

KB00002 : Benutzerfreundliche Kommandozeilen-Anwendungen in C# .NET entwickeln

Mit diesem kleinen Tutorial möchte ich ein paar Erfahrungen beim Erstellen von Programmen in C# und .NET festhalten, die sich sowohl von der Kommandozeile aus, als auch bequem über Windows-Fenster steuern lassen.

Anlass hierfür war die Erstellung einer Reihe von Admin-Tools für eine Windows-Server-Umgebung. Die Programme sollten möglichst viele Funktionen sinnvoll unter einen Hut bringen, sich aber gleichzeitig auch über die Kommandozeile ansprechen lassen. - Zwar werden in der Schönen Neuen Windows Welt fast alle wichtigen Administrationsaufgaben über Fensterchen abgewickelt, aber insbesondere im Hinblick auf die Flexibilität der Shellscript-gesteuerten Systemadministration, wie man Sie traditionell aus Unix-/Linux-Umgebungen kennt und wie sie in Zusammenhang mit der seit 2005 eingeführten „Windows Power Shell“ jetzt einige Jahrzehnte später auch bei Windows möglich wird, wollten wir die Möglichkeit der Steuerung des Programms über Kommandozeilenparameter unbedingt offen lassen.

Erschwerend hinzu kam, dass die Anwender in der Regel „ungeschult“ sind, bzw. so selten mit den Programmen in Berührung kommen, dass ein aufwendiges Einlernen in die Kommandozeilensyntax ineffizient ist und nicht in Frage kommt. Die Programme müssen also trotz Kommandozeile intuitiv zu bedienen sein, bzw. die Kommandozeilensyntax muss auf einfachere Weise zugänglich sein, als durch stundenlanges Lesen der von Ihren Schöpfern meist eher widerwillig zusammengewurstelten Dokumentation.

Die Lösung ist relativ simpel: Wenn der Benutzer nicht zur Kommandozeile kommt, muss die Kommandozeile halt zum Benutzer kommen: Wenn keine Kommandozeilenparameter angegeben werden, dann ist davon auszugehen, dass sich ein "Ottonormalanwender" an dem Programm versucht. In diesem Fall wird ein Kommandozeilen-Assistent gestartet, der bei der Erstellung der richtigen Kommandozeilenparameter hilft. Außerdem eignet sich diese Vorgehensweise als einfache Testmethode für die vollständige Funktionsabbildung über die Kommandozeile.

In diesem Sinne, an die Arbeit!

Zuallererst müssen wir in Visual Studio eine neue Kommandozeilenanwendung erstellen. Das Grundgerüst dieser Anwendung sollte ungefähr so aussehen:

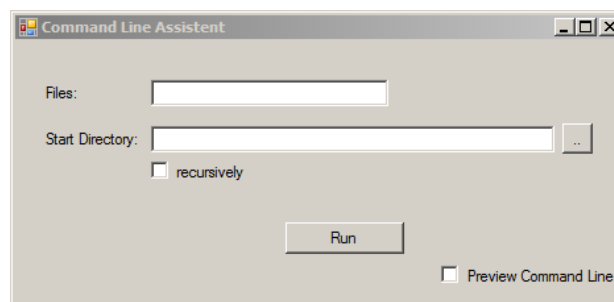
```
using System;
using System.Collections.Generic;
using System.Text;

namespace MyCommandLineApplication
{
    [STAThread]
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

Wichtig ist, das **[STAThread]** vor der Klassendefinition einzufügen, da es ohne diese Compiler-Direktive später Probleme mit bestimmten Steuerelementen wie dem **FolderBrowserDialog** gibt.

Als nächstes benötigen wir ein Formular für den Kommandozeilenassistenten (wir haben es **Form_CLA** getauft):



Als letztes brauchen wir noch ein Kontrollfenster für unsere Kommandozeilenvorschau (wir taufen es **Form_CLD**). Dieses Fenster wird eingeblendet, wenn der Benutzer im Kommandozeilen-Assistenten die "Preview Command Line" Option aktiviert:



Jetzt kommt der Code. Zuerst müssen wir die Entscheidung treffen, ob der Kommandozeilenassistent gestartet werden soll, oder ob gleich mit dem eigentlichen Programm losgelegt werden kann. Dazu überprüfen wir, ob das Array **args[]** mit den Kommandozeilenargumenten leer ist.

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

namespace MyCommandLineApplication
{
    class Program
    {
        [STAThread]
        static void Main(string[] args)
        {
            if (args.Length == 0)
```

```
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form_CLA());
        }
    else
    {
        if ( CheckCmdLineArguments(args) == true)
        {
            RunCoreFunctionalities();
        }
    }
}
}
```

Wenn Parameter gefunden wurden, dann wird die Routine **CheckCmdLineArguments()** gestartet. Diese könnte z.B. so aussehen:

```
static bool CheckCmdLineArguments(string[] args)
{
    string CmdLine = "";

    // Kommandozeile zu einem String zusammenhängenden String zusammenbasteln
    foreach (string arg in args)
    {
        CmdLine = CmdLine + arg + " ";
    }

    try
    {
        // und dann mit regulären Ausdrücken wieder in die Teilvariablen zerlegen -
        (var)={ (value) }

        string input = CmdLine;
        string pattern = @"-(\S*)=([^-]*)|-[^- ]*";
        Regex re = new Regex(pattern);
        MatchCollection matches = re.Matches(input);
        String ParamName, ParamValue = null;

        if (matches.Count > 0)
            foreach (Match m in matches)
            {
                // Treffer und Teilgruppen ausgeben:
                ParamName = m.Groups[1].Value.Trim().ToLower();
                ParamValue = m.Groups[2].Value.Trim();
                ParamValue = ParamValue.Replace("{} ", "");

                if (m.Groups[3].Value.Length > 0)
                {
                    ParamName = m.Groups[3].Value;
                }
            }
    }
}
```

```
    }

    switch (ParamName)
    {
        case "mydir":
            MyDir = ParamValue;
            break;

        case "myfiles":
            MyFiles = ParamValue;
            break;

        case "r" :
        case "recursively" :
            MyRecursively = true;
            break;

        default:
            break;
    }
}
else
    Console.WriteLine("'" + pattern + "' in '" + input + "' nicht
gefunden");

    ConsoleColor oldConsCol = System.Console.ForegroundColor;
    System.Console.ForegroundColor = ConsoleColor.Cyan;
    System.Console.WriteLine(CmdLine + "\r\n");
    System.Console.ForegroundColor = oldConsCol;
}

catch (Exception parseErr)
{
    System.Console.WriteLine("Es trat ein Fehler beim Parsen der Kommandozeile
auf " + parseErr.Message);
    return false;
}
return true;
}
```

Da die Argumente in `args[]` nur nach Leerzeichen getrennt werden, basteln wir zuerst die Kommandozeile wieder zusammen und parsen dann die Kommandozeilenparameter mithilfe regulärer Ausdrücke.

Die erwartete Syntax ist:

-param={values}

oder

-param=value

oder

-flag

In diesem Beispiel gibt es drei mögliche Parameter `-mydir=` oder `-myfiles={..}` oder `-recursively`. Falls keine Fehler auftreten, werden die Werte in die Variablen `MyFiles`, `MyDir` bzw. `MyRecursively` geschrieben und dann die Funktion `RunCoreFunctionalities()` aufgerufen.

Diese gibt, hier nur beispielhaft, die effektiven Kommandozeilenargumente wieder aus. Im echten Programm sollten die geschweiften Klammern entfernt und die Ausdrücke für die Dateifilter weiter aufgesplittet werden.

```
public static string MyFiles = "";
public static string MyDir = "";
public static bool MyRecursively = false;

public static void RunCoreFunctionalities()
{
    MessageBox.Show( "Run Program
using\r\n\r\n[mydir="+MyDir+"]\r\n[myfiles="+MyFiles+
"]\r\n[Recursiv="+MyRecursively+]" );
}
```

Als Nächstes nehmen wir uns das Formular für den Assistenten vor. Zuerst verknüpfen wir die `CheckBox Preview Command Line` mit einem Event zum Öffnen der Kommandozeilenvorschau:

```
public Form_CLP CmdLine = new Form_CLP();

...

private void checkBox_PreviewCL_CheckedChanged(object sender, EventArgs e)
{
    refreshCLP();

    if (checkBox_PreviewCL.Checked)
    {
        CmdLine.Width = Screen.PrimaryScreen.WorkingArea.Width;
        CmdLine.Top = Screen.PrimaryScreen.WorkingArea.Height - CmdLine.Height;
        CmdLine.Left = 0;
        CmdLine.textBox_CL.Width = CmdLine.Width - 20;
    }
}
```

```
        CmdLine.Show();
    }
    else
    {
        CmdLine.Hide();
    }
}
```

Durch die obigen Anweisungen wird das Fenster in den unteren Bereich knapp über der Windows-Startleiste gezwängt, damit es im Blickfeld bleibt. Wichtig, die **StartPosition**-Eigenschaft des CLP-Fensters muss dazu auf Manual stehen.

Danach werden alle nötigen **Changed**-Events für die Eingabefelder mit einer zentralen Routine verknüpft, die die Kommandozeile aus den Werten in den Eingabefeldern zusammensetzt und in die Textbox des CLP-Fensters schreibt:

```
private void refreshCLP()
{
    string cmd = "";

    if (textBox_mydir.Text.Length > 0)
    {
        cmd += " -mydir=" + textBox_mydir.Text;
    }

    if (textBox_FileSpecs.Text.Length > 0)
    {
        cmd += " -myfiles={" + textBox_FileSpecs.Text+"}";
    }

    if (checkBox_Recursively.Checked)
    {
        cmd += " -recursively";
    }

    CmdLine.textBox_CL.Text = "myapp "+cmd;
    MyOptions = cmd;
}
```

Wichtig ist, dass das Eingabefeld der CLP-Box im (Designercode) als public deklariert ist:

```
public System.Windows.Forms.TextBox textBox_CL;
```

Jetzt sollte beim Klick auf, bzw. bei aktivierter Preview Checkbox und gleichzeitiger Eingabe von Text die aktuelle Kommandozeile angezeigt werden.

Als Letztes muss noch der Run-Button scharf gemacht werden. Dazu wird er mit einem Process-Control verknüpft, welches das Programm in einer zweiten Prozessinstanz mit den neuen Parametern startet und gleichzeitig den sich selbst (also den Ursprungsprozess) beendet:

```
private void button_Run_Click(object sender, EventArgs e)
{
    process1.StartInfo.FileName = MyLocation;
    process1.StartInfo.Arguments = MyOptions;

    process1.Start();
    Application.Exit();
}
```

Das vollständige Beispiel kann unter <http://www.jtr.de/kb/KB00002.zip> heruntergeladen werden. Viel Spaß beim Selberbasteln!