Denne guide er oprindeligt
udgivet på Eksperten.dk

COMPUTERWORLD

Plugin-arkitektur med .NET

Denne artikel viser hvordan du kan lave en plugin-arkitektur i dit program. Gør det muligt for andre at skrive udvidelser til det. Smid dem i et dedikeret bibliotek, og lad dit program automatisk opdage og tage dem i brug.

Skrevet den **09. Feb 2009** af **nielle** I kategorien **Programmering / C#** | ★★★☆☆

Indledning

En del færdige programmer giver mulighed for at du som programmør kan lave dine egne udvidelser til programmet i form af *plugins* eller *add-ons*.

I sin simpleste form er en plugin er en lille programstump, f.eks. i form af en DLL, som man kan installere, eller måske blot kopiere til en bestemt lokalisation på harddisken - så vil programmet selv opdage den og gøre den tilgængelig fra programmet i form af ekstra funktionalitet.

Man kan lave en plugin helt uden at man har adgang til kildekoden fra det oprindelige program, eller at nogen behøver at rekompilere noget som helst (andet end selve plugin'en).

Denne artikel giver et forslag til hvordan man kan lave et program i .NET, som stiller en sådan udvidelsesmulighed til rådighed for andre; et program med en plugin-arkitektur.

v. 1.0: 29/12/2007 - Første version.

Demo applikationen

For at demonstrere princippet vil jeg skrive en lille demo applikation. Det er basalt set (starten til) en C# editor, hvor man arbejder med koden via en RichTextBox.

Min plugin-arkitektur vil åbne op for at andre kan lave et plugin som formatere på teksten. I den demo-plugin der kodes her, laves der f.eks. lidt syntax-highlighting på koden.

Demo applikationen er skrevet i C# 2.0 og .Net 2.0, men burde kunne overføres til såvel nyere som ældre versioner; hvis der er noget så er det nok bare Windows-kontrollerne som eventuelt er nogle lidt andre. Selve grundprincippet holder uden tvivl.

Plugin interfacet

Applikationen skal udstikke nogle retningslinjer for hvordan en plugin skal struktureres før at den kan genkende den og bruge den. Dette gøre den ved at definere et interface som plugin's skal implementere:

```
public interface IPlugin
{
    #region Plugin beskrivelse

    string Name { get; }
```

```
    string Description { get; }
    Version PluginVersion { get; }
    string Author { get; }

    #endregion

    #region GUI oplysninger

    string MenuText { get; }
    string MenuToolTipText { get; }

    #endregion
}

public interface IFormatterPlugin : IPlugin
{
    void Format(RichTextBox rtf);
}
```

I dette tilfælde har jeg valgt at definere to:

Det første interface indeholder nogle helt basale oplysninger omkring plugins: Hvem har lavet det, hvad gør det, osv. Man kan sikkert selv komme på flere, som f.eks. copyright.

Desuden indeholder det også nogle oplysninger om hvordan det skal plugges ind i demo applikationens GUI. Dem vil vi komme tilbage til senere.

Det andet interface arver alle egenskaberne fra det første, og desuden definere det en funktion som laver selve formateringen - det var jo en plugin-arkitektur til formatering jeg ville lave.

Hvis demo applikationen skal kunne understøtte flere typer af plugins, skal de hver især blot udvide IPlugin på samme måde som IFormatterPlugin gør det.

Hvor skal interface koden placeres?

Man kan vælge at placere interfacet sammen med resten af koden for demo applikationen, eller man kan vælge at placere den i en separat DLL.

Jeg har her valgt den første mulighed fordi løsningen bliver lidt mere simpel at håndtere, men den ville kræve at folk har din applikation før at de kan bidrage med deres plugins. I det andet tilfælde behøver de kun DLL'en med interfacet for at være kørende.

Plugin handleren

Demo applikationen har behov for at kunne loade og administrere sine plugins. Det er bedst at samle dette i en klasse til formålet:

```
class PluginList<TPluginInterface> : List<IPlugin>
    where TPluginInterface : IPlugin
```

```
{
    public PluginList(string pluginDirectory) : base()
    {
        // Hvis biblioteket ikke eksistere, forbliver listen tom.
        if (Directory.Exists(pluginDirectory))
            DetectPluginCandidates(pluginDirectory);
    }

    #region Detecting plugins.

    private void DetectPluginCandidates(string pluginDirectory)
    {
        // Find alle DLL'er i biblioteket. Det er vores kandidater.
        foreach (string pluginCandidateDll in
Directory.GetFiles(pluginDirectory, "*.dll"))
        {
            try
            {
                // Indlæs dem ...
                Assembly pluginCandidate =
Assembly.LoadFile(pluginCandidateDll);

                // ... og tjek deres indhold for mulige plugins.
                ScanPluginCandidate(pluginCandidate);
            }
            catch (System.Reflection.ReflectionTypeLoadException)
            {
            }
        }
    }

    private void ScanPluginCandidate(Assembly pluginCandidate)
    {
        // Dumt søge-filter, som kun er med fordi at det ikke kan undværes.
        TypeFilter dummyTypeFilter = new TypeFilter(DummyTypeFilter);

        // Hent samtlige typer i filen: class-, interface-, enum-definitioner,
osv.
        Type[] typeArr1 = pluginCandidate.GetTypes();

        // Gennemløb samtlige typer.
        foreach (Type type1 in typeArr1)
        {
            // Vi er kun interesseret i klasser (ikke interfaces), som
            // ikke er abstrakte, og som er erklæret som public.
            if (type1.IsClass && !type1.IsAbstract && type1.IsPublic)
            {
                // Find samtlige interfaces som denne klasse implementere.
                Type[] typeArr2 = type1.FindInterfaces(dummyTypeFilter, null);

                // Gennemløb interface-typerne.
                foreach (Type type2 in typeArr2)
                {
                    // Implementere vores klasse det rette interface?
                    if (type2 == typeof(TPluginInterface))
```

```
                    {
                        // ... så er det en plugin. :^)

                        // Opret en instans af plugin klassen og gem den her i
listen.
                        IPlugin plugin = Activator.CreateInstance(type1) as
IPlugin;
                        Add(plugin);

                        // Afbryd; vi har fundet det vi ledte efter.
                        break;
                    }
                }
            }
        }
    }

    private bool DummyTypeFilter(Type m, Object filterCriteria)
    {
        return true;
    }

    #endregion
}
```

Klasse definitionen for PluginList

PluginList er defineret som en generisk klasse (det er det <...> gør):

```
class PluginList<TPluginInterface> : List<IPlugin>
    where TPluginInterface : IPlugin
{
    ...
}
```

Man opretter et objekt af klassen på denne måde:

```
PluginList<IFormatterPlugin> pluginList =
    = new PluginList<IFormatterPlugin>(@"C:\Plugins");
```

Her har jeg indsat IFormatterPlugin på pladsen <TPluginInterface>, men jeg kunne sagtens have indsat andre interfaces her; ved at gøre klassen generisk har jeg opnået at jeg kan bruge PluginList klassen på forskellige typer af plugins. Jeg skal ikke skrive en klasse for hver plugin-type jeg ønsker at kunne understøtte.

I koden for ScanPluginCandidate() har jeg denne:

```
if (type2 == typeof(TPluginInterface))
{
    ...
}
```

og den sikre at det kun er klasser som rent faktisk implementere interfacet, indsat på TPluginInterface-pladsen, som beholdes som plugins. Hvis man indsætter IFormatterPlugin, er det dette interface plugin klasserne skal implementere.

Denne del af definitionen:

```
...
    where TPluginInterface : IPlugin
...
```

sikre at den generiske klasse kun vil tage imod interfaces (på pladsen TPluginInterface) under forudsætning af at de implementere IPlugin interfacet. Dette er det absolutte minimum for plugins i vores demo applikation.

Endeligt nedarver PluginList fra List<IPlugin> klassen, og den har derfor alle de egenskaber denne har.

Reflektion

Teknikken, som bruges til at inspektører DLL'en for at finde ud af om den indeholder en plugin, kaldes for *Reflektion*. Det vil komme alt for vidt at skulle forklare alle mulighederne i refleksion, men dette er i korte træk hvad der sker:

DLL'en loades først i en Assembly.

Programmet spørger derefter hvilke typer der er i denne. Enhver klasse/interface/enum/delegate/osv. har en *Type* som fortæller om hvordan den ser ud. I dette tilfælde er vi kun interesseret i de typer som repræsentere en ikke-abstrakt public klasse - dette er jo de eneste klasser der er tilgængelige udefra og som kan instantieres.

Når programmet har fundet en sådan klasse, spørger det videre til hvilke interfaces denne klasse så implementerer. Vi går kun videre med den hvis den understøtter det interface som indsættes i stedet for TPluginInterface.

Hvis klassen opfylder denne betingelse så er det en plugin af den rette type. Med kaldet af Activator.CreateInstance() oprettes der en instans af klassen. Denne lægges efterfølgende i listen med Add() - det er en af de metoder som er tilstede fordi PluginList nedarver fra List<IPlugin>.

Hvordan installeres plugins?

I sin nuværende form oprettes objekter af PluginList med angivelse af et bibliotek i filsystemet. Det er i dette bibliotek at den leder efter mulige plugins.

Faktisk er "installeres" en lige lovligt flot betegnelse; man skal simpelthen bare kopiere DLL'en med plugin'en til biblioteket, starte demo applikation og så er man kørende (dette kaldes for *xcopy deployment*).

Der ledes kun efter plugins når objektet oprettes; hvis man efterfølgende lægger nye plugins i biblioteket efter at demo applikationen er startet opdages de ikke før denne genstartes. Der er m.a.o. ingen "hot plugin" funktionalitet. Dette kan man dog lave med en passende brug af FileSystemWatcher hvis man ønsker.

I dette tilfælde har jeg valgt at hardcode biblioteksnavnet. I en rigtig applikation bør man selvfølgelig tilbyde muligheden for at konfigurere hvor at plugins skal ligge.

Nu kan vi indlæse en plugin, hvordan bruger vi den så?

Opret et Windows Application projekt med navnet "PluginDemoApp".

Tilføj koden for IPlugin og IFormatterPlugin til projektet. Dette bør gøres ved at højre-klikke i Solution Exploren og vælge Add > New Item ... og så vælge Interface. Dette gøres to gange, en for hver af de to interfaces.

Jeg har som sagt valgt at gemme interfacene direkte i demo applikationen. Et alternativ kunne være at gemme dem i en separat DLL.

Tilføj koden for PluginList klassen. Dette gøres ved at højreklikke i Solution Exploren og vælge Add Class... .

Træk en MenuStrip, en OpenFileDialog og en RichTextBox ind på formen.

MenuStrip'en: Opret to menupunkter i 1. niveau: "Filer" og "Formatering". Under Filer indsættes en undermenu: "Åben". Vi skal bruge dette menupunkt til at loade cs-filer med. Dobbelt-klik på Åben og indsæt så koden for dette:

```
private void åbenToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (this.openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        using (StreamReader sr = new
StreamReader(this.openFileDialog1.FileName))
        {
            this.richTextBox1.Text = sr.ReadToEnd();
        }
    }
}
```

OpenFileDialog'en: I dennes properties slettes indholdet af FileName og Filter sættes til:
"C# kode (*.cs)|*.cs|Alle (*.*)|*.*".

RichTextBox'en: C# koden fra den loaded cs-fil indlæses i RichTextBox'en. Dennes Dock property bør nok

lige sættes til Fill hvis det skal se ordentligt ud.

Vi skal derefter lave indlæst de plugins der måtte være. Dobbelt-klik på formen og indsæt koden:

```
PluginList<IFormatterPlugin> pluginList;
private void Form1_Load(object sender, EventArgs e)
{
    pluginList = new PluginList<IFormatterPlugin>(@"C:\Plugins");
    AddPluginsToGUI();
}
```

Initialiseringen af pluginList indlæser alle plugins fra det angivne bibliotek. Nu skal de så bare gøres tilgængelig fra brugergrænsefladen og det er det AddPluginsToGUI() gør. Tidligere oprettede vi menupunktet Formatering, og det er under dette at vi ønsker at indsætte dem:

```
private void AddPluginsToGUI()
{
    // Løb igennem alle de loadede plugins.
    foreach (IPlugin plugin in pluginList)
    {
        // Opret et menupunkt til hvert plugin.
        ToolStripMenuItem pluginToolStripMenuItem = new ToolStripMenuItem();

        // Menupunktet initialiseres med informationer hentet direkte fra
plugin'et.
        pluginToolStripMenuItem.Size = new System.Drawing.Size(152, 22);
        pluginToolStripMenuItem.Text = plugin.MenuText;
        pluginToolStripMenuItem.ToolTipText = plugin.MenuToolTipText;

        // Sammenkæd menupunktet med plugin'en.
        pluginToolStripMenuItem.Tag = plugin;

        // Definere en eventhandler til menupunktet.
        pluginToolStripMenuItem.Click += new
EventHandler(pluginToolStripMenuItem_Click);

        // Indsæt menupunktet under Formatering.

this.formateringToolStripMenuItem.DropDownItems.Add(pluginToolStripMenuItem);
    }
}
```

Derefter mangler vi blot at definere event-handleren, og så er demo applikationen faktisk færdig:

```
void pluginToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Find den menu som kaldte eventhandleren.
```

```
    ToolStripMenuItem temp = sender as ToolStripMenuItem;

    // Find den plugin som høre sammen med det menupunkt.
    IFormatterPlugin plugin = temp.Tag as IFormatterPlugin;

    // Kald plugin'ens Format() metode med formens RichTextBox som input.
    plugin.Format(this.richTextBox1);
}
```

Vi har hermed en kørende applikation som kan loade cs-filer og vise dem i formens RichTextBox.

Desuden kan man kan smide plugins ned i plugin-biblioteket uden at det kræver at applikationen skal re-kompileres før at den kan bruge dem. Andre kan derfor lave plugins til den uden at de skal have adgang til din kildekode, eller at du skal kompilere deres plugin for dem.

Der er endnu ikke nogen menupunkter under Formatering - vi har jo endnu ikke lavet en eneste plugin. Det kommer, det kommer! Faktisk nu:

Endelig, en plugin!

Opret et Class Library projekt og kald det SyntaxHighlightingPlugin.

Opret en reference til PluginDemoApp exe-filen. Dette gør du i Solution Explorer; højre-klik på References, vælg Add Reference og tryk derefter på Browse tabben i den resulterende menu. Find frem til exe-filen.

Dette skal følges op af en

```
using PluginDemoApp;
```

i din kode.

Den skal også have en tilsvarende reference til System.Windows.Forms.dll da det er der RichTextClass er defineret. Dette gøres på tilsvarende facon, men under .NET tabben i stedet for Browse.

Omdøb klassen fra det kedelige "Class1" til at hedde f.eks. "SyntaxHighlighter", og angiv at den arver fra IFormatterPlugin:

```
public class SyntaxHighlighter : IFormatterPlugin
{
    ...
}
```

Udfyld nu de metoder som interfacet kræver. Resultatet kunne se nogenlunde sådan her ud:

```
```

```
public class SyntaxHighlighter : IFormatterPlugin
{
    #region IFormatterPlugin Members

    public void Format(RichTextBox rtf)
    {
        MarkupText(rtf, new Regex(@"\b(class|interface)\b"), Color.Blue);
    }

    private void MarkupText(RichTextBox rtf, Regex re, Color color)
    {
        Match rem = re.Match(rtf.Text);

        while (rem.Success)
        {
            rtf.SelectionStart = rem.Index;
            rtf.SelectionLength = rem.Length;
            rtf.SelectionColor = color;
            rtf.SelectionLength = 0;

            rem = rem.NextMatch();
        }

        rtf.Select(0, 0);
    }

    #endregion

    #region IPlugin Members

    public string Author
    {
        get { return "Nielle"; }
    }

    public string Description
    {
        get { return "En demo plugin til PluginDemoApp applikationen"; }
    }

    public string MenuText
    {
        get { return "Syntax highlight"; }
    }

    public string MenuToolTipText
    {
        get { return "C# syntax color highlighting"; }
    }

    public string Name
    {
        get { return "C# syntax highlighting plugin"; }
    }
```

```
    public Version PluginVersion
    {
        get { return new Version(1, 0, 0, 0); }
    }

    #endregion
}
```

De fleste properties returnerer blot en streng. Det saftige foregår i metoden Format(), men den sender i virkeligheden blot formaterings-jobbet videre til MarkupText() metoden. Pointen er at Format() eventuelt kan kalde MarkupText() flere gange med andre input.

I dette tilfælde gør koden ikke så meget - den finder samtlige steder hvor at der står "class" eller "interface" i koden og markere dem med blåt. Dette gøres vha. regulære udtryk og med noget kreativt RichTextBox gymnastik.

Kompilér projektet, og smid den resulterende DLL ind i biblioteket C:\Plugins. Start derefter PluginDemoApp (helst uden at re-kompilere). Se nu under Formatering - der skulle gerne være kommet et menupunkt som hedder "Syntax highlight" og hvis man holder musen over dette skulle teksten "C# syntax color highlighting". Dette er egenskaber som er hentet fra plugin'en selv.

Load en cs-fil, og aktiver menupunktet. Så skulle alle forekomster af ordene "class" og "interface" gerne fremhæves med blåt.

Afslutning

Mere kode skal der i virkeligheden ikke til før at man har en fungerende plugin arkitektur.

Den viste model er i relativt simplistisk og man kan sagtens spekulere i meget mere avancerede modeller som tilbyder yderligere faciliteter.

Fælles for dem er at de bruger refleksion til at loade og inspektionere DLL'er på run-time tidspunktet, i stedet for at det skal gøres på compile-time. Loadningen kan enten ske ved at applikationen selv kigger i et bestemt bibliotek, eller ved at man specificere hvilke plugins der skal loades vha. en konfigurationsfil.

.oOo.

Hvis man er interesseret i en langt mere avanceret model for hvordan man laver plugins, kan man f.eks. læse denne artikel som handler om hvordan plugin arkitekturen er lavet i SharpDevelop programmet:

http://www.icsharpcode.net/TechNotes/ProgramArchitecture.pdf

Her er desuden en artikel fra Microsoft om emnet:

http://www.dotnet247.com/247reference/a.aspx?u=http://msdn.microsoft.com/msdnmag/issues/03/10/Plug-Ins/default.aspx

Kommentar af jps6kb d. 02. Jan 2008 | 1

Nice :)

Kommentar af webben d. 30. Dec 2007 | 2

Kommentar af coderdk d. 14. May 2008 | 3

:)

Kommentar af cwboy d. 31. Dec 2007 | 4

Super!

Kommentar af _basil d. 24. Jun 2008 | 5

Kommentar af fedora d. 30. Dec 2007 | 6

Jeg synes det er meget god artikel, desværre for mit vedkommende endnu lidt højere niveau end jeg er klar til, men har god forståelse for princippet.

Kommentar af mr-kill d. 02. Jan 2008 | 7

God læsning

Kommentar af dotdonk d. 07. Jan 2008 | 8

Awsome stuff