# XML Activator

All too often I see people using switch/Select Case statements when using a Factory pattern. The problem with this is if you wish to add the ability to instantiate a new class in your Factory, you need to add a new “case” statement, re-compile the code and redeploy that DLL back out to your client machines, or your server(s). Another way to implement a Factory pattern is to use Reflection and Interfaces to dynamically create an instance of a class. This blog post will show you how to use an XML file, an Interface and the Assembly class to dynamically load a list of assemblies and classes to load into an application at runtime.

# An XML File of Assembly/Class Names

You will need a data store to place a list of assembly names and class names that you wish to dynamically create. You could use a database table, but I am going to use an XML file that contains the Assembly Name to load and a Class Name to instantiate from that assembly. Any additional information you wish to add, feel free. In the XML listing below you can see a VendorId, VendorName as well as the AssemblyName and ClassName elements.

```
<Vendors>
  <Vendor>
    <VendorId>1</VendorId>
    <VendorName>Microsoft</VendorName>
    <AssemblyName>Vendor.Microsoft</AssemblyName>
    <ClassName>Vendor.Microsoft</ClassName>
  </Vendor>
  <Vendor>
    <VendorId>2</VendorId>
    <VendorName>Apple</VendorName>
    <AssemblyName>Vendor.Apple</AssemblyName>
    <ClassName>Vendor.Apple</ClassName>
  </Vendor>
  <Vendor>
    <VendorId>3</VendorId>
    <VendorName>Google</VendorName>
    <AssemblyName>Vendor.Google</AssemblyName>
    <ClassName>Vendor.Google</ClassName>
  </Vendor>
</Vendors>
```

# Load the XML File into a Collection Class

When you have an XML file like that shown above, you will typically create a class with properties that match each element in the XML file. Below is a VendorClass that has properties for each element.

```
public class VendorClass
{
  public int VendorId { get; set; }
  public string VendorName { get; set; }
  public string AssemblyName { get; set; }
  public string ClassName { get; set; }
}
```

Next you create a collection class using the Generic List<VendorClass> class. In this class there is a LoadVendors() method. This method uses the XElement class to load the Vendors.xml file into memory. LINQ to XML is used to iterate over the XML elements and create a collection of VendorClass objects. This resulting collection of objects is added to the List using the AddRange() method.

```
public class VendorClasses : List<VendorClass>
{
  public VendorClasses LoadVendors()
  {
    // Load XML file into memory
    var xElem = XElement.Load(
      GetCurrentDirectory() +
      ConfigurationManager.AppSettings["VendorFile"]);

    // Read Vendors using LINQ to XML
    var vendors = from vend in xElem.Descendants("Vendor")
       select new VendorClass
       {
         VendorId =
           Convert.ToInt32(vend.Element("VendorId").Value),
         VendorName = vend.Element("VendorName").Value,
         AssemblyName = vend.Element("AssemblyName").Value,
         ClassName = vend.Element("ClassName").Value
        };

    // Add vendors read in to the List of Vendor classes
    this.AddRange(vendors.ToList());

    return this;
  }
}
```

To load an XML file you need to find the location on disk. There is a GetCurrentDirectory() method in this class. This method gets the current directory and removes any \bin\Debug from the end of the path if present. This allows you to get the current path whether you are running in Visual Studio or not.

```
private string GetCurrentDirectory()
{
  string path = null;

  path = AppDomain.CurrentDomain.BaseDirectory;
  if (path.IndexOf("\\bin") > 0)
  {
    path = path.Substring(0, path.LastIndexOf("\\bin"));
  }

  if(!path.EndsWith(@"\"))
    path += @"\";

  return path;
}
```

# Create a Vendor Object Dynamically

Now that you have the list of assembly and class names loaded into a collection you can now create an instance of a specific class. In order to do this you must have an Interface or a base class. I have created an Interface called IVendor. This interface will need to be implemented by each Vendor class that you will be dynamically loading. This interface is located in a separate assembly from all other classes and from the main application. This assembly will need to be referenced by the main application as well as from each assembly where your Vendor classes are located.

```
public interface IVendor
{
  int VendorId { get; set; }
  string VendorName { get; set; }
  string GetVendorInfo();
}
```

Below is an example of a vendor class named “Apple” that implements the IVendor interface. The GetVendorInfo() method will simply return a string just so we can verify that the class did get created correctly.

```
public class Apple : IVendor
{
  public int VendorId { get; set; }
  public string VendorName { get; set; }

  public string GetVendorInfo()
  {
    return "Apple Corporation";
  }
}
```

Below is another example of a vendor class named “Microsoft” that also implements the IVendor interface.

```
public class Microsoft : IVendor
{
  public int VendorId { get; set; }
  public string VendorName { get; set; }

  public string GetVendorInfo()
  {
    return "Microsoft Corporation";
  }
}
```

Each of the above classes are located in separate assemblies as you can see by the AssemblyName element in the XML file. These DLLs do NOT need to be referenced from your project, and in fact, should not be. The Create() method shown below uses the Assembly.LoadFile() method to load the assembly from the current directory. That means that the assembly where each Vendor class is located must be in the current directory for this sample. After the DLL has been loaded, you use the CreateInstance() method on the loaded assembly to load the specific class name. Finally you take the VendorId and the VendorName and place it into the Vendor object’s appropriate properties. This newly created Vendor object is returned from this method.

```
public class VendorClass
{
  // Properties here…

  public IVendor Create()
  {
    Assembly assm;
    IVendor vendor = null;

    // Load Assembly File
    assm = Assembly.LoadFile(
      AppDomain.CurrentDomain.BaseDirectory +
       AssemblyName + ".dll");

    // Create new instance of Class
    vendor = (IVendor)assm.CreateInstance(ClassName);

    // Set properties in the Provider class from the XML file
    vendor.VendorId = VendorId;
    vendor.VendorName = VendorName;

    return vendor;
  }
}
```

# The Sample to Test Activation

To test activating each of these vendor classes I have created a WPF application (Figure 1). The Load Vendors button will call the LoadVendors() method to build the collection of VendorClass objects from the XML file.

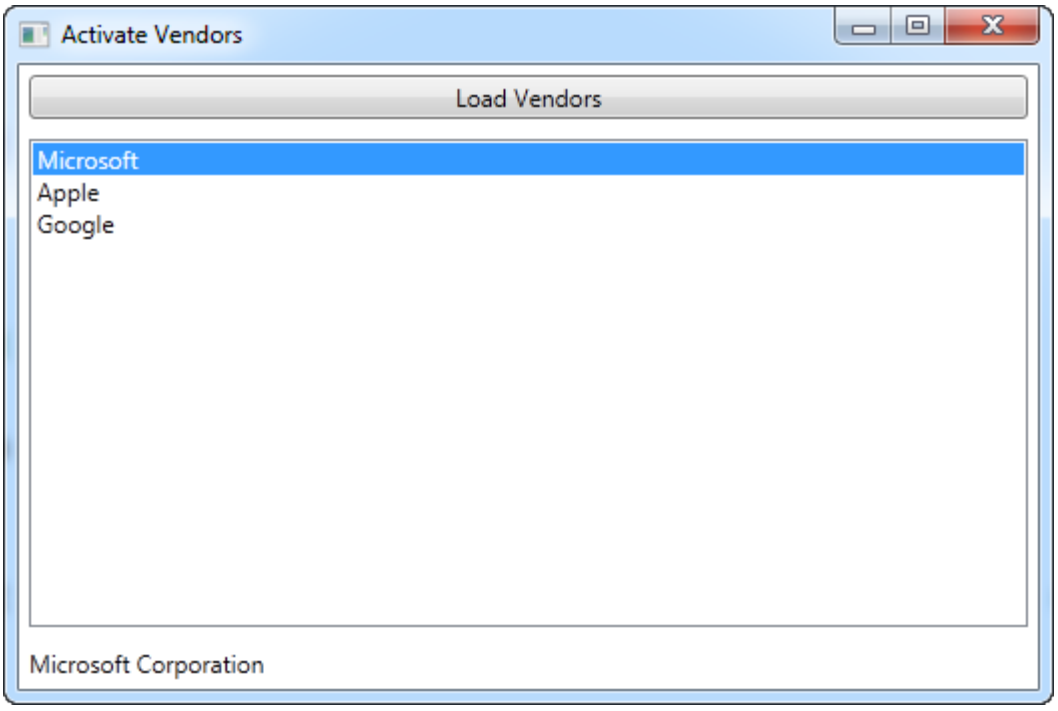


Figure 1: A sample to test our activation

Below is the code in the btnLoad_Click event procedure that loads the collection of VendorClass objects.

```
private void btnLoad_Click(object sender, RoutedEventArgs e)
{
  VendorClasses classes = new VendorClasses();

  // Load all vendors in the XML file
  classes.LoadVendors();
  lstVendors.DataContext = classes;
}
```

Once the VendorClass collection is in the List Box you can click on the list box to fire the SelectionChanged event procedure. In this event procedure you will cast the SelectedItem property of the list box into a VendorClass object. You can then call the Create() method to instantiate the Vendor object by loading the assembly and creating an instance of the Vendor object. After the Vendor object has been created you can call the GetVendorInfo() method to verify that the assembly has been loaded and an instance of the class has been created.

```
private void lstVendors_SelectionChanged(object sender,
  SelectionChangedEventArgs e)
{
  VendorClass vendor;
  IVendor vend;

  // Get the currently selected vendor
  vendor = (VendorClass)lstVendors.SelectedItem;
  // Create an instance of a Vendor class
  vend = vendor.Create();

  tbVendorName.Text = vend.GetVendorInfo();
}
```

# Summary

In this blog post you learned how to use LINQ to XML to load a collection of classes that contain information about other classes that you will eventually load. You saw how to use the Assembly class to load an assembly from disk and instantiate a class from a string value. An Interface is used to ensure that each class to be instantiated has the same set of methods and properties. Using this technique can eliminate a switch/Select case statement and allow you to dynamically add classes to any application.

NOTE: You can download the sample code for this article by visiting my website at http://www.pdsa.com/downloads. Select “Tips & Tricks”, then select “XML Activator” from the drop down list.