An Introduction to Plugins in C#

By: Jonathan Dick (JonD@Net-Lutions.com)

Preface:

First things first: I wrote this article mainly because I've found a complete lack of comprehensive information on the subject. Sure there are projects out there that show you how to make a simple plugin that lets you do something dumb like feed it two numbers, and then one plugin may add the two numbers together and return the result, while the other plugin may multiply them instead.

This is where I began, and is not bad reading as a prep tutorial to this tutorial. It might not be a bad idea to understand those tutorials first. However, when I began to search for how to bring GUI's into the plugins, I was completely lost. My aim is to show you how to do this, and provide some clear explanations.

This tutorial will not be a step by step guide to making the program, as if you have the tutorial, you should also have the source code for the Tutorial project. I feel it unnecessary to go through this all, so instead I will only be covering some explanations on the code that is tricky.

The Tutorial:

Notice that we have a Solution with 6 different projects in it. The first project is the Plugin Application Tester thingy. I like to refer to it as the Host, since it is the application that your plugins will be plugging-in to. This of course is the heart and soul of the whole Plugin get-up. The Plugins themselves are really not that complex as you will see.

But before I go into the Host Application, Let's talk about the PluginInterface Project. If you are unfamiliar with interfaces, Please go find a tutorial and familiarize yourself with them. In a nutshell, they are 'outlines' of what properties and methods a Class that inherits a certain interface should have in them. There is very little code in this project. In fact, the only code we're worried about is this:

```
using System;

namespace PluginInterface
{
        public interface IPlugin
        {
                IPluginHost Host {get;set;}

                string Name {get;}
                string Description {get;}
                string Author {get;}
                string Version {get;}

                System.Windows.Forms.UserControl MainInterface {get;}

                void Initialize();
                void Dispose();

        }

        public interface IPluginHost
        {
                void Feedback(string Feedback, IPlugin Plugin);
        }
}
```

In this code, we have 2 Interfaces declared. These are the legends, the roadmaps, the keys to what a class inheriting these interfaces should do. For example, if we were to declare a new class in code and have it inherit IPlugin interface, the class would HAVE to employ four readonly string properties (Name, Description, Author, Version), a IPluginHost type Property, a UserControl property, and two voids: One

called Initialize, the other called Dispose. Only if a class has all of that as said above, can it legally inherit the IPlugin Interface.

This of course makes sense if you think about it. If we're going to make plugins, we need some establish some ground rules. Just think of what would happen if we tried calling a method in a plugin that didn't exist! This way, we establish a set of guidelines that every plugin must follow, and we can predict just what the plugin is going to be capable of doing.

When all is said and done, we have 2 interfaces: IPlugin, and IPluginHost. It should be clear that IPlugin is the interface that all Plugins will have to inherit. However, we also will make our Plugin host inherit the IPluginHost interface. This way, we can send the Plugin a reference to the host later on. This will allow the Plugin some access to things of its host.

When this project is compiled, it produces PluginInterface.dll file. We made this in its very own Dll for a reason. Now we have a common roadmap to share between plugins and the plugin host that contains defenititons for each object.

Next up, we have the Host Project. This one is a bit more confusing. First of all let's get this cleared out of the way. I used a technique that may be a bit confusing at first for some, but once scaled up can make things a bit simpler later on.

```
using System;

namespace Host
{
	/// <summary>
	/// Holds A static instance of global program shtuff
	/// </summary>
	public class Global
	{
		public Global(){} //Constructor

		//What have we done here?
		public static Host.PluginServices Plugins = new PluginServices();

	}
}
```

The above Code is a trick I learned a little while ago. I have a class called Global. Inside this class is a static public object. This object in this case happens to hold an instance of the PluginServices object. All that this little trick does is makes sort of a Global instance of the (PluginServices) object that can easily be accessed from the entire application. This is instead of declaring an object and making a new instance in the Main form of the host application. Now, I can access the instance of the PluginServices object anywhere in the program by typing Global.Plugins. ! It's that simple!

The next bit of code I will introduce is the code that actually finds any compatible plugin files in a certain folder, and adds them to a collection of AvailablePlugins:

```
public void FindPlugins(string Path)
{
	//First empty the collection, we're reloading them all
	colAvailablePlugins.Clear();

	//Go through all the files in the plugin directory
	foreach (string fileOn in Directory.GetFiles(Path))
	{
		FileInfo file = new FileInfo(fileOn);
```

```
                //Preliminary check, must be .dll
                if (file.Extension.Equals(".dll"))
                {
                        //Add the 'plugin'
                        this.AddPlugin(fileOn);
                }
        }
}
```

This is a very simple method that accepts a path to look for plugins in. For example, you could tell it to look in your application's Plugins folder. All it does is loop through all the files in the given directory, and check to see if the extension is of .dll type. Now you may be wondering if that's enough to check for. What if someone renamed some random file as .dll and stuck it in? We'll get to that later.

You may have noticed in the last method, a call to another method AddPlugin(). I'm not going to paste the whole function in here, as that would be entirely redundant, but I will explain what's going on in a few spots. First of all, we're declaring a new Assembly type, and loading the passed filename into it:

```
//Create a new assembly from the plugin file we're adding..
Assembly pluginAssembly = Assembly.LoadFrom(FileName);
```

Next, we're iterating through all the different types that the particular plugin contains. It would be wise to note at this time, that you may want to put in some extra error catching around this area. I would imagine if you tried loading an Assembly that isn't a proper assembly, and getting its types, you may run into some problems. But we'll leave that for you to play with. The next few if statements are just checking some attributes of the given type found in the assembly. We want to make sure it's public and accessible to us, as well as not an abstract class. Here is the next chunk of code:

```
//Gets a type object of the interface we need the plugins to match
Type typeInterface = pluginType.GetInterface("PluginInterface.IPlugin", true);

//Make sure the interface we want to use actually exists
if (typeInterface != null)
{
        //Create a new available plugin since the type implements the //IPlugin interface
        Types.AvailablePlugin newPlugin = new Types.AvailablePlugin();

        //Set the filename where we found it
        newPlugin.AssemblyPath = FileName;

        //Create a new instance and store the instance in the collection for later use
        //We could change this later on to not load an instance.. we have 2 options
        //1- Make one instance, and use it whenever we need it.. it's always there
        //2- Don't make an instance, and instead make an instance whenever we use it, then close it
        //For now we'll just make an instance of all the plugins
        newPlugin.Instance =
(IPlugin)Activator.CreateInstance(pluginAssembly.GetType(pluginType.ToString()));

        //Set the Plugin's host to this class which inherited IPluginHost
        newPlugin.Instance.Host = this;

        //Call the initialization sub of the plugin
        newPlugin.Instance.Initialize();

        //Add the new plugin to our collection here
        this.colAvailablePlugins.Add(newPlugin);
}
```

This bit of code is the heart of the whole plugin project. First, we declare a new Type object, and make it the type of our IPlugin interface. Notice here, we're calling the GetInterface() method. What this

does is tries to sort of cast the Type that we are on in the for loop to the IPlugin interface. If the Type does actually implement the IPlugin interface, typeInterface will not be null. That's really the hardest step. Next we're just making a new instance of the class we created 'AvailablePlugin'. We set the AssemblyPath property. On the next line, we're doing something else. The AvailablePlugin type has a property called Instance. This is where we will be storing the actual instance of the loaded plugin, ready to use.

Now, for the purpose of this tutorial, I tried to keep things easier. I decided that my program would simply load an instance of every plugin it finds and have it ready for use, regardless of whether or not the plugin will actually be used. There is another way to doing this as well. What if you don't want to use all the plugins? It's a bit of a waste of memory to load them all up if you're not going to use them all. You could implement some sort of Loading and Unloading of plugins if you choose to go this route. Like I said though, that is beyond the scope of this article.

So, we create a new instance of the plugin with the following line:

```
newPlugin.Instance = (IPlugin)Activator.CreateInstance(pluginAssembly.GetType(pluginType.ToString()));
```

After this is done, we can use the new instance of the plugin, and set its Host property. Since the class that we are working in happens to inherit the IPluginHost interface, we can conveniently set the Host property to 'this'. Finally, we call the Initialize() method of the plugin to alert it to do anything it needs to start, and then we add it to the collection of AvailablePlugins.

The only other bit of code I think we should cover is the FormFeedback() function implemented by the PluginServices class:

```
public void Feedback(string Feedback, IPlugin Plugin)
{
        //This sub makes a new feedback form and fills it out
        //With the appropriate information
        //This method can be called from the actual plugin with its Host Property

        System.Windows.Forms.Form newForm = null;
        frmFeedback newFeedbackForm = new frmFeedback();

        //Here we set the frmFeedback's properties that i made custom
        newFeedbackForm.PluginAuthor = "By: " + Plugin.Author;
        newFeedbackForm.PluginDesc = Plugin.Description;
        newFeedbackForm.PluginName = Plugin.Name;
        newFeedbackForm.PluginVersion = Plugin.Version;
        newFeedbackForm.Feedback = Feedback;

        //We also made a Form object to hold the frmFeedback instance
        //If we were to declare if not as  frmFeedback object at first,
        //We wouldn't have access to the properties we need on it
        newForm = newFeedbackForm;
        newForm.ShowDialog();

        newFeedbackForm = null;
        newForm = null;

}
```

This is really simple actually. The whole idea of the IPluginHost interface is to allow some communication between the main application that runs the plugins, and the plugins themselves. In this example, we didn't take advantage of this idea to a great extent, however in your own applications, this idea can become extremely powerful. Think about it. You could expose a LOT of functionality of your main program. For example, lets say you were making an mp3 player, you could make the IPluginHost interface have methods such as PlayMp3(), Stop(), Previous(), Next(), SetVolume(), etc. This would allow your plugins to take some control over the main program. Hopefully this idea makes sense to you. In our example here, all we've done is incorporated a method that plugins can call which displays a new form with a string that the plugin passed along, as well as some information on the plugin that called the method.

Ok, so we talked about the actual host application for the plugins, but we have yet to talk about the plugins themselves. The plugins themselves are not that complicated. All you have to do to make one is create a new Class Library project, and make a class that inherits the IPlugin interface. Remember, you will need to add a reference to the project to the PluginInterfaces Project that our interfaces reside in. Also, another tip: When you add the reference, make sure the CopyLocal property of the reference is set to false! Otherwise, you will have some unexplained problems. The only place where the CopyLocal property should be True is on the Host application's reference to the PluginInterfaces project. This really makes sense if you think about it, since the Host will always have the PluginInterfaces.Dll file. The plugins just use it from the Host program.

I'll let you take a look at the actual plugins yourself. Plugins 1 through 3 are all made pretty much the same. They all have a Class which inherits the IPlugin interface and implements all its methods and properties. The plugins also contain a UserControl class in them. This is the Gui for the plugin. It is exposed as a property in the IPlugin interface (MainInterface). This allows the host to add the UserControl to a panel or something to expose a Gui to the plugin.

Notice that in Plugin3, in the UserControl itself, notice we have a properties of IPlugin and IPluginHost types. This allows the UserControl access to the Plugin Host's methods (in this case, ShowFeedback(), which is the whole point of this plugin), and the Plugin's methods itself.

Finally, in Plugin4, we demonstrate that you don't need to have a separate Plugin class and UserControl class for a plugin. We created a UserControl that also inherits the IPlugin interface. We've combined two classes into one.

Conclusion:

And that's about it for plugins. Once you've grasped the concept of Interfaces, and have understood the code for actually initializing instances of plugins, creating an actual plugin is not a very difficult task. I will leave you to explore the rest of the code by yourself. Hopefully you have learned something from this tutorial, and can successfully implement plugins into your own programs!