**Introduction to Building a Plug-In Architecture Using C#**

**Part I. Overview**

Using this technique, we will use the standard GOF State Pattern by having a host application expose a piece of functionality as interface. The host will then load different implementations based on some criteria that we'll be choosing. Once we have a class that implements this interface, it can be "plugged" into the host application by dropping the containing dll into a specified folder which provides the host application with a "pluggable" implementation of the exposed interface.

For this article we'll be using a the following very simple interface that will allow us to build plug-in components that the host can use to perform a calculation on two integers and expose the symbol representing the calculation. (Of course, in a real application the interface for the plug-in would probably be much more complex but we'll keep it simple here to keep the focus on the technique.)

public interface ICalculator  
{  
     int Calculate(int a, int b);  
     char GetSymbol();  
}

We could have multiple implementations of this interface in our main project or in seperate assemblies that would all look similiar to this class with slightly different implementations:

class Divider:ICalculator  
{  
     #region ICalculator Members

     public int Calculate(int a, int b)  
     {  
          return a / b;  
     }

     public char GetSymbol()  
     {  
          return '/';  
     }

     #endregion  
}

Injecting our implementation of the ICalculator interface with the constructor, we could provide a default behavior (division) for a host class while at the same time allowing for other implementations to be injected into our host class. This pattern also makes the host class easier to unit test.

public class CalculatorHost       
{  
     public CalculatorHost(ICalculator calculator)  
     {  
          m\_calculator = calculator;  
     }

     public CalculatorHost() : this(new Divider()) { }

     private int m\_x, m\_y;  
     private ICalculator m\_calculator;

     public int X       
     {  
          get { return m\_x; }  
          set { m\_x = value; }  
     }

     public int Y  
     {  
          get { return m\_y; }  
          set { m\_y = value; }  
     }

     public int Calculate()  
     {  
          return m\_calculator.Calculate(m\_x, m\_y);  
     }

     public override string ToString()  
     {  
          return string.Format("{0} {1} {2} = {3}",  
               m\_x.ToString(),  
               m\_calculator.GetSymbol(),  
               m\_y.ToString(),  
               m\_calculator.Calculate(m\_x, m\_y));  
     }

}

**Part II. Late Binding**

What we want to be able to do at the end of the day, is drop a new dll implementing ICalculator into a folder and have the application be able to consume and use the new functionality through a late binding mechanism. For this particular implementation we'll be placing all the plug-in dlls in a folder called "Plugins" which will a sub directory where the main application's assembly lives. (Note: Because this is just a sample app and not bullet-proof, when you unzip and build the samples, you may have to manually add the "Plugins" sub-folder to avoid a runtime exception. Of course, you'll also have to place the plug-in dlls into this folder for the host application to load them.).

Because our late-binding mechanism uses reflection we will be taking a perf hit. One way to minimize the impact is to try to cache the results of this operation. In order to do this, we'll use a static class to hold the results of our binding. One approach would be to perform our late binding when the application starts. In this particular implementation we'll use the lazy-load approach and create a loader method that will populate a list of CalculatorHost objects and so we will take the hit the first time the application requests the plugins (which will probably be early in the life of the application).

public static class CalculatorHostProvider  
{

     private static List<CalculatorHost> m\_calculators;

     public static List<CalculatorHost> Calculators  
     {  
          get   
          {  
               if (null == m\_calculators)  
                    Reload();

               return m\_calculators;   
          }  
     }  
}

When Reload() is called for the first time, we will create the new list. We also may want to reload the plugins while our application is running. For instance, if we have dropped a new implementation in the "Plugings" subdirectory and can't afford to restart the application. In this case, we'll clear the existing calculators list.

Next we'll load all the assemblies in the "Plugins" subdirectory and iterate through them to locate the ones that we can use for creating a new CalculatorHost object.

public static void Reload()  
{

     if (null == m\_calculators)  
          m\_calculators = new List<CalculatorHost>();  
     else  
          m\_calculators.Clear();

     m\_calculators.Add(new CalculatorHost()); // load the default  
     List<Assembly> plugInAssemblies = LoadPlugInAssemblies();  
     List<ICalculator> plugIns = GetPlugIns(plugInAssemblies);

     foreach (ICalculator calc in plugIns)  
     {  
          m\_calculators.Add(new CalculatorHost(calc));  
     }  
}

**Part III. Attributes**

While not absolutely required for this technique, it is a good idea to explicitly declare our plugins to ensure that the intent of the interface implementation is actually for a plug in component for our host application. In order to do this we'll use a custom attribute decorator for any class that will be implementing ICalculator and is supposed to function as a plugin for our host. There may be cases where we would only want one implementation of a particular plug in instead of this articles approach of having multiple plugins or maybe we would want to have priorities assigned to each plug-in. In order to do that, we could put some identifier in the attribute class by which we could sort and filter the plugins to get the one(s) we want. We can also look at the assembly versions. It is probably something we would run into at some point using this technique and using attributes for providing metadata on the plugins is a pretty good solution.

[AttributeUsage(AttributeTargets.Class)]  
public class CalculationPlugInAttribute : Attribute  
{  
     public CalculationPlugInAttribute(string description)  
     {  
          m\_description = description;  
     }

     private string m\_description;

     public string Description  
     {  
          get { return m\_description; }  
          set { m\_description = value; }  
     }  
}

So now, when we build a plug-in, we'll make sure to decorate it to explicitly declare the intent of the implementation. This is especially helpful if there are other developers building plugins for our application. In a seperate project we have an implementation of the ICalculator used for adding two numbers together. We'll build this solution, take the resulting dll and drop it in the "Plugins" folder of the host application.

[CalculationPlugInAttribute("This plug-in will add two numbers together")]  
class Adder: ICalculator  
{  
     #region ICalculator Members

     public int Calculate(int a, int b)  
     {  
          return a + b;  
     }

     public char GetSymbol()  
     {  
          return '+';  
     }

     #endregion

}

**Part IV. The Guts**

Loading the assemblies from the "Plugins" folder is a straightforward process. We'll find all the dlls in the folder and use Assembly.LoadFile() to add them to a list.

private static List<Assembly> LoadPlugInAssemblies()  
{  
     DirectoryInfo dInfo = new DirectoryInfo(Path.Combine(Environment.CurrentDirectory, "Plugins"));  
     FileInfo[] files = dInfo.GetFiles("\*.dll");  
     List<Assembly> plugInAssemblyList = new List<Assembly>();

     if (null != files)  
     {  
          foreach (FileInfo file in files)  
          {  
               plugInAssemblyList.Add(Assembly.LoadFile(file.FullName));  
          }  
     }

     return plugInAssemblyList;

}

Next, we'll take the resulting list and get all the types that implement our ICalculator interface and have the CalculationPlugInAttribute declared. We'll use the Activator class to instantiate an instance of each class we found and return the resulting list of instantiated calculators.

static List<ICalculator> GetPlugIns(List<Assembly> assemblies)  
{  
     List<Type> availableTypes = new List<Type>();

     foreach (Assembly currentAssembly in assemblies)  
          availableTypes.AddRange(currentAssembly.GetTypes());

     // get a list of objects that implement the ICalculator interface AND   
     // have the CalculationPlugInAttribute  
     List<Type> calculatorList = availableTypes.FindAll(delegate(Type t)  
     {  
          List<Type> interfaceTypes = new List<Type>(t.GetInterfaces());  
          object[] arr = t.GetCustomAttributes(typeof(CalculationPlugInAttribute), true);  
          return !(arr == null || arr.Length == 0) && interfaceTypes.Contains(typeof(ICalculator));  
     });

     // convert the list of Objects to an instantiated list of ICalculators  
    return calculatorList.ConvertAll<ICalculator>(delegate(Type t) { return Activator.CreateInstance(t) as ICalculator; });

}

**Part V. Using Our Plug-In Architecture**

I have two projects (a console app and a windows app) implementing the plug in architecture in the code accompanying this project.

The console app is simple and consists of less than a dozen lines:

static void Main(string[] args)  
{

     int  
          x = 34,  
          y = 56;

     Console.WriteLine(String.Format("x={0} y={1}", x.ToString(), y.ToString()));

     foreach (CalculatorHost calculator in CalculatorHostProvider.Calculators)  
     {  
          calculator.X = x;  
          calculator.Y = y;  
          Console.WriteLine(calculator.ToString());  
     }  
          Console.ReadLine();

}

The windows application uses our plugins for the contents of a drop down (bound when the form loads) and is also fairly simple:

private void Form1\_Load(object sender, EventArgs e)  
{  
     m\_cbCalculation.DisplayMember = "Operator";  
     m\_cbCalculation.DataSource = CalculatorHostProvider.Calculators;  
}

Remember... both of these projects need a subdirectory called "Plugins" to run and you will have to manually drop the implementation of each plugin into this directory for the host application to consume it. Also, nither of these projects are bullet-proof by any means and are demonstration purposes only. We would have to add error handling before they would be production ready. Also, we would probably want to add enough unit tests to our projects to ensure everything is working properly before a distribution build.

Anyways... the important thing is the technique. We could adapt this approach for many differerent types of applications such as creating windows or web services with pluggable functionality or building a mission-critical application architecture where we can plan for future version deployment such that a customer would just have to drop a new dll in a folder in order to avoid application down-time as a result of a new installation process.

I hope you found this article useful.

Until next time,  
Happy coding