Tutorial – Reflection

In this tutorial we're going to shake things up a bit and use reflection for a real-world, practical purpose that's actually kind of cool… writing a plug-in.

For the sake of argument, let's say that you had a sprite map editor that saved output in XML. Now, XML is useful, but you've received a lot of feedback from your users saying that they would love to be able to export to JSON, or CSV, or some other format. You want to support your users, but you'd rather concentrate on other features for the moment.

The solution to your conundrum is to allow your users to write their own export format using plugins! Using C#'s Reflection mechanism this isn't as complicated as it sounds.

This tutorial will walk through a trivial example for simplicity. But why not extend this approach and include some plugins with your assignment?

Starting off:

We are going to make a simple Windows Forms application containing a plugin mechanism that will search and load plugins from a predefined location (in this instance the application's executable directory). The created plugins are projects in the form of built assemblies (.dlls).

Before we make anything, we need a solution. Go ahead and create a new Windows Forms project. I called min *pluginApp*. We'll code the application last, so for now just leave it as a blank form.

The Plugin Interface:

First we need to define an Interface that all plugins must implement. We will include this as a separate project so that other developers will only need the assembly of this project to write their own plugins.

The members of this interface will depend on what your application does, and what functionalities plugins are intended to perform. For this tutorial we will create a trivial interface containing one property that is returning a name, and one method that is doing something.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PluginContracts
{
    public interface IPlugin
    {
        string Name { get; }
        void Do();
    }
}
```

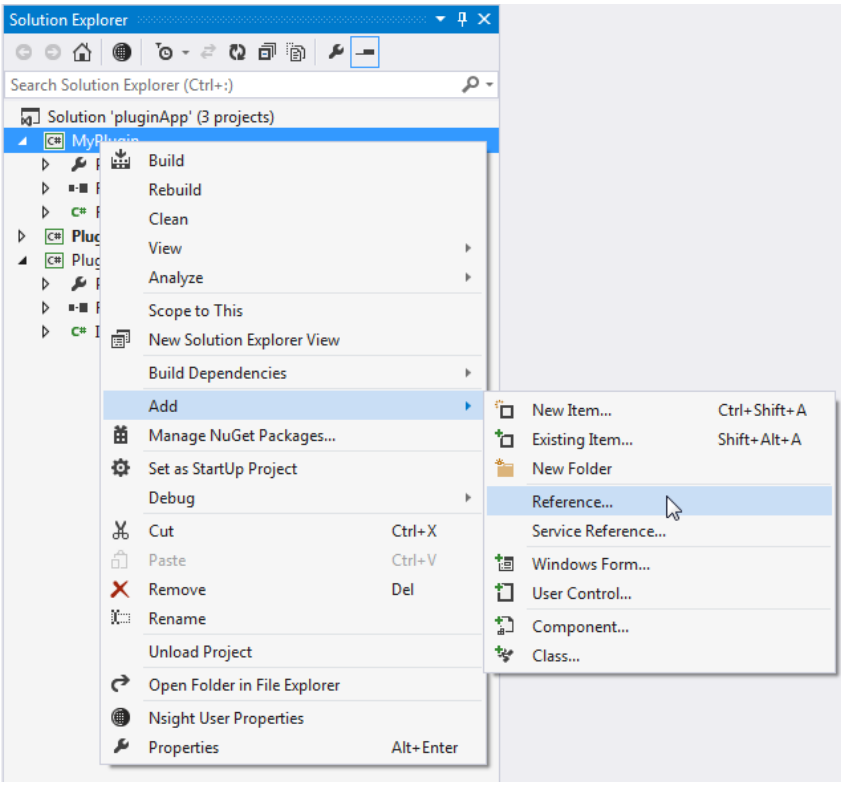
(This code goes in its own .dll project called, perhaps unsurprisingly, PluginContracts)

Creating the Plugin:

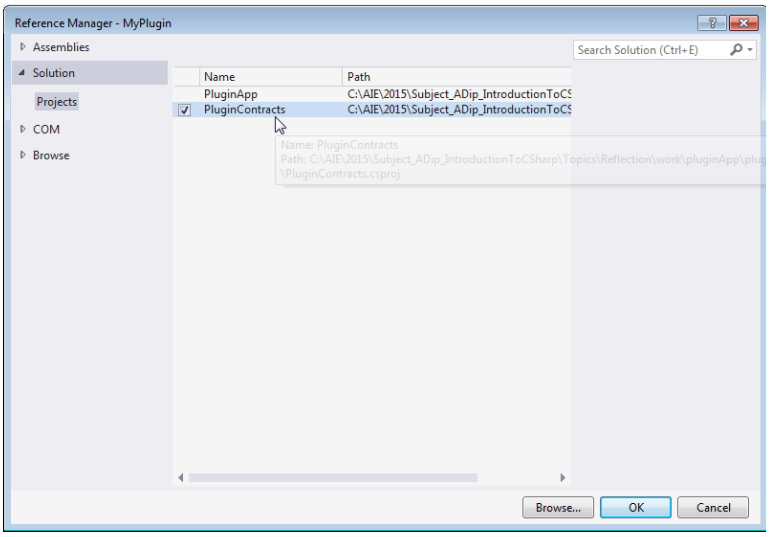
To provide a plugin, you have to create a new project and add a reference to PluginContracts.

The first step is to create the new project (which we add to our solution) called MyPlugin.

Once you have added the new project to your solution, you need to add a reference to the PluginContracts project (which builds the .dll containing the IPlugin interface).



Right-click on the MyPlugin project and select Add -> Reference. Find the MyPlugin project and add the reference.



	Name	Path
	PluginApp	C:\AIE\2015\Subject_ADip_IntroductionToCS
✓	PluginContracts	C:\AIE\2015\Subject_ADip_IntroductionToCS

Name: PluginContracts
Path: C:\AIE\2015\Subject_ADip_IntroductionToCSharp\Topics\Reflection\work\pluginApp\plug
\PluginContracts.csproj

Within your MyPlugin project, create a new class called MyPlugin and add the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using PluginContracts;

namespace MyPlugin
{
    public class MyPlugin : IPlugin
    {
        #region IPlugin Members

        public string Name
        {
            get
            {
                return "My Plugin";
            }
        }

        public void Do()
        {
            System.Windows.Forms.MessageBox.Show("Do Something in My Plugin");
        }

        #endregion
    }
}
```

This is all you need to do to provide a plugin.

The final step is to implement the framework in our main application that knows how to find and handle the plugins.

The Application Framework:

First of all we have to know where to search for plugins. Usually we will specify a folder where all plugins are put. For this tutorial we will hard code the search path to be the current directory (which will be either the Debug or Release folder for your application, depending on your build settings). This means that you will need to manually copy the MyPlugin.dll to the application executable directory before you run your application.

To start, we get the names of all .dll files in the search directory:

```
            string[] dllFileNames = null;
            if (Directory.Exists(path))
            {
                dllFileNames = Directory.GetFiles(path, "*.dll");
            }
```

Next we load all the assemblies:

```
// Next we have to load the assemblies. This is where Reflection comes in
ICollection<Assembly> assemblies = new List<Assembly>(dllFileNames.Length);
foreach (string dllFile in dllFileNames)
{
       AssemblyName an = AssemblyName.GetAssemblyName(dllFile);
       Assembly assembly = Assembly.Load(an);
       assemblies.Add(assembly);
}
```

With the assemblies loaded we can search for all types that implement our Interface IPlugin:

```
    Type pluginType = typeof(IPlugin);
    ICollection<Type> pluginTypes = new List<Type>();
    foreach (Assembly assembly in assemblies)
    {
        if (assembly != null)
        {
            Type[] types = assembly.GetTypes();
            foreach (Type type in types)
            {
                if (type.IsInterface || type.IsAbstract)
                {
                    continue;
                }
                else
                {
                    if (type.GetInterface(pluginType.FullName) != null)
                    {
                        pluginTypes.Add(type);
                    }
                }
            }
        }
    }
```

Finally, we create instances from our found types using Reflections.

```
    // Last we create instances from our found types using Reflections
    ICollection<IPlugin> plugins = new List<IPlugin>(pluginTypes.Count);
    foreach (Type type in pluginTypes)
    {
        IPlugin plugin = (IPlugin)Activator.CreateInstance(type);
        plugins.Add(plugin);
    }
```

This is what the final function looks like:

```
public static ICollection<IPlugin> LoadPlugins(string path)
{
    string[] dllFileNames = null;
    if (Directory.Exists(path)) {
        dllFileNames = Directory.GetFiles(path, "*.dll");
    }

    // Next we have to load the assemblies. This is where Reflection comes in
    ICollection<Assembly> assemblies = new List<Assembly>(dllFileNames.Length);
    foreach (string dllFile in dllFileNames) {
        AssemblyName an = AssemblyName.GetAssemblyName(dllFile);
        Assembly assembly = Assembly.Load(an);
        assemblies.Add(assembly);
    }

    // Now we have loaded all assemblies from our predefined location,
    // we can search for all types that implement our Interface IPlugin
    Type pluginType = typeof(IPlugin);
    ICollection<Type> pluginTypes = new List<Type>();
    foreach (Assembly assembly in assemblies) {
        if (assembly != null)
        {
            Type[] types = assembly.GetTypes();
            foreach (Type type in types)
            {
                if (type.IsInterface || type.IsAbstract)
                {
                    continue;
                }
                else
                {
                    if (type.GetInterface(pluginType.FullName) != null)
                    {
                        pluginTypes.Add(type);
                    }
                }
            }
        }
    }

    // Last we create instances from our found types using Reflections
    ICollection<IPlugin> plugins = new List<IPlugin>(pluginTypes.Count);
    foreach (Type type in pluginTypes) {
        IPlugin plugin = (IPlugin)Activator.CreateInstance(type);
        plugins.Add(plugin);
    }

    return plugins;
}
```

This function is static, so go ahead an added to your applications main class (the class containing the Main() function).

The Form:

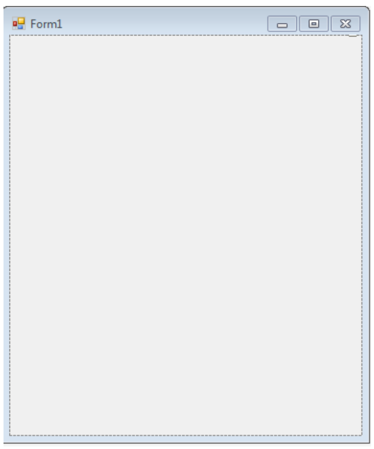
Switch over to the Form designer now and let’s work on how we are going to display the plugins.

We are going to add a button for each plugin we load. So that our buttons will be arranged correctly on the form, we are going to place a TableLayoutPanel on the form.

Find the TableLayoutPanel in the tools window and drag it onto the form. Set the following properties:

Name: _tableLayoutPanel
AutoSize: True
ColumnCount: 1
Dock: Fill
Location: 0,0
Row Count: 1

The Form should look like this when you’re done:



Jump to the code, and lets fill in the definition of the class.

Using our LoadPlugins function we search for and initialize the plugins so that we can use their implemented properties and methods. To demonstrate this we will create a button for each loaded plugin and connect the content and the click event of the button to the property and the method of the plugin.

Here is the complete constructor for our form:

```
public Form1()
{
    InitializeComponent();

    // First of all we have to know where to search for plugins.
    // Usually we will specify a folder in that all plugins are put in.
    // In this folder we search for all assemblies.
    // I'm using the current directory (ie, the directory the executable is in), but you
    // would normally provide an option for the user to set somewhere in your application
    string path = Directory.GetCurrentDirectory();

    ICollection<IPlugin> plugins = Program.LoadPlugins(path);

    // create a button for each loaded plugin and connect the content and the click event
    // of the button to the property and the method of the plugin.
    foreach (var item in plugins)
    {
        _plugins.Add(item.Name, item);

        Button b = new Button();
        b.Text = item.Name;
        b.Click += ButtonOnClick;
        _tableLayoutPanel.Controls.Add(b);
    }
}
```

We are using a Dictionary with the name of the plugin as key to keep track of which button content belongs to which plugin. This is so we can execute the correct plugin method according to which button is clicked (provided of course that we add more than one plugin).

The final step is to add the ButtonOnClick function, which is called attached to each button’s click event.

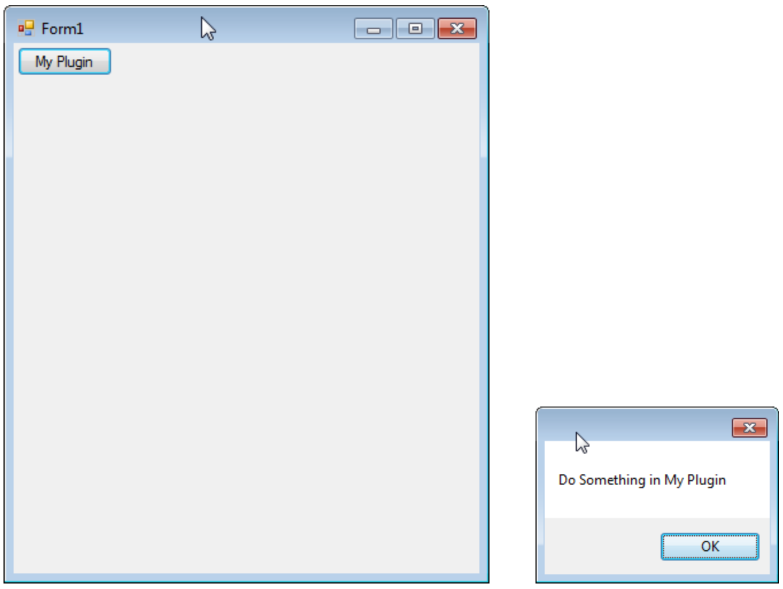
```
private void ButtonOnClick(object sender, System.EventArgs e)
{
    Button b = sender as Button;
    if (b != null)
    {
        string key = b.Text.ToString();
        if (_plugins.ContainsKey(key))
        {
            IPlugin plugin = _plugins[key];
            plugin.Do();
        }
    }
}
```

And that’s it! To run your application, make sure you have performed the following steps:

1. Build the MyPlugin project.

2. Build the application.
3. Copy the MyPlugin.dll to the Debug or Release folder of the application
4. From Visual Studio run the application.

If all went well, and you have all the code in the right places, your application should launch, load the plugin assembly and create a new button on the form corresponding to your plugin.



Do Something in My Plugin

Challenge:

For the adventurous, use plugins in your assignment to add the ability to output your xml sprite sheet data in a different format.