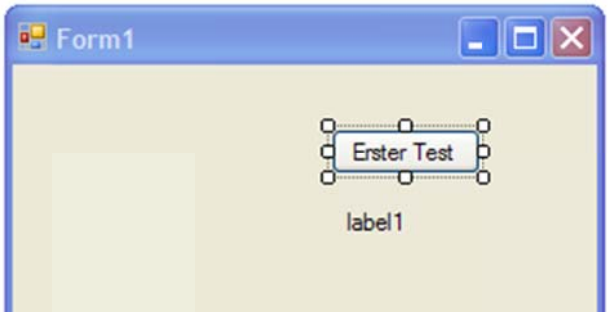
Unit- Source Code

Alle Design- Informationen des Formulars

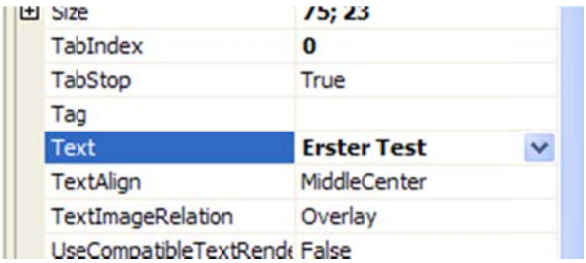
Jetzt kann die erste Komponente auf das Formular platziert werden. Dazu muss die Toolbox geöffnet werden (Menü Ansicht →Toolbox)
Wir fügen einen *Label* und einen *Button* hinzu und setzen sie irgendwo auf das leere Formular

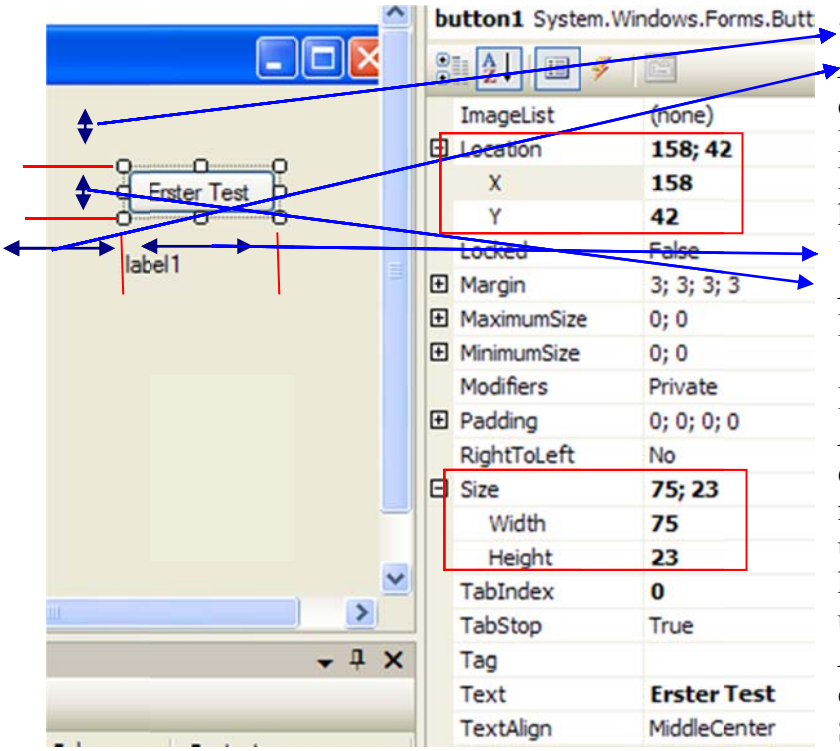


Das Ergebnis:



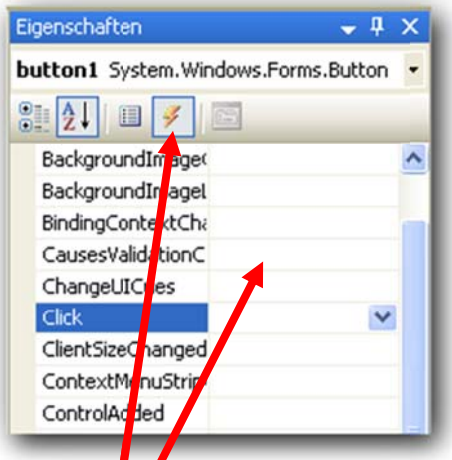
Mit dem *Eigenschaften Fenster* können nun Eigenschaften der einzelnen Komponenten sichtbar und verändert werden. Im Beispiel ist der *button1* angewählt (Quadratpunkte als Objektmarkierungen) und wir haben die Eigenschaft „*Text*“ verändert auf „Erster Test“. Z. B. ist *Text* immer die Beschriftung einer Komponente, (*Name)* der Name, der C-Namenskonventionen entsprechen muss (z. B. keine Leer- oder Sonderzeichen). Mit *Name* kann die Komponenten im C- Source-code angesprochen werden. Die ersten wichtigsten Eigenschaften:





Top und *Left* geben die Pixel- Entfernung des obersten linken Pixels der Komponente zu der Container-Komponente an (Container ist hier das Formular),
Width und *Height* sind die Dimensionen der Komponente in Pixel.

Diese Werte können allerdings im *Eigenschaften Fenster* nicht direkt eingegeben werden, sondern dort nur über die Eigenschaften *Size* und *Location*. Im Programm können dabei sowohl über *Left* usw. als auch über *Size* oder *Location* Werte verändert werden, es ist Geschmackssache. Selbstverständlich werden auch diese Werte sofort verändert, wenn man per Maus mit „Anfassen“ der Komponente die Lage oder Größe in gewohnter Weise verstellt. Die Werte werden dann automatisch aktualisiert.

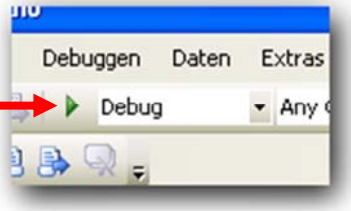


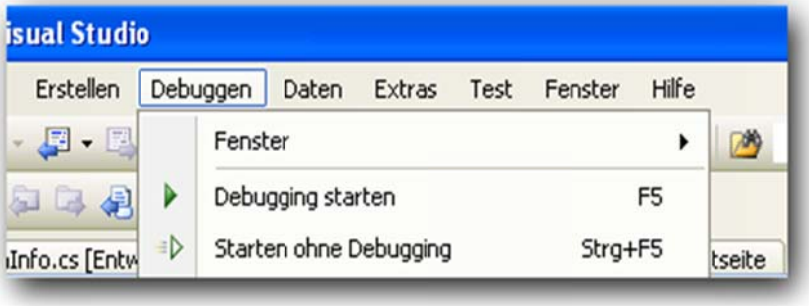
Noch ist keine einzige Zeile C eingegeben worden. Dies geschieht jetzt.
Gewünschte Funktion: bei Druck auf die Taste soll im Label- Text der Schriftzug „Hello World“ erscheinen.
Alle Funktionen der Programme werden über Ereignisse (Events) gesteuert. Jede Komponente hat eine Liste von möglichen Ereignissen, die in dem Eigenschaften- Fenster angesprochen werden können. Wir benötigen den Click-Ereignis von *button1*. Man erzeugt automatisch eine Ereignis- Behandlungs- Funktion durch Doppelklick auf den rechten Eintrag neben dem Event:
Symbol für die Ereignisse- Seite
Doppelklick
Es öffnet sich der Editor für den C#- Source- Code und man kann jetzt den C#- Source eingeben, der ablaufen soll, wenn der User später im laufenden Programm auf die Taste klickt.

Wir geben folgenden rot umrandeten Text ein:

```
}

private void button1_Click(object sender, EventArgs e)
{
    label1.Text = "Hello World";
}
```

Nun kann das erste Programm kompiliert werden, entweder mit F5 oder mit Klick auf
oder über Menü:

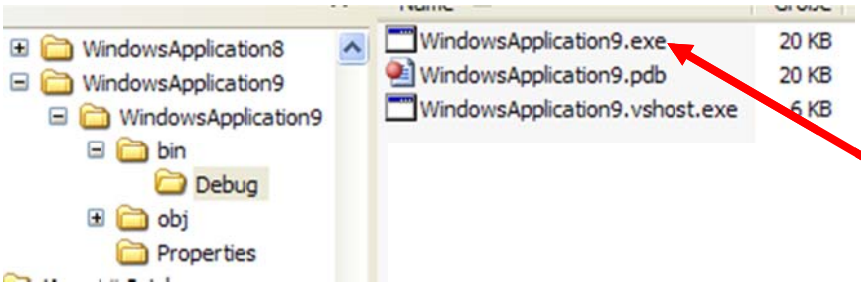




Das Programm wird neu abgespeichert, kompiliert, gelinkt und anschließend gestartet. Man erhält letztendlich folgendes Ergebnis:

Vor Klick:

Nach Klick



Es sind nun eine Menge neuer Dateien auf der Festplatte gelandet:

Die exe ist jetzt das ausführbare Programm, wohl logo.

Die anderen Dateien enthalten weitere Projektinfos, welche, spielt hier erst mal keine Rolle.

Das Programm sollte normal über die X-Taste geschlossen werden. Niemals über den Task- Manager, denn dann kommt die IDE (Intergrated Development Enviroment) von C# durcheinander und muss neu gestartet werden.
Bei Endlosschleifen, die vom IDE gestartet werden, ist ein Abbruch mit Shift + F5 oder über Menü *Debuggen/Debugging beenden* möglich.
Noch ein paar Tipps zum Zurechtfinden:
Öffnen des Source- Codes mit Menü *Ansicht → Code* (F7)
Öffnen des Form- Designers mit Menü *Ansicht →Designer* (Shift+F7)

Öffnen der Toolbox mit den Komponenten: Ansicht → Toolbox (Strg+w, dann x)
Öffnen des Eigenschaftenfensters mit Ansicht → Eigenschaftenfenster (Strg+w, dann p)

Portieren der exe- Datei auf andere PC

Die Datei erwartet einige DLL und die .Net- Umgebung auf dem Zielrechner, die sie auf einem andern PC ohne .Net nicht findet. Will man sein Programm anderen Usern ohne C# zur Verfügung stellen, so muss man alle Dateien des Verzeichnisses „bin“ kopiert mitliefern und .Net. falls auf Zielrechner nicht vorhanden als Redistributable (kostenlos von MS) mit liefern.

Löschen von Komponenten und Source Code

Will man Komponenten löschen, so ist es einfach: Markieren und Taste „Entf“ / „Del“. Nur: Hat man Event – Source für diese Komponenten geschrieben, bleiben diese erhalten und man hat toten Source in seinem Programm, der nie wieder benutzt wird. Hat man Programmtext geschrieben, der auf Eigenschaften dieser Komponenten zugreift, dann wird es beim Kompilieren Fehlermeldungen geben.
Also am Besten: Erst alle Verweise auf diese Komponente löschen, dann alle Event- functions leeren, dann erst die Komponente entfernen.
Achtung : Niemals Source löschen, den die IDE automatisch erzeugt hat. Nur Ihren Text, den Sie eingegeben haben löschen, sonst wird es schwierig.
Also: in obigem Programm nur die Zeile
Label1->Caption="Hello World";
löschen, sonst nichts mehr!

Wenn es nun doch passiert ist und Sie die Funktion komplett gelöscht haben, erscheint nach F5 folgende linke Fehlermeldung.

Jetzt mit Strg+z (Undo) den gelöschten Text wiederholen und nur Ihren Text löschen.

Die function muss erst im Eigenschaftenfenster im Event Click entfernt werden, z. B. durch rechten Mausklick auf Click und dann Reset wählen.

Danach ist wieder alles OK.

Jetzt noch ein paar einfache Übungen:

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = "Hello World";
    button1.Top++;
    button1.Left++;
    button1.Width++;
    button1.Height++;

}
```

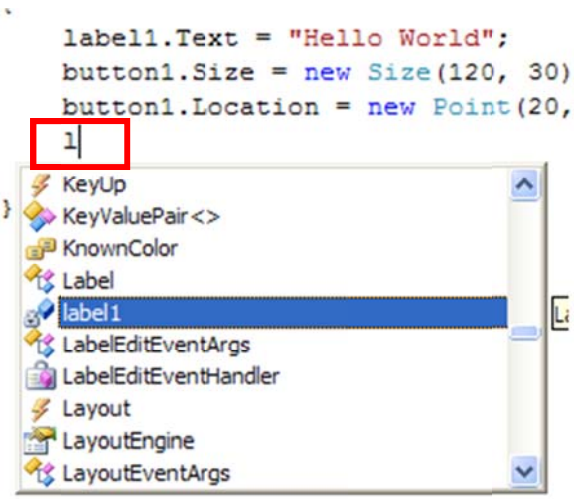
Verändern von Lage und Position des Buttons per Programm über *Top / Left* usw.

Das Gleiche über *Size* und *Location*, der Code wird etwas komplizierter:

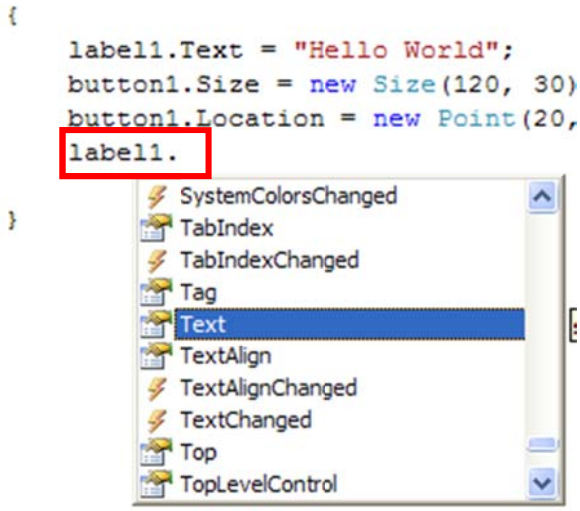
```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = "Hello World";
    button1.Size = new Size(120, 30);    //gleichbedeutend mit Width=120, Height=30
    button1.Location = new Point(20, 50);//gleichbedeutend mit Left=20, Top=50;

}
```

Man kann jetzt sehr gut die hervorragende Unterstützung mit dem „Code Completion Window“ nutzen. Jedes Mal, wenn man Code eingibt, dann werden intelligente Vorschläge gemacht, die dann mit den Tasten Up/down selektiert und mit der Return-Taste ausgewählt werden können.

Gibt man in dieser Eventfunction/Ereignisfunktion die neue Zeile
`label1.Text = button1.Size.Height.ToString();`
ein, so braucht man nur einen Buchstaben eingeben und schon erhält man sinnvoll Vorschläge.

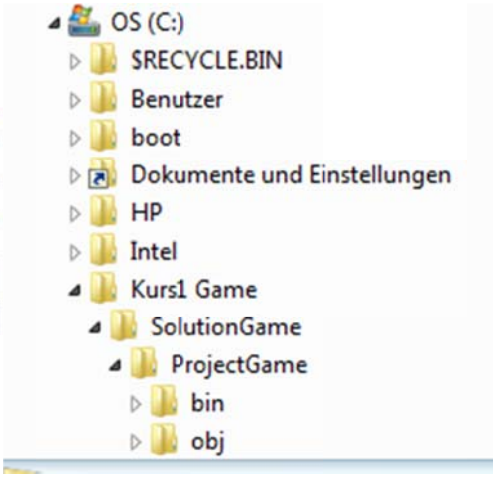


Z.B. Nach Eingabe von „l“ sieht man linkes Bild. Nur durch Taste „Return“ wählt man jetzt label1 aus. Nach dem Punkt sieht man dann den nächsten Vorschlag usw.



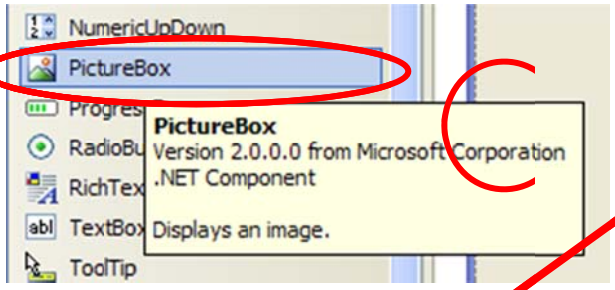
Kurs 1 Spiel

In diesem Kurs soll gezeigt werden, wie einfach und schnell und mit wie wenig echtem C#-Code schon funktionierende Programme geschrieben werden können.
Wir schließen alles Vorherige und starten eine neue Applikation (*Datei→Neu→Projekt*), dann Geben wir folgende Daten im Fenster “Neues Projekt“ ein und wählen *Windows Forms-Anwendung*. Dann wird alles in dem Verzeichnis *C:\Kurs1 Game* gespeichert.

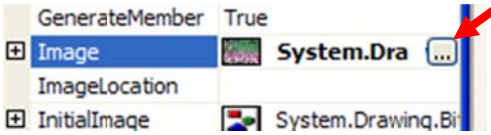


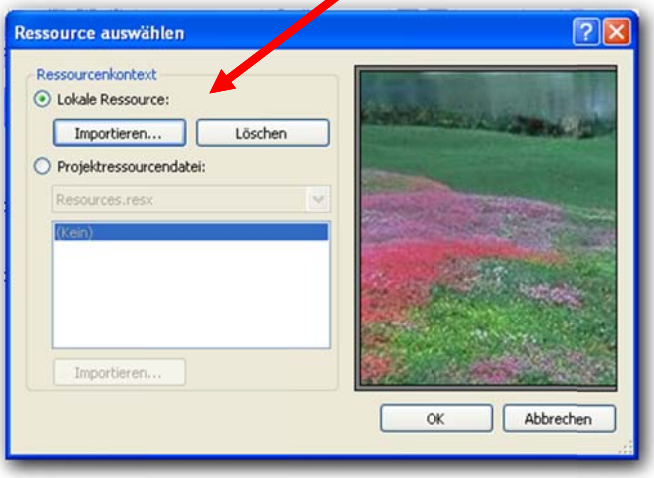
Jetzt brauchen wir 2 Bilder, die vom ftp-server geladen und auch dort auf die Festplatte kopiert werden sollten. Es können selbstverständlich auch eigene Bilder sein, aber nicht alle Formate arbeiten korrekt. Hintergrundbild:

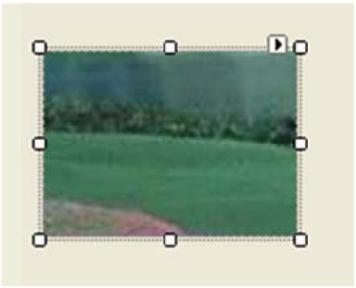
Blumen.JPG aus meinem FTP- Verzeichnis *MS Studio2008 C#/Kurs1# Spiel* und das Mohrhuhn *grHuhn5 echt.bmp*.
Wer sich das fertige Spiel vorher mal ansehen und ausprobieren möchte, der lade sich die Datei *WindowsApplication1.exe* aus dem Verzeichnis*WindowsApplication1/bin/Debug* vom Server und starte es. Man kann jetzt nach Taste „Start“ versuchen, auf das sich bewegende Moorhuhn zu klicken. Die Treffer und die Fehlversuche werden gezählt. Die Zeit zwischen zwei Bewegungen und die Größe des Moorhuhnes können verstellt werden.
Dieses Spiel wollen wir jetzt programmieren. Dazu brauchen wir einige Komponenten, die wir auf das leere Formular ziehen:



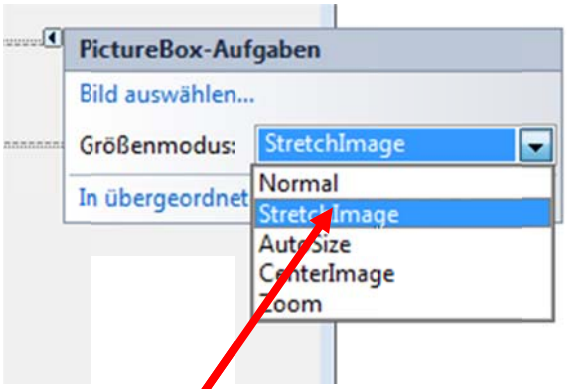
Erst zwei Komponenten *PictureBox* von der Seite Allgemeine Steuerelemente.
pictureBox1 soll das Hintergrundbild werden, wir aktivieren die Eigenschaft *Image* im Objektinspektor durch Klicken auf die Taste mit den drei Punkten und können nun unser Bild *Blumen.JPG* mit „*Lokale Ressource*“ und *Importieren* laden.







Nach Laden sehen wir in der Komponente einen Ausschnitt des geladenen Bildes. Erst ganz zum Schluss wollen wir dieses Bild unter das ganze komplette Formular legen.



Die zweite *pictureBox2* füllen wir mit dem Moorhuhn. Wenn es nach dem ersten Bild auf das Formular gezogen wurde, dann liegt es über dem ersten Bild.
Man kann die Reihenfolge aber in dem PopUp-Menu jeder Komponente leicht verändern:
Dort kann man „*In den Vordergrund*“ oder „*In den Hintergrund*“ wählen. Das Moorhuhn muss vor dem Hintergrundbild liegen.
Soll das Bild die *pictureBox* vollkommen ausfüllen, so setzt man die Eigenschaften *SizeMode* auf *StretchImage*. Den gleichen Effekt erhält man, wenn man auf den kleinen Pfeil oben rechts in der PictureBox klickt und dann die SizeMode/Größenmodus entsprechend wählt.

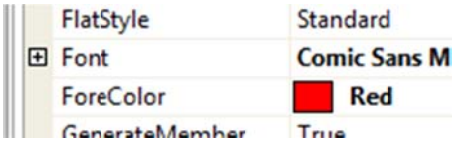
Übrigens können Eigenschaften der Sorte Bool mit einem Doppelklick auf *true* oder *false* in die jeweils andere Eigenschaft geschaltet werden.

Will man jetzt das Hintergrundbild formatfüllend in der Form darstellen, auch dort die *SizeMode* auf *StrechImage* stellen und die *Dock*- Property auf „*Fill*“ stellen.
Soll das Programm den Bildschirm komplett füllen, dann setzt man in der Form die Eigenschaft *WindowsState* auf *Maximized*. Diese Voreinstellungen kann man auch programmieren, das sieht dann so aus:

```
public Form1()
{
    InitializeComponent();
    pictureBox1.Dock = DockStyle.Fill;
    pictureBox2.SizeMode = PictureBoxSizeMode.StretchImage;
}

private void Form1_Activated(object sender, EventArgs e)
{
    Form1.ActiveForm.WindowState = FormWindowState.Maximized;
}
```

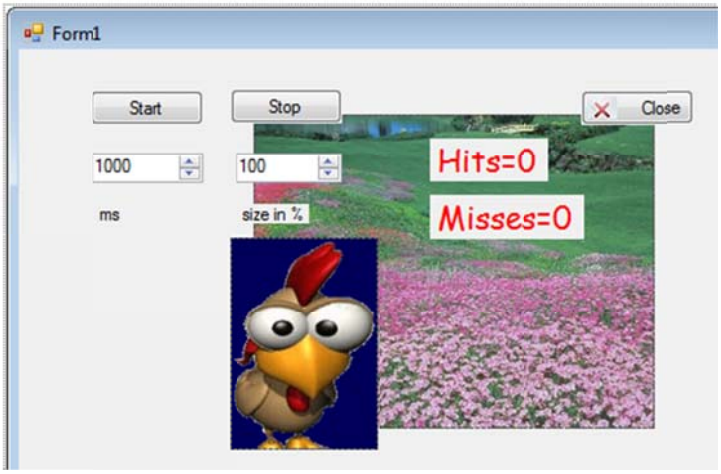
Hinter *public Form1()* kann man Initialisierungen durchführen, jedoch kann man die Form erst auf *Maximized* stellen, wenn sie aktiviert ist. Die Ereignisfunktion *Form1_Activated* erzeugt man durch Markieren von *Form1*, selektieren der Events im Property Manager und dann Doppelklick hinter *Activated*. Jetzt füllt das Bild mit den Blumen allerdings noch nicht die volle Form aus, da es noch nicht „gestreched“ wurde. Also auch in *PictureBox1* die *SizeMode* anpassen.

Jetzt werden noch benötigt:
2 Labels, *Transparent* geht noch nicht, *Font*-Eigenschaft nach Geschmack (ich mag ComicSansMS), Schriftgröße auf 14pt. Schriftfarbe Rot stellt man in der Eigenschaft *ForeColor*



ein. Die gleichzeitige Veränderung von Eigenschaften von mehreren Komponenten geht, wenn man die Komponenten gemeinsam anwählt. Die Anfasspunkte werden dann grau. Das gemeinsame Anwählen geht mit z. B. gedrückter Shift- Taste oder mit anderen Windows-typischen Wegen. Den Text auf Hits=0 und Misses=0 in Eigenschaft *Text* ändern.

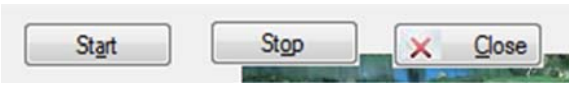
Weiter 2 *Buttons*, eine *Name=buttonStart*, *Text=Start*, die andere Taste *Name=buttonStop, Text=Stop*.
Dann brauchen wir zwei Versteller für Zahlen, zu finden in Allgemeine Steuerelemente, dort *NumericUpDown*. Den voreingestellten Wert von *NumericUpDown*1 in *Value* auf 1000 ändern. Das geht auf Anhieb nicht, da der Maximalwert in *Maximum* auf 100 steht. Den also erst auf 2000 und dann *Value* auf 1000 und *Increment* auf 100. Und den *Value* von *NumericUpDown*2 auf 100 verstellen. Beide mit einem Label beschriften mit „ms“ und „size in %“.Ergebnis:



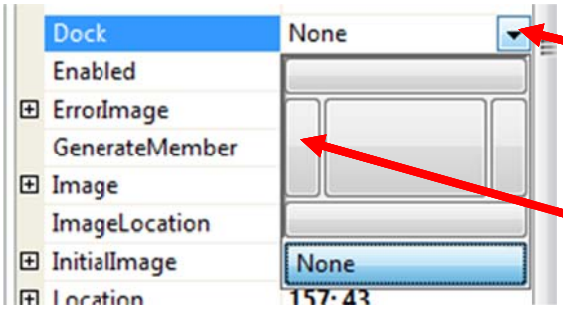
Userfreundliche Programmierer spendieren noch eine Close-Taste. Diese ist mit einem *Button* leicht zu erreichen. Habe diese Taste *buttonClose* genannt, *Text* auf *Close* und in *Image* das Bild *close.jpg* einbinden, *TextAlign* auf *MiddleRight* und *ImageAlign* auf *MiddleLeft*. In der Click Funktion dann folgenden Text:

```
private void button1_Click(object sender, EventArgs e)
{
    Close();
}
```

Will man aber mehrere solcher Buttons mit Icons haben, so warte man auf die dafür viel bessere Komponente *ToolStrip*.
Noch einen Hinweis für Freunde der Steuerung über Tastatur. Macht man irgendwelche Bedienschritte immer wieder, dann ist das Bedienen über Tastatur einfach schneller und bequemer als mit Maus. Dies wird auch hier unterstützt. Man sieht unten in der Close- Taste, dass das C unterstrichen ist. Schreibt man in der Eigenschaft *Text* ein & vor dem Buchstaben C, definiert man diesen Buchstaben als Short-Cut- Taste. Wollen wir z. B. in unserer Anwendung die Start-Taste mit der Tastatur über „a“ schnell ansprechen (und bei Stop z. B. mit „o“), so schreiben wir in *Text* vor diesen Buchstaben ein „&“. Wir haben jetzt:



Diese Tasten lassen sich jetzt z. B. mit ALT+“o“ bedienen.



Die Eigenschaft *Dock* führt eine automatische Plazierung und Vergrößerung von Komponenten durch. Man kann die Seiten der Komponenten damit an die Seiten des „Containers“ kleben. Bei uns ist die *Form1* der Container. Mit Klick auf den Pfeil unten in der Eigenschaft *Dock* öffnet sich ein Fensterchen mit einigen Tasten. Klickt man z.B. auf die Taste, auf die der rote Pfeil zeigt, dann wird Dock auf *Left* gestellt und bei der Komponente wird z. B. die linke, obere und untere Seite an den Container geklebt, die rechte Seite ist frei verschiebbar. Wir wollen alle vier Seiten kleben, das geht mit Klick auf die mittlere große Taste, *Dock* wird dann= *Fill*. Macht man dieses im Eigenschaftenfenster, hat das den Nachteil, dass schon in der IDE das Formular ausgefüllt wird und die Form durch Anklicken nicht mehr selektiert werden kann. Besser man klebt erst nach dem Start des Programms, z. B. in dem dafür bestens geeigneten Event „*Form_Activated*“, wie oben bereits schon gezeigt.

Danach sollte alles wie gewünscht aussehen, nur es funktioniert noch nichts.
Zu einem Spiel wird es erst mit den nächsten Schritten.
Was soll passieren: Nach Start soll die Position von *PictureBox2* mit einem Zufallsgenerator verändert werden. Klickt der User auf das Bild *PictureBox2*, soll ein Zähler inkrementiert werden, trifft er daneben (also auf *PictureBox1*), sollen diese verfehlten Treffer gezählt werden. Bei Stop soll alles anhalten, bei Start die Zähler wieder auf null gesetzt werden.

Dazu brauchen wir eine Timer-Komponente, die regelmäßige Ereignisse zum Verstellen der Position erzeugen kann. Diese finden wir in der Toolbox unter „Komponenten“ als *Timer*. Er hat die Eigenschaft „*Enabled*“, die im Eigenschaftsfenster auf *False* gesetzt wird. Wenn *Enabled* auf *True* steht, dann erzeugt er alle in *Interval* spezifizierten Millisekunden den Event *Tick*. Also folgende Schritte sind zu tun:
timer1 auf *Form1* ziehen, *Enabled* auf *False*, *Interval* auf 1000.
Jetzt Start und Stop- Taste programmieren mit Doppelklick auf die jeweilige Taste und füllen der *Click*- Funktion mit untenstehendem rot umrahmtem Text.

```
private void buttonStart_Click(object sender, EventArgs e)
{
    timer1.Enabled = true;
}

private void buttonStop_Click(object sender, EventArgs e)
{
    timer1.Enabled = false;
}
```

Im Ereignis *timer1_Tick* wird mit dem Zufallszahlengenerator R*andom()* die Position des oberen Pixels vom Moorhuhn in *PictureBox2* im erlaubten Bereich verändert. Der erlaubte Bereich berechnet sich aus der freien inneren Dimension der *Form1* (abgefragt in *ActiveForm.ClientSize.Width* und …*Height*) abzüglich der Dimension der*PictureBox2*.
Der vollständige Code in *timer1_Tick* lautet:

```
Random rand = new Random();
pictureBox2.Left = rand.Next(Form1.ActiveForm.ClientSize.Width - pictureBox2.Width);
pictureBox2.Top = rand.Next(Form1.ActiveForm.ClientSize.Height - pictureBox2.Height);
```

Mit der Methode .*Next(zahl)* wird eine Zufallszahl zwischen 0 und zahl-1 erzeugt.
Jetzt austesten mit F5. Das Moorhuhn muss nach Start über den Bildschirm springen.

Jetzt fehlen noch Zähler und Ergebnisausgabe und Veränderung der Bewegungsgeschwindigkeit. Letztes ist einfach: im *ValueChanged*- Ereignis von *numericUpDown1*kopiere man den Wert von *numericUpDown1 .Value* nach *Timer1.Interval*.
Code:

```
private void numericUpDown1_ValueChanged(object sender, EventArgs e)
      {
            timer1.Interval = (int) numericUpDown1.Value;
      }
```

Das expliziete Casting auf int ist nötig, da Value ein Typ „Decimal“ ist.

Den Zähler realisiert man mit zwei Variablen außerhalb der Funktionen. Dieses sind keine globalen Variablen (gibt es in C# nicht mehr), sondern zwei Variablen des Objektes Form1.

.

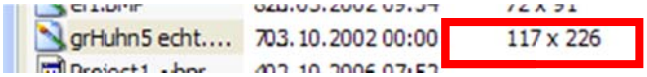
Ich mache es mit den beiden Variablen cntOK und cntNOK.
Also bei der Start- Taste auf null setzen, bei Click auf *pictureBox1* (daneben) cntNOK ++, bei Click auf *pictureBox2* Treffer cntOK++ und Ergebnis in l*abel1* / 2 anzeigen mit der Methode *ToString(),* die den Intergerwert in einen String umwandelt.
Fertiger Source:

In timer1_Tick:
```
      label1.Text = "Hits= " + cntOK.ToString();
      label2.Text = "Misses= " + cntNOK.ToString();
```
In pictureBox2_Click
```
cntOK++;
```
In pictureBox1_Click
```
cntNOK++;
```

Nun hat diese Programmierweise den Nachteil, dass die Fehlschüsse auf null bleiben, wenn man nichts tut. Wertet man ein Nichtstun auch als Fehlschuss, muss das Programm etwas abgeändert werden. Zudem sollte in einem Durchgang nur ein Klick gezählt werden, z.Zt. kann man pro Durchgang mehrere Treffer auslösen. Im Timer wird in einer Bool- Variable namens „free“ ein Schuss freigegeben und schon mal ein Fehlschuss gezählt. Nur in *pictureBox2_Click* wird jetzt ein Treffer gezählt, der Zähler für die Fehlschüsse korrigiert und weiter Schüsse blockiert (siehe Kommentar).
Im neuen Source sind t*imer1_Tick* und *pictureBox2_Click* unten abgebildet. Die Reihenfolge im Source ist beliebig veränderbar. Im Editor kann man vor jeder Funktion mit den + oder – Zeichen mit quadratischem Rahmen den Code verbergen oder öffnen.

```
int cntOK = 0;
int cntNOK = 0;
bool free = false;

private void Form1_Activated(object sender, EventArgs e)...

private void button1_Click(object sender, EventArgs e)...

private void buttonStart_Click(object sender, EventArgs e)...

private void buttonStop_Click(object sender, EventArgs e)...

private void timer1_Tick(object sender, EventArgs e)
{
    Random rand = new Random();
    pictureBox2.Left = rand.Next(Form1.ActiveForm.ClientSize.Width - pictureBox2.Width);
    pictureBox2.Top = rand.Next(Form1.ActiveForm.ClientSize.Height - pictureBox2.Height);
    label1.Text = "Hits= " + cntOK.ToString();
    label2.Text = "Misses= " + cntNOK.ToString();
    free = true;
    cntNOK++;

}

private void numericUpDown1_ValueChanged(object sender, EventArgs e)...

private void pictureBox2_Click(object sender, EventArgs e)
{
    if (free)
    {
        cntNOK--;  // Misses corrected
        cntOK++;   // count hits
        free = false;// stop counting
    }
```

Zum Abschluss dieses ersten kleinen Projektes wird jetzt noch die Größe des Zieles variiert mit



numericUpDown2. Dort soll eine Prozentzahl die tatsächliche Größe bestimmen. Steht dort 50, so sollen *Height* und *Width* eben nur 50% der ursprünglichen Größe besitzen. Im Explorer kann man die Größe des Bildes erkennen mit 117* 226 Pixel. Im Ereignis *Changed* von *numericUpDown2* wird dann folgender Text hinzugefügt:
Man beachte, dass vor Start die Originalwerte *Height=226* und *Width*=117 in *pictureBox2* stehen sollten und dass die Eigenschaft *SizeMode* auf *StrechImage* steht, sonst geht es nicht.
Code:

```
private void numericUpDown2_ValueChanged(object sender, EventArgs e)
      {
            pictureBox2.Width = 117 / 100 * (int)numericUpDown2.Value;
            pictureBox2.Height = 226 / 100 * (int)numericUpDown2.Value;
      }
```

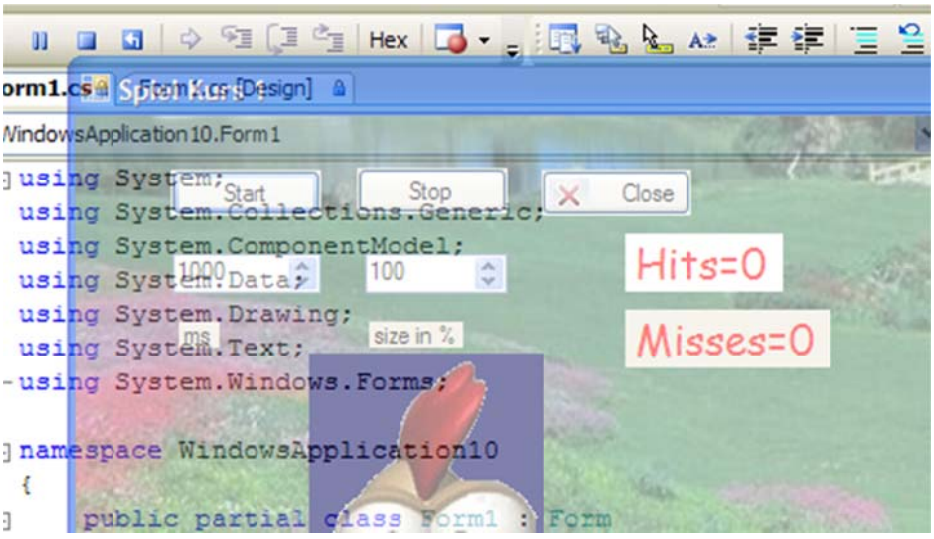
Ein letzter Schönheitsfehler wird beseitigt. Wenn man die Bewegung des Zieles beobachtet, so erscheint ganz kurz aber sichtbar das Ziel erst horizontal neben der alten Position. Das liegt daran, dass erst *Left* und dann *Top* verstellt wird.
Abänderung durch das Verstellen von *Location*, dann werden x und y- Koordinate gleichzeitig verstellt.
Code in timer1_Tick:

```
int left= rand.Next(Form1.ActiveForm.ClientSize.Width - pictureBox2.Width);
int top= rand.Next(Form1.ActiveForm.ClientSize.Height - pictureBox2.Height);
Point loc1 =new Point(left,top);
pictureBox2.Location = loc1;
```

Weiterer Variationen wie z. B. High Score Liste, einstellbare Laufzeit, mehrere Ziele sind keine Grenzen gesetzt.

Stellt man die Zeit auf <700 ms, so wird es schon sehr schwierig, alle Ziele zu treffen.

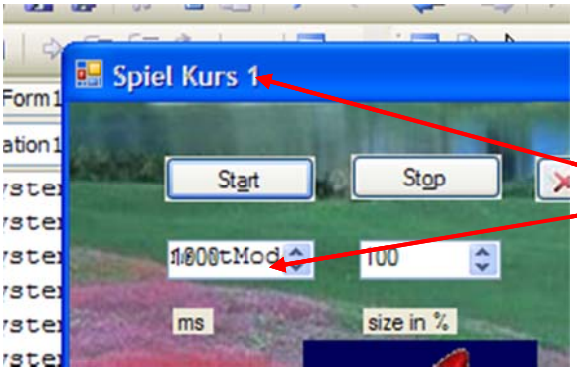
Alle fertigen Dateien sind auf dem ftp-Server unter „Kurs1#Spiel“ zu finden. Dort habe ich auch noch den Titel der Form (in *Form1.Text*) geändert.
Jetzt fehlt eine kleine Schönheitskorrektur: Es ist die Transparenz der Label und des Zieles, was nicht so trivial ist.



Zur Transparenz einige Infos:

Eine Form hat die Eigenschaft *Opacity*, die zwischen 0 und 100% eingestellt werden kann. Damit werden alle Elemente mehr oder weniger durchscheinend. Bei 50% sieht das Spiel z.B. so aus und der Inhalt dahinter

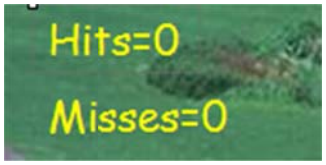
schimmert schon mächtig durch. Dies ist für Auf- und Abblendungen beim Start wohl mal ganz witzig.



Stellt man die Eigenschaft *TransparencyKey* z.B. auf *White*, dann bekommt die Anwendung an allen Stellen, die weiß sind, Löcher und der Hintergrund ist zu sehen. Sowohl in der Überschrift als auch die *numericUpDown* sind jetzt stellenweise durchsichtig. Selbst die Maus klickt durch die Löcher durch. Das Close- Kreuz ist auch durchsichtig!

Nur leider gilt dies nicht für ein BMP, das in eine *pictureBox* geladen ist.
Transparenz ist mit BMP- Bildern nicht möglich. Aber dafür kann man gif und png- Bilder laden, und die bringen eine Transparenz- Eigenschaft mit. Ich habe das Moorhuhn- Bild in ein GIF konvertiert und die Außenfarbe als Transparenz Farbe deklariert (das geht mit einschlägigen Bildbearbeitungsprogrammen, z. B. ULead Photoimpact). Es heißt jetzt *grHuhn5 echtweiß3.gif*. Dann muss man noch *BackColor* von *PictureBox2* auf *Transparent* stellen, dann sollte es gehen. Den *TranparencyKey* der Form wieder auf eine Farbe stellen, die nicht vorkommt, z.B. red.
Nun sollte es gehen. Damit beim Bewegen des Moorhuhns keine weißen Felder zu sehen sind, stellt man dann noch die Eigenschaft *DoubleBuffered* von *Form1* auf *true*.

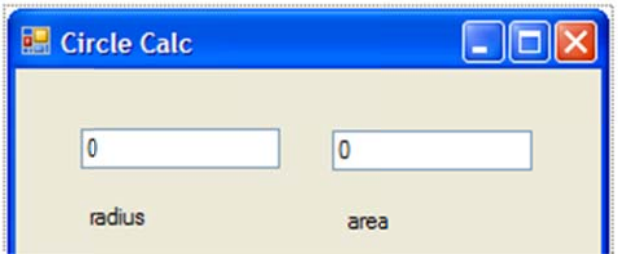
Die Label werden mit folgendem Schritt gegen das Blumenbild transparent: *pictureBox1* löschen samt aller Verweise darauf. Das Bild wird als Hintergrundbild der Form eingebunden. Dazu setzt man dort die Eigenschaften so, wie im Bild setzen. Dann in allen Labels die Eigenschaft *BackColor* auf *Transparent* setzen. Diese ist unter „WEB“ zu finden. Ich hab die Farbe in ForeColor auf Yellow gesetzt, dann sieht ein Label so aus:



Kurs 2 Rechnen, Multiwindowing

Es wird die Berechnung, die Ein-sowie Ausgabe von Zahlen gezeigt sowie der Umgang von Projekten mit mehreren Fenstern.

Start mit neuer Anwendung. Es werden zwei *TextBox*- Komponenten mit zwei Beschriftungen (*Label*) auf das leere Formular platziert. Bei mir ist es WindowsApplication11.



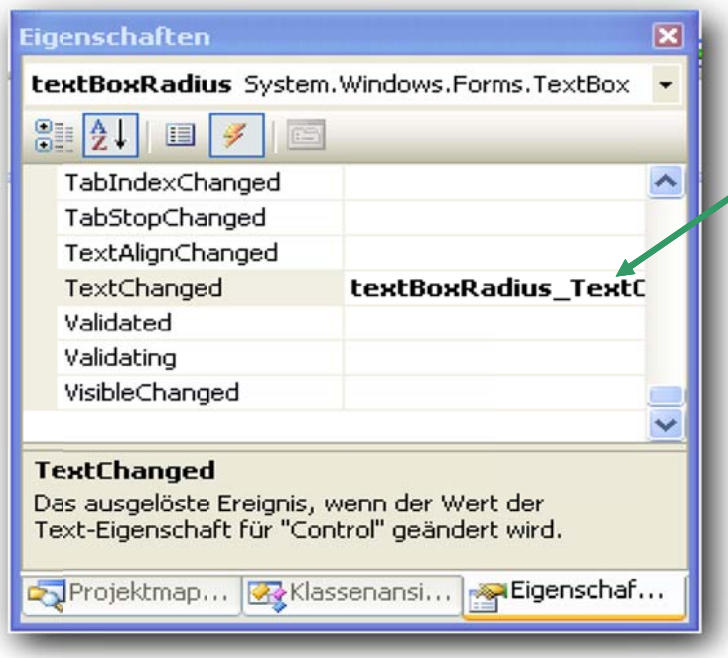
Ändere *Label->Text* Inhalte und *Edit->Text* Eigenschaften wie links gezeigt.
Nach Eingabe einer Zahl in Feld Radius soll die Fläche ausgegeben werden und nach Eingabe der Fläche in Area soll der Radius berechnet werden. Die Formeln lauten bekanntlich:
$A= \pi r^2$ sowie $r=\sqrt{(A/\pi)}$.

Nun muss überlegt werden, in welchem Ereignis die Berechnung durchgeführt werden soll. Es soll kein Button zum Start der Berechnung geben.
Möglichkeiten: **1. *TextCanged*** der *textBox*- Komponenten. Nachteil: gibt man falsche Werte ein (z. B. Buchstaben) führt die Berechnung sofort zu Fehlermeldung, da ja sofort der Stringinhalt des *Text*- Feldes (.*Text* ist eine Zeichenkette, Typ String) in eine Zahl konvertiert wird und der User keine Chance hat, es zu korrigieren. Die Konversion des Strings in eine Floating- Point – Zahl geht mit *Double.Parse()* oder mit *Convert.ToDouble()*. Ausprobieren:
Code:

```
private void textBoxRadius_TextChanged(object sender, EventArgs e)
        {
            double radius=Double.Parse(textBoxRadius.Text);
            double area = radius*radius*Math.PI;
            textBoxArea.Text = area.ToString();
        }
```

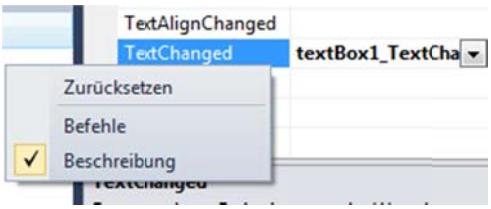
Hinweise zur Berechnung: Die Ein- und Ausgabe erfolgt über die *Text*- Eigenschaft vom Typ String. Es muss dann eine Konvertierung von Float nach String und umgekehrt durchgeführt werden. Es wird hier der Floating-Point Typ „*double*“ benutzt (ca. 15 Stellen), der viel genauer ist als *float* (7 Stellen). Die Konstante π ist in der mathematischen Bibliothek *Math* zu finden.
2. *Leave*: Dann wird das Ereignis ausgelöst, sobald der Focus auf eine andere Komponente übergeht, z. B. durch Klick des Benutzers auf *textBox2*. Nur der User muss das wissen.
Also Ändern von *OnChange* auf *Leave*. Das geht so, dass man das Feld hinter dem Ereignis *Leave* doppelklickt, im Code den Inhalt von *TextChanged* in die neue Funktion *Leave* schiebt. Jetzt **nicht im Code** den Funktionsaufruf löschen, sondern den Eintrag im Eigenschaftsfenster unter *TextChanged*. Dann wird in der IDE automatisch der Funktionsaufruf gelöscht.

```
private void textBoxRadius_TextChanged(object sender, EventArgs e)
        {
        }                              NICHT LÖSCHEN !!

private void textBoxRadius_Leave(object sender, EventArgs e)
{
      double radius = Double.Parse(textBoxRadius.Text);
      double area = radius * radius * Math.PI;
      textBoxArea.Text = area.ToString();
}
```



Diesen Eintrag Löschen !

Oder Cursor auf TextChange, rechter Mausknopf, dann « Zurücksetzen »



3. Möglichkeit. Nach Druck der Eingabe- Taste (auch return- Taste genannt) auf dem Keyboard soll Berechnung durchgeführt werden. Das ist nicht mit dem Ereignis *Enter* zu realisieren (*Enter* wird ausgelöst, wenn die jeweilige Komponente den Focus bekommt). Dazu gibt es das Ereignis *KeyPress*. In dem Header der Funktion wird im Parameter e der Char-Wert der Taste in e.*KeyChar* übergeben, der besagt, welche Taste gedrückt worden ist. Die return / Eingabe- Taste hat den Wert HEX 0D. Also Code:

```
private void textBoxRadius_KeyPress(object sender, KeyPressEventArgs e)
      {
            if (e.KeyChar == 0x0d) textBoxRadius_Leave(this,e);
      }
```

Oder

```
private void textBoxRadius_KeyPress(object sender, KeyPressEventArgs e)
       {
           if (e.KeyChar == (char)Keys.Return) textBoxRadius_Leave(this, e);
       }
```

Hinter der if- Abfrage wird nun die Berechnung ausgelöst. Hier könnte eine Kopie der Zeilen unter *Leave* folgen, aber man kann auch eine existierende Ereignisfunktion direkt aufrufen. Im Zeiger „*sender*“ wird die Adresse des aufrufenden Containers übergeben.
Vervollständigen Sie das Programm zur Berechnung auch noch des Radius aus der Fläche nach Druck auf die Eingabe- Taste, wenn die Textbox für „area“ den Focus hat.

Ein Hinweis sei noch getan: Die Ausgabe einer Floating- point Zahl kann auch formatiert erfolgen, indem man in der Methode ToString(IFormatProvider) den IFormatProvider benutzt.

Beispiele: Ausgabe von 12345.6789012345 in einer Textbox mit ergibt

x.ToString();	12345.6789012345	Keine Formatierung
x.ToString("f2");	123456.79	Festkomma 2 Stellen
x.ToString("g2");	1.2e+05	2 Stellen „general“

`x.ToString("c2");`	123.456,79 €	„Currency“ 2 Stellen
`x.ToString("e2");`	1.23e+005	„exponential“
`x.ToString("p2");`	12,345,678.90%	Prozent

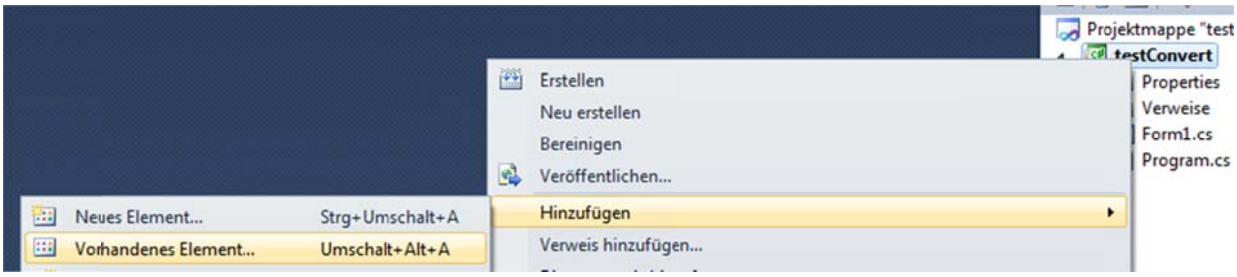
Dann gibt es noch die Buchstaben „d“ für Dezimalausgabe, dann muss x jedoch ein Typ „decimal“ sein.

Hexadezimalausgabe: x=255 führt mit `x.ToString("x10");` zu 00000000ff
Und mit `x.ToString("X10");` zu 00000000FF

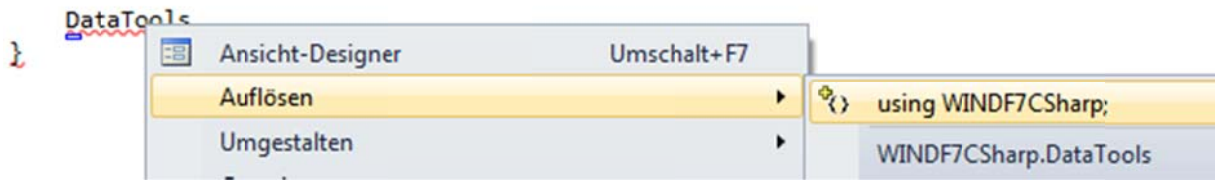
Das Ergebnis bis hierhin ist in Kurs 2a# Berechnung auf dem ftp zu finden.

Komma- Problematik

Wenn man Texte / Strings in Fließkommazahlen umwandelt, gibt es ein grundsätzliches Problem: Auf deutschen PCs ist der Dezimaltrenner ein Komma, der Punkt ist für Tausendergruppierung vorgesehen. Bei US- Rechnern ist es genau umgekehrt. Da bei uns auch meine Programme von US- Studierenden benutzt werden sollen, muss das Programm mit beiden Versionen klarkommen. Werden Textdateien zwischen beiden Nationalitäten ausgetauscht, dann gibt es immer wieder Konflikte genau deshalb. Der deutsche Rechner erzeugt Zahlen mit Kommas z.B. 1,234, die dann von US- Rechnern als Tausendertrennzeichen gelesen werden und diese Zahl als 1234 liest.
Eine Lösung bietet meine selbstgeschriebene eigene ConvertToDouble- Funktion. Wenn in einer Zahl nur ein Trennzeichen auftaucht, wird dies als Dezimaltrenner aufgefasst. Wenn mehr als ein Trenner auftaucht, so ist das rechte immer der Dezimaltrenner, die linken immer Tausendergruppierungen. Also egal ob 1,234.78 oder 1.234,56 beide Zahlen werden richtig als 1234 *Komma* 78 interpretiert, wobei *Komma* den Dezimaltrenner meint. Dabei ist es egal, auf was für einem Dezimaltrenner der PC gestellt ist, dies wird vorher von der Funktion geprüft.
Der einzige nichtlösbare Konflikt taucht auf, wenn der einzige Trenner in einer Zahl ein Tausendertrenner sein soll. Bei dieser Funktion wird er immer als Dezimaltrenner gelesen. Also egal ob 1,234 oder 1.234, diese Zahl wird dann immer als 1 *Komma* 234 gelesen.
Beispielanwendung:
Neues Projekt nach Kurs 2c# erzeugen. Zum Projekt Hinzufügen die Datei DataToolsMini.cs:



Dann diese Datei DataToolsMini.cs suchen und auswählen.
Außerhalb einer function jetzt eine neue Instanz von DataTools erzeugen, indem man das Wort DataTools eintippt und rechte Maustaste → Auflösen wählt:



Dann wird automatischer der namespace von DataTools hinzugefügt, nämlich WINDF7CSharp.
Weiter:

```
DataTools dt = new DataTools();
```

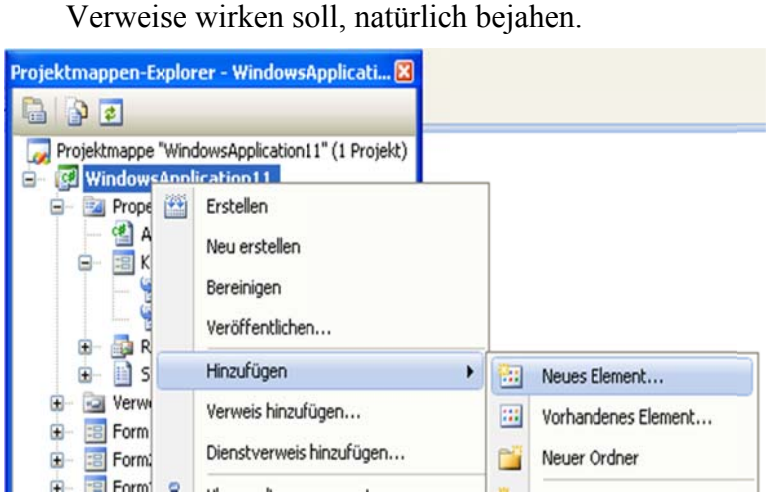
Nun eine Textbox und ein Label hinzufügen.
Hinter dem Click- Ereignis des Labels dann folgenden Code ergänzen:

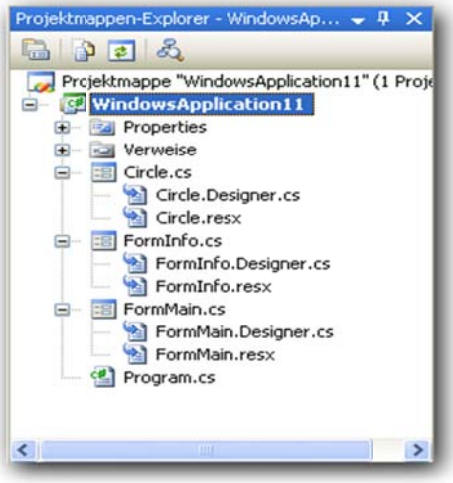
```
double X = 0;
dt.ConvertToDouble(textBox1.Text, ref X);
label1.Text = X.ToString();
```

Dann kann man das Programm ausprobieren.

Mehrere Fenster

Ein Projekt hat normalerweise mehr als ein Fenster. Wir wollen jetzt ein Projekt mit drei Fenstern generieren. Ein Fenster soll das eben entwickelte sein, ein weiteres soll das Hauptfenster werden, von dem dieses geöffnet wird und ein drittes soll ein Infofenster sein. Dazu fügen wir zwei Forms zu unserm Projekt hinzu. Am besten öffnet man dazu den Projektmappen-Explorer (Strg+w,s). Dort kann man das Projekt selektieren (bei mir WindowsApplication11), dann im Popup wählen „Hinzufügen“, dann „Neues Element“, dann eine Windows Form selektieren. Die zwei Male wiederholen. Nun alle Forms mit neuem Namen versehen, so dass anschließend wie im Bild die drei gewünschten Forms zu sehen sind. Die Abfrage, ob das „Übernennen“ auch auf Verweise wirken soll, natürlich bejahen.





Name sowie Text der zweiten Form: *FormMain*, Name sowie Text der dritten Form: *FormInfo*. Größen und Position etwa so wie im nächsten Screenshot.
Nun wird allerdings beim Start immer noch das alte Circle- Fenster geöffnet, nicht wie gewünscht FormMain. Dieses kann man ändern in der Datei *Program.cs*, die wir jetzt öffnen. Dort in der Funktion *Main()* sieht man folgenden Code:

```
static void Main()
 {
      Application.EnableVisualStyles();
      Application.SetCompatibleTextRenderingDefault(false);
      Application.Run(new Circle());
 }
```

Im roten Rahmen steht der Name der Form, die als erstes sichtbar wird. Hier einfach auf FormMain ändern. Dann öffnet sich nach Start unser FormMain:

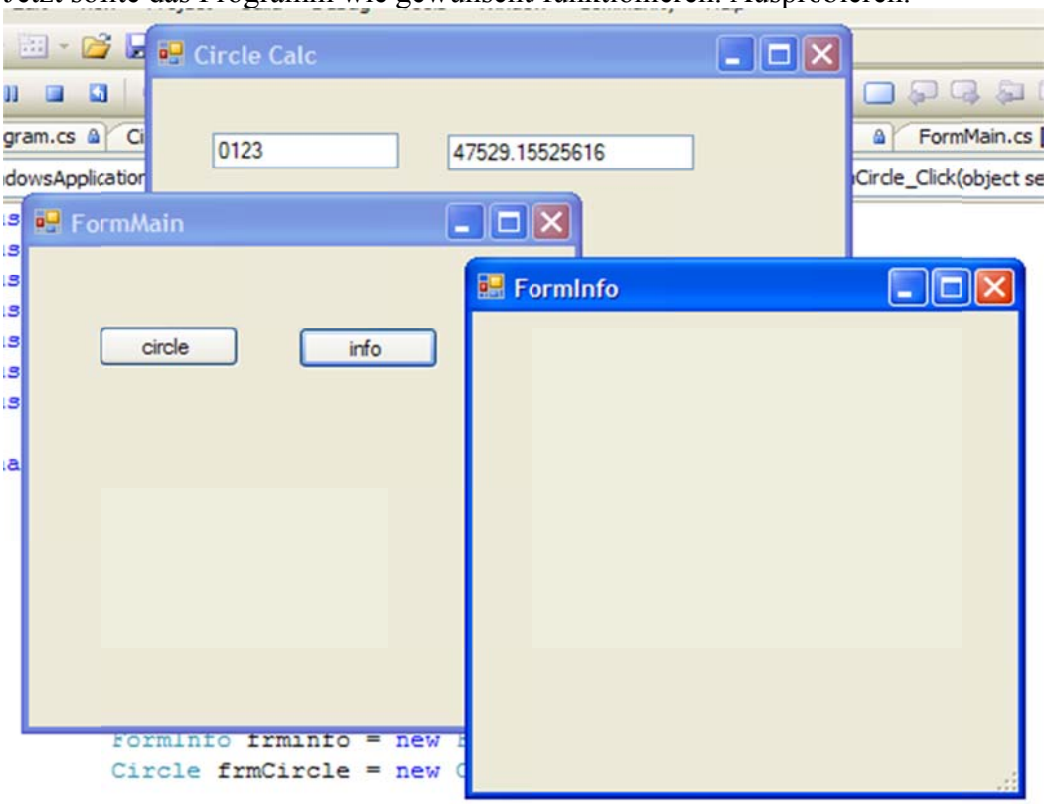
```
Application.Run(new FormMain());
```

Bevor wir jetzt weitermachen, sollte erstmal die Location der Formulare auf verschiedene Werte gestellt werden, sonst liegen alle Fenster genau übereinander. Also *Location* von *FormMain* lassen auf (0;0), *FormInfo* auf (200;200) und *FormCircle* auf (100;100).

Jetzt zwei Buttons auf das Hauptfenster, also in *FormMain* plazieren, beschriften mit Circle und Info. Mit „Circle“ soll jetzt unser ursprüngliches Fenster geöffnet werden und zwar „nichtmodal“, d. h. der User kann in andere Fenster klicken. Das Info-Fenster soll „modal“ geöffnet werden, d.h. der User muss erst dieses Fenster schließen, bevor er andere Fenster der Applikation erreichen kann. Welche Sorte sinnvoll ist, bestimmen Sie als Programmierer. Die Methoden zum Öffnen der Forms heißen einfach *Show()* und *ShowDialog()*.Code:

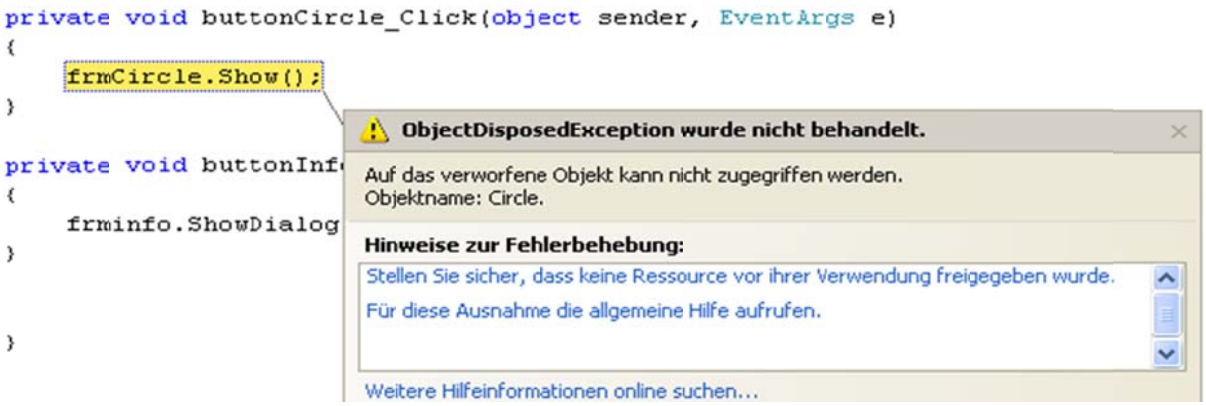
```
FormInfo frminfo = new FormInfo();
Circle frmCircle = new Circle();

private void buttonCircle_Click(object sender, EventArgs e)
{
            frmCircle.Show();
}

private void buttoninfo_Click(object sender, EventArgs e)
{
            frminfo.ShowDialog();
}
```

Erst werden Instanzen der Formulare erzeugt mit *new*, dann kann man diese Formulare ansprechen mit den beiden Show- Methoden. Die Instanziierung muss dabei außerhalb einer Funktion erfolgen, damit der Name der Instanz (also z.B. frminfo) für andere Funktionen erreichbar ist.
Jetzt sollte das Programm wie gewünscht funktionieren. Ausprobieren.



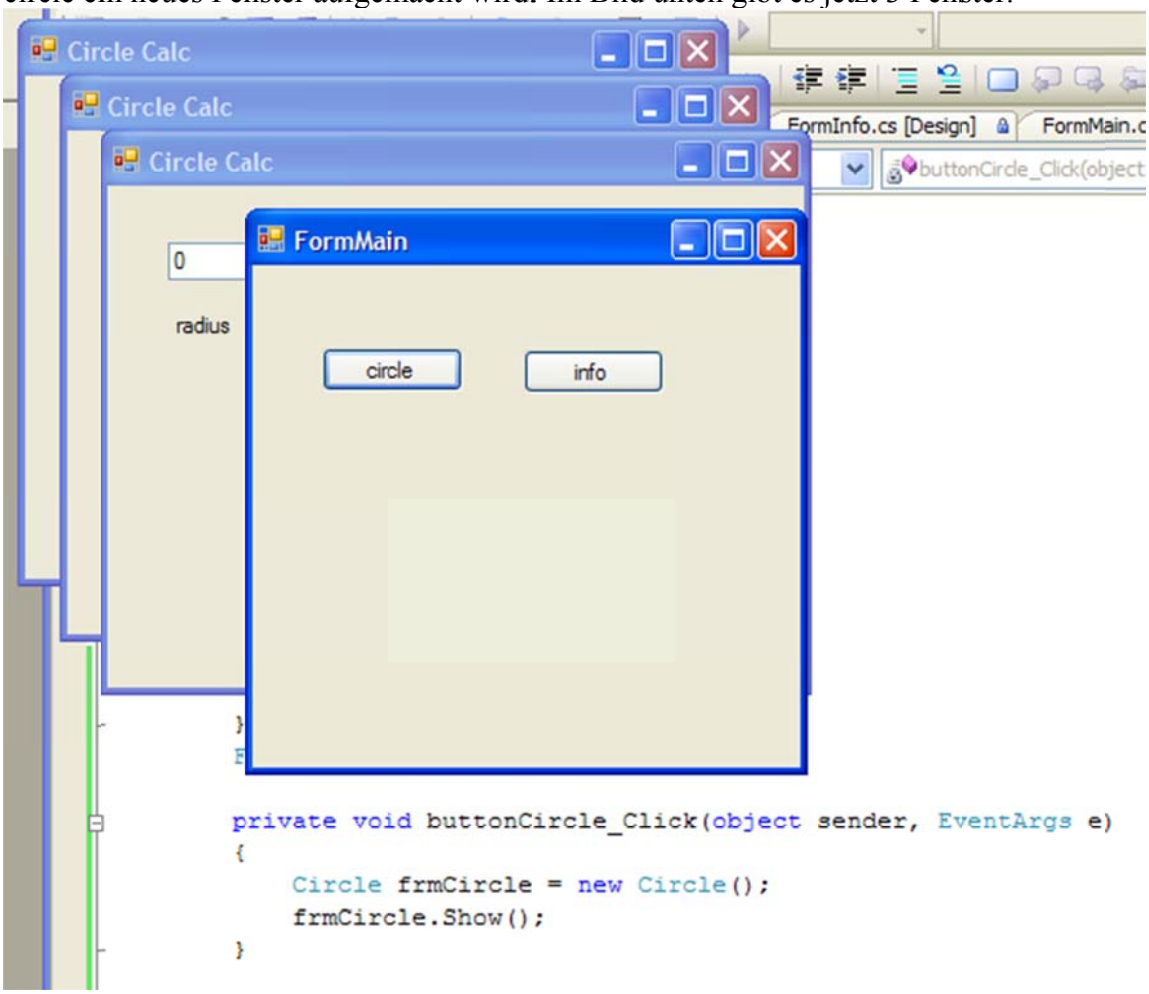
Man kann jetzt beliebig zwischen *Circle* und *FormMain* hin und her springen, aber *FormInfo* muss geschlossen werden, um wieder an *Circle* oder *FormMain* zu gelangen. Wird *FormMain* geschlossen, werden die anderen Fenster auch geschlossen und aus dem Speicher entfernt. Nun gibt es ein Problem. Schließt man die Info- Form, dann kann man sie beliebig oft wieder öffnen, die Instanz wird beim Schließen nicht entfernt, sondern nur unsichtbar. Öffnet man aber eine Form mit *Show()* und schließt diese mit Close(), dann wird die Instanz aus dem Speicher

entfernt. Beim nächsten Klick auf die Taste circle gibt es dann eine Fehlermeldung, da auf eine Instanz zugegriffen wird, die es nicht mehr gibt:



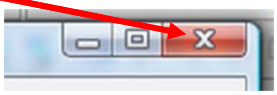
Übrigens jetzt über Menü *Debug* nach *Stop Debugging* gehen und Programm beenden.

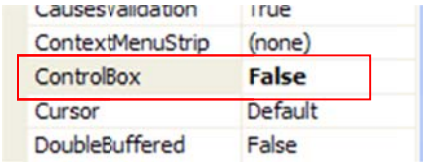
Abhilfe 1: Man verlegt die Instanziierung in die Click- Funktion des Buttons. Ergebnis ist, dass nach jedem Wiederöffnen von circle die Daten weg sind und schlimmer bei jedem Klick auf circle ein neues Fenster aufgemacht wird. Im Bild unten gibt es jetzt 3 Fenster.



Abhilfe 2:
Man spendiert eine Close- Taste auf Circle, in der nur die Form unsichtbar gemacht wird mit Aufruf der Methode *Hide()*.Code in Circle:

```
private void buttonClose_Click(object sender, EventArgs e)
{
      Circle.ActiveForm.Hide();
}
```

Dann sollte man aber die Close- Taste im Kopf des Windows-Rahmens auch verschwinden lassen z. B. mit der Property  *ControlBox* der Form, die dann auf *False* gesetzt wird.



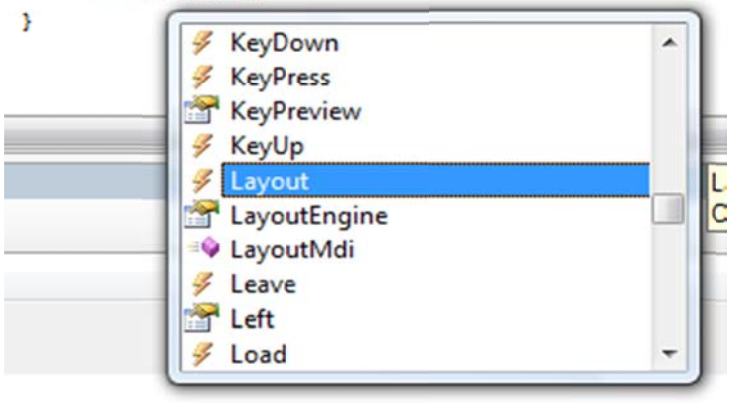
Noch besser: ControlBox lassen (der User erwartet das ja irgendwie und kennt es) und den Close- Befehl abfangen mit folgendem Event:

```
private void frmCircle_FormClosing(object sender, FormClosingEventArgs e)
{
            e.Cancel = true;
            Visible = false;
}
```

Dann geht es.

Schaut man jetzt auf unser Projektverzeichnis, so erkennt man folgende Dateien links. Jedes Fenster hat eine .cs - Datei für den Source, eine .Designer.cs für das Design der Oberfläche, eine .resx. Im Pfad „bin“ findet man dann die exe-Datei. Bitte die .Designer.cs nicht öffnen und nicht verändern. Es ist eine vom IDE automatisch erzeugte Textdatei, die das graphische Design festlegt. Zwischen Code (.cs) und Design nur mit Menü View umschalten.


```
private void buttonTransfer_Click(object
{
    frminfo.la
}
```

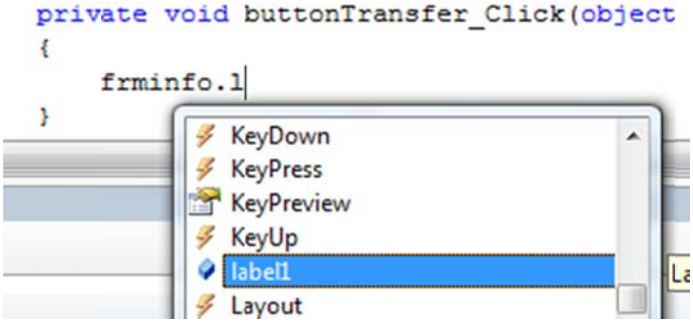


In FormInfo soll mit Druck auf einen Button der Inhalt von *textBoxRadius.Text* von Circle herübergeholt werden. Realisierung folgendermaßen:

Der Zugriff von einer Form auf Properties, Methoden und Events anderer Forms ist möglich, sofern man von der Main Form auf die dort instanzierten Forms zugreifen will. Also im obigen Beispiel kann man von *FormMain* auf *frmCirce* und *frmInfo* zugreifen. Was muss man tun: Beispiel: Auf *FormMain* soll mit einem Button „*buttonTransfer*“ der Inhalt (*.Text*) von l*abel1* auf *FormInfo* verändert werden. Versucht man

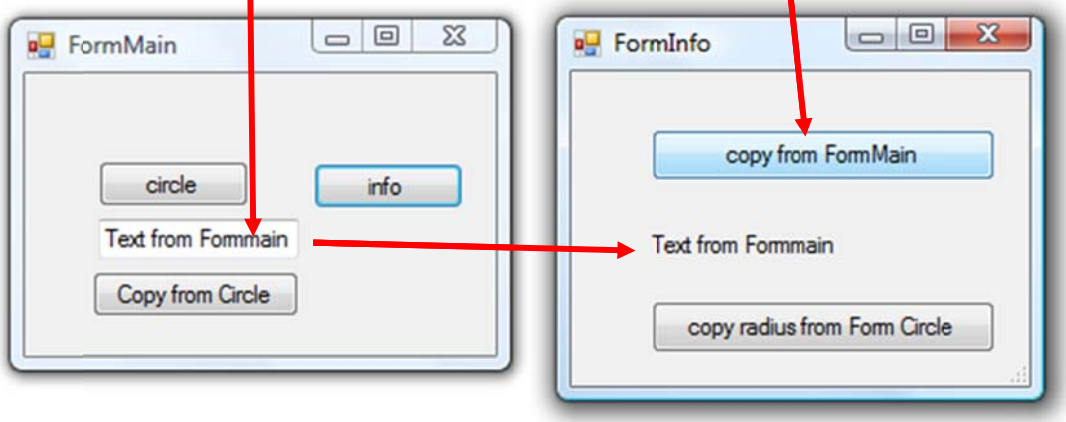
es, so geschieht erst einmal folgendes: Ein Zugriff auf *frminfo.label1* ist nicht möglich und wird im CodeCompletion- Fenster nicht angeboten. Das liegt daran, dass die Komponente *label1* als „private“ deklariert ist. Diese muss auf „public“ geändert werden. Dies ist mit der Property „*Modifiers*“ des *label1*möglich. Also in *FormInfo* diesen *Modifiers* auf *Public* umstellen. Dann ist ein Zugriff auf *label1* möglich.

Jetzt wird in *FormMain* ein Zugriff auf *label1* angeboten. Fertiger Code:




```
private void buttonTransfer_Click(object sender, EventArgs e)
      {
            frminfo.label1.Text = "This Message comes from FormMain";
      }
```

Somit ist also ein Zugriff vom Hauptprogramm, in dem der Konstruktor mit dem Schlüsselwort new aufgerufen wird, problemlos möglich.
Aber wie kann jetzt vom Sub- Fenster (Forminfo oder Circle) auf das FormMain oder von Sub nach Sub zugegriffen werden. Das geht folgendermaßen. Für den Zugriff braucht das Programm in der Sub- Form eine Referenz (ist natürlich so etwas wie ein Pointer / Zeiger). Diese Referenz muss vom Main an die Sub- Fenster übergeben werden.
Dazu wollen wir jetzt folgendes Beispiel realisieren. Ein Tastendruck auf FormInfo soll jetzt den Inhalt einer TextBox auf FormMain auslesen in ein Label. Ein zweiter Tastendruck soll aus Circle den Radius aus der entsprechenden Textbox auslesen und in ein Label in FormInfo schreiben.



In FormMain folgende Veränderung durchführen:	In FormInfo folgende Veränderung durchführen:

`FormInfo frminfo;`
 	
`public FormMain()`
`  {`
`  InitializeComponent();`
`frminfo = new FormInfo(this);`
`}`
	`FormMain meinMain;`
	
	`public FormInfo(FormMain m)`
	`  {`
	`  InitializeComponent();`
	`  meinMain = m;`
	`  }`

In der ersten Zeile wird ein globaler Zeiger auf FormInfo ohne Konstruktor deklariert. Im Konstruktor (nach new) wird jetzt mit Schlüsselwort `this` der Zeiger auf FormMain an den Konstruktor in FormInfo übergeben	In der ersten Zeile wird ein globaler Zeiger auf FormMain ohne Konstruktor deklariert. In der zweiten Zeile wird `this` in *m* übergeben und in der Zeile *meinMain=m* wird der globale Zeiger auf FormMain gestellt. Ab dann ist FormMain über *meinMain* erreichbar.

Mit der Zeile

```
label1.Text = meinMain.textBox1.Text;
```

Kann dann in FormInfo die Textbox auf FormMain ausgelesen werden.

Der Zugriff von FormInfo auf Circle geht jetzt nicht direkt, sondern über FormMain. Will man den Radius von Circle nach FormInfo auslesen, geht das einfach mit

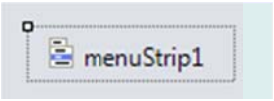
```
label1.Text =meinMain.frmCircle.textBoxRadius.Text ;
```

Das war‘s.

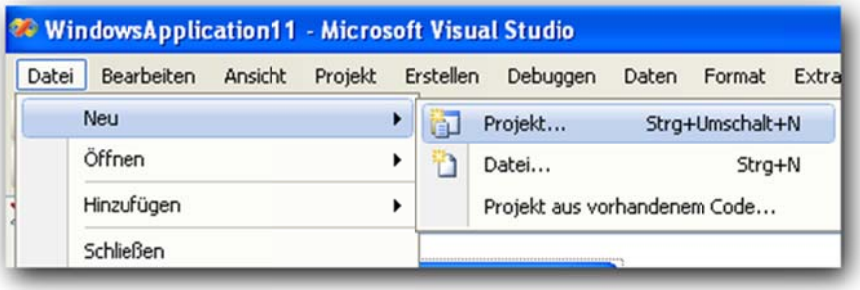
Kurs 3 Professionelles Design

In diesem Kurs werden die Komponenten *StatusBar, Menu, PopupMenu* mit einigen wichtigen Eigenschaften wie ToolTips (früher: *Hint)* vorgestellt.

Menüs



Wir wollen mit dem Ende des letzten Kurses starten. Die Dateien sind verfügbar auf dem FTP- Server *studpub.fh-luebeck.de* unter *bayerlej/MS Studio2008/kurs2b# Multiwindows.*
Wir wollen als erstes dem Hauptfenster *FormMain* ein *MenuStrip* hinzufügen. Diese Komponente ist in der Toolbox „Menüs & Symbolleisten“ zu finden. Jetzt wird mit einem Doppelklick auf diese Komponente der Menü- Designer geöffnet. Hier kann man jetzt seine Menüstruktur bequem definieren.
Wir wollen zur Übung einfach mal die ersten Einträge des Menüs vom C# abkupfern, allerdings vorerst mal ohne Icons. Bevor man ein Menü definiert, muss man sich als Programmierer sehr genau die Bedienphilosophie überlegen. Dieses ist natürlich nicht Gegenstand dieser Vorlesung, wie man das macht. Also erst mal das Menü unter „Datei“ ansehen und dann nach und nach eingeben:

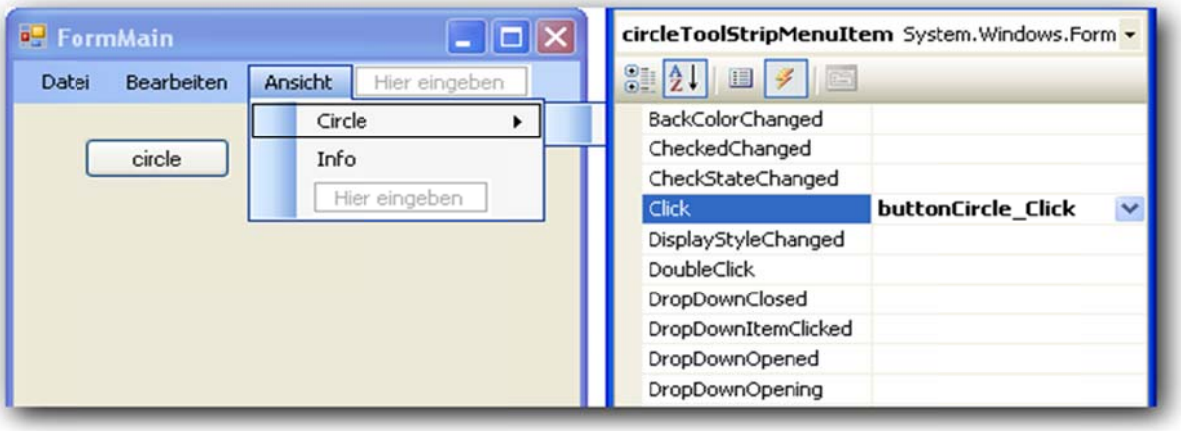


Unter Datei finden sich die Einträge Neu, Öffnen, Hinzufügen, Schließen, Querstrich (einfach Rechte Maustaste, dann Separator einfügen) usw. Man gebe diese Namen sukzessive ein. Das Menü ist im IDE sofort auszutesten. Es fehlen noch die beiden Pfeile hinter Neu und Öffnen, die ein SubMenu öffnen. Dort dann weiter die entsprechenden Menüeinträge eingeben.

Die Verknüpfungen sind einzelne Eigenschaften der Menüeinträge: *ShortcutKeys*. Dort kann man in einer Liste den gewünschten wählen.

Das Menü ist natürlich noch ohne Funktion. Jetzt muss für jeden Menüeintrag natürlich noch ein OnClick- Ereignis definiert werden. Entweder macht man einen Doppelklick auf einen Menüeintrag, so dass die IDE ein Funktionsaufruf erzeugt oder man wählt in der Ereignis-Liste ein vorhandenes Ereignis aus. Wir ergänzen einen dritten Hauptmenüeintrag „Ansicht“ und erzeugen darunter die beiden Menüs „Circle“ und „Info“. Diesen ordne man dann die den bisherigen Tasten *buttonCircle* oder *buttonInfo_Click* zugehörigen Click- Ereignis zu. Dann können wir auch mit dem Menü die beiden anderen Fenster öffnen.

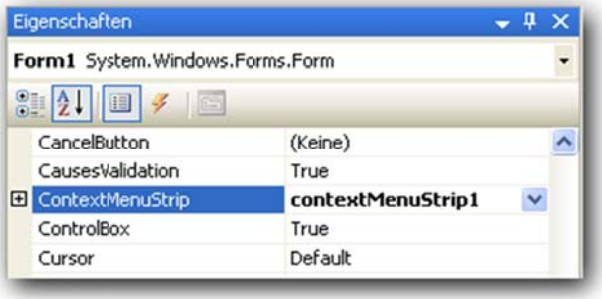
Einige zusätzliche Möglichkeiten: Ein Menü lässt sich als Ein/ Ausschalter verwenden. Dann muss die Eigenschaft *CheckOnClick* auf *True* stehen. Bei jedem Klick auf diesen Menupunkt wird dann die Eigenschaft *Checked* zwischen *true* und *false* umgeschaltet, erkennbar an einem Häkchen vor dem Eintrag.



Jeder Menüeintrag kann wie im Vorbild mit einem Icon versehen werden. Diese Icons werden mit Image einzeln geladen . (*ImageList* aus *Components* funktioniert hier leider nicht wie bei BCB). Für ein Menü sollten diese Icons nicht zu groß sein, sonst sieht es hässlich aus. Ich habe als Beispiel aus einem Windows- Ordner einige Icons mit der Größe 16*16 Pixel zusammengesucht. Diese sind im Ordner Icons dieses Kurses zu finden.

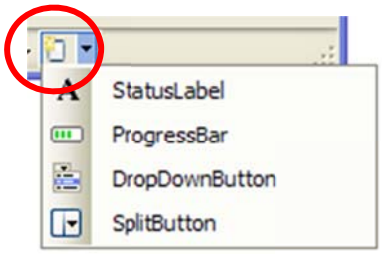
PopUp-Menüs mit ContextMenuStrip

Die Arbeit mit PopUps ist sehr ähnlich. Soll Ihr Projekt solcher Art Menüs bekommen, dann muss der User das natürlich wissen. Er muss dies ja mit einem rechten Mausklick aktivieren. Ich hatte mal ein Toolprogramm mit solchen Popups geschrieben, aber aus Unwissenheit hat keiner diese genutzt. Also jetzt ein *ContextMenuStrip* hinzufügen (aus Menüs & Toolbars). Auch hier kann man leider keine *ImageList* daran knüpfen.

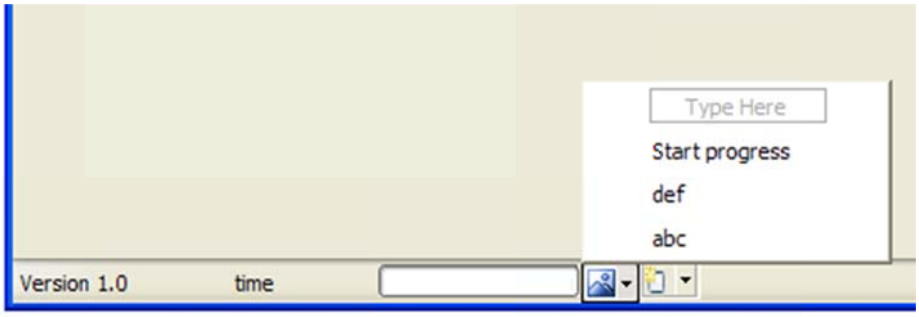


Da nun jede Komponente ein eigenes PopUp-Menü haben kann, muss man Komponenten über die Eigenschaft *ContextMenuStrip* verbinden. Dort kann man vorhandene selektieren. Wir wählen *FormMain*. Nach Programmstart mit F5 kann man mit Rechtsklick auf FormMain das PopUp-Menü aufrufen.

StatusBar mit StatusStrip



Diese Komponente ist bei den meisten üblichen Programmen auch vorhanden. Man findet sie in Menüs & Toolbars. Man ziehe eine auf das Hauptformular *FormMain*. Die Eigenschaft *Dock* steht automatisch auf *Bottom*, somit ist sie gleich an der richtigen Stelle. Will man sie nicht weiter unterteilen in so genannte *Panels*, dann kann man den Ausgabetext in die Eigenschaft *Text* schreiben. Aber meistens nutzt man die Unterteilung in mehrere Abschnitte. Diese kann man hinzufügen mit dem rot umkreisten Symbol. Es werden vier Typen von Abschnitten angeboten. Wir fügen 2 *StatusLabel* zur Ausgabe von Text hinzu, dann eine *ProgressBar* und ein *DropDownButton*. *StatusLabel1* bekommt den festen Text (hier „Version 1.0“in Eigenschaft *Text*), dann im nächsten die aktuelle Uhrzeit (erstmal „time“). In die *DropDownButtons* gebe man ein paar Einträge an, einer sollte „Start Progress“ heißen. Ergebnis:



Mit Doppelklick auf „Start Progress“ geben wir dort folgenden Text in die Click-Funktion ein:

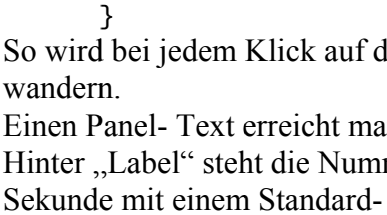
```
private void ghiToolStripMenuItem_Click(object sender, EventArgs e)
      {
            toolStripProgressBar1.Value += 10;
      }
```

So wird bei jedem Klick auf diesen Eintrag zur Laufzeit der Progress- Balken weiter nach links wandern.
Einen Panel- Text erreicht man mit der Syntax `toolStripStatusLabel2.Text`....
Hinter „Label“ steht die Nummer des Panels beginnend links mit 1. Die Uhrzeit wird alle Sekunde mit einem Standard- Timer (in „Komponenten“) im TimerTick- Ereignis- zum Panel geschrieben. Timer- Eigenschaften: *Interval* auf 1000, *Enabled* auf *true*. Vom System erhält man die Systemzeit mit *DateTime.Now*, diese muss dann noch in einen String konvertiert werden: Code:

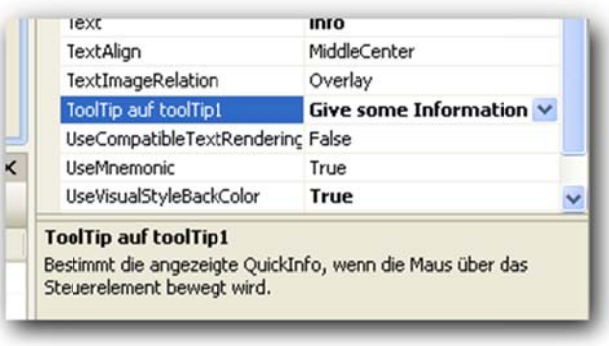
```
private void timer1_Tick(object sender, EventArgs e)
      {
            toolStripStatusLabel2.Text = DateTime.Now.ToString();
      }
```

Hints mit ToolTips

Viele Windows- Standardprogramme wie z. B. Word oder Paint sind mit den kleinen gelben Hinweistexten (=Hints oder Tooltips) ausgestattet, die beim „Hoovering“, also beim Bewegen des Mauszeigers z.B. über Tasten, kurz erscheinen und dem Benutzer Hilfestellung leisten sollen (siehe Beispiel Paint). Dieses wird auch von C# unterstützt. Alle Tooltip-fähigen Komponenten, also alle sichtbaren, haben die Eigenschaft T*oolTip*, in der ein Hilfetext stehen kann, das aber erst, wenn die Anwendung die Komponente *ToolTip* aus „Allgemeine Steuerelemente“ bekommt. Jetzt bei den beiden Buttons irgendeinen sinnvollen Text in *ToolTip on*



toolTip1 eingeben, z. b. in der Info- Taste: Dieser wird mit gelbem Minifenster angezeigt, wenn die Eigenschaft *isBallon* auf *true* steht, in Ballon- Design.



Jedes Programm sollte meiner Meinung eine Möglichkeit haben, diese gelben Dinger auszuschalten. Den erfahrenen User nerven sie nur. Hat das Projekt aber viele 1000 Komponenten mit ebenso

vielen Hint- Texten, ist es mühsam. Will man z. B. alle *Tooltips* der gesamten Applikation an / und ausschalten, dann muss man nur die Eigenschaft *Active* in der Komponente *Tooltip* auf *false* setzen.
Will man per Programm den Inhalt des ToolTips verändern so gibt es abhängig von der Komponente zwei Versionen, wie das geht:

1. Steht im Eigenschaften- Editor „ToolTip on toolTip1“, dann geht das mit der Methode SetToolTip(object,string); Beispiel:

   ```
   toolTip1.SetToolTip(Button1, "Schließen");
   ```

2. Steht im Eigenschaften- Editor ToolTipText, dann einfach z.B.
   ```
   ToolButton11LoadFile.ToolTipText = "Lade Datei";
   ```

Mit folgendem Code kann der ToolTip- Text auch z.B. in der Statusleiste angezeigt werden:

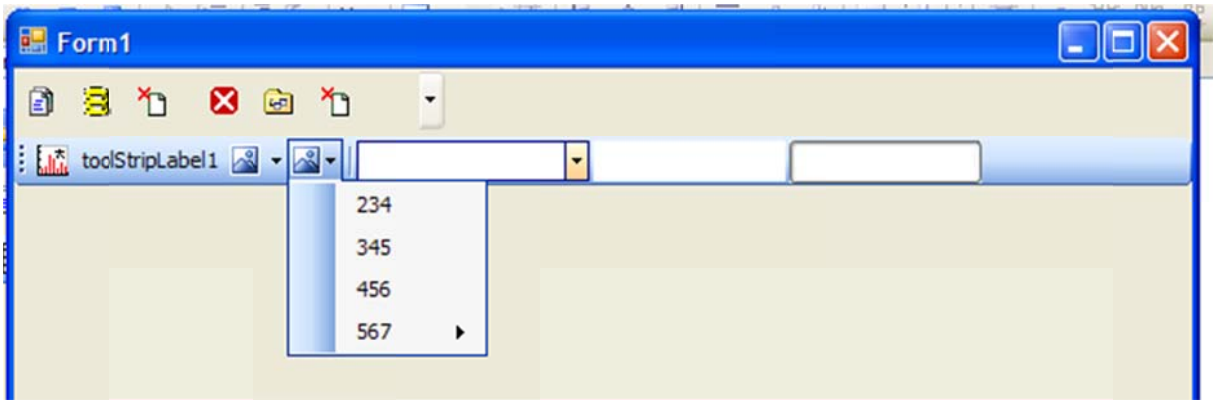
```
private void toolTip1_Popup(object sender, PopupEventArgs e)
        {
            toolStripStatusLabel1.Text = toolTip1.GetToolTip(e.AssociatedControl);
        }
```

Das geht aber leider nicht, wenn die toolTips inaktiv sind (Active=false). D.h. wenn einer diese ToolTips abgeschaltet hat, steht auch nichts in der Statusleiste.

Kurs 4 weitere Komponenten

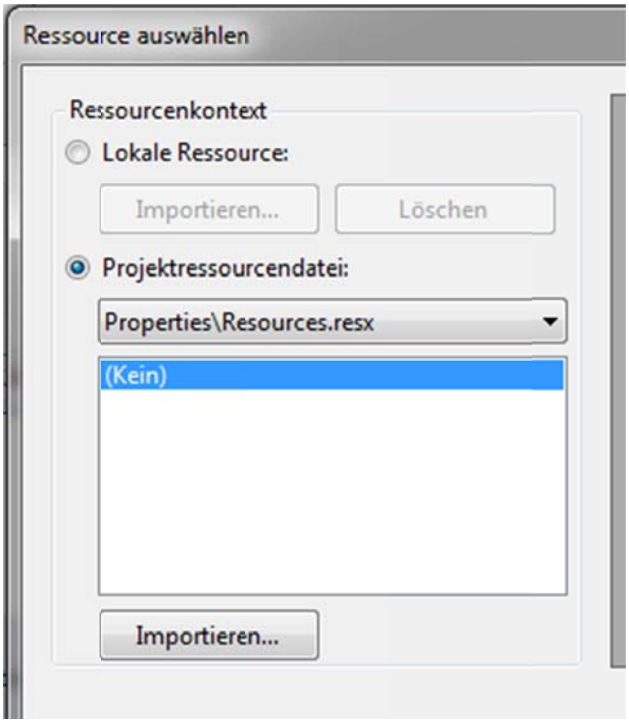
ToolStrip

Start dieses Kurses mit leerer neuer Anwendung.
Will man auch eine *ToolBar* wie z. B. in Word haben, so ist das leicht möglich. Ich hab sie im unteren Bild mal in der ersten Zeile hinzugefügt. Sie ist aber veraltet. Moderner ist die Komponente ToolStrip. Wir fügen sie hinzu.

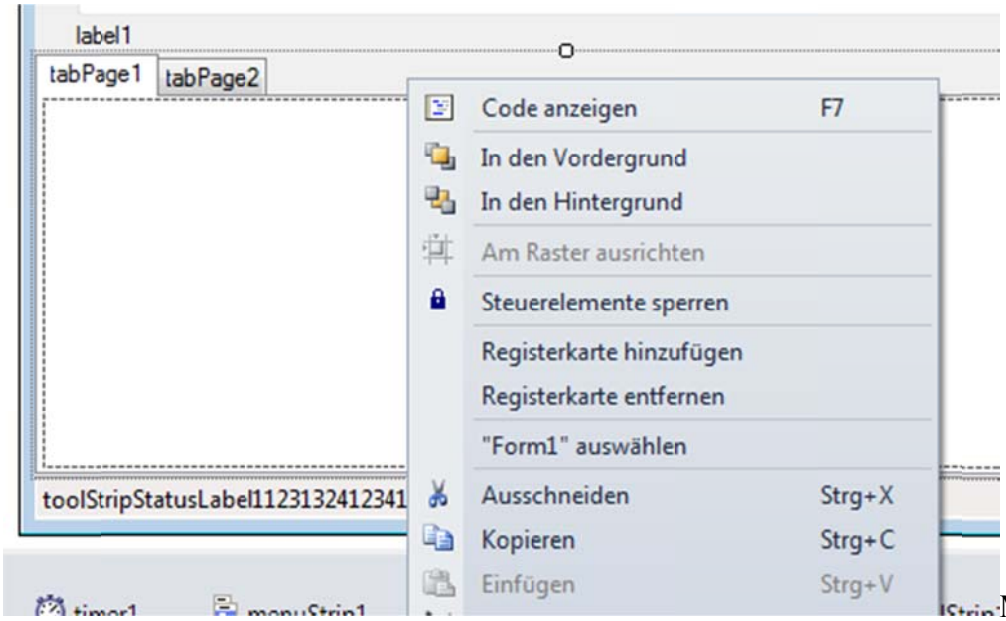


Dort kann man (siehe jetzt im Bild in der zweiten Zeile) mit hübschen Verlaufsfarben ein modernes Design bekommen. Von links nach rechts sichtbar die Möglichkeiten der Elemente: toolStripButton (normale Taste wie *Toolbar*, jedoch ohne *ImageList*- Möglichkeit, sondern mit Einzelbildchen), *toolStripLabel* für Textausgabe / Beschriftungen, dann *toolStripSplitButton* und *toolStripDropDownButton*, mit denen man ein Menübaum designen und öffnen kann. Den Unterschied hab ich noch nicht erkannt, erscheint mit identischer Funktion. Dann sind noch *toolStripTextBox* und *toolStripProgressBar* möglich mit entsprechender Funktion. Da ist XP – und Vista- Design leicht möglich. Hinzu kommt eine ComboBox- Möglichkeit.

Beim Hinzufügen von Images zwei Hinweise: Zum Einen muss die Bildgröße passend sein zur Bildgröße Imagesize in der ToolStrip- Komponente, dann sollte sie als png oder gif mit transparenter Außenfarbe versehen werden.
Zum anderen muss man sich überlegen, ob diese Bildchen nicht bei größeren Projekten mehrfach Verwendung finden. Dann sollte man sie in die „Projektresourcendatei“ einfügen und steht dann überall im Projekt zur Verfügung. Nach Klick auf eigenschaft „Image“ des Buttons dann so:

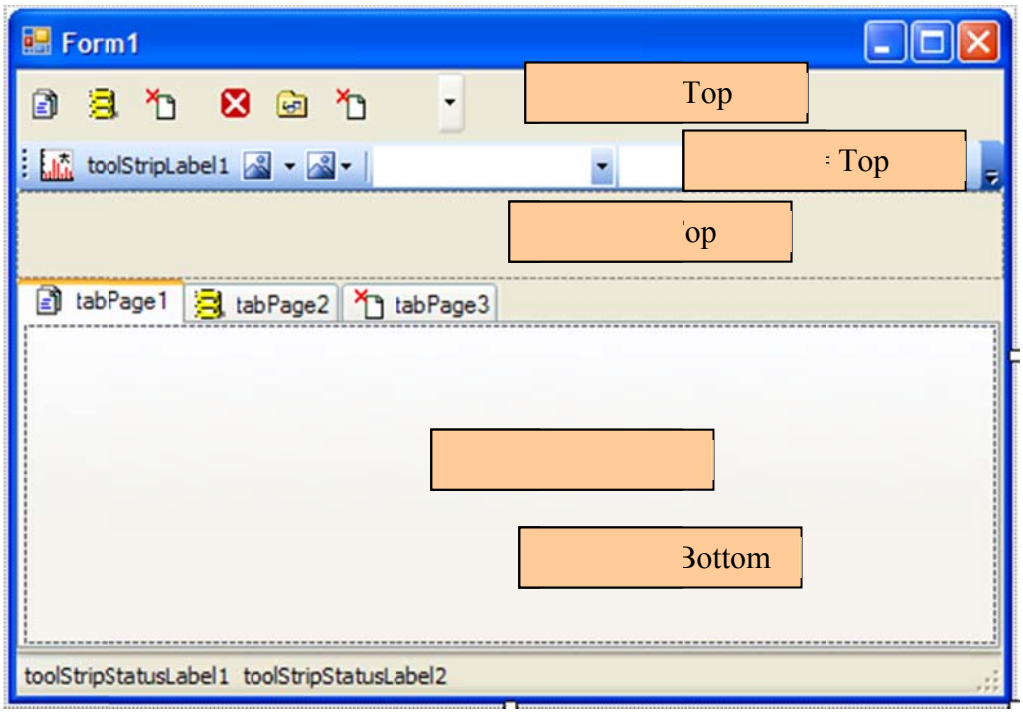


TabControl



Mit der *TabControl*- Komponente können Sie in einem Fenster die Oberfläche mehrfach nutzen. Das Umschalten geschieht wie in einem Karteikasten. So kann man solche Oberflächen wie in typischen Windowsprogrammen programmieren.
Also aus dem Folder Container eine *TabControl* auf das Formular ziehen. Normalerweise wird diese Komponente nach unten gedockt. Damit automatisch bei Fenstergrößenänderung die einzelnen Elemente sich vernünftig mit ändern, sollte man folgende Strategie anwenden (hab

noch ein Panel als reservierte Fläche für Tasten oder andere Elemente auf das Formular gezogen):

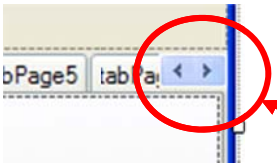


Oben *ToolBar* auf *Top*, dann *ToolStrip* auf *Top*, dann *Panel* (eigener Container aus „Containers“) wieder *Top*, der klebt dann unter dem *ToolStrip*.
Dann *StatusStrip* auf *Bottom* und zum Schluß *TabControl* auf *Fill*.
Nun wieder zurück zur *TabControl*.

Neue Pages werden mit rechtem Mausklick→ *PopUpMenu* erzeugt, dort „*Registerkarte hinzufügen*“. Bitte jetzt mal 7 neue Pages erzeugen. Diese heißen automatisch tabpage1 bis 7. Jetzt achte man darauf, wie man einzelne Pages im IDE anwählt. Erst Klick auf den Anfasser (Tab), dann ist aber die *TabControl* angewählt. Dann nochmal unten auf die Seite klicken. Man achte auf die weißen kleinen Punkte. Erst dann ist die Page selektiert (Kontrolle im Eigenschaftenfenster).

TabControl selektiert

Page selektiert



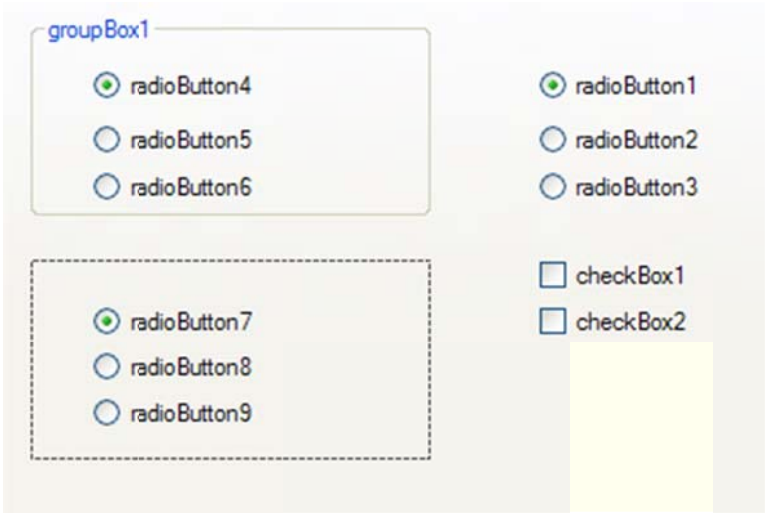
Multiline = false

Multiline = true

Reicht die Seitengröße für alle Pages nicht aus, so erscheinen die von anderen Anwendungen gewohnten beiden Pfeile zur Verschiebung der Anfasser. Will man bei zu engem Platz mehrere Zeilen für die Anfasser vorsehen, so setze man in der *TabControl* (erst selektieren!) die Eigenschaft *MultiLine* auf *true*.
Hier müssen Images per *ImageList* ergänzt werden, die Auswahl der Icons in den einzelnen Seiten wieder mit der Eigenschaft *ImageIndex*. Also vorher eine *ImageList* auf die Form ziehen und mit einer Liste von Bildchen füllen. Die Größe der Icons in ImageList einstellen.

Komponenten zur Gruppierung und Selektion

Jetzt sollen Komponenten zur Gruppierung und Selektion gezeigt werden.
Wir fügen auf *tabPage1* hinzu (alles Standard):

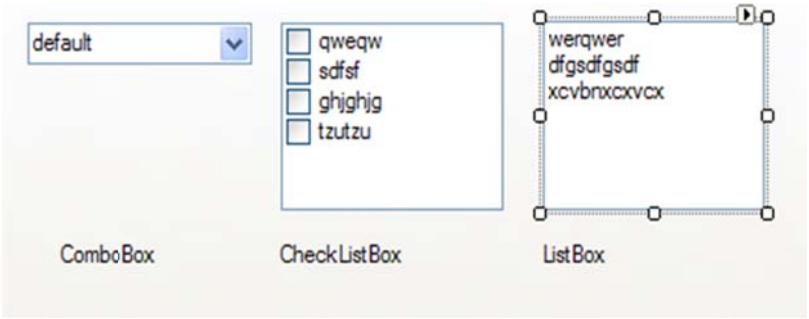


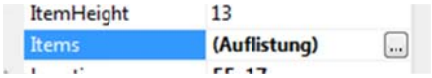
1 GroupBox
1 Panel

Auf tabPage1 3**RadioButton1*-3
Auf *groupBox1* 3**RadioButton4*-6
Auf das *Panel*3* *RadioButton7*-9

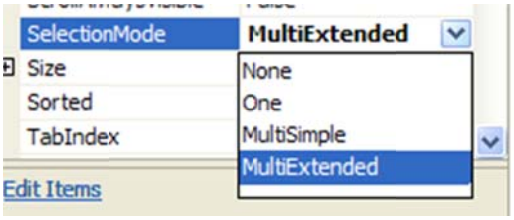
Bewegt man nun einen „Container“, z. B. das *Panel*, dann bewegen sich die darauf befindlichen Elemente mit. Zuordnung geht mit Hinenschieben,
Die ***CheckBox***-en lassen eine einzelne Auswahl zu. Ob angewählt oder nicht, ist in Eigenschaft *Checked* zu sehen.
Die ***RadioButton*** haben auch eine Eigenschaft *Checked*, jedoch gilt hier: auf einem Container kann nur immer ein *RadioButton* angewählt sein. Wählt man ein anderes, wird automatisch der andere deselektiert. Dieser Zusammenhang besteht nicht zwischen *RadioButtons* anderer Container, wodurch der nützliche Sinn der Container wohl hier zu erkennen ist.
Panel und *GroupBox* unterscheiden sich eigentlich nur in der Überschrift der *GroupBox*. Das Aussehen des Randes von *Panel* kann man leider nicht gleich dem der *GroupBox* machen, dann nehme man halt eine Group- Box und lösche die Beschriftung in *Text*.
Auf *tabPage2* weitere Selektionsmechanismen:

Weitere Selektionskomponenten sind *ListBox*, *ComboBox* aus Standard und *CheckListBox* aus „Allgemeine Steuerelemente“. Alle drei Komponenten enthalten als Textinhalt eine *Collection* oder „*Auflistung*“ namens *Items*. Alle Funktionen, die mit *Collections* getan werden können, sind auch hier möglich (dazu siehe späteres Kapitel). Füllen der einzelnen selektierbaren Zeilen mit dem Collectioneditor im Eigenschaftenfenster unter *Items*. Hier mal ein paar Zeilen Quatschtext in alle drei Komponenten eingeben.





Bei der ***ComboBox*** sieht man das selektierte Element oben im Edit- Feld. Es benötigt nur wenig Platz, man muss es halt mit einem zusätzlichem Mausklick aufmachen.



Bei der ***ListBox*** sieht man alle Möglichkeiten, wenn die Komponente groß genug ist. Sonst gibt es einen Scrollbalken. Mit der Eigenschaft *Sorted=true* lassen sich die Einträge alphabetisch sortieren. Wenn

SelectionMode auf *One* steht, dann kann man nur ein Element selektieren. Bei *MultiSimple* wird jede Zeile selektierbar und die Selektion *toggelt*, bei *MultiExtended* selektiert man wie beim Explorer gewohnt, die Shift und Strg- Tasten sind dann auch möglich.
Die Abfrage, welche Zeile selektiert ist, kann man mit der Eigenschaft *SelectedIndices*[] durchführen. Es ist eine Eigenschaft vom Typ int- Array und ist im Eigenschaftenfenster nicht zu sehen. Jede Zeile startet bei Nr. 0. Ist z.B. Zeile 3 selektiert, dann ist der Wert von `SelectedIndices.Count` gleich 1 und in *SelectedIndices*[0] steht die Nummer der Zeile, also 2. Dieses mache man sich deutlich mit folgendem einfachen Code in dem Ereignis `SelectedValueChanged`.

```
private void listBox1_SelectedValueChanged(object sender, EventArgs e)
      {
            label1.Text = listBox1.SelectedIndices[0].ToString();
            label2.Text = listBox1.SelectedIndices.Count.ToString();
      }
```

In *label1* wird die Nummer der ersten selektierten Zeile und in *label2* die Anzahl der selektierten Zeilen dargestellt.

Die ***CheckListBox*** ist ähnlich, nur man sieht vor jeder Zeile ein *CheckBox*- Feld. Diese kann man wie bei einer *CheckBox* bei jedem Klick toggeln. Abfrage nun allerdings über *CheckedIndices*[], auch hier ein Int Array mit den Zeilennummern. Auch ein kleines Programm dazu: Nur wenn die dritte Zeile einen Pfeil trägt, ist die Farbe der *tabPage* rot, sonst weiß.Code:

```
private void checkedListBox1_SelectedIndexChanged(object sender, EventArgs
e)
      {
            if ((checkedListBox1.CheckedIndices.Count==1)&&
                (checkedListBox1.CheckedIndices[0] == 2))
                tabPage2.BackColor=Color.Red;
            else tabPage2.BackColor=Color.Transparent;
      }
```

Der Zugriff auf den Text der Zeilen geht mit .Items[0] wie bei einem Array, die Nummer wird einem NumericUpDown der Eigenschaft Value entnommen :

```
label2.Text = listBox1.Items[(int)numericUpDown1.Value].ToString();
label3.Text = checkedListBox1.Items[(int)numericUpDown1.Value].ToString();
```

Bei der CheckListBox kann man das Kästchen mit folgendem Befehl direkt setzen (true) oder zurücksetzen (false):

```
checkedListBox1.SetItemChecked((int)numericUpDown1.Value, true);
```

Abfragen mit

```
checkedListBox1.GetItemChecked((int)numericUpDown1.Value)
```

liefert bool- Wert zurück.

Ergebnis: Siehe *Kurs4# weitere Kompos.*

Kurs5 Textverarbeitung mit Stringlisten

Das Visual C# - Entwicklungswerkzeug verfügt über mächtige Objekte zur Textverarbeitung. Diese sind teilweise einfach aus der Verwandtschaft zu Borland entstanden. Ansi-C kennt nur die Zeichenketten in Form von Char- Arrays, Textverarbeitung damit wird sehr mühsam.
Zum Start dieses Kurses öffnen Sie bitte eine neue Applikation und fügen eine *TabControl* hinzu, Dock=*Bottom*, dann 4 pages hinzufügen. Namen alle so lassen. Abspeichern, bei mir heißt diese „Projektmappe“ WindowsApplication14.

Der Typ String

Auf der ersten Seite *TabPage1* werden zum Testen einiger Stringeigenschaften 8 Label untereinander plaziert. Mit Doppelklick auf *Label1* eine Click-Event-Funktion erzeugen, in der einiges gemacht werden soll. Die Tooltip- oder *Text*- Eigenschaften der Komponenten sind auch Eigenschaften vom Typ s*tring*. Will man eigene Variablen deklarieren, so verwende man den Typbezeichner *string* (wohlgemerkt mit kleinem s!), S*tring* gibt es auch, ist Name der Klasse!).
Als erstes folgende Zeilen in diese *label1Click()* eingeben, grüner Text kann natürlich weg, dient nur zur Info, was nicht geht.

```
private void label1_Click(object sender, EventArgs e)
{
//char a[20]="12345678901234567890"; funktioniert in C# nicht
char[] a ={ '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '1', '2' };
string b = new String(a);
label1.Text=b;  // ...=a; // ...=a.ToString(); funktioniert nicht
String...
}
```

Es wird in der ersten Zeile ein *Char-Array* deklariert und initialisiert. In der zweiten Zeile wird dieses Array dem *String b* zugewiesen. Erst dann kann man diese Zeichenkette nach *Text* von *Label1* zuweisen.
Die Zuweisung anders herum geht allerdings so nicht. Versucht man dem *char- Array* einen String zuzuweisen, geht es schief, da "a" ein Zeiger ist. Diese Zuweisung geht nur mit der Methode *ToCharArray* etwa so:

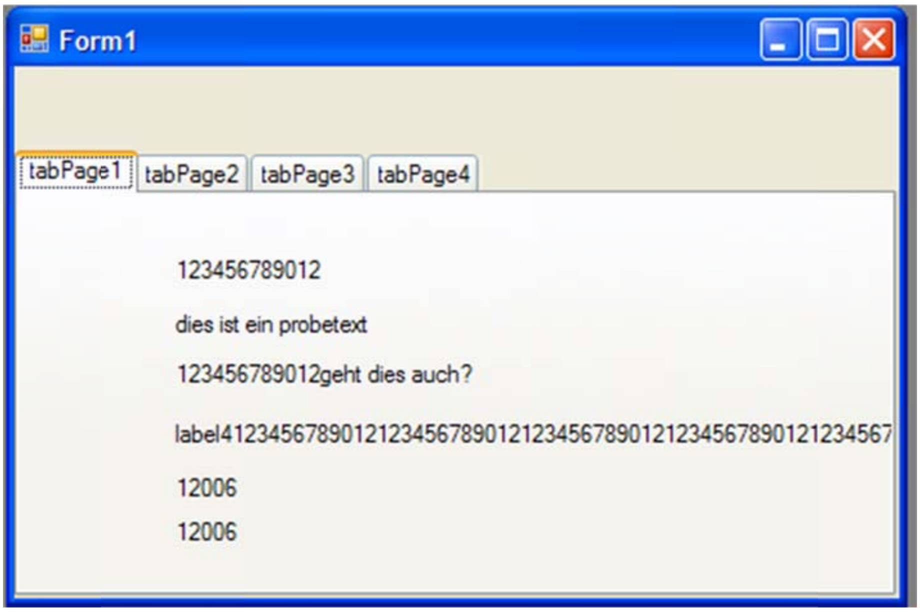
```
a = b.ToCharArray();
```

Der ganz große Nachteil von *Char- Arrays* in Standard herkömmlichen C ist die statische Größe. Das ist jetzt in C# anders. Das *Char- Array* wird mit dynamischer Größe gemanaged!!! Siehe dazu folgender Code:

Die Länge eines Strings wird auch dynamisch verwaltet. Er kann beliebig lang sein. Beispiel: Eine Schleife addiert immer 20 Zeichen hinzu. Der Operator „+“ ist ein einfacher Verkettungsoperator, der zwei Strings zu einem neuen einfach verknüpfen kann. Folgender Code liefert folgendes Ergebnis:

```
char[] a ={ '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '1', '2' };
string b = new String(a);
label1.Text=b;  // ...=a; // ...=a.ToString(); funktioniert nicht
 a = b.ToCharArray();
 label2.Text = "dies ist ein probetext";
 label3.Text = b + "geht dies auch?";
for (int i = 0; i < 1000; i++)
     label4.Text = label4.Text + b;
 label5.Text = label4.Text.Length.ToString();
```

```
a = label4.Text.ToCharArray();
label6.Text = a.Length.ToString();
```



Der String b mit 12 Zeichen wird 1000 mal zu label4.Text hinzuaddiert, dieser hat anschließend (mit *Length*) ermittelt 12006 Zeichen. Anschließend wird label4 in das Char-Array kopiert, auch dieses hat dann die 12006 Zeichen.

Wichtige Methoden der Strings sind noch *.Substring(x,y)*, wobei hier ein Teilstring als Ergebnis herauskommt mit Startzeichen bei x anfangend mit 0 und Länge y. Also liefert

```
label7.Text = label2.Text.Substring(13, 5);
```

als Ergebnis das Wort "probe". Die Methode *IndexOf*(„suchstring") kann die Position eines Teilstrings suchen. So liefert die Zeile

```
label8.Text = label2.Text.IndexOf("probe").ToString();
```

die Zahl 13, da in label2 der Teilstring „probe" bei Zeichen 13 beginnt.

Weitere Methoden finden Sie in der Hilfe: Cursor auf das Wort string, dann F1:
ToLower() ersetzt alle Großbuchstaben durch kleine
ToUpper() dito durch Großbuchstaben
Replace() ersetzt einen Teilstring.

Fazit strings: niemals wieder sich mit Char-Arrays herumquälen.

Der mehrzeilige Text

Die StringList direkt wie bei Borland gibt es in C# oder .NET nicht. Aber ähnliche Strukturen sind möglich. So gibt es die *Klasse Collections*, die als *Collection* beliebiger Typen arbeiten kann, also auch für strings. Dann sind eine Vielzahl von Methoden möglich wie Insert(), Remove() Contains() Clear() usw. Man kann es dann einfach als ein dynamisch verwaltetes Array von AnsiStrings betrachten. Jede Zeile dieser *Collection* ist wieder ein beliebig langer String mit allen Methoden wie oben beschrieben. Ab .NET V3 gibt es auch die ArrayList mit ähnlichen Methoden und Eigenschaften. Man kann aber auch einfach ein Stringarray benutzen:

```
string[] abc = new string[5];
```

Nur ist die Anzahl der Elemente dann fest auf 5 und kann nicht verändert werden.

Besser wäre dann die Collection:
```
Collection<string> strcoll = new Collection<string>();
```
Fügt man diese Zeile in C# ein, so kennt es erstmal das Schlüsselwort Collection nicht, es ist schwarz. Dann kann man aber mit rechtem Mausklick „Auflösen“ wählen und er fügt automatisch die fehlende *using* hinzu, in diesem Fall `using System.Collections.ObjectModel;`

Hinzufügen eines Elementes mit : `strcoll.Add("12345");`

Zugriff wie Array mit `label2.Text = strcoll[(int)numericUpDown1.Value];`
Die Größe wird dynamisch verwaltet.

Nun zu Komponenten mit mehrzeiligem Text:
Die *TextBox* wird mehrzeilig, wenn man die Eigenschaft *Multiline* auf *true* stellt. Wir ziehen eine *textBox* auf t*abPage2*, und ein *RichTextBox* auf *tabPage3* und setzen beide *Dock=Fill*.
Schauen wir uns zuerst einmal *textBox1* an. Einige wichtige Eigenschaften:

- ->*ReadOnly* bestimmt, ob der User Text verändern kann (false) oder nur angezeigt wird (true)
- ->*Font* bestimmt Schriftart Farbe etc
- ->*ScrollBars* bestimmt, ob diese vorhanden sind. Es sollte immer eine Vertikale ScrollBar eingestellt werden
- ->*WordWrap*, wenn true, dann werden längere Zeilen umgebrochen, sonst verschwinden sie rechts, sind mit Scrollbar aber zu erreichen: ich würde immer auf false stellen, dann bleiben die Zeilenumbrüche von Originaltexten erhalten.
- *Maxlength,* steht defaultmäßig auf 32767, begrenzt die Anzahl der Zeichen des Users auf diese Zahl. Sollte auf 0 stehen, dann ist die Grenze bei 2^32 = 4GByte Zeichen.

Die Vorbelegung des Textbereiches geschieht per Eigenschaftenfenster bei der Eigenschaft *Lines*. Damit öffnet sich ein Texteditor zum Eingeben des Starttextes.
Wichtig zu wissen: Der Inhalt wird immer in einem einzigen string, nämlich Text gespeichert. Ruft man die Eigenschaft .Lines auf, so wird intern der Einzelstring in ein Textarray umgebrochen, was bei großen Texten sehr zeitraubend sein kann. Will man oft auf große Texte per Lines zugreifen, dann am Besten vorher einmalig einen Umbruch erzwingen und dann auf das lokale Textarray zugreifen:
```
string [] LocalArray = textBoxRadius.Lines;
```
Wird der Text verändert, muss man dieses natürlich wiederholen.

Für das Programm in diesem Kurs sind zum Laden und Speichern noch zwei Dialoge wichtig: in Dialogs findet man die Komponenten *OpenDialog* und *SaveDialog*. Bitte jetzt zwei Tasten oben auf die Form, Name Load und Save. Dann die beiden Dialoge hinzufügen.
In der Click- Funktion von *Load* folgt jetzt der Standard- Code zum Öffnen von Datei-Auswahl- Boxen, die dem üblichen Windows- Design entsprechen:

```
if (openFileDialog1.ShowDialog() == DialogResult.OK)
      {  // ……. Öffnen der Datei, Einlesen Abspeichern etc
      }
```
Die Methode *ShowDialog*() öfffnet das typische Auswahlfenster. Nur wenn der User auf die OK- Taste klickt, wird die Aktion ausgeführt, bei Taste Abbruch nicht.

Das Laden einer Text-Datei in die Textbox geschieht nun über die *Klasse StreamReader* mit folgendem Code: (vorher oben mit using…):
```
using System.IO;

StreamReader myStream1 = new StreamReader(openFileDialog1.OpenFile());
textBox1.Text = myStream1.ReadToEnd();
```

```
myStream1.Close();
```

Mit der Methode *ReadToEnd()* wird dann der Textinhalt der Datei (muss eine ASCII sein) in die *textBox* eingelesen. Problem mit dem Text gibt es allerdings mit den deutschen Sonderzeichen ü, ö ä ß, die einfach ausgelassen werden. Dieses kann man verhindern, wenn man die Instanziierung nach new mit folgender Ergänzung durchführt:

```
...new StreamReader(openFileDialog1.OpenFile(),Encoding.Default);
```

Tipp: Bei großen Texten dauert insbesondere das "scrolling" der sich füllenden textbox sehr lange, besser vor Einlesen Textbox Visible auf false, danach wieder auf true. Geht so viel schneller.
Abspeichern mit der Save- Taste geht so:

```
if (saveFileDialog1.ShowDialog() == DialogResult.OK)
 {
  StreamWriter myStream1 = new
  StreamWriter(saveFileDialog1.OpenFile(),Encoding.Default);
  myStream1.WriteLine(textBox1.Text);
  myStream1.Close();
}
```

Die Dialoge sollten noch etwas angepasst werden: im *openDialog1* folgende Eigenschaften setzen:
Auf jeden Fall sollten die *Filter* gesetzt werden, z. B. so wie links. Immer sollte ein Filter „Alle" mit *.* dabei sein, damit man immer sehen kann, was alles auf dem Verzeichnis vorhanden ist. Die Syntax ist: (*Bezeichner*)|*.xxx|(*Bezeichner*)|*.yyy usw. Also für Text und C#- Files z. B.
Filter=*Text*|**.txt*|*C#*|**.cs*|*All*|*.*.
In *InitialDir* kann das Startverzeichnis gewählt werden, in *Title* die Überschrift der *FileSelectBox* und in *FileName* der voreingestellt Name.

Ganz wichtig in der *SaveDialog*- Komponente ist *DefaultExt*, die den Suffix bestimmt, wenn der User keinen eintippt. Dort sollte also immer *.txt* stehen, dann wird die Datei ohne Angabe eine txt- Datei.
Ganz wichtig: der *OverwritePrompt* in Options sollte immer auf *true* gestellt werden, damit der User vor Überschreiben gewarnt wird.

```
label9.Text = "Char= "+textBox1.Text.Length.ToString();
label10.Text = "Lines= " + textBox1.Lines.Length.ToString();
```

Mit den beiden oberen Zeilen werden sowohl Anzahl der Zeichen in *Label9* und Anzahl der Zeilen in *Label10* ausgegeben. *Lenght* ist eine Eigenschaft vom typ int.

Man kann jetzt einzelne Zeilen erreichen durch Ansprechen der Eigenschaft *Lines* in der *Textbox.*
Dabei benutzt man die Array- Indizierung mit [ ]. Folgender Code liest die Zeile in eine zweite *textBox2*, dessen Zeilennummer startend mit Null von einem *numericUpDown* eingestellt wird. Um Zugriffsfehler zu vermeiden, sollte vorher immer der Maximalwert des *UpDown* auf die Zeilenzahl *-1* gestellt werden.

```
private void numericUpDown1_ValueChanged(object sender, EventArgs e)
{
            numericUpDown1.Maximum = textBox1.Lines.Length-1;
```

```
            textBox2.Text = textBox1.Lines[(int)numericUpDown1.Value];
}
```

Die *textBox* kann nur alle Zeichen mit einen einzigen Font darstellen, die RichTextBox dagegen lässt Formatierungen für jedes einzelne Zeichen zu.

Nun noch ein bisschen Textverarbeitung:

Sortieren

Es sollen alle Zeilen alphabetisch sortiert werden. Nun lässt die *Textbox* dies direkt nicht zu. Also muss man den Umweg über z.B. die *Listbox* machen, die kann sortieren. Eine *Collection* leider auch nicht. Damit das nicht sichtbar wird, kann man die *Listbox* mit *Visible=false* unsichtbar machen. Und natürlich in der *Listbox die Eigenschaft Sorted=true*.
Source hinter einer Taste Sort:

```
1 listBox1.Items.Clear();
2 for (int i = 0; i < textBox1.Lines.Length; i++)
3     {
4           label10.Text = "Lines= " + i.ToString();
5           listBox1.Items.Add(textBox1.Lines[i]);
6           Application.DoEvents();
7     }
8 textBox1.Clear(); textBox1.Visible = false;
9 for (int i = 0; i < listBox1.Items.Count-1; i++)
10   textBox1.AppendText(listBox1.Items[i] + "\x0d" + "\x0a");
11textBox1.AppendText(listBox1.Items[listBox1.Items.Count - 1].ToString());
```

Kommentar dazu:

Zeile 1: Zwischenspeicher Listbox leeren.
Zeile 2: for- Schleife zum Umschaufeln und gleichzeitigem Sortieren
Zeile 4: Fortschrittsanzeige, sinnvoll bei langen Texten, dann dauert es sehr lange
Zeile 5: Umschaufeln
Zeile 6: Dieser Befehl arbeitet alle Ereignisse ab, die während der Schleife auflaufen. Damit wird auch ein Neuzeichnen des *label10* erzwungen, sonst würde man den Fortschritt nicht sehen.
Zeile 8 Start Zurüchschaufeln mit Leeren. Gleichzeitig wird die Textbox unsichtbar gemacht, um das sehr zeitaufwendige Scrollen zu vermeiden.
Zeile 9: Schleife zum Zurückschaufeln
Zeile 10: Zurückschaufeln, leider hat Textbox kein .Add(), geht nur mit TextAppend, dann muss aber Zeilenende mit CR und Linefeed per Escape Sequenz programmiert werden.
Zeil1 11: Die letzte Zeile des Textes soll kein Linefeed bekommen.

Leerzeilen weg

Code hinter einer Taste mit Namen „Leerzeilen weg“. Es wird als Zwischenspeicher hier mal eine „Collection“ benutzt. Ginge aber auch mit Listbox wie oben.

```
Collection<string> strcoll = new Collection<string>();
for (int i = 0; i < textBox1.Lines.Length; i++)
if (textBox1.Lines[i].Length > 0) strcoll.Add(textBox1.Lines[i]);
      textBox1.Clear();// copy Back to Textbox
for (int i = 0; i < strcoll.Count - 1; i++)
      textBox1.AppendText(strcoll[i] + "\x0d" + "\x0a");
textBox1.AppendText(strcoll[strcoll.Count - 1]); // last line
```

Größe von StringListen

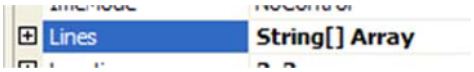
Die Größe ist im Prinzip nicht begrenzt. Wir fügen Taste hinzu, die *textBox1* über *ListBox1*immer verdoppelt. Also kopiere *textbox1* nach *Listbox1* und zurück ohne zu sortieren und Textbox zu leeren. Anzeige der Zeilen in *Label9* und der Zeichenzahl in *Label10*:

```
listBox1.Items.Clear(); listBox1.Sorted = false;
for (int i = 0; i < textBox1.Lines.Length; i++)
    {
      label10.Text = "Lines= " + i.ToString();
      listBox1.Items.Add(textBox1.Lines[i]);
     Application.DoEvents();
    }
textBox1.Visible = false;
for (int i = 0; i < listBox1.Items.Count - 1; i++)
      textBox1.AppendText(listBox1.Items[i] + "\x0d" + "\x0a");
textBox1.AppendText(listBox1.Items[listBox1.Items.Count - 1].ToString());
textBox1.Visible = true;
label9.Text = "Char= " + textBox1.Text.Length.ToString();
label10.Text = "Lines= " + textBox1.Lines.Length.ToString();
```

Beschleunigen eventuell mit

```
string [] LocalArray = textBoxRadius.Lines;
```

TextBox und *RichTextBox* speichern den Text in einem *String[] Array* namens *.Lines*. Es gibt zwar eine Methode *.CopyTo()*, aber der Befehl



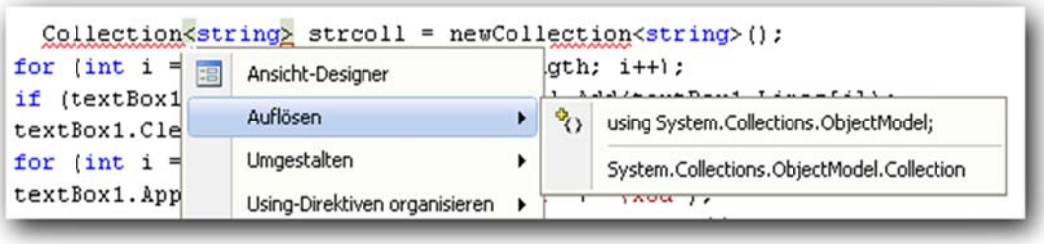

```
textBox1.Lines.CopyTo(richTextBox1.Lines, 0);
```

wird ohne Fehler kompiliert, aber es passiert nicht das erwartete Kopieren, **sondern nichts**!!.

ListBox hat seine String in *Items*, das scheint eine String Collection zu sein. Dort gibt es eine *CopyTo()* – Methode, sie funktioniert aber nicht so wie man das als User wünscht.


```
Collection<string> strcoll = new Collection<string>();
listBox1.Items.CopyTo(strcoll, 0);// syntax error
```

wird mit einer Fehlermeldung abgebrochen und nicht kompiliert. Wieso nicht??



Noch ein Tip, wie man herausbekommt, welche Using – Anweisung nötig ist. Gibt man die oberste Zeile mit „Collection“ ein, so ist das Wort zunächst schwarz, da die nötige Using – Anweisung fehlt. Mit Rechtsklick auf das Wort kann man im PopUp unter „Auflösen“ die passende using- Anweisung „System.Collections.ObjectModel“ automatisch hinzufügen lassen.

Leider auch hier: eine Kopiermethode wie *AddStrings* oder meinetwegen auch CopyTo() von String[] Array nach Collection gibt es nicht. Dabei sind die Daten einfach nur mehrzeilige Texte.

Wörter zählen

Noch ein Codebeispiel, das oft nützlich ist: das Auffinden von Teilstrings. Damit kann man z.B. zählen, wie oft ein Wort in einem Text vorkommt. Dazu ist die Methode .Substring(a,b) der Strings noch zu beschreiben. Sie liefert einen String zurück, der im Ausgangsstring bei Buchstabe an Position a (Start von Null) beginnt und b Zeichen enthält. Also nach Ausführung folgenden Codes

```
label2.Text = "dies ist ein probetext";
label7.Text = label2.Text.Substring(13, 5);
```

steht in *label7* das Wort "probe". Ein Zeilenumbruch in Text sind zusätzlich noch einmal 2 Zeichen (\n\r).
Hinter der Taste z.B. „Item Search" steht folgender Code, der in Textbox1 nachzählt, wie oft dort der Begriff vorkommt, der in Textbox2 steht.

```
string item = textBox2.Text;
int len=item.Length;
int cnt = 0;
for (int j = 0; j < textBox1.Text.Length-len+1; j++)
   if (textBox1.Text.Substring(j, len) == item) cnt++;
      label9.Text = "Hits= " + cnt.ToString();
```

Das dauert aber leider bei großen Texten sehr lange. Sucht man in Bibel.txt (Auf ftp bei mir zu finden) nach dem Wort „Gott", so dauert es Stunden.
Etwa 5 sec dauert gleiche Suche mit folgendem leicht modifiziertem Programm:

```
string item = textBox2.Text;
string text = textBox1.Text;
int len=item.Length;
int textlen=text.Length-len+1;
int cnt = 0;
for (int j = 0; j < textlen; j++)
      if (text.Substring(j, len) == item)
            {
              cnt++;
              label9.Text = j.ToString();
              label10.Text = cnt.ToString();
              Application.DoEvents();
            }
          label9.Text = "Hits= " + cnt.ToString();
```

In der Schleife mit dem ca 4,6 Millionen Zeichen der Bibel wird nur in einem einfachen string gesucht. Dauert bei meinem Lap ca. 5s.
Eine andere Alternative ist die Methode *.IndexOf(a,b),* die sehr schnell ist. Außerdem sind jetzt die Ausgaben in die Label, die auch sehr zeitintensiv sind, weg. Neue for- Schleife:

```
            for (int j = 0; j < textlen; j++)
            {
                startindex=text.IndexOf(item,j);
                if (startindex >=0)
                {
                    cnt++;
                    j = startindex+1;
               }
```

```
        }
```

Jetzt merkt man die Rechenzeit kaum.
Alle diese Teilprogramme sind in dem Verzeichnis *Kurs 5# Strings* zu finden.

Kurs 6 Projekt Diashow

Ehemals früher als Hausaufgabe, jetzt als Projekt1 wollen wir ein Programm schreiben, das aus einer Serie von Fotos (JPG, gif, Png in einem beliebigen Verzeichnis) in einer Diashow darstellen.
Dazu braucht man als erstes einen Zugriff auf das Filesystem des Rechners. Dazu sind Komponenten nützlich, die auch in Visual Basic benutzt wurden. Diese sind nicht Standard in der Toolbox und man muss sie erst aktivieren. Das geht über das Menü „Extras → Toolboxelemente auswählen“. Im kleinen Check-Kästchen markieren: *FileListBox*, *DirListBox* und *DriveListBox*. Sie sind zwar schon etwas altbacken, aber die notwendigen Funktionen für unseren Zweck stellen sie zur Verfügung. Meines Wissens will .NET diese Komponenten nicht mehr unterstützen, es gibt aber auch keine Alternative: Danke liebe MS-Informatiker!

Die Plazierung der Komponenten soll jetzt so geschehen, dass ein automatisches Resizing erfolgt, und der User selbst die Aufteilung noch mit der Maus per Splitter verändern kann. Also folgendes Vorgehen ist zu tun:
FileListBox auf neue Seite eines *TabControl, Dock=Left*.
Dann Komponenten *Splitter* (Alle Windows Forms), voreingestellt ist schon passende *Dock=Left*.
Mit einem *Panel* den Rest der Seite mit *Dock=Fill* ausfüllen.
Auf dieses Panel dann eine *DirListBox* plazieren, *Dock=Top*. Darunter auf das *Panel* dann die Komponente *DriveComboBox.*
Die drei File- system- Kompos müssen noch verbunden werden. Das geht leider nicht mehr wie bei Borland mit einfachem Setzen der Eigenschaften im Eigenschaftenfenster, sondern nur noch programmiert z. B. in dem Ereignis Form_Activated. Warum einfach, wenn es auch komplizert geht: Danke MS- Informatiker!
In dem Ereignis

```
private void driveListBox1_TextChanged(object sender, EventArgs e)
      {
            dirListBox1.Path = driveListBox1.Drive;
      }
```

schreibt man das gewählte Laufwerk (es heißt nicht drive, sondern path!) nach dirListBox.
Um jetzt die Dateien im selektierten Verzeichnis darzustellen, muss man wieder tricksen. Die Komponente dirListBox liefert zwei Informationen: in .Path den geöffneten (nicht selektierten Path) und in .Text den selektierten Text ohne Pfadangabe. Daraus kann man dann mit folgendem Programm im Ereignis den selektierten Pfad errechnen und nach fileListBox schreiben. Man hat ja sonst nichts zu tun. Warum sollten die Informatiker auch diese Information gleich fertig zur Verfügung stellen. Man muss dann eben schnell folgendes Programm schreiben:

```
privatevoid dirListBox1_SelectedValueChanged(object sender, EventArgs e)
{
// in Text appears text of the selected line
// in path appears the open path. the selected path can now be constructed:
// if I select a path under open path: selpath=Path+/+Text
// if I select a path above open path: selpath= find pos of text, delete
// after text.
string selpath;
int pos=dirListBox1.Path.IndexOf(dirListBox1.Text);
```

```
//pos is Position of text, if -1 not found
if (pos < 0)
     selpath = dirListBox1.Path + "\\" + dirListBox1.Text;
else
     selpath = dirListBox1.Path.Substring(0, pos + dirListBox1.Text.Length);
fileListBox1.Path = selpath;
}
```

Will man an die Daten in der FileListBox drankommen, so geht das mit der Eigenschaft Items. Folgende for- Schleife kopiert alle Dateien in eine TextBox:

```
for (int i=0;i<fileListBox1.Items.Count;i++)
 textBox1.AppendText(fileListBox1.Items[i].ToString()+ "\x0d" + "\x0a");
```

Jetzt soll ein kleines Vorschaufenster programmiert werden. Wenn man auf eine Datei in f*ileListBox1* klickt, soll in einer *PictureBox* eine Bildvorschau zu sehen sein. Also Plazieren eines *PictureBox*auf die Seite mit den obigen Komponenten. Größe 100*100 Pixel.
In f*ileListBox1.SelectedItem* steht die Nummer der selektierten (blau eingefärbten) Datei. Jetzt kann man in einem Event *FileListBox1Click* diese Bilddatei nach *PictureBox1*laden. Syntax:

```
string fn = fileListBox1.Text;  // selected Line
int len=fn.Length;
string ext = fn.Substring(len - 4, 4).ToLower();
if ((ext == ".jpg") || (ext == ".png") || (ext == ".bmp"))
      {
            pictureBox1.Load(fileListBox1.Path + "\\" + fn);
      }
```

es wird zuvor abgefragt, ob es sich um eine Bilddatei handelt. Die Methode .ToLower() wandelt alle Großbuchstaben in kleine um. Der Substring- Befehl schneidet die letzten vier zeichen heraus.
Der Dateiname wird mit *fileListBox1.Text* ausgelesen. Mit der Methode *Load* kann eine Bilddatei vom Typ *.ico, *.bmp oder *.jpg in die *pictureBox1* gelesen werden.
Die Bilder werden jetzt alle nur mit 100*100 Pixel angezeigt, man sieht nur einen Ausschnitt. Das ganze Bild sieht man mit der Eigenschaft *pictureBox1->SizeMode=StrechImage;*

Die Originalbildgröße kann mit

```
  label1.Text = pictureBox1.Image.Height.ToString();
  label2.Text = pictureBox1.Image.Width.ToString();
```

in zwei *Label* dargestellt werden. Das Bild ist zwar in *pictureBox1*gestaucht, aber im Speicher noch in Originalgröße erhalten.

Will man jetzt noch das Seitenverhältnis erhalten, dann stelle man die Eigenschaft „SizeMode“ der pictureBox1 auf Zoom.

Will man jetzt noch bei Verstellung der Selektion mit den Up- oder Down- Tasten ein neues Bild sehen, dann wähle man in den Ereignissen der *fileListBox1 „SelectedIndexChanged“* die schon existierende Ereignisfunktion *fileListBox1_Click*. Testen Sie das Ergebnis. Wenn jetzt eine Bilddatei gewählt ist, wird sie in *pictureBox1* angezeigt. Wenn man mit den Pfeiltasten die Selektion hoch und herunterbewegt, wird jeweils das aktuelle Bild angezeigt. Lädt man allerdings eine Datei mit einer anderen als der erlaubten Endung, passiert natürlich nichts.
Siehe fertiges Programm in Kurs 6# Prepare HW1.

Projektaufgabe 1 Diashowprogramm

Schreiben sie jetzt ein Programm, dass alle Dateien aus dem gewählten Pfad in entweder einstellbarer Zeit (mit einem *Timer* und mit in einem *numericalUpDpwn*- wählbaren Sekunden) oder auf Mausklick hintereinander endlos möglichst groß (also in einem neuen Formular) anzeigt. Abbruch mit einem *ContextMenuStrip*- Menu, das mit einem Eintrag „end“ das Formular wieder schließt.

Hinweise:
Eigenschaften des neuen Formulars: *Name=“FormDia“, FormBorderStyle=None; BackColor* auf Schwarz. *WindowState=wsMaximized*. Achtung, bevor Sie es starten und mit *Show()* aufrufen, müssen Sie das Beenden programmiert haben (das *ContextMenuStrip* an das Formular binden), da das Fenster nun keine Tasten dafür mehr hat. Sonst geht es nur über Alt F4 oder andere Maßnamen, z. B. „Debugging beenden“ mit Shift + F5!
Beenden mit *ContextMenuStrip*- Menu. Ein Eintrag „end“, mit Doppelklick in der Ereignisfunktion *Click()* das Schließen des Fensters auslösen z.B. mit *Close();*. In dem Ereignis von *FormDia* namens *FormClosing* dann wie oben schon erwähnt den Befehl *e.Cancel=true;* programmieren, damit wird das Schließen und Löschen der Instanz verhindert. Stattdessen dort *Hide();* programmieren.
Um den Zugriff von FormDia zurück auf das Hauptformular zu ermöglichen, muss man diese Schritte tun:
In Form1, dem Hauptformular:

```
public Form1()
{
    InitializeComponent();
    frm2 = new FormDia(this);
}
FormDia frm2;
```

In FormDia:
Bei den using muss der Namespace von Form1 angegeben werden. In meinem Beispiel – kann bei ihnen anders heißen- : `namespace WindowsApplication14`
Also in Form Dia: `using WindowsApplication14;`
Wenn nicht der namespace von FormDia nicht sowieso schon den gleichen namen hat.
Dann dort im Konstruktor die rot gerahmten Zeilen/ Teile ergänzen:

```
Form1 meinmain;

public FormDia(Form1 m)
{
    InitializeComponent();
    meinmain = m;
}
```

Auf *FormDia* eine *pictureBox*- Komponente ziehen, diese mit *Dock=fill* auf das ganze Formular ausdehnen. Dann auch an *pictureBox1*das *ContextMenuStrip* dranlinken.
Jetzt kann man in einem Ereignis *MouseClick* ein Bild aus der Liste von FileListBox laden.
Um an diese dranzukommen, muss sie erst im *Modifier* auf *public* gestellt werden.
Benötigt wird noch ein

```
public int diacnt = 0;
```

mit dem der index der Filelistbox hochgezählt wird.

Das Laden erfolgt wie schon gehabt, alles hinter ereignis MouseClick:

```
string fn = meinmain.fileListBox1.get_Items(diacnt);  // selected Line
int len = fn.Length;
string ext = fn.Substring(len - 4, 4).ToLower();
if ((ext == ".jpg") || (ext == ".png") || (ext == ".bmp"))
{
    pictureBox1.Load(meinmain.fileListBox1.Path + "\\" + fn);
}
diacnt++;
if (diacnt >= meinmain.fileListBox1.Items.Count) diacnt = 0;
```

Jetzt sollte es schon gehen, nur in der FileListBox sollte das Pattern so gestellt werden, dass dort nur Bilder drin stehen, also =*.jpg;*.png;*.bmp.
Damit gleich nach Druck auf Start-Buttom ein Bild gezeigt wird, im Click- Ereignis des Buttons das MouseClick- Ereignis von FormDia aufrufen. Das geht erst, wenn man dieses Ereignis public macht:

```
public void FormDia_MouseClick(object sender, MouseEventArgs e)
```

dann in `buttonStartDia_Click(….`

.

```
frm2.FormDia_MouseClick(null, null);
```

Benötigt wird auf *FormDia* noch ein *Timer, Interval*=2000, *Modifier=public*, damit er von *Form1* aus erreicht und aktiviert werden kann.

FormDia, dort *picturebox1.Sizemode=Zoom;*

Will man mit der ESC- Taste (Code 27) schließen, muss man auf dem Formular *FormDia* eine *KeyPress*- Ereignisfunktion erzeugen und dort

```
if (e.KeyChar == 27) Visible=false ;
```

programmieren.
Nach Klick auf die Start- Taste muss der Timer bei aktivierter CheckBox für den Timer auf *Enabled=true* geschaltet werden.
Nun nur noch im Timer_Tick – Ereignis das mouseClick- Ereigniss aufrufen.

```
FormDia_MouseClick(null, null);
```

Beispiel des fertigen Programms in dem Verzeichnis HA1NeuV1, dort natürlich ohne Source.
Damit in der *FileListBox* nur Bilddateien aufgelistet werden, sollte vorher die Eigenschaft *.Pattern* von der *FileListBox* auf ein passendes Filter gesetzt werden. Das Format dafür: **.jpg* , wenn ein Typ zugelassen werden soll, wenn es mehrere sind, mit Semikolon trennen: **.jpg;*.gif;*.png;*.bmp.*
Ein Problem sollte noch abgefangen werden: Wenn gar kein Bild in der FileListBox ist, gibt es eine Fehlermeldung, diese wird noch nicht abgefangen.

Kurs 7 Zeichnen mit dem Graphics - Objekt

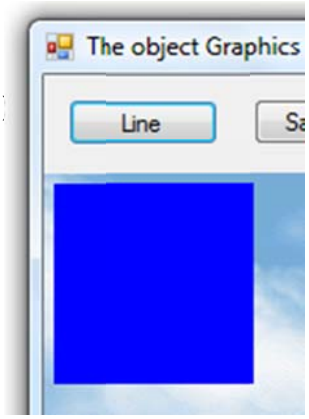
Das Graphics-Objekt

Das Objekt *Graphics* dient zum pixelorientierten Zeichnen auf Komponenten als Ausgabeeinheiten. Jeder einzelne Pixel kann individuell definiert werden, mit komfortablen Methoden ist ein Zeichnen von fast beliebigen Figuren (Linien, Rechtecken, Kreisen usw.)

möglich. Auch Text kann der Zeichnung hinzugefügt werden. Mit diesem Objekt lässt sich ein eigenes Paint- Programm leicht selber schreiben.
Es gibt einige Komponenten, die ein *Graphics*- Objekt haben:

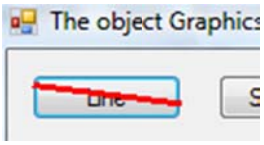
- Panel
- PictureBox
- Form

Will man auf dem Bildschirm eine Grafik erzeugen, so nehme man am besten direkt die *Graphics* der *Form*, will man später sein Werk als Bitmap speichern, so muss man erst eine Bitmap definieren, doch dazu später. Diese Bitmap verfügt dann über die Methoden *Load()* und *Save()*.
Also platzieren wir eine *PictureBox* auf eine neue Form einer neuen Anwendung. Oben wird wieder ein *Panel1* und *Dock=Top* als Bereich für Tasten definiert, die *pictureBox1* füllt den Rest der Form (*Dock=Fill*).
Wollen wir auf einem hübschen Hintergrund malen, so laden wir das Bild (im Eigenschaftenfenster die Eigenschaft *Image*) zu Beginn mit einem Hintergrundbild, z. B. Wolken.bmp (in Pfad bayerlej\ MS Studio2008 C#\ Images\).
Am allereinfachsten zeichnet man im Paint-Ereignis, sofern in der Komponente vorhanden.
Einige Beispiele:
erzeugt von folgendem Programm.




```
private void pictureBox1_Paint(object sender, PaintEventArgs e)
{
      SolidBrush pbrush = new SolidBrush(Color.Blue);
      e.Graphics.FillRectangle(pbrush, 5, 5, 100, 100);
}
```

In den *PaintEventArgs* wird der Bezug auf das Graphics- Objekt von der *Picturebox1* übergeben. Die Zeichenmethoden mit Fill vor dem Namen erwarten ein *SolidBrush*, mit dem die Füllung definiert wird. Das Rechteck selbst kann mit diversen Parametern definiert werden, hier mit den beiden Koordinaten der Eckpunkte oben links (5,5) und unten rechts (100,100).
Oder Ähnliches mit dem Button (auch der hat ein Paint- Event):

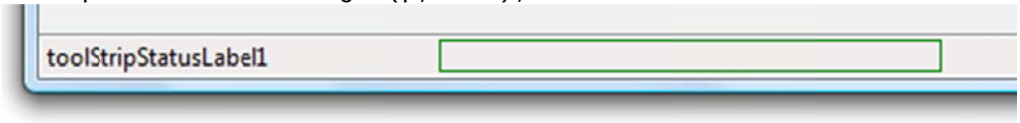



```
Pen p = new Pen(Color.Red, 3);
Point poi1 = new Point(0, 5);
Point poi2 = new Point(80, 15);
e.Graphics.DrawLine(p, poi1, poi2);
```

DrawLine() zeichnet eine Linie, hier mit der Eigenschaft von *Pen*, der mit Farbe und Strichdicke definiert ist. Die Koordinaten diesmal mit zwei *Point*- Objekten.

Oder mal in einem *StatusStrip1*:

```
Pen p = new Pen(Color.Green, 1);
Rectangle rec = new Rectangle(200, 3, 250, 15);
e.Graphics.DrawRectangle(p, rec);
```



In DrawRectangle steht mit (200,3) die Koordinate oben links und dann folgt Width und Height.

Allen diesen Figuren werden jetzt sofort zu Anfang beim Start gezeichnet, denn dann wird der Paint- Ereignis ausgelöst.
Will man in einem anderen Ereignis zeichnen, z.B. in einem Click des Buttons, so gibt es natürlich die *PaintEventArgs* nicht. Dann kann der Bezug folgendermaßen erzeugt werden:

```
privatevoid buttonLine_Click(object sender, EventArgs e)
{
      Graphics gPB = Graphics.FromHwnd(pictureBox1.Handle);
      SolidBrush pbrush = new SolidBrush(Color.LightSkyBlue);
      gPB.FillRectangle(pbrush, 0, 0, 100,100);
}
```

es wird eine lokale Instanz von Graphics (gPB) erzeugt, die per Handle auf die pictureBox1 zeigt. Der Rest geht wie gehabt. Jetzt wird auf Click ein Rechteck gezeichnet. Die erste Zeile mit der Methode .*FromHwnd* kann auch völlig gleichbedeutend mit der Alternativcodierung ersetzt werden. Beides erfüllt den gleichen Zweck:

```
Graphics gPB = pictureBox1.CreateGraphics();
```

Bei einer solchen lokalen Instanz sollte man nach dem Zeichenvorgang die Graphics- Instanz gPB wieder entfernen mit

```
gPB.Dispose();
```

Die Instanziierung von gPB kann auch global erfolgen, dann können andere Ereignisfunktionen auch darauf zugreifen:
Erst global- also außerhalb einer function, deklarieren:

```
public partial class Form1 : Form
   {
      Graphics gPB;
      SolidBrush pbrush = new SolidBrush(Color.Blue);
.....
```

Dann instanziieren:

```
public Form1()
        {
            InitializeComponent();
            gPB = Graphics.FromHwnd(pictureBox1.Handle);
            pbrush = new SolidBrush(Color.LightSkyBlue);
```

Dann benutzen, z. B. mit zwei *Trackbars* wird ein *Pie* gezeichnet, dessen Umrandung eine Ellipse ist und die Randbegrenzungen werden mit einem Winkel (0-360°) definiert, wobei 0° nach rechts zeigt, von dort startet ein Kuchenstück mit der Winkelbreite eingestellt mit *trackBar2*:

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
pbrush.Color=Color.Red;
gPB.FillPie(pbrush, 20, 30, 100, 200, trackBar1.Value,
trackBar2.Value);
```

Links steht *trackBar1* etwa auf 5° und trackBar2 auf 90°

Nun wird direkt auf die Komponenten gezeichnet. Will man eine Zeichnung erzeugen, die man abspeichern kann, so muss man mit einer Bitmap

arbeiten.

Der erste Versuch bietet sich an: Man nehme die fertige BitMap von *PictureBox1* namens *Image*.

Man kann zur Laufzeit ein Bild wie gehabt laden mit

```
pictureBox1.Load("Wolken.BMP");
```

wobei jetzt natürlich das Bild „Wolken.bmp“ im aktuellen Verzeichnis ….Debug stehen muss. Führt man diese Zeile aus, so wird gleich wieder das blaue Rechteck darüber gemalt, da dies in der Paint- Ereignisfunktion `pictureBox1_Paint` so programmiert ist. Beim Laden wird diese natürlich ausgeführt.
Will man jetzt das Bild mit Zeichnung abspeichern, so stellt man erstaunt fest, dass *pictureBox1* keine Save- Methode hat. Man findet aber bei *Image* ein *.Save*, nur das funktioniert nicht, warum weiß ich zurzeit nicht:

```
pictureBox1.Image.Save("test.bmp");   //→ Fehlermeldung
```

Mal ein Versuch mit einer Bitmap, z. B. so :

```
Bitmap pic = new Bitmap(200, 200);
```

erzeugt eine leere 200*200 Pixel große Bitmap,

```
Bitmap pic = new Bitmap("Blumen.jpg" );
```

lädt das Bild aus dem aktuellen Verzeichnis und definiert eine Bitmap in der Größe des Bildes. Es darf kein indiziertes Farbformat sein (wie z.B. „Wolken.BMP“), dann gibt es einen Laufzeitfehler. Es müssen TrueColor- Bilder sein.

Wie zeichnet man nun in eine Bitmap und speichert sie dann folgendermaßen ab:
Z.B. in einer Click-Funktion könnte man schreiben:

```
Graphics g1 = Graphics.FromImage(pic);
Rectangle rec = new Rectangle(0, 3, 25, 15);
g1.FillRectangle(pbrush, rec);
```

Jetzt wird ein Rechteck nach g1 gezeichnet, nur man sieht nichts. Man kann es aber abspeichern, z. B. mit folgender Zeile:

```
pic.Save("testa.bmp", System.Drawing.Imaging.ImageFormat.Bmp);
```

Mit dem *ImageFormat* teilt man der *Save*- Methode mit, in welchem Format man es denn gerne hätte. Als .jpg- Datei geht dann so:

```
pic.Save("testa.jpg", System.Drawing.Imaging.ImageFormat.Jpeg);
```

Wenn man sich diese Dateien mit Windows-Programmen ansieht, dann ist das Rechteck zu erkennen, der Hintergrund ist aber schwarz, wenn man mit Bitmap(200,200) eröffnet hat, sonst eben das geladene Bild. Will man die Zeichnung sehen, so muss man in ein Graphics- Object zeichnen, das zu einer sichtbaren Komponente gehört, z.B. unser gPB, das auf die *pictureBox1* zeichnet. Der Code dazu lautet:

```
  gPB.DrawImage(pic,10,10);
```

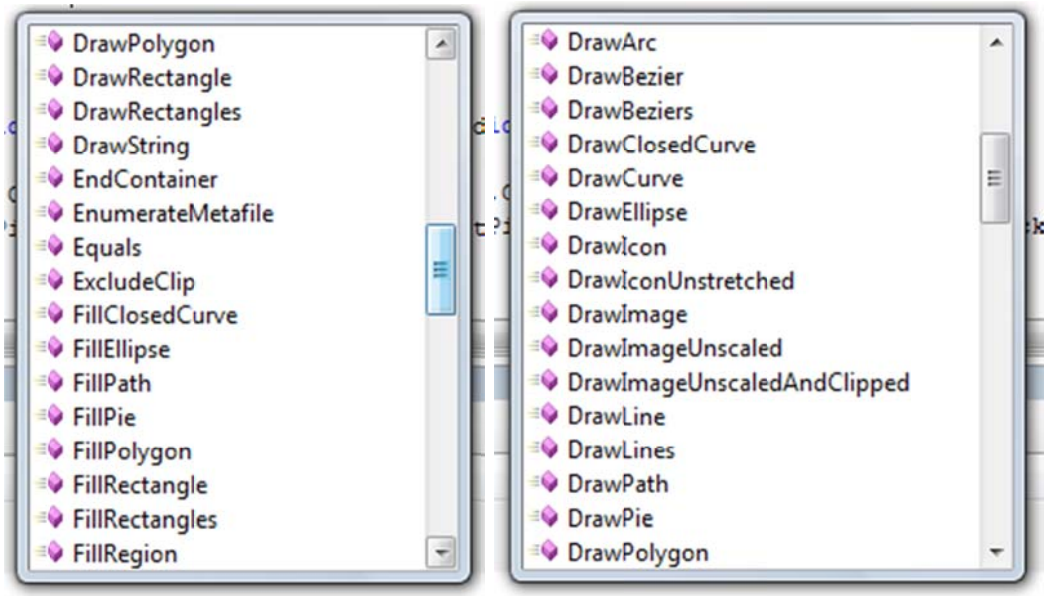
wobei die beiden Koordinaten in diesem Aufruf den linken oberen Punkt von *pic* festlegen.

Will man jetzt g1 von oben nicht auf eine Bitmap, sondern auf das Image- Object von *pictureBox1* legen, so geht das leider auch nicht: Laufzeitfehler

```
Graphics g1PB = Graphics.FromImage(pictureBox1.Image);// ->Laufzeitfehler
```

Man kann nun Zeichenfiguren mit verschiedenen Methoden erzeugen. Die wichtigsten sind:

- *DrawLine(int X, int Y, int x1, int Y1);* zeichnet Linie von (X,Y) bis (X1,Y1). Mit 4 verschiedenen überladenen Parameterlisten, siehe CodeCompletion- Fenster.
- *DrawRectangle(int X1, int Y1, int X2, int Y2);* Zeichnet ein Rechteck mit Ecke oben links bei X1,Y1 und unten rechts bei X2,Y2. Mit 3 verschiedenen überladenen Parameterlisten, siehe CodeCompletion- Fenster. *FillRectangle* ähnlich mit Füllung
- *DrawEllipse(int X1, int Y1, int X2, int Y2);* Zeichnet Kreis oder Ellipse mit umgebendem Rechteck mit den Koordinaten wie beim Rectangle.
- *DrawPie(int X1, int Y1, int X2, int Y2, int X3, int Y3, int X4, int Y4);*s.o.
- Weitere Methoden von *Graphics* siehe CodeCompletion



- Die Methoden mit „Draw…..“ erwarten einen Pen, der mit new definiert werden muss und legt den Rand fest. Die Methoden mit „Fill….“ Erwarten einen Brush oder SolidBrush, ebenfalls mit new definiert, der die Fülleigenschaften festlegt. Man sehe in den Beispielcode hinein.

Wir schreiben nun zwei kleine Programme, die folgendes machen: Mit Mausklick folgt der Pen der Maus und zeichnet eine freihändige Figur. Mit Tastendruck auf „Artwork“ Start/Stop-Betrieb wird die Zeichenfläche mit einer Serie von Rechtecken gefüllt (mit Timer gesteuert), deren Größe und Position und Farbe mit einem Zufallsgenerator erzeugt werden. Mit einer Taste Save soll das Werk dann abgespeichert werden.

Wir brauchen vier Tasten mit *Text* Artwork, Save Pic, Pen-Color und Fill-Color. Zum Abspeichern ist ein *SaveDialog1* nötig, für die Farbwahl ein *ColorDialog*, für die Strichdicke

brauchen wir noch ein *numericUpDown*- Element, Default Wert für *Value=1*. Das sieht dann so aus:

Die Komponenten haben folgende Ereignisfunktionen:
Artwork(*Button1*): Schaltet den *timer1* an und aus, Vorher *timer1.Enabled=false*; als Startzustand festlegen.

```
if (timer1.Enabled) timer1.Enabled = false; else timer1.Enabled = true;
```

Save (*Button2*), es wird als .jpg abgespeichert:

```
if (saveFileDialog1.ShowDialog() == DialogResult.OK)
pic.Save(saveFileDialog1.FileName,System.Drawing.Imaging.ImageFormat.Jpeg);
```

PenColor und BrushColor: Hinter *Button3* und *Button4* wird die Farbe des Stiftes und des Pinsels eingestellt, p und pbrush sind globale Instanzen von Pen und SolidBrush:

```
if (colorDialog1.ShowDialog() == DialogResult.OK)
                p.Color = colorDialog1.Color;

if (colorDialog1.ShowDialog() == DialogResult.OK)
                pbrush.Color = colorDialog1.Color;
```

Mit dem *OnChange*-Event wird die Stiftdicke gewählt:

```
private void numericUpDown1_ValueChanged(object sender, EventArgs e)
  {   p.Width = (int)numericUpDown1.Value;  }
```

Zuerst jetzt die Freihandmalfunktion. Dazu lernen wir die Mausevents kennen. Das Zeichnen selbst geschieht mit dem Ereignis *MouseMove*, der Aufruf sieht so aus:

```
private void pictureBox1_MouseMove(object sender, MouseEventArgs e)
{
  if (draw)
   {
      Point endp = e.Location;
      g1pic.DrawLine(p, start, endp);
      gPB.DrawImage(pic, 0, 0, pic.Width, pic.Height); // Draw on screen
      start = endp;
   }
```

Die aktuellen Mouse Koordinaten werden in e übergeben, z.B. in e.X und e.Y oder in e.Location. Diese Funktion wird jedes Mal aufgerufen, wenn die Maus sich bewegt. Also wird auch gezeichnet, wenn man keine Maustaste drückt, also immer. Deswegen gibt es nun eine Steuerung mit der bool- Variablen *draw*. Sie wird im *MouseDown*- Ereignis auf *true* und im *MouseUp*- Ereignis zurück auf *false* gesetzt.
Dann wird wie oben beschrieben erst mit *g1pic* in die Bitmap pic gezeichnet und dann mit *gPB* in *pictureBox*1, damit es auf den Bildschirm gelangt.

Es wird immer eine Linie von start nach endp gezeichnet, wenn *draw true* ist. Deswegen muss jetzt noch im Ereignis *MouseDown* die globale Variable *start* auf die aktuellen Mauskoordinaten gesetzt werden mit:

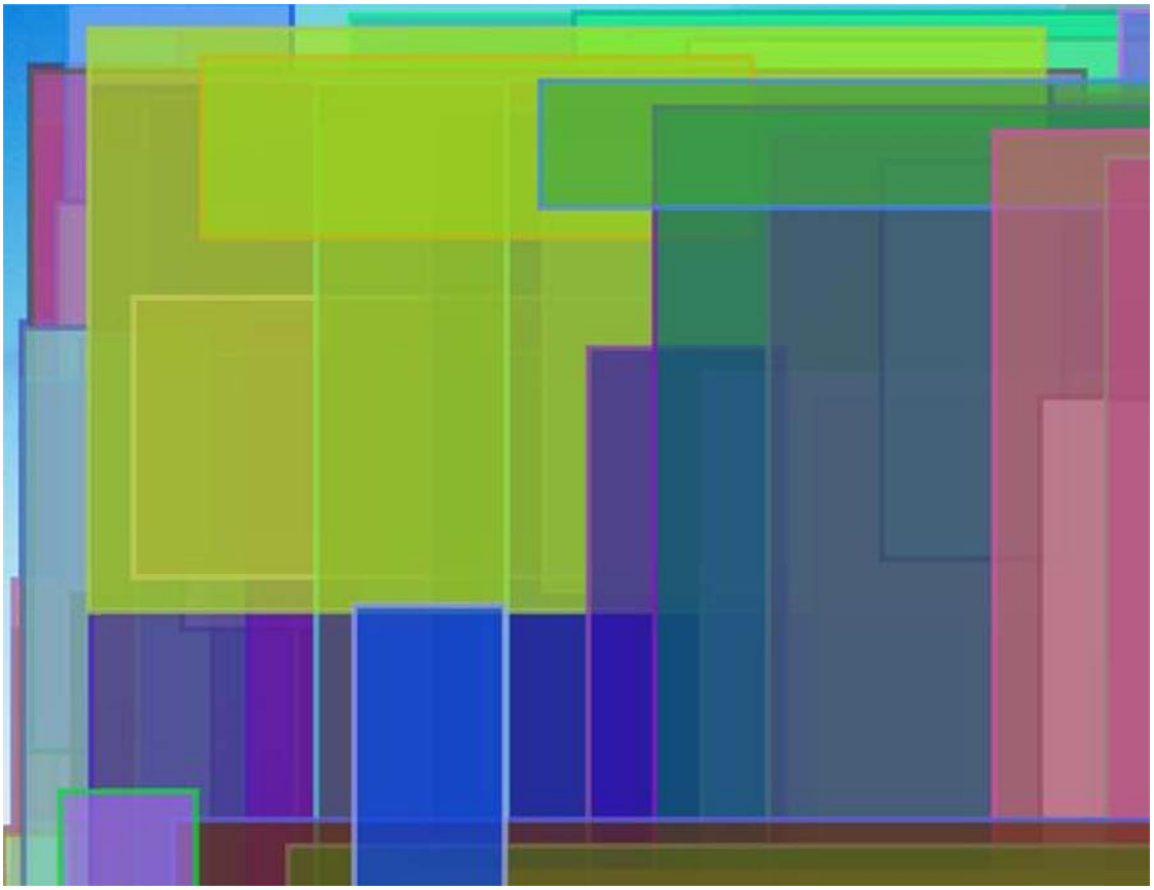
```
private void pictureBox1_MouseDown(object sender, MouseEventArgs e)
      {
            draw = true;
            start = e.Location;
      }
```

Ein paar Bemerkungen:

- Die Zeichenfläche wird beim ersten Zeichnen festgelegt. Ändert man sie hinterher, findet keine Größenänderung mehr statt.
- Lädt man als Hintergrund das Bild „Wolken.BMP“, so gibt es einen Laufzeitfehler. Das liegt daran, weil dieses Bild ein 256- Index-Farben-Bild ist. Wandelt man es in truecolor, so geht es.
- Das Bild darf nicht auf SizeMode =*StrechImage*; gestellt werden, da dann die Mauskoordinaten nicht mit der Position übereinstimmen.

Nun das Artwork. Im Timer sollen per Random- Generator farbige Rechtecke erzeugt werden. In der Timer- Funktion steht bei mir:

```
Random r = new Random();

int x1 = r.Next(0, pic.Width);      //x point top left
int y1 = r.Next(0, pic.Height);     // y point top left
int x2 = r.Next(0, pic.Width-x1);   // width
int y2 = r.Next(0, pic.Height-y1);  // height

Rectangle rect = new Rectangle(x1, y1, x2, y2);

int red = r.Next(0, 255);
int green = r.Next(0, 255);
int blue = r.Next(0, 255);
//int alpha = r.Next(0, 255); // this value with trackbar1
            pbrush.Color = Color.FromArgb(alpha,red, green, blue);

int red2 = r.Next(0, 255);
int green2 = r.Next(0, 255);
int blue2 = r.Next(0, 255);

p.Color = Color.FromArgb(alpha, red2, green2, blue2);
p.Width = r.Next(1, 10);

g1pic.FillRectangle(pbrush,rect);
g1pic.DrawRectangle(p, rect);
gPB.DrawImage(pic,0,0, pic.Width, pic.Height);//Draw on screen
```

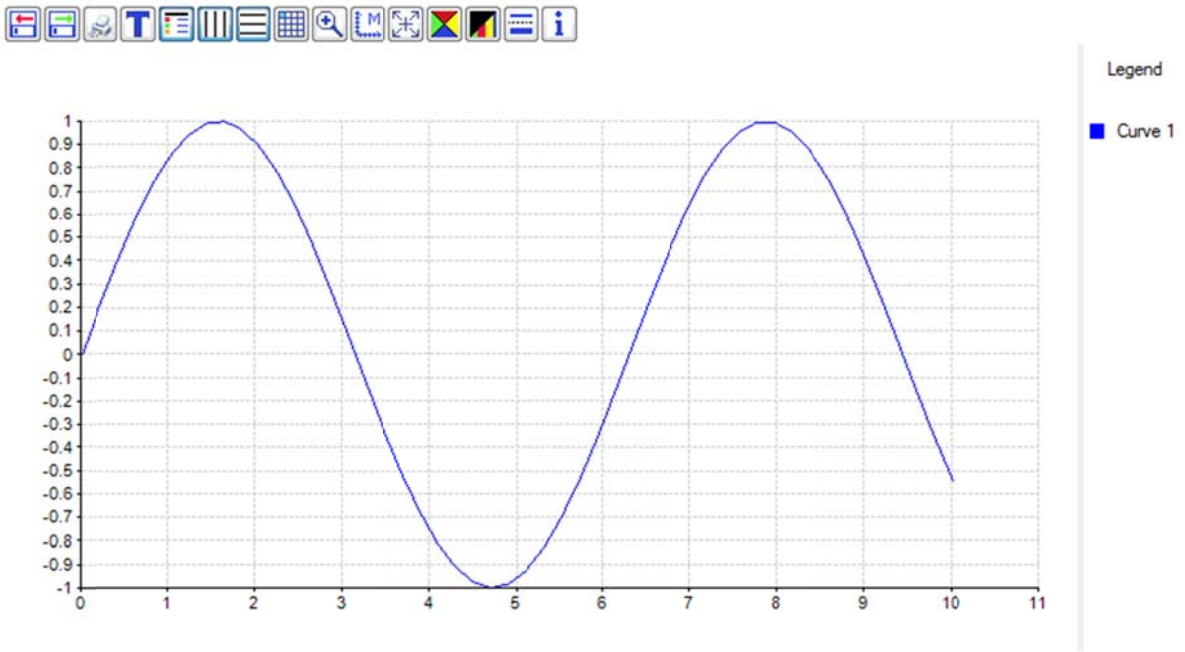


Die Rechtecke orientieren sich an der Größe von pic. Stellt man *Interval* vom *timer1* auf 100 ms, dann werden alle Sekunde 10 Rechtecke erzeugt.
Die Farben werden mit *.FromArgb()* erzeugt. Schön ist auch, dass der Transparentwert alpha hier auch geändert werden kann. alpha=0 heißt voll transparent, also unsichtbar, alpha = 255 nicht transparent. Im fertigen Programm verändert man alpha mit dem linken *trackBar1*. Das Programm zeichnet erst die Füllung und dann einen Rahmen mit anderer Farbe.
Die fertigen Dateien finden Sie in *Kurs7# Graphics*.

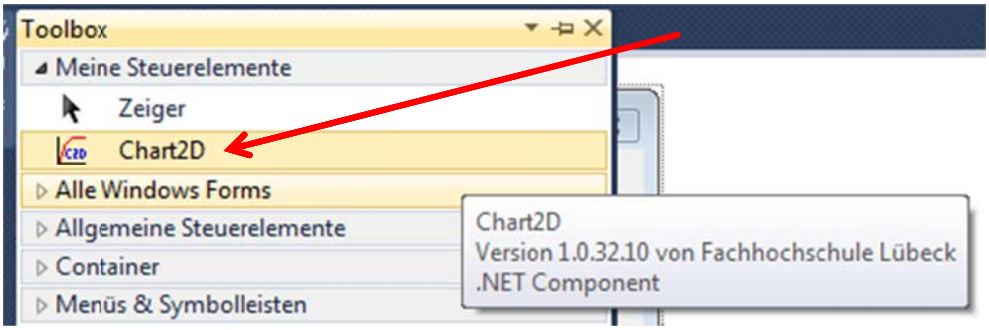
Viel Spaß bei weiteren Experimenten.

Kurs 8 Chart2D

Siehe Hinweis auf die in DotNet 4.0 neu verfügbare Komponente Chart ab S. 85.
In diesem und dem nächsten Kurs wird die mächtige Komponente Chart2D vorgestellt. Diese Komponente ist an der FHL neu entwickelt worden und vollständig in C# geschrieben. Sie ersetzt die alte bisher seit Jahren benutzte Komponente Chartfx, die bei Borland Verwendung fand. Damit ist der ganze Ärger mit ActiveX usw. endlich vom Tisch.
Der Sourcecode liegt auch vor und Erweiterungen sind damit jederzeit möglich.
Einige Eigenschaften: Hauptzweck ist das Zeichnen von Diagrammen anhand von z. B. Messdaten. Es können maximal 10 farbige Kurven mit einer beliebigen Punktzahl gezeichnet werden. Es gibt eine Autoskalierung und mehrere Toolbuttons, mit denen die Darstellung angepasst werden kann. Eine Zoomfunktion, eine Koordinatenanzeige sowie eine Umschaltung in einen Punkteditor per DataGrid runden die Funktionen ab. Hier ein Beispielscreenshot:



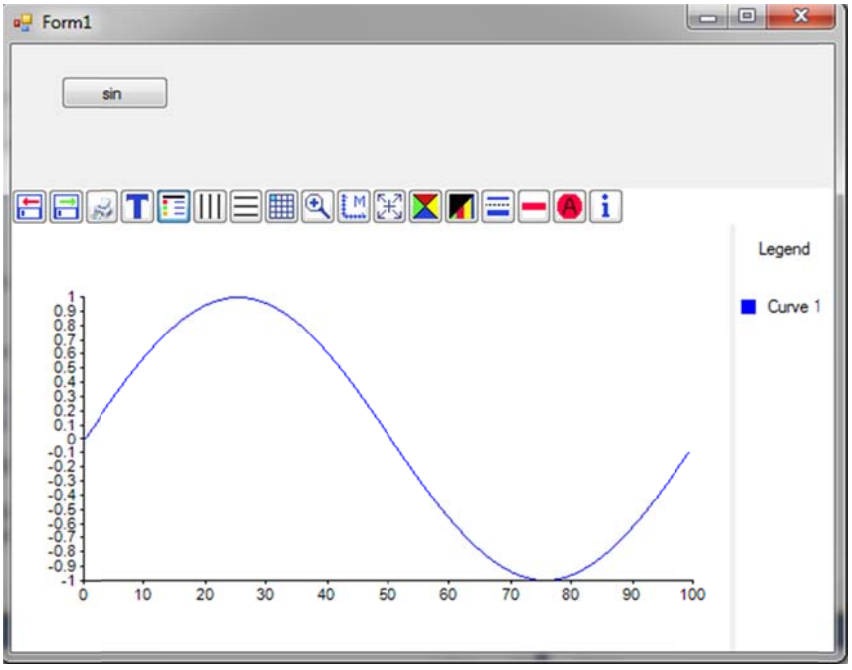
Das Abspeichern und Reloaden sowie ein Ausdruck sind vorgesehen. Ich benutze diese Komponente in meinem Großprojekt WindfC#. Das erste Regelungstechnik Praktikum damit ist ohne Probleme gelaufen und die Kinderkrankheiten sind beseitigt. Es sind für diese Komponente 2 separate Dokumente verfügbar. Auf studpub findet sich in
bayerlej\ MS Studio2008 C#\ Kurs 8 neu Chart2D von Lars Seckler
die Datei *Benutzeroberfläche von Chart2D_2.pdf* . Diese beschreibt für Anwender eines Programms die Bedienung der Benutzeroberfläche.
Die Datei *Verwendung der Komponente Chart2D in C_2.pdf* beschreibt für den Programmierer, wie man diese Komponente einbindet in seinen eigenen C#- Code. Deswegen soll hier nur kurz beispielhaft die Verwendung beschrieben werden, weitere Details entnehme man diesen Dokumenten.
Zur Installation benötigt man die Datei Chart2D.vsi. Diese darf nur bei ausgeschaltetem Visual Studio gestartet werden. Danach sollte die Komponente in der Toolbox erscheinen.
Das Ergebnis bei MS Studio 2010 sieht so aus:



Erstes kleines Beispiel:
In einem neuen Projekt erst ein *Panel* mit *Dock Top* uns dann diese Komponente auf die Form ziehen mit *Dock Fill*.
Dann einen Button mit Namen ButtonSin mit text sin ergänzen. Hinter dem Clickereignis dieser Code:

```
int i, nb = 100, serie = 1;
for (i = 0; i < nb; i++)
{
    chart2D1.setPoint(0, i, Math.Sin(i / 16.0 * serie));
}
chart2D1.startDrawing();
chart2D1.autoScaling();
```

Es sieht nach Ausführung dann so aus:



Mit Setpoint wird das interne dynamisch verwaltete Datenarray gefüllt. Die erste Zahl ist die Nummer der Kurve (von 0 bis 9), die zweite der x- Wert und die dritte Zahl der y- Wert eines Punktes. Mit dem Befehl startDrawing() wird die Kurve gezeichnet. Mit autoScaling() findet eine automatische Skalierung statt.
Die Anzahl der Punkte pro Kurve ist beliebig und kann für jede Kurve individuell verschieden sein.
Drückt man diese Taste sin jedoch ein zweites Mal, so erscheint eine blauer Querstrich. Das liegt einfach daran, dass jetzt zwei gleiche Kurven im Speicher hintereinander liegen und alle

Punkte verbunden werden. Also wird der letzte Punkt der ersten Sin- Kurve mit dem ersten Punkt der zweiten verbunden.
Um dies zu vermeiden, sollte vor dem Neuzeichnen erst alter Inhalt einer Kurve mit der Methode *clearLast()* gelöscht werden.
Eine einfache Variation liegt jetzt hinter einer neuen Taste namens buttonMulti. Folgender Code zeichnet immer neue Kurven:
Außerhalb einer function: `short serie = 1;`
Dann innerhalb der Click-funktion:

```
int i, nb = 100;
for (i = 0; i < nb; i++)
{
    chart2D1.setPoint(serie - 1, i, Math.Sin(i / 16.0 * serie));
}
chart2D1.startDrawing();
chart2D1.autoScaling();
serie++;
```

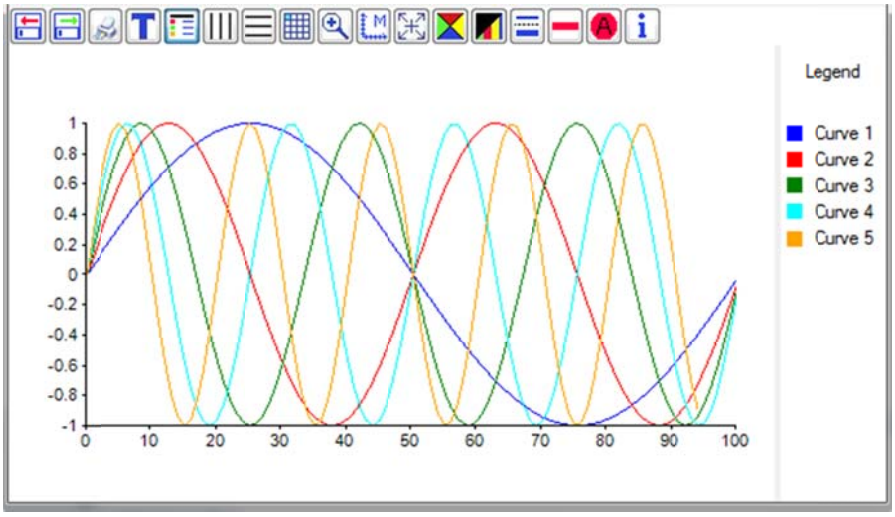
Nach der 10. ten Kurve wird nicht mehr gezeichnet.
Jetzt testen wir mal die Echtzeitzeichenfunktion: Hinzufügen eines Timers (Interval=20, Enabled=false) und einer Taste ButStart.
Hinter diese Taste:

```
if (timer1.Enabled)
    timer1.Enabled = false;
else
{
    timer1.Enabled = true;
    chart2D1.clearAll();
    chart2D1.XMax = 100;
    chart2D1.XGap = 10;
    chart2D1.YMax = 1;
    chart2D1.YMin = -1;
    chart2D1.YGap = 0.2;
    serie = 1;
}
```

Außerhalb: `int itemp = 0;`
In timer1_Tick:

```
chart2D1.realTimeChart((serie - 1), itemp, Math.Sin(itemp / 16.0 * serie));
itemp++;
if (itemp > 100)
{
    itemp = 0;
    serie++;
}
if (serie > 5)
{
    serie = 1;
    chart2D1.clearAll();
}
```

Dann werden nacheinander 5 Kurven in Echtzeit gezeichnet:



Jetzt noch einige einfache Codebeispiele:
Änderung eines Kurvennamens:

```
chart2D1.CurveNames[0] = "Kurve 1";
chart2D1.updateNames();
```

Titel: `chart2D1.TopTitle = "Top Header Title";`
Löschen der letzten Kurve: `chart2D1.clearLast();`
Ausblenden z.B. der ToolTaste“ Lösche alle Kurven“: `chart2D1.AllClearButton = false;`
Abfrage der Anzahl von Kurven: Counter, keine Kurve: -1
Abfrage der Punkte pro Kurve: in CurvePoints[], es ist ein int- Array.
Abfrage von Punktkoordinaten: einmal getPoint(i,j), wobei i die Kurvennummer und j die Punktnummer ist beide startend mit Null. Beispiel:

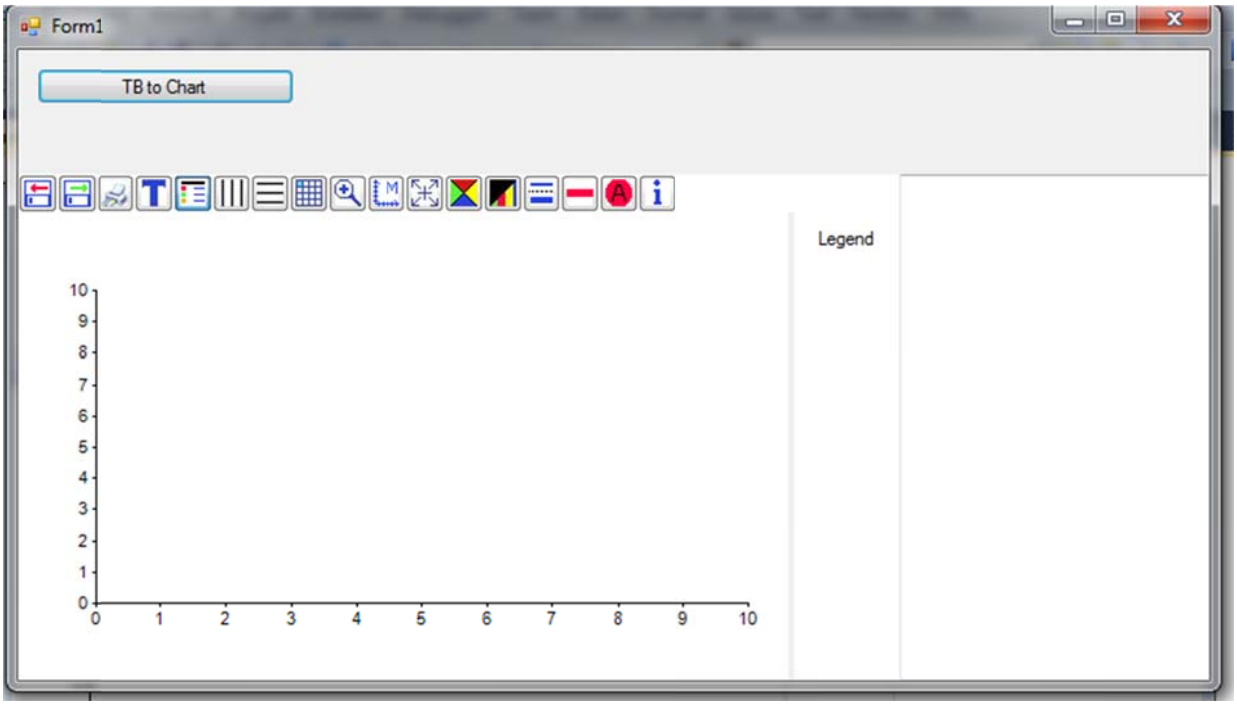
```
double[] xy;
xy = chart2D1.getPoint((int)nudKurve.Value, (int)nudPunkt.Value);
labelXval.Text = xy[0].ToString();
labelYval.Text = xy[1].ToString();
```

Natürlich dürfen die Werte in i und j die Arraygrößen nicht überschreiten, sonst Fehler. i und j werden im Beispiel mit je einem NumericalUpDown eingestellt. Der Typ von Value ist dort ein *decimal*, deswegen das „typecast“ mit *(int).*

Will man übrigens eine Grafikdarstellung mit der alten chartFX in C# (Balken, Kuchendiagramme auch in 3D) so lese man diesen Skript in einer Version 3. Ab Version 4 ist sie verschwunden, da die dort benutzte ActiveX große Installationsprobleme verursacht und außerdem ein 64-Bit programmierung verhindert. Ich bin froh, dass ich sie los bin.

Kurs 9 DataTools - Einbinden von Tool- Units

In meinem Regelungstechnik- Tool- Programm WindfC# benutze ich Units zu finden dort in *DataTools.cs*, mit deren Hilfe man z.B. den Inhalt einer Textbox, falls die ein bestimmtes Format aufweist, komplett als Kurve darstellen kann. Das Format sollte so sein, dass pro Punkt die Koordinaten x und y in einer Zeile stehen und mit einem Separator (kann Komma, Tab oder Leerzeichen sein) getrennt sind.
Vorgehensweise ähnlich schon wie im Kurs 2c#:
Neues Projekt nach Kurs 9# abspeichern.
Der Form ein Panel mit Dock Top hinzufügen.
Zweites Panel mit dock Left hinzufügen
Chart2D mit Dock Fill hinzufügen.
Auf panel1 ButtonTB2Chart hinzufügen mit Text TB to Chart
Auf panel2 Textbox hinzufügen mit Dock Fill.
Resultat:



Nun folgende Instanzen erzeugen (außerhalb des Konstruktors von Form1):

```
DataTools dt = new DataTools();
 LoadOpt lo = new LoadOpt();
 CHMinMax mm = new CHMinMax();
```

Im Konstruktor dann :

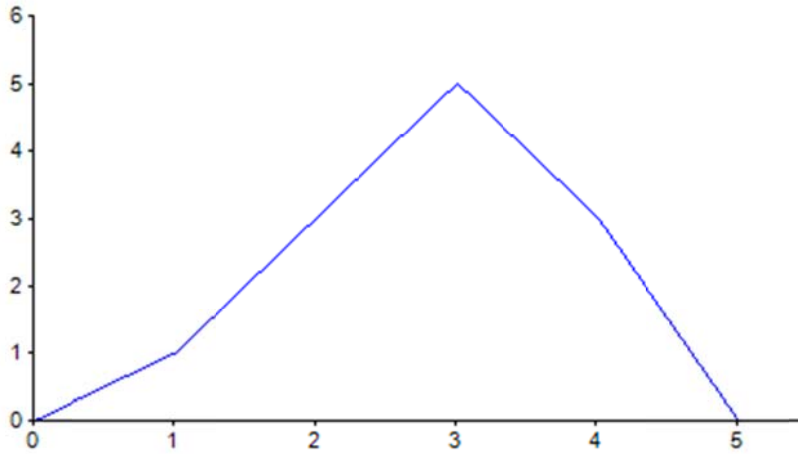
```
public Form1()
{
    InitializeComponent();
    lo.init();
    mm.init();
}
```

Hinter der Clickfunction des Button folgenden Code:

```
dt.ConvertTB2ChartAdd(ref chart2D1, textBox1,ref mm, lo);
```

Jetzt mal folgenden Inhalt in die rechte Textbox tippen oder kopieren:
0 0
1 1
2 3
3 5
4 3
5 0

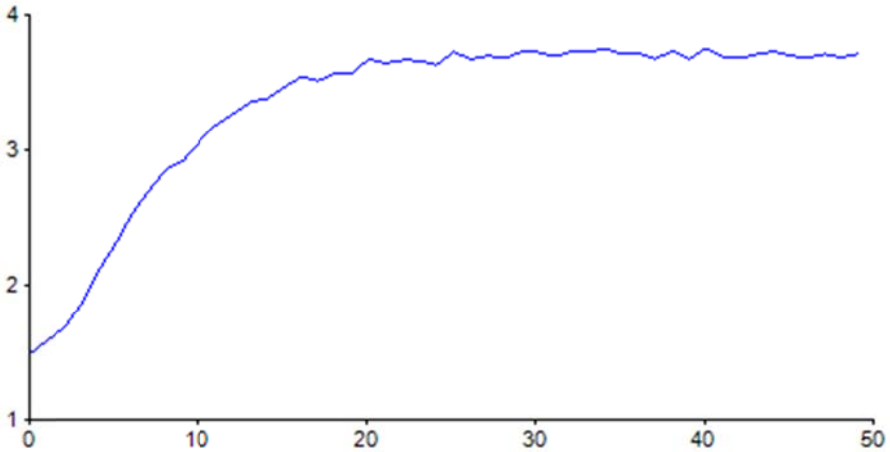
Ergebnis:



Nun kann man ganze Dateien mit Messwerten laden und so einfach grafisch anzeigen. Dazu ein Load- Button spendieren, noch einen openFileDialog1 hinzufügen und hinter die Clickfunctions des Loadbuttons dann wie schon oben in Kurs 5# erklärt eine Datei laden mit:

```
            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                StreamReader myStream1 = new StreamReader(openFileDialog1.OpenFile(),
Encoding.Default);
                textBox1.Text = myStream1.ReadToEnd();
            }
```

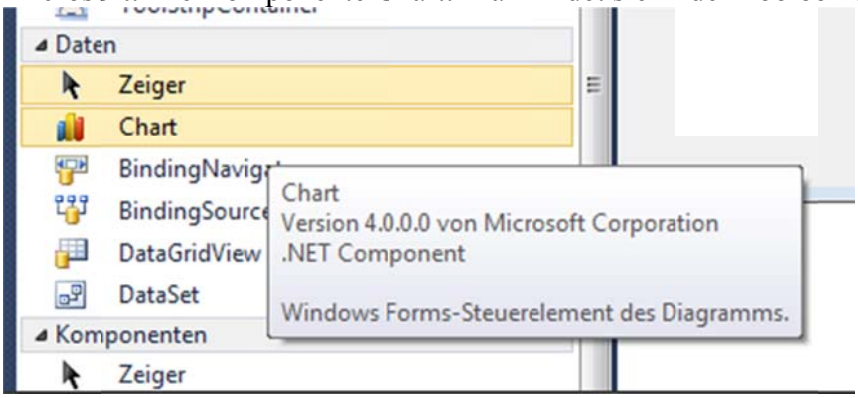
Als Beispiel ist im Kurs 9 die Datei *2PT1_2.sim* zu finden. Die Grafik sieht dann so aus:



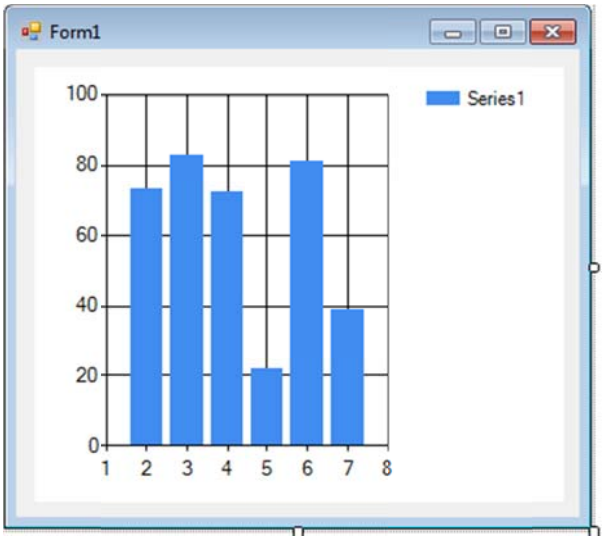
In den LoadOptions kann man dann auch andere Dateiformate, logarithmische Skalierung etc voreinstellen.

Kurs 9a Zeichnen von Kurven mit Chart DotNet 4.0

Seit DotNet 4.0 gibt es nun endlich eine Komponente zum Zeichnen beliebiger Grafiken von Microsoft: Die Komponente Chart: Man findet sie in der Toolbox unter Daten:



Wenn man diese auf das Formular zieht, sieht es so aus:



Die Befüllung mit Daten geschieht so (Sinuskurve mit 100 Punkten):

```
int i, nb = 100, serie = 1;
chart1.Series[0].ChartType = SeriesChartType.Line;
for (i = 0; i < nb; i++)
{
    chart1.Series[0].Points.AddXY(i, Math.Sin(i / 16.0 * serie));
}
```

Will man eine neue Kurve (genannt serie) hinzufügen, so muss man dies machen:

```
chart1.Series.Add("Kurve 1);
chart1.Series[1].ChartType = SeriesChartType.Line;
```

Die einzelnen Kurven werden dann wie ein Array mit .Series[i] erreichbar.

Nur leider gibt es die Toolbar nicht, die dem User bei einem fertigen Programm auch Möglichkeiten gibt, seine Darstellung während der Laufzeit zu ändern. Allerdings sind die Darstellungsmöglichkeiten dieser Chart – Komponente gigantisch. Will man nicht nur Kurven, sondern beliebige Grafiken darstellen, so muss man sich mit dieser sehr komplexen

Komponenten auseinander setzen. Eine sehr gute Einführung bekommt man mit einem Tutorial, das ich bei Microsoft gefunden habe und es auf studpub zur Verfügung gestellt habe: Siehe *bayerlej\ MS Studio2010 C# WS2012_13\ DotNetChart*.
Entpackt man die dort zu findende Datei *WinSamplesChart von MSDN(1).zip*, so kann man das Programm *WinFormsChartSamples.exe* starten und bekommt eine Einführung in alle Möglichkeiten mit passenden Source – Code Zeilen. Besser kann man ein Tutorial nicht aufbauen und ich kann dem nichts mehr hinzufügen. Ich werde sicher auch demnächst mein Toolprogramm Windfc# damit ausstatten, da insbesondere auch Bodediagramme damit sehr gut darstellbar werden, was bisher nicht ging. Folgende Tabelle soll Entscheidungshilfe geben, welche Komponente man wann nehmen könnte:

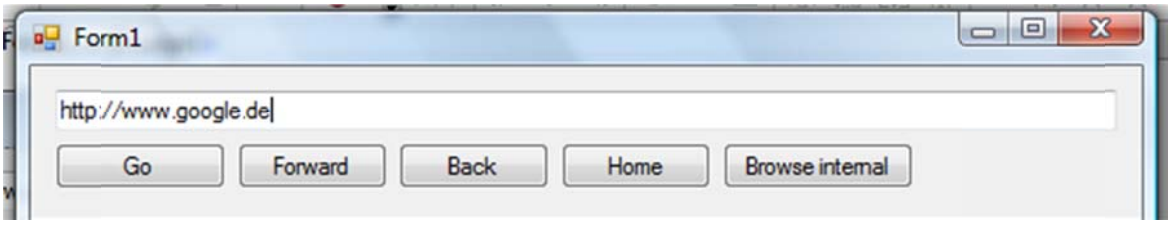
Chart2D von Herrn Seckler	**Chart in DotNet 4.0**
Source Code verfügbar und dadurch anpassbar an eigene Wünsche	Anpassbar durch tausende Optionen
Toolbar für User während Laufzeit	Keine Toolbar
Einfaches Interface	Komplexes Interface
Nur Kurven, max. 10	Beliebige Grafiken
Keine Bodediagramme	Bodediagramme möglich

In Arbeit ist zurzeit ein Projekt, welches die Vorteile beider Komponenten vereint. Es existiert schon eine Komponente, die der DotNet- Chart meine Toolbar verpasst. Es fehlt noch eine Erweiterung des Interfaces, so dass ich in meinen Programmen einen einfachen Austausch vornehmen kann, ohne dort tausende Änderungen vorzunehmen. Das könnte ein nettes Studierenden – Projekt sein. Bei Interesse bitte melden.

Kurs 10 Weitere Komponenten

WebBrowser

Mit der Komponente *WebBrowser* (in Allgemeine Steuerelemente) kann man sich seinen eigenen Browser basteln. In Anwendungen kann man dann damit auf seine eigene Homepage verweisen oder Hilfetexte anbieten. Da in der Regel ein pdf- Reader installiert ist, kann man mit dieser Komponente dann auch pdf- Dateien in seiner Anwendung anzeigen. Wir starten mit einer neuen Anwendung und platzieren in der oberen Hälfte ein Panel (Dock auf Top) für die Bedienelemente unseres Browsers und darunter mit Dock auf Fill den *WebBrowser*. Mit einer textBox1 und einigen Buttons lässt sich ganz einfach ein Browser zusammenstellen.



Hinter der Go- Taste:
```
webBrowser1.Navigate(textBox1.Text);
```

Hinter der Forward- Taste:
```
webBrowser1.GoForward();
```

Hinter der Back- Taste:
```
webBrowser1.GoBack();
```

Hinter der Home- Taste:
```
webBrowser1.GoHome();
```

Hinter der Browse- Taste:

```
if (openFileDialog1.ShowDialog() == DialogResult.OK)
{
    webBrowser1.Navigate(openFileDialog1.FileName);
}
```

Die Home- Seite ist die, die beim Internet- Explorer als Home definiert ist, die Browse- Taste setzt eine Komponente *openFileDialog1* voraus. Es läuft im Hintergrund wie zu erwarten eine eingebettete Form des auf dem PC installierten IE in der entsprechenden Version. Lädt man mit der Browse- Taste Dateien der Art pdf, doc, jpg usw., so wird je nach Installation auf dem Rechner entweder die Anwendung selbst geöffnet, die der Datei zugeordnet ist oder ein Viewer im IE wird aufgerufen und man verlässt die Anwendung nicht.

Mediaplayer

Es gibt nun keine Extra- Komponente als Mediaplayer, sondern nur ein Objekt, mit dem man Mediendateien abspielen kann. In .NET heißt es *SoundPlayer* und ist verfügbar nach Hinzufügen von dem Namespace *System.Media* :

```
using System.Media;
```

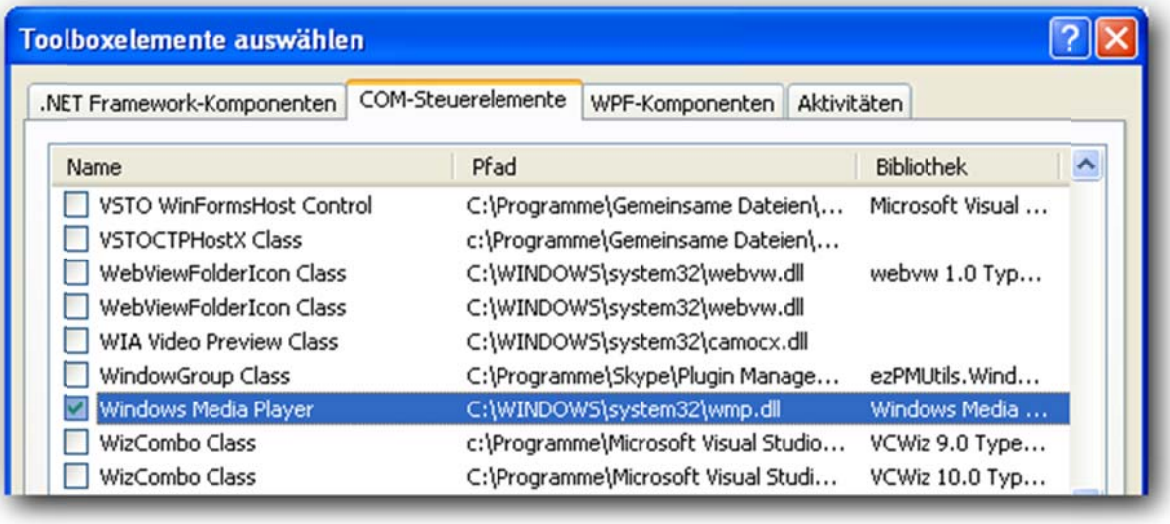
Hat man seinem Projekt eine Dialogkomponente *openFileDialog1* hinzugefügt, dann kann man mit folgendem Code eine *.wav- Datei laden und abspielen:

```
      if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                  SoundPlayer player = new SoundPlayer();
                  player.SoundLocation = openFileDialog1.FileName;
                  player.Load();
                  player.Play();
            }
```

mp3 oder videos gehen aber nicht.

Die gehen aber mit einem eingebetteten Widows- Media –Player. Dann kann man alles machen, was der Media- Player auch kann:
Vorgehen:

1. Toolboxelemente auswählen→ COM , dort WindowsMediaPlayer suchen und wählen.



2. Dann Komponente Windows Media Player in Toolbox suchen, bei mir ganz unten, auf Form ziehen
3. Dann laden einer Media- Datei and abspielen mit

```
if (openFileDialog1.ShowDialog() == DialogResult.OK)
      {
                axWindowsMediaPlayer1.URL=openFileDialog1.FileName;
                axWindowsMediaPlayer1.Ctlcontrols.play();
      }
```

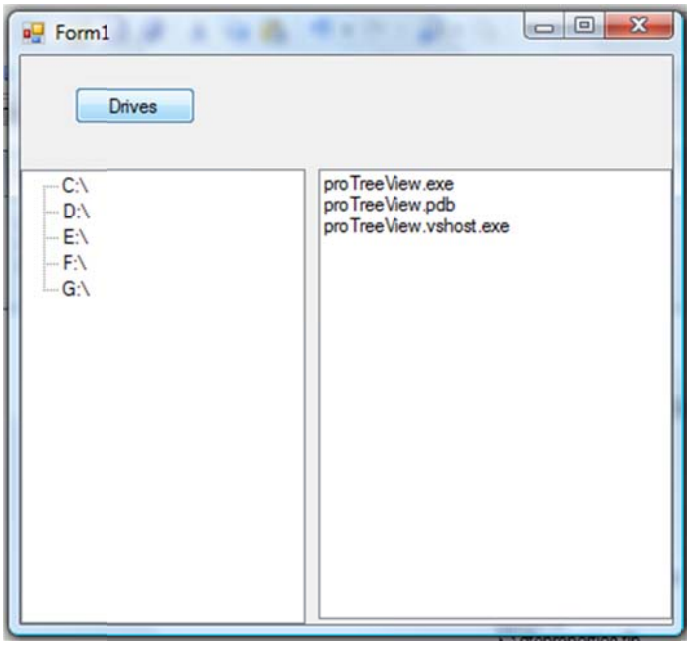
Der spielt insbesondere auch mp3, aber auch Videos ab.

Kurs 11 Email- Programm

Inzwischen hat ein Diplomand Erfahrungen gesammelt mit dem Versenden von Emails ohne externe Klassen. Es ist jetzt mit DotNet- Klassen möglich. Sie finden einfache Beispielprogramme in StudPub in MS Studio 2012 im Kurs 11. Dort sind ein einfaches Sende- und Empfangs- Programm zu finden. Zum Test müssen Sie allerdings eigene Zugangsdaten ihrer Email- Konten eingeben, sonst geht es natürlich nicht. In einer Diplomarbeit ist unter anderem ein kleines Emailprogramm (Verzeichnis Email) erarbeitet worden, in dem auch eine Adressliste, ein HTML- Editor und die Arbeit mit Attachments zum Einsatz kommen. Ein Test dieses Programmes ist allerdings noch nicht erfolgt, eine Fehlerfreiheit kann nicht garantiert werden.

Kurs 12 TreeView

Eine sehr leistungsfähige Komponente ist TreeView. Damit kann man seinen eigenen Explorer bauen. In einem Projekt hab ich damit ein vereinfachtes Dateisystem benutzt mit nur zwei Ebenen (Anfänger- Version).
Wir öffnen ein neues Projekt (ProTreeView) und platzieren ein Panel (Dock Top) einen TreeView (Dock Left) und eine FileListBox (erst über Menü „*Extras -> Toolboxelemente auswählen*“ hinzufügen, wenn nicht in ToolBox bereits verfügbar) mit Dock Fill.
Die Ideen zu diesem Projekt hab ich von Andreas Kühnel übernommen.



Hinter einem Button wird nun der folgende Text gelegt:

```
string[] drives = Directory.GetLogicalDrives();
TreeNode node;
foreach (string drv in drives)
{
            node = treeView1.Nodes.Add(drv);
            if (drv == "C:\\")                // C: selektieren
               treeView1.SelectedNode = node;
```

Dann erfolgt nach Tastendruck obige Reaktion, alle Drives werden untereinander angezeigt. Die Klasse *Directory* wird erst möglich nach Einbindung von *using System.IO* . Zuerst werden alle Laufwerke in das Stringarray *drives* eingelesen, dann wird mit foreach dieses Stringarray

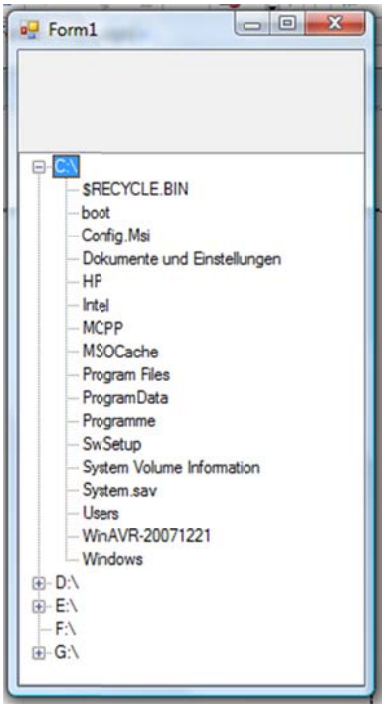
durchgearbeitet. Mit *.Add(drv)* werden Einträge hinzugefügt, in *drv* stehen die einzelnen Elemente des Stringarays.
Nach Einlesen wird noch der Node mit C: selektiert.

Jetzt generieren wir eine Funktion „ReadSubDirs“, die für jeden Knoten im Parameteraufruf alle Unterverzeichnisse einliest. Code.

```
privatevoid ReadSubDirs(TreeNode node)
        {
            DirectoryInfo[] arrDirInfo;  // Array für Subdirectories
            DirectoryInfo dirinfo = new DirectoryInfo(node.FullPath);
            try
            {
                arrDirInfo = dirinfo.GetDirectories();
            }
            catch
            {
                  return;
            }
            foreach (DirectoryInfo DI in arrDirInfo)
            {
                node.Nodes.Add(DI.Name);
            }
        }
```

DirectoryInfo enthält dabei den vollen Namen eines Pfades, den man sich aus dem Node auslesen kann mit *.FullPath.* Bindet man jetzt diese Funktion in die obere foreach- Schleife ein, so sieht der Code so aus: Die Button Click- Funktion hab ich entfernt, alles wird jetzt im Konstruktor von Form1() ausgeführt.

```
public Form1()
        {
            InitializeComponent();
            treeView1.Sorted = true;
            string[] drives = Directory.GetLogicalDrives();
            TreeNode node;
            foreach (string drv in drives)
            {
                  node = treeView1.Nodes.Add(drv);
                  ReadSubDirs(node);
                  if (drv == "C:\\")               // C: selektieren
                        treeView1.SelectedNode = node;
            }
        }
```



Außerdem wird die Sortieren- Eigenschaft von TreeView aktiviert. Nach Programmstart sieht es dann so aus, die „plus“ Kästchen öffnen dann Unterverzeichnisse der ersten Baumebene. Jetzt kann man im Click- Ereignis von TreeView alle Dateien des selektierten Pfades in FileListBox anzeigen lassen mit

```
private void treeView1_Click(object sender, EventArgs e)
     {
                                        TreeNode node;
           node = treeView1.SelectedNode;
           fileListBox1.Path = node.FullPath;
           fileListBox1.Update();
                                    }
```

Dann sieht man rechts die Dateien der selektierten Pfade. Nur ganz zuverlässig funktioniert es nicht, denn manchmal muss man zweimal klicken. Der Ereignis scheint nicht der richtige zu sein. Besser ist der Ereignis:

```
private void treeView1_NodeMouseClick(object sender,
TreeNodeMouseClickEventArgs e)
        {
            fileListBox1.Path = e.Node.FullPath;
            fileListBox1.Update();
        }
```

dann wird in e der angeklickte Knoten übergeben. Dann funktioniert es sicher. Jetzt fehlen noch zwei Dinge, dann ist der Miniexplorer fertig. Bei jedem Öffnen eines Knotens sollten die Unterverzeichnisse dieses Knotens eingelesen werden, so dass man sich durch den gesamten Baum bewegen kann. Das geschieht in dem folgenden Ereignis so:

```
private void treeView1_BeforeExpand(object sender,TreeViewCancelEventArgs e)
        {
            foreach (TreeNode node in e.Node.Nodes)
            {
                ReadSubDirs(node);
            }
        }
```

Jetzt fügen wir noch hübsche Bildchen hinzu. Dies geschieht mit einer ImageList-Komponente, die mit der TreeView- Komponente verlinkt wird in der Eigenschaft „ImageList“ von treeView1.
Bei mir sieht die imageList1 so aus:

Jetzt bekommen schon mal alle Einträge das Icon mit Nummer 0. Will man bei geöffnetem Directory das grüne Icon anzeigen, dann muss man allen Knoten, die man mit Add hinzufügt, mit

```
e.Node.SelectedImageIndex = 1;
```

dieses mitteilen. Ich hab diese Zeile im Click- Ereignis ergänzt.

Kurs 13 Drucken

Diesen Kurs habe ich mehr oder weniger direkt von Andreas Kühnel [1], Beispiel 18 übernommen. Was muss man tun:

1. Hinzufügen der Print- Komponenten

2. `using System.Drawing.Printing;`
3. Im Konstruktor:`printDialog1.PrinterSettings = new PrinterSettings();`
4. Deklaration folgender Variablen:

```
privateint startSeite;        // Anzahl der Druckseiten
privateint anzahlSeiten;            // aktuelle Seitenzahl
privateint seitenNummer;            // zu druckender Text
privatestring strPrintText;    // der Name der geöffneten Datei
privatestring strDateiName;
```

5. In einer Druck- Funktion (hinter einem Button Druck oder so):

```
printDialog1.AllowSomePages = true;
if (printDialog1.ShowDialog() == DialogResult.OK)
{
      printDocument1.DocumentName = "Dokument";
      // Startwerte abhängig vom zu druckenden Text initialisieren
      switch (printDialog1.PrinterSettings.PrintRange)
      {
            case PrintRange.AllPages:
             strPrintText = textBox1.Text;
             startSeite = 1;
             anzahlSeiten = printDialog1.PrinterSettings.MaximumPage;
             break;
            case PrintRange.SomePages:
             strPrintText = textBox1.Text;
             startSeite = printDialog1.PrinterSettings.FromPage;
             anzahlSeiten =printDialog1.PrinterSettings.ToPage-startSeite+1;
             break;
      }
      // Drucken starten
        seitenNummer = 1;
        printDocument1.Print();
      }
```

6. In dem Ereignis von PrintDocument1_`PrintPage()` folgenden Code

```
StringFormat stringFormat = new StringFormat();
RectangleF rectFPapier, rectFText;
int intChars, intLines;

// Ermitteln des Rectangles, das den gesamten Druckbereich
// beschreibt (inklusive Kopf- und Fusszeile)
      rectFPapier = e.MarginBounds;

// Ermitteln des Rectangles, das den Bereich für den
// Text beschreibt (ausschließlich Kopf- und Fusszeile)
rectFText = RectangleF.Inflate(rectFPapier, 0, -2 *
textBox1.Font.GetHeight(e.Graphics));

// eine gerade Anzahl von Druckzeilen ermitteln
int anzahlZeilen = (int)Math.Floor(rectFText.Height /
textBox1.Font.GetHeight(e.Graphics));
```

```
// die Höhe die textbeinhaltenden Rechtecks festlegen, damit die
// letzte Druckzeile nicht abgeschnitten wird
      rectFText.Height = anzahlZeilen * textBox1.Font.GetHeight(e.Graphics);

// das StringFormat-Objekt festlegen, um den Text in einem Rechteck
// anzuzeigen - Text bis zum nächstliegenden Wort verkürzen
      stringFormat.Trimming = StringTrimming.Word;

//    legt die Druckstartseite fest, wenn es sich nicht um die
// erste Dokumentenseite handelt
while ((seitenNummer < startSeite) && (strPrintText.Length > 0))
      {
        e.Graphics.MeasureString(strPrintText, textBox1.Font,
        rectFText.Size, stringFormat, out intChars, out intLines);
        strPrintText = strPrintText.Substring(intChars);
        seitenNummer++;
      }

// Druckjob beenden, wenn es keinen Text zum Drucken mehr gibt
if (strPrintText.Length == 0)
      {
            e.Cancel = true;
            return;
      }

// den Text an das Graphics-Objekt übergeben
e.Graphics.DrawString(strPrintText, textBox1.Font, Brushes.Black, rectFText,
stringFormat);

// Text für die nächste Seite
// intChars - Anzahl der Zeichen in der Zeichenfolge
// intLines - Anzahl der Zeilen in der Zeichenfolge
e.Graphics.MeasureString(strPrintText, textBox1.Font, rectFText.Size,
stringFormat, out intChars, out intLines);
strPrintText = strPrintText.Substring(intChars);

// StringFormat restaurieren
stringFormat = new StringFormat();

// Dateiname in der Kopfzeile anzeigen
      stringFormat.Alignment = StringAlignment.Center;
      e.Graphics.DrawString(this.strDateiName, textBox1.Font, Brushes.Black,
      rectFPapier, stringFormat);

// Seitennummer in der Fusszeile anzeigen
      stringFormat.LineAlignment = StringAlignment.Far;
      e.Graphics.DrawString("Page " + seitenNummer, textBox1.Font,
      Brushes.Black, rectFPapier, stringFormat);

// ermitteln, ob weitere Seiten zu drucken sind
      seitenNummer++;
      e.HasMorePages = (strPrintText.Length > 0) && (seitenNummer <
      startSeite + anzahlSeiten);

// Neuinitialisierung
if (!e.HasMorePages)
{
      strPrintText = textBox1.Text;
      startSeite = 1;
      anzahlSeiten = printDialog1.PrinterSettings.MaximumPage;
      seitenNummer = 1;
}
```

Zuvor müssen noch bei den Komponenten *printDialog1, printPreviewDialog1* und *pageSetupDialog1* die Eigenschaft Document auf *printDocument1* gestellt werden.

Kurs 14 DLL

Das Einbinden von DLL – ist relativ einfach, wobei ich feststellen muss, dass es verschiedene Typen von DLL- Dateien gibt. Ich zeige zuerst die Methode, die für die Dll geeignet ist, die für Nutzung unter Visual C++ oder Borland C++ geschrieben wurden. Dazu gehören in der Regel auch die Interface – DLL von AD- Karten wie Orlowski- USB Karte, Vellemann USB- Karte, oder andere Hardwareergänzungen. Wenn allerdings, wie vermehrt zu beobachten auch C# - DLLs geliefert werden, benutzt man natürlich diese.
Dazu zuerst einmal der Hinweis auf die unterschiedliche Bezeichnung der Variablentypen in C/C++ und C#. Hier ein Auszug aus Frank Eller [3]: **C#- Typen**

Datentyp	Größe	Wertebereich	Alias
bool	8 Bit	true, false	System.Boolean
byte	8 Bit	0 bis 255	System.Byte
sbyte	8 Bit	–128 bis +127	System.Sbyte
char	16 Bit	ein Unicode-Zeichen	System.Char
decimal	128 Bit	$\pm 1.0 \times 10^{28}$ to $\pm 7.9 \times 10^{28}$	System.Decimal
double	64 Bit	$\pm 5.0 \times 10^{324}$ to $\pm 1.7 \times 10^{308}$	System.Double
float	32 Bit	$\pm 1.5 \times 10^{45}$ to $\pm 3.4 \times 10^{38}$	System.Single
int	32 Bit	–2,147,483,648 bis +2,147,483,647	System.Int32
uint	32 Bit	0 bis 4,294,967,295	System.UInt32
long	64 Bit	-9,223,372,036,854,775,808 bis 9,223,372,036,854,775,807	System.Int64
ulong	64 Bit	0 bis 18,446,744,073,709,551,615	System.UInt64
short	16 Bit	–32768 bis +32767	System.Int16
ushort	16 Bit	0 bis 65535	System.UInt16

Tabelle 4.1: Die integralen Datentypen von .NET

Im Gegensatz dazu sehen Sie hier die ganzzahligen Datentypen in C/C++:

Aus Kaiser C++ mit Borland C++ Builder

3.3 Ganzzahldatentypen

Variablen, deren Datentyp ein Ganzzahldatentyp ist, können ganzzahlige Werte darstellen. Je nach Datentyp können dies ausschließlich positive Werte oder positive und negative Werte sein. Der Bereich der darstellbaren Werte hängt dabei davon ab, wie viele Bytes der Compiler für eine Variable des Datentyps reserviert und wie er diese interpretiert.

In C++ gibt es die folgenden Ganzzahldatentypen:

Datentyp	Wertebereich im C++Builder	Datenformat
signed char ***char*** (Voreinstellung)	–128 .. 127	8 bit mit Vorzeichen
unsigned char	0 .. 255	8 bit ohne Vorzeichen
short int	–32768 .. 32767	16 bit mit Vorzeichen

Datentyp	Wertebereich im C++Builder	Datenformat
short *signed short* *signed short int*		
unsigned short int *unsigned short* ***wchar_t***	0 .. 65535	16 bit ohne Vorzeichen
int *signed* *signed int* ***long int*** *long* *signed long* *signed long int*	–2,147,483,648.. 2,147,483,647	32 bit mit Vorzeichen
unsigned int *unsigned* ***unsigned long int*** *unsigned long*	0 .. 4,294,967,295	32 bit ohne Vorzeichen
long long (siehe Abschnitt 3.3.8)	–9223372036854775808 ..9223372036854775807	64 bit mit Vorzeichen
unsigned long long (siehe Abschnitt 3.3.8)	0 .. 18446744073709551615	64 bit ohne Vorzeichen
bool	*true, false*	

3.3 Ganzzahldatentypen 81

Die Funktionsaufrufe der DLL sind in der Regel mit den C++ Datentypen deklariert, da muss man dann aufpassen. Z.B. ist der *char* in C ist 8 bit, der *char* in C# ist 16 Bit.
Die Header Dateien, die meist mit den DLL geliefert werden, sind leider für C# nicht direkt nutzbar. Am sichersten ist die Erzeugung einer Lib- Datei aus der DLL, aus der man dann genau entnehmen kann, welche Funktionen wirklich in der DLL verborgen sind.

```
using System.Runtime.InteropServices;
```

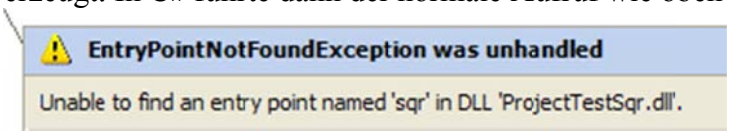
Dann außerhalb einer Funktion im Deklarationsteil:

```
[DllImport("ProjecttestSqr.DLL")]
public extern static double sqr(double x);
```

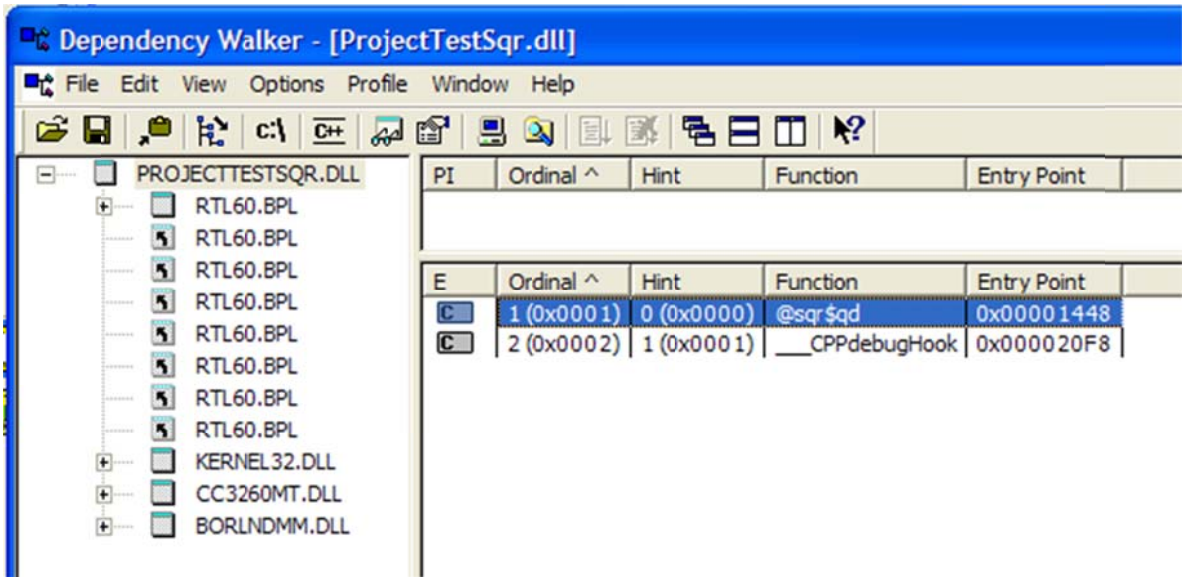
Diese Datei muss das Programm finden können, deshalb am Einfachsten kopieren nach *Projekt…\bin\debug*. Sollte nun der Fall auftreten, dass C# diese Funktion *sqr* nicht finden kann, obwohl Sie sicher sind, dass diese Funktion drin sein muss, dann hilft nur die Möglichkeit, diese Dll zu untersuchen. Ich habe in Foren den Hinweis auf ein Freeware Tool namens DependencyWalker gefunden, mit dem eine DLL analysiert werden kann. Man lädt die DLL und er listet alle möglichen exportierten Funktionen auf. Beispiel: Hab mit dem „alten C++ Builder eine Dll erzeugt mit folgendem Code:

```
double __declspec(dllexport)  sqr(double x)
{
    return (x*x);
  }
```

Also eine einfache Quadratur der übergebenen Zahl x.
Daraus hab ich die DLL namens `ProjectTestSqr.dll`
erzeugt. In C# führte dann der normale Aufruf wie oben beschrieben zu der Fehlermeldung



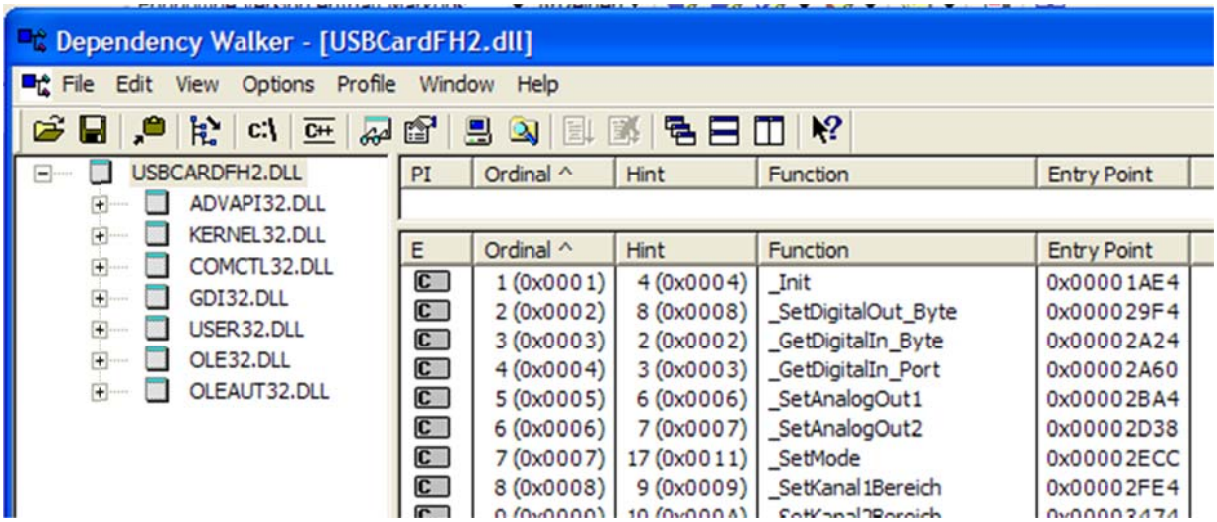
Erst eine Suche mit DependencyWalker brachte des Rätsels Lösung: Screenshot:



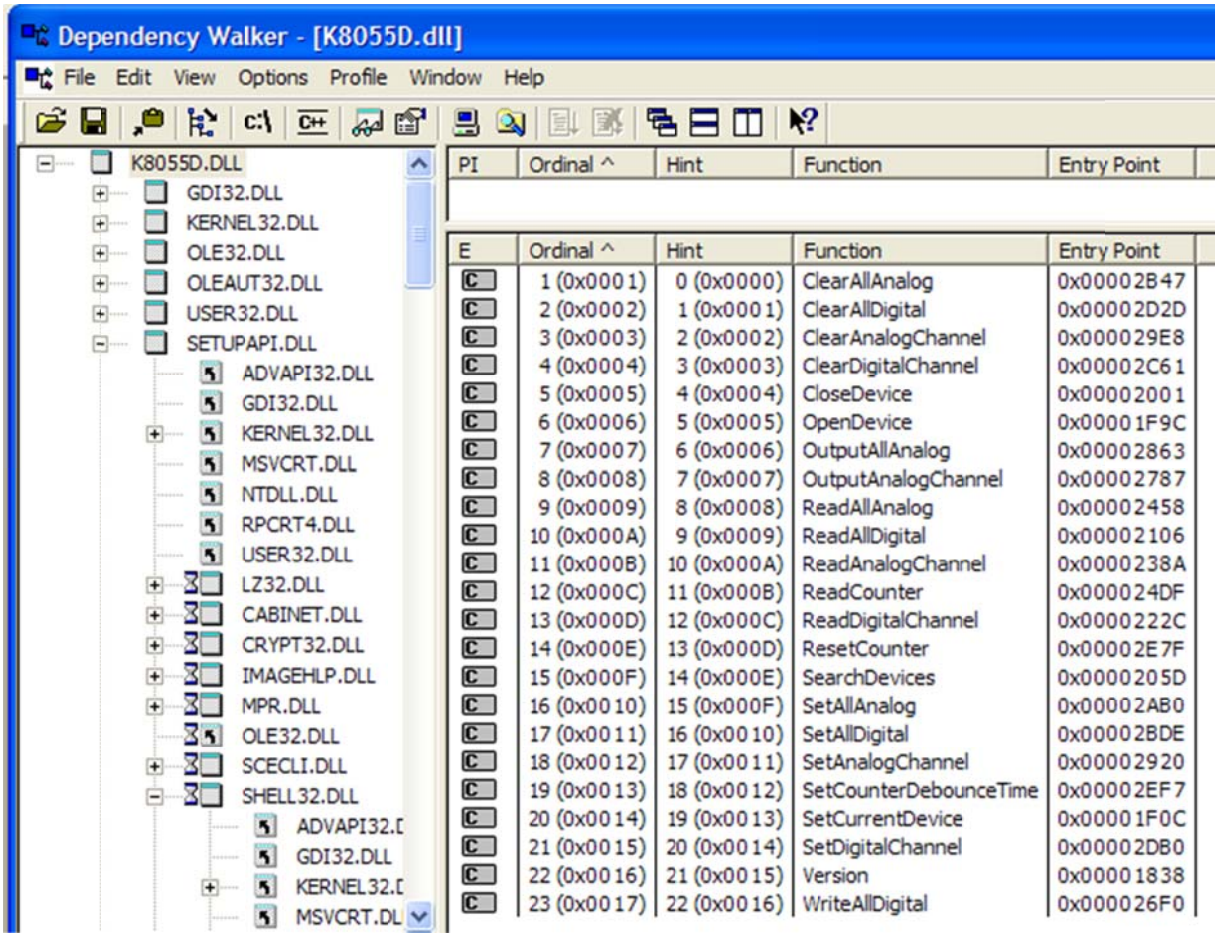
Der Name der Funktion ist dort mit „@sqr$qd“ angegeben. Damit führte dann folgender Aufruf zum Erfolg:

```
[DllImport("ProjectTestSqr.dll",EntryPoint="@sqr$qd")]
public extern static double sqr(double x);

private void button1_Click(object sender, EventArgs e)
{
      double y = Convert.ToDouble(textBox1.Text);
      textBox1.Text = sqr(y).ToString("F5");
```

So zeigte sich, dass die DLL für das USB- ADC- Interfaceboard von Prof. Dr. Orlowski (in FH erhältliche) namens „USBCardFH2.dll“ jetzt alle Namen mit einem Unterstrich davor wieder findet: Screenshot DependencyWalker dieser Dll:



Die Anbindungen der DLL der Vellemann – USB- Karte mit der DLL K8055D lieferte folgendes Bild, alle Namen tauchen wie in der Doku auf:



Natürlich ist hier nicht entnehmbar, wie die Parameterübergabe definiert ist, dass muss der Doku oder der Header – Datei der Dll entnommen werden.

Zusammenfassung AD- DA mit dll für ME2600, USB Orlowski:

ME2600

Vorher einbinden von *ME2600.dll.* und `using System.Runtime.InteropServices;`
DA und AD haben 12 Bit Auflösung. Die übergebenen Werte bilden -10 bis +10 Volt ab in iValue 0 bis 4095.

Initialisieren

```
[DllImport("ME2600.DLL")]
public extern static int me2600AOSetMode(int iBoardNumber, int iChannel, int
iRange, int iMode);
```

Beispiel

```
ex=me2600AOSetMode(0, 0, me2600Def.AO2600_MINUS_10,
me2600Def.AO2600_TRANSPARENT);
```

Auf DA Schreiben

```
[DllImport("ME2600.DLL")]
public extern static int me2600AOSingle (int iBoardNumber, int iChannel,
short iValue);
```

Beispiel

```
short outval = (short)numericUpDown1.Value;
int ex=me2600AOSingle(0, 0, outval);
```

Von DA Lesen

```
[DllImport("ME2600.DLL")]
public extern static int me2600AISingle(int iBoardNumber, int iChannel, int
iMode, refshort iValue);
```

Beispiel

```
short val=0;
int ex=me2600AISingle(0, (int)numericUpDownADChan.Value,
me2600Def.AI2600_MINUS_10 + me2600Def.AI2600_SINGLE_ENDED, ref val);
```

USB- Orlowski – Karte – alte USB 1.1 - Version

Vorher einbinden von der *USBCardFH2.dll* und
`using System.Runtime.InteropServices;`

Initialisieren

```
[DllImport("USBCardFH2.dll")]
public extern static bool _Init();
[DllImport("USBCardFH2.dll")]
public extern static bool _SetKanal2Bereich(byte a);
[DllImport("USBCardFH2.dll")]
public extern static bool _SetMode(byte a);
```

Beispiel

```
bool OK = _Init();
_SetKanal2Bereich(1);//bip1_4V=1, bipolar -4 bis +4Volt
_SetMode(2);          //StartRealtime=2
```

Auf DA Schreiben

```
[DllImport("USBCardFH2.dll")]
public extern static bool _SetAnalogOut1(ushort a);
```

Beispiel

```
ushort Ua1=3000;
_SetAnalogOut1(Ua1);   // Einstellen einer Spannung in 3V am D/A-Ausgang 1
```

Von DA Lesen

```
[DllImport("USBCardFH2.dll")]
public extern static bool _RealtimeData(ref float x);
```

Beispiel

```
float [] buf = new float [8];
double fak = 10.0 / 4096.0;
_RealtimeData(ref buf[0]);  // Abholen der Messwerte von Kanal 2;
label2.Text = "Ua= " + (buf[1] * fak).ToString("F3") + " V";
```

Neue DLL / Klassenbibliothek mit C# erzeugen

Um eine eigenen neue Dll zu erzeugen, öffne man ein neues Projekt mit der Eigenschaft

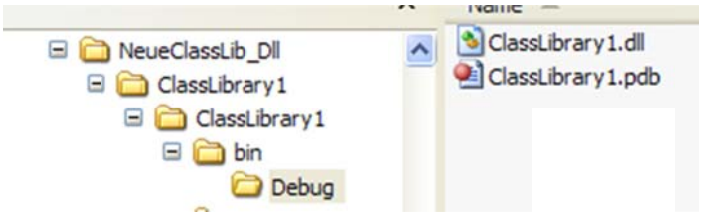
Klassenbibliothek.

Ich habe dort die gleiche DLL mit der Quadratfunktion erzeugt wie mit dem Borland CBuilder:
Syntax:

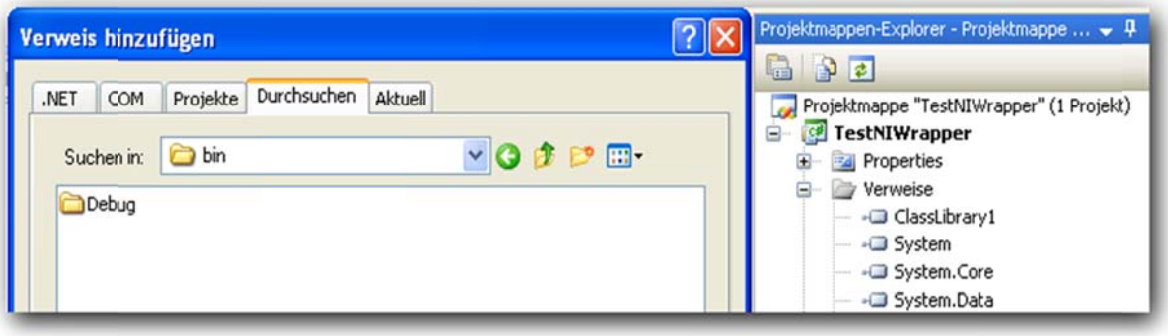
```
namespace ClassLibrary1
{
      public class Class1
    {
            public static double sqrCSharp(double x)
            {
                  return (x * x);
            }
    }
}
```

Wichtig ist vor der Methodendefinition das Schlüsselwort *static*, erst dann wird die Funktion *sqrCSharp* exportiert und benutzbar.

Nach „Build Project“ ist eine DLL verfügbar. Mit default – Namen sieht es jetzt so aus:



Will man diese DLL in einem anderen Projekt benutzen, so füge man sie als Verweis hinzu: (rechte Maus auf Verweise, dann Verweis hinzufügen, dann kommt linkes Bild.)



anschließend ist dann die Funktion *sqrCSarp* verfügbar und kann benutzt werden:

Beispiel:
```
double y = Convert.ToDouble(textBox1.Text);
textBox1.Text = ClassLibrary1.Class1.sqrCSharp(y).ToString("F5"); ;
```

Kurs 15 Zeitmessung und 1ms Timer

Die Zeitmessung kann sehr genau mit der Klasse Stopwatch durchgeführt werden. Mit

```
Stopwatch stopwatch1 = new Stopwatch();
```

wird eine Instanz erzeugt. Mit der Eigenschaft `Stopwatch.Frequency` kann abgefragt werden, mit welcher Frequenz der Zähler dieser Stopwatch hochläuft. Bei mir auf dem PC waren es 2,6 GHz !!! Diese Zahl ist Motherboard- abhängig. Mit der Methode *.Start()* läuft der Zähler los, mit *.Stop()* bleibt er stehen. Die Eigenschaft *.ElapsedTicks* enthält dann die Anzahl der Takte zwischen Start und Abfrage . Die Zeit in sec bekommt man dann heraus durch Division von *ElapsedTicks* durch *Frequency*. Laufzeiten kann man dann messen, indem man vor und nach dem zu messenden Befehl *.ElapsedTicks* abfragt. Beispiel:

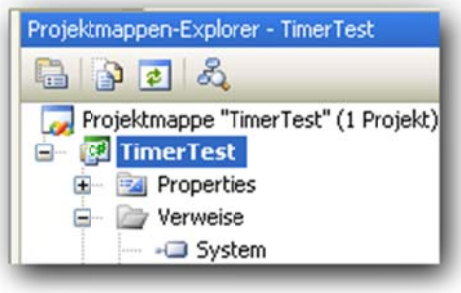
```
long dtstart = stopwatch1.ElapsedTicks;
```
dann kommt der Befehl, von dem man die Laufzeit messen will…
```
long  dtend = stopwatch1.ElapsedTicks;
```

dann
```
long  dt = (dtend - dtstart);
```

Anzeige in Millisekunden:
```
double dtsec = dt/ (double)Stopwatch.Frequency * 1000;
Label1.Text = "Laufzeit= " + dtsec.ToString("f2");
```

Im Projekt „*Test StandardTimer mit Zeitmessung Bay*“ wird nun die Periodendauer des System-Timers gemessen. Dabei läuft *stopwatch* endlos vor sich hin und im *TimerTick* wird er lediglich periodisch abgefragt, die Abstände der Ticks berechnet und in einem Speicher TimeAvr aufakkumuliert, um den Durchschnittswert zu bestimmen. Mit einem zweiten Timer wird dann zweimal in der Sekunde das Ergebnis angezeigt.
Man stellt fest, dass der System - Timer keine 1ms – Auflösung besitzt. Die Periode kann nur in groben Schritten verstellt werden. Bei meinem Desktop ist dies etwa 15 ms, die Intervallzeit kann nur ein Vielfaches dieser 15 ms betragen.
Ich frage mich, wie es möglich ist, dass die C# -Entwickler wirklich einfach nicht in der Lage sind, diesen Timer auf 1 ms genau hinzubekommen. Seit Jahrzehnten (schon bei Borland) bietet man einen Timer an, in dem das Intervall auf 1 ms verstellt werden kann, tatsächlich ist er in Wirklichkeit nur 10 – 15 ms genau.
Ich bin froh, dass es meinem Drittherstellter Mario Prantl (mario.prantl@hh-system.com) genauso geht wie mir. Er hatte für Borland einen 1 ms Timer entwickelt, ist inzwischen auch auf C# umgestiegen und bietet auch dafür seinen 1 ms Multimediatimer an. Die FH hat eine Campuslizenz für 150 € gekauft. Industrienutzer müssen sich eine eigene Lizenz besorgen.

Dieser Timer ist als DLL verfügbar. Man fügt ihn wie in Kurs 14 schon beschrieben als Referenz hinzu. Im Projekt „*Test MMTimer mit Zeitmessung Bay*“ findet man dann den um diesen Timer erweiterten Code. Nach Ergänzen von




```
using MPS;
```

instanziiert man ihn mit

```
TMultimediaTimer MMTimer1;
```

und initialisiert ihn mit

```
MMTimer1 = new TMultimediaTimer(null);
```

Er verfügt ebenfalls über die Eigenschaft *.Interval* und wird gestartet mit der Methode *.Enabled()*. Interessant sind der Zugriff und die Erzeugung der Ereignisfunktion. Da es ja keine Komponente ist, sondern nur ein Objekt einer Klasse, kann man auch auf keine Eigenschaften und Ereignisse im Property – oder Eigenschaftsfenster zugreifen.
Bei den Email – Objekten ging dies mit

```
pop3.OnWork +=new TWorkEvent(this.Work);
```

mit dem das Ereignis Work dann verfügbar war.
Hier wird dies mit „Delegates“, einem neuen Konzept von C# realisiert. Das Konzept ist schwierig zu verstehen (an dieser Stelle nicht nötig) die Durchführung sieht furchtbar einfach aus. Mit

```
MMTimer1.OnTimer = EventVomMMTimer;
```

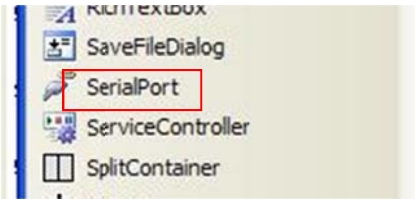
wird die Ereignisfunktion `EventVomMMTimer` erzeugt und kann dann wie die beim Standard – Timer bekannte _Tick( ) – Ereignisfunktion benutzt werden:

```
void EventVomMMTimer(TMultimediaTimer sender)
        {
            ……
         }
```

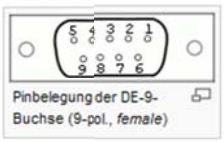
Im Projekt erkennt man, dass jetzt der Multimediatimer eine Auflösung von 1 ms besitzt.
Eine genaue Überprüfung dieser Periodendauer mit gemessener Zeit und einem eventuell vorhandenen Jitter (Schwankung der Periodendauer) ist möglich, wenn man mit diesem Timer über eine AD- Karte, die schnell genug ist, in jedem Event einen Impuls auf einem DA-Ausgang erzeugt und mit einem Digitalscope misst. Dies wird in dem Projekt *MMTimer mit*

DA Impuls auf ME2600 realisiert. Dies Programm läuft natürlich nur mit einer ME2600 – Meilhauskarte im PC. Aber mit einer anderen Karte (keiner USB, USB ist nicht schnell genug) ginge es natürlich auch. Das Verhalten ist identisch dem Multimediatimer unter Borland CBuilder. Es gibt eine von der Last von Windows abhängigen Schwankung, und statt 1 ms sind es 1024 usec Periodendauer auf einem „BlueChip – PC.

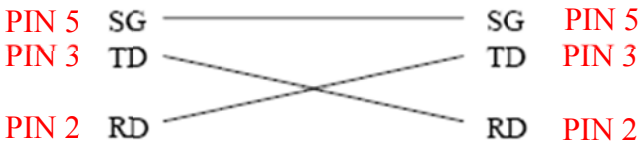
Kurs 16 RS232



Eine Verbindung mit einer seriellen Schnittstelle lässt sich sehr einfach aufbauen. Man benutzt dazu die Komponente „SerialPort“. Über eine 3- Draht- Verbindung kann man jetzt eine Voll- Duplex- Verbindung zweier Geräte erreichen. Dies bedeutet eine gleichzeitige Übertragung von Daten in beide Richtungen. Man muss nur darauf achten, dass erstens der Sendeausgang („Transmit“) des einen Gerätes mit dem Sendeeingang („Receive“) des anderen Gerätes zusammengeschaltet sind. Zwei Transmit-Ausgänge dürfen auf keinen Fall miteinander verbunden werden. Bei dem 9 –poligen Sub- D Stecker eines PC sind dies die PIN 3 (Ausgang Transmit) und PIN 2 (Eingang Receive). Die Ground- PIN müssen natürlich auch verbunden werden (PIN 5).



Will man 2 PCs miteinander verbinden, dann benötigt man ein „Nullmodemkabel“, bei dem PIN 2 und 3 über Kreuz verbunden sind:



Damit die Geräte sich aber synchronisieren können, müssen die „Sendefrequenzen“ gleich sein. Diese werden mit der Eigenschaft *BaudRate* eingestellt, Standardwert sind 9600 Baud, das sind 9600 Bits pro Sekunde. Ebenso müssen *Parity* (None,Odd, Even, Mark, Space) und Anzahl der *StopBits* (One, Two, OnePointFive) und die *DataBits* (7 oder 8) übereinstimmen.
Dann gibt es noch Geräte (z.B. die Mikrocontroller) die nur einen 0-5V TTL- Pegel auf der seriellen Leitung haben, die genormte RS232 hat aber einen Pegel von +15 V bis – 15V und ist invertiert zu dem 0-5V -Pegel. Will man solche Geräte verbinden, ist ein Pegelwandler (z.B. der Max232) nötig. Dort sind 2 Wandler für 2 Schnittstellen mit integrierter Spannungserzeugung zu finden. (Conrad Elektronik 1,50 €).
Hat ein PC keinen COM- Port (das ist diese serielle RS232) mehr, dann kann man sich einen USB- Seriell – Adapter kaufen (z.B. DIGITUS USB 2.0 SERIELL ADAPTER, bei Conrad 10 €).
Bevor man startet, kann man mit

```
foreach (string s in SerialPort.GetPortNames())
cbPort.Items.Add(s);
```

z.B. in eine ComboBox cbPort alle auf dem PC verfügbaren COM- Ports ermitteln. Die tatsächlich angeschlossene muss dann in der Eigenschaft „PortName“ eingestellt werden, also z.B.

```
serialPort1.PortName =“COM1“
```

Dann kann dieser Port mit *.Open()* geöffnet werden.

Steht ein Text in einem string tx, so kann man nun einfach Daten senden mit

```
serialPort1.Write(tx);
```

Benutzt man WriteLine, so wird dem Text ein LineFeed (Hex0a) anghängt, allerding kein CarriageReturn (Hex0d). In einer Textbox wird dann kein Zeilenvorschub gemacht. Also am Besten nur Write() benutzen und ggf. einen Zeilenvorschub mit „\x0d“ und „\x0a“ anhängen.

Empfangene Daten lesen kann man mit *.ReadByte()* oder *.ReadChar()*. Damit keine Daten verloren gehen, wenn der Eingangsbuffer voll ist, sollte man dieses in einem Timer periodisch z.B. alle 100 ms durchführen. Erwartet man nur Text, so benutzt man *ReadChar(),* bei binären Daten (z.B. aus unserem Scope) muss man mit *ReadByte()* lesen. Also in einem Timer_Tick – Event (inp ist eine globale string- Variable):

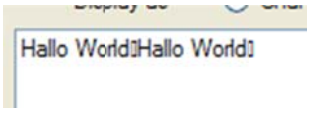
```
if (serialPort1.BytesToRead > 0)
      {
            for (int i = 0; i < serialPort1.BytesToRead; i++)
            {
                  char ch=(char) serialPort1.ReadByte();
                  inp=inp+ch;
            }
                  textBox1.AppendText(inp);
                  inp = "";
            }
```

Voreingestellt ist eine ASCII – Codierung, das heißt, deutsche Sonderzeichen gehen verloren, im Empfangenen Text steht dort ein “?”. Will an diese Sonderzeichen auch lesen können, dann muss z.B. nach Initialisierung der Befehl

```
serialPort1.Encoding = Encoding.Default;
```

ausgeführt werden.
In der bool - Eigenschaft *.IsOpen* kann man abfragen, ob ein Öffnen geklappt hat, denn andere Prozesse können ja auch einen COM- Port belegen.
Will man Eigenschaften wie *BaudRate* verändern, muss zuvor der Prot geschlossen werden mit *.Close().*

Noch ein Hinweis zur *TextBox*. Schreibt man die empfangenen Zeichen in die *Textbox* mit *AppendText*(), so wird dort nur eine neue Zeile gezeichnet, wenn die Sonderzeichen „\x0d“ und „\x0a“ hintereinander auftauchen oder nur LineFeed („\x0a“) auftaucht. Löst man dann aber den Text mit *.Lines* in ein String- array auf, dann wird in diesem Stringarray auch schon bei einem CarriageReturn - Zeichen („\x0d“) eine neue Zeile erzeugt. Will man also auf einzelne Zeilen zugreifen, kann man sich verzählen. Beispiel:



Schreibt man „Hello World“ + „\x0d“ +„Hello World“ in eine *textBox1*, so sieht man auf dem Bildschirm linkes Bild, aber sowohl in *textBox1.Lines[0]* als auch in *textBox1.Lines[1]*steht dann „Hello World“.

Will man unsere Scopes (z.B. Phillips Scope PM3394 oder baugleich PM 3380) auslesen, dann muss man zuerst einen Befehl z.B. QW001 mit einem anschließenden CarriageReturn („\x0d“) senden (kein LineFeed!!! hinterher). Die Nummer nach QW gibt die Kanalnummer an, 001 steht für Kanal a usw.
Dann wird bei Fehlerfreiheit ein „0“ mit CarriageReturn gesendet, anschließend kommt ein ASCII – Header aus 17 mit Komma getrennten Informationen. Siehe dazu Handbuch Kapitel 6 ab Seite 6-33.

Anschließend nach diesem Header werden die Werte der Abtastpunkte in Binärform übertragen. Das erste Byte ist ein vorzeichenbehaftetes High-Byte (Type sbyte), das zweite das vorzeichenlose low-Byte (type byte). Der Spannungswert des Punktes ergibt sich zu Y_zero+Y_resolution*(High*256+low) und der Zeitwert errechnet sich zu t=X_zero + X_resolution*num, wobei num die Nummer des Abtastpunktes ist. Das Auslesen der seriellen Schnittstelle muss hier mit ReadByte() durchgeführt werden.
Manche Multimeter benötigen als Sparversion für eine Pegelwandlung auf dem DTR- Signal (PIN 4) eine positive Spannung, sonst geht gar nichts. Dies kann man erreichen, wenn man

```
serialPort1.DtrEnable = true;
```

einmal ausführt
Auszug aus Scope – Bedienanleitung siehe nächste Seite.

wave_nr Die Quelle der Oszilloskopwellenform:
001 - 004 für CH1 - CH4
011 - 084 für m1.1 - m8.4
091 - 504 für m9.1 - m50.4 (nur bei erweitertem Speicher)

admin 16 Parameter, getrennt mit einem ",":

PARAMETER	TYP	BEISPIEL	ANMERKUNGEN
trace_name	Zeichenfolge	m4.1	
Y_unit	Zeichenfolge	V	
X_unit	Zeichenfolge	s	
Y_zero	Zahl	3	
X_zero	Zahl	-8.625E-6	
Y_resolution	Zahl	78.13E-3	
X_resolution	Zahl	1E-6	
Y_range	Zahl	65535	
date	Zeichenfolge	0000-00-00	
time	Zeichenfolge	00:00:00	
dT-corr	Zahl	375E-3	dT = dT-corr * X-Auflösung
min/max	Zahl	0	1 = min/max-Spur
reserved	Zeichenfolge	0	
reserved	Zeichenfolge	0	
reserved	Zeichenfolge	0	
reserved	Zeichenfolge	0	

count Acquisitionslänge:
512, 8192,16384 oder 32768
(Erweiterter Speicher: 2048 und mehr)

sample 2 Bytes (Höchstwertiges Byte zuerst), die den 16-Bit Probenwert darstellen (Bit 16 = -32768 ... Bit 1 = 1)
Bereich: -32k (down) ... +32k (up)

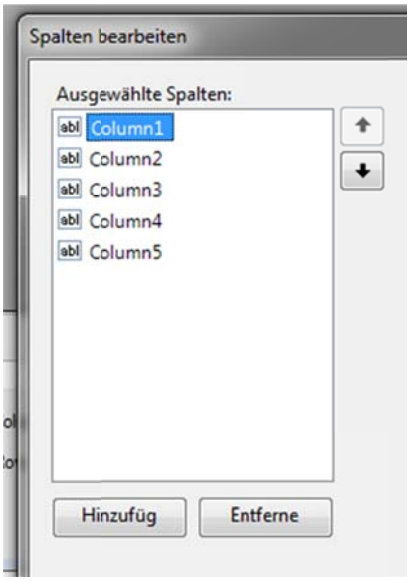
checksum 1 Byte Prüfsumme über sämtliche Proben-Bytes

Kurs 17 bis 20 entfernt

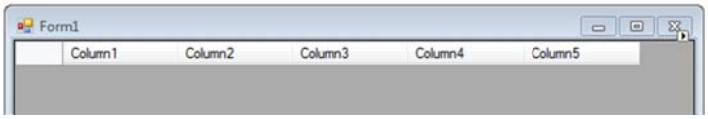
Kurs 21 DataGridView

In diesem Kurs soll die recht komplexe Komponente DatDataGridView eingeführt werden.
Dazu starten wir ein neues Projekt im Ordner Kurs 20# und nennen das Projekt TestGridView.
Dann eine Komponente DataGidView auf das Formular ziehen mit Dock Top.
Wir öffnen den Spalteneditor im Eigenschaftenfenster, dort unter „Columns“ die Auflistung öffnen:

Dann fünf Spalten hinzufügen:



Das Formular sieht dann so aus:



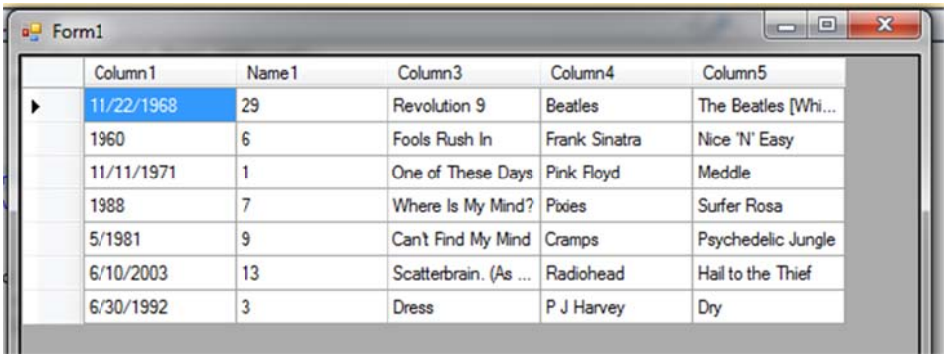
Es soll diese Komponente mit fest programmierter Information gefüllt werden. Dazu den nächsten Programmabschnitt in ihr Programm außerhalb des Konstruktors einfügen:

```
string[] row0 = { "11/22/1968", "29", "Revolution 9", "Beatles", "The Beatles [White Album]" };
string[] row1 = { "1960", "6", "Fools Rush In","Frank Sinatra", "Nice 'N' Easy" };
string[] row2 = { "11/11/1971", "1", "One of These Days", "Pink Floyd", "Meddle" };
string[] row3 = { "1988", "7", "Where Is My Mind?", "Pixies", "Surfer Rosa" };
string[] row4 = { "5/1981", "9", "Can't Find My Mind","Cramps", "Psychedelic Jungle" };
string[] row5 = { "6/10/2003", "13", "Scatterbrain. (As Dead As Leaves.)","Radiohead","Hail to Thief"};
string[] row6 = { "6/30/1992", "3", "Dress", "P J Harvey", "Dry" };
```

Damit werden 7 string- Arrays vorbelegt mit je 5 Elementen.
Hinter einer Taste mit Namen buttonLoad dann folgenden Text kopieren:

```
dataGridView1.Rows.Add(row0);
dataGridView1.Rows.Add(row1);
dataGridView1.Rows.Add(row2);
dataGridView1.Rows.Add(row3);
dataGridView1.Rows.Add(row4);
dataGridView1.Rows.Add(row5);
dataGridView1.Rows.Add(row6);
dataGridView1.Columns[1].HeaderText = "Name1";
```

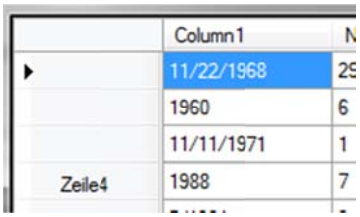
Nach Start und Klick auf diesen button sieht dann die Form so aus:



Man beachte, wie die Spaltenüberschrift der zweiten Spalte mit dem Befehl
`dataGridView1.Columns[1].HeaderText = "Name1";`
verändert werden kann, also Zählung von Null.

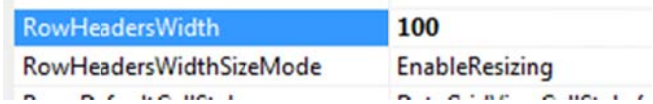
Fügt man folgende Zeile hinzu:

```
dataGridView1.Rows[3].HeaderCell.Value = "Zeile4";
```



Dann wird so auch eine Zeilenbeschriftung ganz rechts möglich. Man kann die Größe automatisch einstellen lassen mit z.B.

Oder man gibt in der Eigenschaft „RowHeaderWidth“ die Größe in Pixel an. Das geht aber nur so mit „EnableResizing:



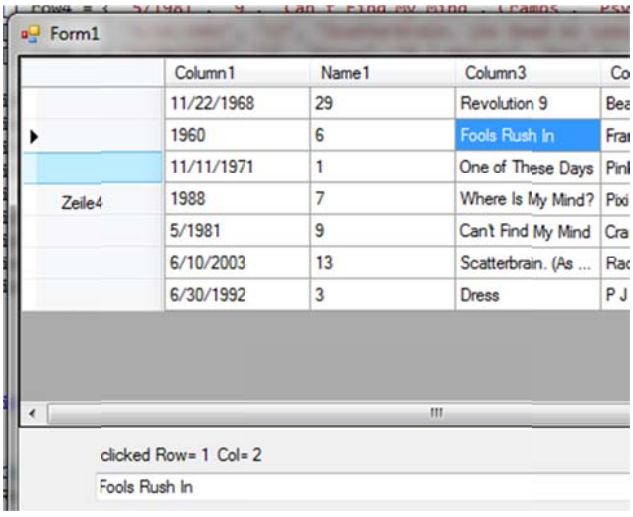
Jetzt sollen die Daten aus der Komponente ausgelesen werden. Wie kommt man an die Inhalte dran: dazu bitte eine label3 und eine textBox1 hinzufügen und einen Eventhandler hinter einem CellClick erzeugen. Dann folgenden Code dort eingeben:
Außerhalb der function:

```
int c=0, r=0;
```

Innerhalb:

```
c = e.ColumnIndex;
r = e.RowIndex;
label3.Text="clicked Row= "+r.ToString()+"  Col= "+c.ToString();
if ((c>=0)&&(r>=0)&&(c<dataGridView1.ColumnCount)&&(r<dataGridView1.RowCount))
      textBox1.Text = dataGridView1[c,r].Value.ToString();
```

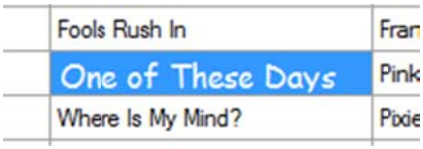
In dem Parameter e kann man den Spalten- und Zeilenindex abfragen, diese werden nach c und r geschrieben und in label3 ausgegeben. Das Auslesen geschieht wie bei einem zweidimensionalen Array mit …[c,r] per Eigenschaft Value.
Ein Beispiel erkennt man im nächsten Screenshot, auch da hat die Zelle oben links die Zeile Null und die Spalte Null:



Man kann jetzt z. B. in jeder Zelle einen eigenen Font einstellen. Beispielprogramm dazu hinter einer Taste buttonFont, vorher einen FontDialog auf die Form ziehen:

```
  if (fontDialog1.ShowDialog() == DialogResult.OK)
              dataGridView1[c, r].Style.Font = fontDialog1.Font;
```

Z.B. Zelle 2,2 mit Comic Sans MS, Schriftgröße 12 :

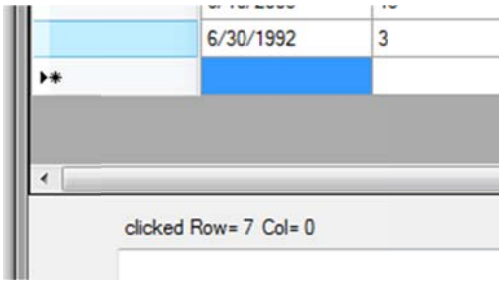


Man muss darauf achten, dass keine Zelle angesprochen wird, die es nicht gibt, denn sonst bricht das Programm mit Fehlermeldung ab. Da bietet sich eine try – catch – Abfrage an.

Soll der User Zeilen hinzufügen können, dann muss folgende Eigenschaft auf true gesetzt werden:

```
dataGridView1.AllowUserToAddRows = true;
```

dann gibt es eine Leerzeile am Ende der Tabelle. Wenn dort etwas hineingeschrieben wird, sieht es so aus:



Nur dann gibt es bei einem Klick in diese neue Zeile sofort eine Fehlermeldung, da der Inhalt sofort in die Textbox geschrieben werden soll. Dieser existiert aber noch nicht. Die Eigenschaft RowCount ist schon um eins erhöht worden, aber für den Inhalt ist noch kein Speicherbereich zugewiesen worden. Das kann man abfangen mit folgendem if:

```
if (dataGridView1[c, r].Value!=null) textBox1.Text = dataGridView1[c, r].Value.ToString();
```

Erst wenn die Zeile verlassen wird, wird der Inhalt aktualisiert und der Stern verschwindet.

Kurs 22 C# DirectX

Hier sind alle Informationen und ein kleines Beispielprogramm auf der Seite http://www.fh-luebeck.de/Inhalt/03_Hochschulangehoerige_Ch031/04_FB_Elektrotechnik/04_Labore_und_Institute/63_Labor_ESA_RT/Visual-C_/DirectX_mit_C_/index.html
zu finden. Es wird eine x- Datei angezeigt, die natürlich die 3D- Bildinformation enthält. Solche Dateien kann man mit dem Freewareprogramm Googlesketch erstellen, das man im Internet findet. Auf der Seite sind Screenshots des C# - Programmes und vergleichsweise eines DirektX- Viewers zusehen.

Kurs 23 MySql und C#

Auf der Seite http://www.fh-luebeck.de/Inhalt/03_Hochschulangehoerige_Ch031/04_FB_Elektrotechnik/04_Labore_und_Institute/63_Labor_ESA_RT/Visual-C_/MYSQL/index.html
findet man alles, was man wissen muss, um mit C# eine mySL- Datenbank anzusprechen. Dies wurde in einer Diplomarbeit von Frau Yannan Shen erarbeitet, die dort auch zur Verfügung gestellt wird. Ein Beispielprogramm vervollständigt das Angebot. Dort ist ein

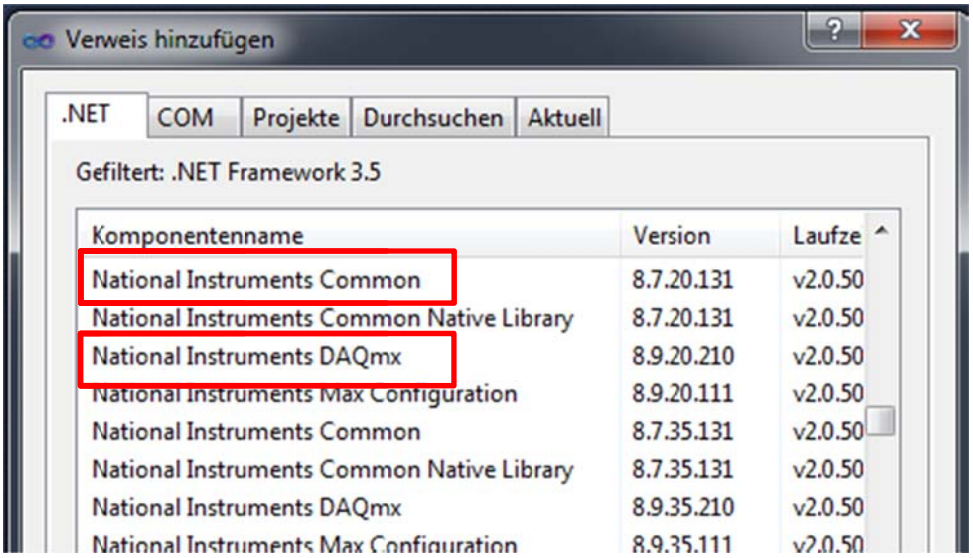
Anwesenheitszähler mit Source zu finden, der es ermöglicht, in einer Veranstaltung mit Studierenden den Barcode des Ausweises zu scannen. Die Anwesenheitszeit und Datum werden dann in einer MySQL- Tabelle und die Studierendendaten in einer zweiten tabelle einer Datenbank abgelegt.

Kurs 24 Meilhaus ME 4660

Inzwischen sind auch modernere Versionen der Meilhauskarte (ME 4660) und der Orlowskikarte (USB 2.0 – Version) entstanden und im Labor verfügbar. Ich habe die Anleitungen dazu im Internet veröffentlicht.
ME4660: (geht sowohl mit 32.Bit als auch mit 64 Bit- Windows!)
http://www.fh-luebeck.de/Inhalt/03_Hochschulangehoerige_Ch031/04_FB_Elektrotechnik/04_Labore_und_Institute/63_Labor_ESA_RT/Visual-C_/Kurs_24_AD_Meilhaus_4660/index.html

Kurs 25 NI mit NIDAQmx

Ab Win7 ist nur noch die neue Interface- Version mit DAQmx zu verwenden, die veraltete NI – traditional darf nicht mehr benutzt werden. Diese hatte zu Beginn den Nachteil, dass die Rechenzeit im Vergleich zu „traditional“ gut um den Faktor 10 größer ist. Dauert eine Single-AD-Abfrage mit „traditional“ ca 300 µsec, so dauerte es auf dem gleichen PC mit gleicher Karte 2-3 ms. Ein Takten mit 1 ms ist dann unmöglich. Dies lag aber daran, dass ein Fehler bei unserem Programm vorhanden war. Eine Task muss gestartet werden, dann gehen die AD- DA – Wandlungen schneller. Wir hatten den Startbefehl nicht gegeben, so hat das Interface bei jeder Wandlung die Task gestartet und danach beendet. Und ein Start dauert halt ein paar msec. Eine richtige Lösung für DAQmx gibt es in meinem Windfc#- Projekt an Version 7.4.15 in der Datei „Hardwaretools /DAQmx_Basics_NChannel.cs“, siehe dort.
Dazu zuerst zwei Verweise hinzufügen:



Anschließend muss dann im Projektmappen-Explorer dann folgendes stehen:

Dann
```
using NationalInstruments.DAQmx;
```

Und dann sind folgende Befehle möglich:
AD auslesen:
```
double val = rschan.ReadSingleSample();
```

Ausgabe auf DA
```
wschan.WriteSingleSample(true, val);
```

zuvor muss man die beiden Tasks rschan und wschan „kreieren“ mit
```
        AnalogSingleChannelReader rschan;
        AnalogSingleChannelWriter wschan;
```

Und das geht so: Erst die Instanziierung der AD- Tasks: (DA geht ähnlich, siehe Datei .cs):
```
Task taskAI = new Task("abci");
```
dann
```
taskAI.AIChannels.CreateVoltageChannel(actdev+"/ai0", "aiChannel0",
       AITerminalConfiguration.Rse,MinI, MaxI, AIVoltageUnits.Volts);
```

Dann
```
rschan = new AnalogSingleChannelReader(taskAI.Stream);
```

dann sollte es nach `taskAI.Start();` gehen.
Weitere Hinweise kann man auf den Internetseiten von mir zu diesem Punkt finden.

Kurs 26 USB- Orlowski 2.0

USB – Orlowski USB 2.0 Version: (geht nur auf 32. Bit Windows)
http://www.fh-luebeck.de/Inhalt/03_Hochschulangehoerige_Ch031/04_FB_Elektrotechnik/04_Labore_und_Institute/63_Labor_ESA_RT/Visual-C_/Kurs_26_AD-_USB_-_Karte_2_0_Orlowski/index.html

Kurs 27 Interface mit TCP/IP

Viele Geräte kommunizieren heute über eine schnelle Ethernet- Verbindung oder über Internet. Das wird in C# auch gut unterstützt. Als Beispiel soll die Kommunikation mit einer WAGO-Klemme gezeigt werden, die in der Industrie als Analog – und Digital- Interface Verwendung findet.
Man benötigt Zugriff auf diesen Namespace:
```
using System.Net.Sockets;
```

dann kann man einen TCP – Client instanziieren:
```
TcpClient tcpClientWAGO = new TcpClient();
```

Hinter einem Button „Connect“ kann man dann mit folgendem Programm eine Verbindung aufbauen:

```
try
  {
    if (!tcpClientWAGO.Connected)
        {
          tcpClientWAGO.Connect("192.168.0.2",502);//tcpClient.Connect("193.175.124.132", 502);
        if (tcpClientWAGO.Connected) toolStripStatusLabel1.Text = "Wago is connected";
        }
        else MessageBox.Show("Verbindung besteht bereits","TCP Connection Info", MessageBoxButtons.OK);
        }
        catch (SocketException)
        {
         MessageBox.Show("Verb.-aufbau nicht möglich","TCP Connection Error", MessageBoxButtons.OK);
        }
```

Dabei wird eine Nachricht in ein toolStripStatusLabel geschrieben, ein StatusStrip muss natürlich dann vorhanden sein.
Wenn eine Verbindung gelungen ist, kann man Lesen und Schreiben mit einem NetworkStream:

```
NetworkStream netStreamWAGO;
```

Den muss man mit der Verbindung zusammenführen. Das Schreiben geht so:

```
netStreamWAGO = tcpClientWAGO.GetStream();
if (netStreamWAGO.CanWrite)
{
   byte[] sendBytes = new byte[12]; //lokales Array mit Daten
   netStreamWAGO.Write(sendBytes, 0, sendBytes.Length);
}
```

Das Lesen geht so:

```
netStreamWAGO = tcpClientWAGO.GetStream();
if (netStreamWAGO.CanRead)
{
    byte[] readBytes = new byte[tcpClientWAGO.ReceiveBufferSize];
    netStreamWAGO.Read(readBytes, 0, (int)tcpClientWAGO.ReceiveBufferSize);
}
```

Anschließend sind im Byte –Array die Daten.

Um jetzt mit den Wago –Klemmen einen DA- Wert auszugeben, muss man das Mod- Bus – Protokoll bedienen. Bei anderen Geräten muss man dann das entsprechend Protokoll den Datenblättern entnehmen.
Das Wago- Protokoll ist den Versuchsunterlagen zu entnehmen und wird am Versuchstag zur Verfügung gestellt.

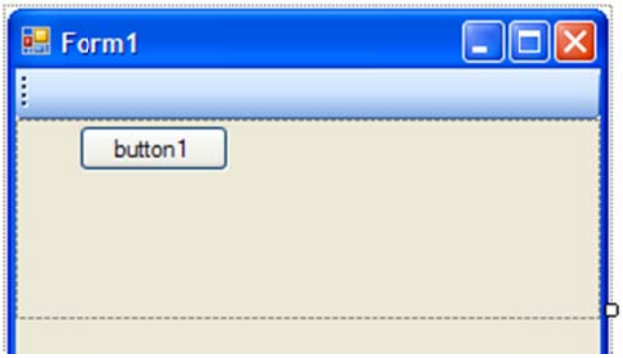
WindfC#, mein großes Regelungstechnik- Toolprogramm.

Weitere Informationen kann man erhalten, wenn man sich das folgende Projekt, das auch als Source- Code in C# erhältlich sind - quasi als Freeware – anschaut:
http://www.fh-luebeck.de/Inhalt/03_Hochschulangehoerige_Ch031/04_FB_Elektrotechnik/04_Labore_und_Institute/63_Labor_ESA_RT/WindfCSharp/index.html

Einige Hinweise

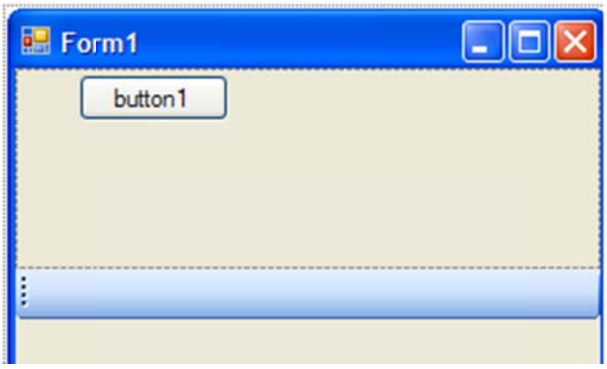
Problematik mit Dock

Werden zwei Elemente mit Dock auf Top gesetzt, erhebt sich die Frage, welcher wirklich zuerst oben und welche dann darunter platziert wird. Beispiel ToolStrip und Panel:




```
// Form1
//
this.AutoScaleDimensions = new System.Drawing.Size
this.AutoScaleMode = System.Windows.Forms.AutoScal
this.ClientSize = new System.Drawing.Size(292, 266
 this.Controls.Add(this.statusStrip1);
this.Controls.Add(this.panel1);
this.Controls.Add(this.toolStrip1);
```

So soll es sein. Manchmal aber passiert es, dass bei beiden Elementen, die auf Dockstyle.Top gesetzt werden, eine falsche Platzierung erfolgt, z.B. so:




```
//
// Form1
//
this.AutoScaleDimensions = new System.Dra
this.AutoScaleMode = System.Windows.Forms
this.ClientSize = new System.Drawing.Size
this.Controls.Add(this.toolStrip1);
this.Controls.Add(this.statusStrip1);
this.Controls.Add(this.panel1);
this.Name = "Form1";
```

Es hängt von der Reihenfolge ab, mit der diese Elemente der Form bzw. dem Container hinzugefügt worden sind. Wie kann man diese nachträglich verändern? So: Man muss die Datei „designer“, die normalerweise nicht angefasst wird, nun doch öffnen. In dem Teil, der oben rechts ausgeschnitten ist, kann man die Zuordnung erkennen. Das Element, das unter dem anderen steht, gewinnt!

Parameter in Funktionsaufrufen und Arrays

Dort gibt es einige Änderungen gegenüber C++:

Beispiele:
Deklaration von Arrays:

C++	C#
double xein[max2n]	`double[] xein = new double[max2n]`

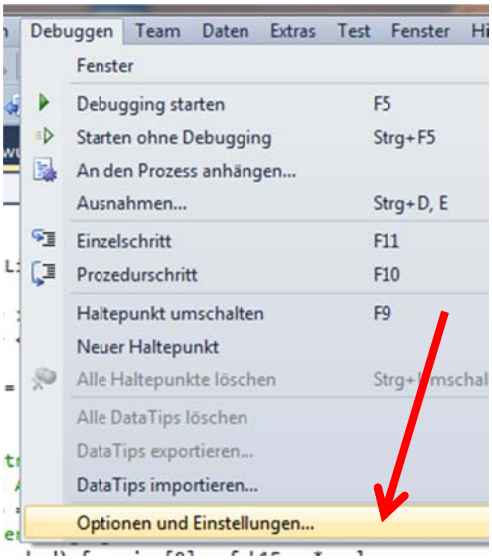
Funktionsaufruf

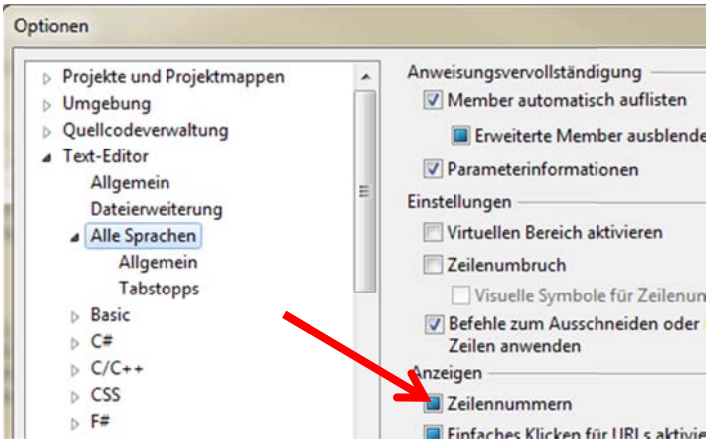
C++	C#
void xr_db(double& xrr, double p[10], double& q[10], int m)	`void xr_db(ref double xrr, double[] p, ref double[] q, int m)`

Xrr wird referenziert aufgerufen, damit können Werte zurückgegeben werden, es wird nur der Pointer übergeben. Das Array p wird in die Funktion hineingegeben und von der Funktion nicht verändert. Das Arry q wird wieder referenziert, also lediglich mit Pointer übergeben und kann von der Funktion verändert werden.

Zeilennummern im Editor

Manchmal helfen Zeilennummern im Texteditor. Sie aktiviert man mit Menü Debuggen → Optionen…
Dann im Text-Editor → Alle Sprachen
Dort Klick auf Zeilennummern:





Weitere Projekte werde ich auf den Internetseiten unter www.fh-Luebeck.de, →Studierende →FB EI → Labore & Institute, → Labor Regelungstechnik → Visual-C# -Page oder Windfc#
demnächst in loser Folge veröffentlichen.

Stand November 2012
Prof. Dr. Bayerlein