



CODE
PROJECT®
For those who code



Greg Osborne
31 Jul 2007 CPOL

Extend your applications by providing a framework for other developers



Download source - 49.21 KB

Introduction

Would you like to be able to supply to the users of your applications an extensible framework that they can use to expand the functionality of your base application? We see this all the time – in Visual Studio and Office products they are called add-ins; in Eclipse, they are called plug-ins; in other apps I've heard them called as extensions or snap-ins or modules. Whatever they are called, they do one thing; they allow your application functionality to be expanded with functions you may never have dreamed of.

In this article, I'm going to show you how to create a simple framework for extensibility that you can use to expand the functionality of your application.

Using the Code

It's actually very simple to allow other developers to extend your application. All you have to do is provide a base class to extend from and then when loading the extensions, make sure that a class extends from that class. This class could be either an Interface or a Class. I chose to use an Interface, but if you have common code that you would like to be able to execute from any extension, you should use a Class. If you don't use an interface, you will need to provide another method of checking the type (see below in the **Extensions** collection class, **searchDir** method).

Below is the base class for the simple extension:

C#

```
using System.Drawing;
using System;
using System.Collections.Generic;
using System.Text;

namespace Extensibility {

    public interface IExtender {
        String Name {
            get;
        }

        String Description {
            get;
        }

        String MenuText {
            get;
        }
    }
}
```

```

    }

    String DLLPath {
        get;
    }

    Image Image {
        get;
    }

    String Provider {
        get;
    }

    Object Execute();
}
}

```

You can expand this with whatever properties and methods you want. I also like to type-safe my collections so I've provided an **Extensions** collection class. This also allows me a class to use for other things, like populating the extensions and the **MenuItem** if they are to be accessed from a menu, and providing an event to notify the application when an extension has been clicked. Below is my **Collection** class:

C#

```

using System.Windows.Forms;
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.Reflection;

namespace Extensibility {
    public class Extensions : System.Collections.CollectionBase {
        public delegate void extensionClickedEventHandler(object sender,
            EventArgs e);
        public event extensionClickedEventHandler ExtensionClicked;
        public int add(IExtender extension) {
            return this.List.Add(extension);
        }
        public void remove(int index) {
            this.List.Remove(index);
        }
        public IExtender item(int index) {
            return (IExtender)this.List[index];
        }

        public IExtender item(String name) {
            IExtender ret = null;
            foreach (IExtender ext in this.List) {
                if(ext.Name.Equals(name)){
                    ret = ext;
                    break;
                }
            }
            return ret;
        }

        public int populate(String directory, bool searchSubs,
            ToolStripMenuItem menuItem) {
            int ret = this.populate(directory, searchSubs);
            if (ret > 0) {
                foreach (IExtender ext in this.List) {
                    ToolStripMenuItemEx item = new
                        ToolStripMenuItemEx();
                    item.Click += new EventHandler(item_Click);
                    if (ext.MenuText != null) item.Text =
                        ext.MenuText;
                }
            }
        }
    }
}

```

```

        if (ext.Image != null) item.Image = ext.Image;
        if (ext.Description != null)
            item.ToolTipText = ext.Description;
        item.Extension = ext;
        menuItem.DropDownItems.Add(item);
    }
}
return ret;
}

void item_Click(object sender, EventArgs e) {
    ExtensionClicked.Invoke(sender, e);
}

public int populate(String directory) {
    return this.populate(directory, false);
}

public int populate(String directory, bool searchSubs) {
    if(!Directory.Exists(directory)){
        throw new IOException("Directory not found");
    }
    DirectoryInfo dir = new DirectoryInfo(directory);
    is.searchDir(dir, searchSubs);
    return this.Count;
}

private void searchDir(DirectoryInfo dir, bool searchSubs) {
    FileInfo[] files = dir.GetFiles("*.dll");
    if (files.Length == 0) {
        throw new IOException("No dll files found in " +
            dir.FullName);
    }
    foreach (FileInfo f in files) {
        String fileName = f.FullName;
        Assembly assy = Assembly.LoadFile(fileName);
        Type[] types = assy.GetTypes();
        foreach (Type t in types) {
            if (t.GetInterface("IExtender") != null) {
                IExtender ext =
                    (IExtender)assy.CreateInstance(t.FullName);
                this.add(ext);
            }
            if (searchSubs) {
                foreach(DirectoryInfo subDir in
                    dir.GetDirectories()){
                    this.searchDir(subDir, searchSubs);
                }
            }
        }
    }
}
}

```

Some notes about the load code. Reflection is used to load the extensions from all libraries in the specified directory. The code goes through all DLL files located in the specified directory and queries all types to see if they implement the **IExtender** interface. If it does, we load it into our collection. I've chosen to use the directory method (loading all extensions) over specifically identifying extensions because of ease of use. I could have easily provided registry entries or a file to specify which extensions to load or ignore.

Back to business - optionally, when we call the **populate** method, we can supply a **ToolStripMenuItem** and it will also populate that **menuItem** with all the extensions it finds. In this regard, I've also provided an extended **ToolStripMenuItem** class (**ToolStripMenuItemEx**) that will hold the extension so I can pass back an extension to the application when it is clicked. The **populate** method will also create the internal event handler, which will raise the external event handler in your application and pass back an extender.

Below see the **ToolStripMenuItemEx** class:

C#

```

using System;
using System.Collections.Generic;
using System.Text;

namespace Extensibility {
    public class ToolStripMenuItemEx : System.Windows.Forms.ToolStripMenuItem {
        private IExtender _Extension = null;
        public IExtender Extension {
            get { return _Extension; }
            set { _Extension = value; }
        }
    }
}

```

That's all there is to the framework.

Extending Application Functionality With an Extension

Now that we have our framework, all we have to do is create libraries of extensions. In a new class library project (any .NET language will work), add a reference to the **Extensibility** project, paste the following code into a new class:

```

C#

using System.Windows.Forms;
using System.Drawing;
using System;
using System.Collections.Generic;
using System.Text;
using Extensibility;
using System.Reflection;

namespace ExtensionsTest {
    public class SomeExtension_1 : IExtender {

        public SomeExtension_1() {
            String imagePath = System.IO.Path.GetDirectoryName
            (Assembly.GetExecutingAssembly().Location);
            Icon icn = new Icon(System.IO.Path.Combine(imagePath, "18.ico"));
            _Image = icn.ToBitmap();
        }

        private String _Name = "Some extension #1";
        private String _Description = "Description of Some Extension #1";
        private String _MenuText = "Extension 1";
        private Image _Image = null;
        private String _Provider = "My Company Name";

        #region IExtender Members

        string IExtender.Name {
            get { return _Name; }
        }

        string IExtender.Description {
            get { return _Description; }
        }

        string IExtender.MenuText {
            get { return _MenuText; }
        }

        string IExtender.DLLPath {
            get {
                return Assembly.GetExecutingAssembly().Location;
            }
        }
    }
}

```

```

    Image IExtender.Image {
        get { return _Image; }
    }

    string IExtender.Provider {
        get { return _Provider; }
    }

    object IExtender.Execute() {
        return "The Execute method or operation for
        SomeExtension_1 is not implemented.";
    }

    #endregion
}
}

```

This will create an extension called **SomeExtension_1**. Note that all the identification information should be provided through read-only properties. The DLL path is returned by reflection automatically. This would have been a good reason to use a class vs. an interface so this does not have to be provided in every extension. The image is optional as it will check on **menuItem** creation, and it is set when the extension is created.

Putting It All Together

Here is the final result. From this point, it's very simple to create and execute extensions in your application. Create a new Windows application, again, add a reference to the **Extensibility** project, add in a new form and a button (**button1**), and **menustrip** (**menuStrip1**) with a tools **menuItem**, and under the tools **menuItem**, an **extensions menuItem**. Note that we don't have to reference any of the **Extensions** library (DLL) files; we don't want to specify any DLLs because we want the selection process to be dynamic. Then paste the following code:

C#

```

void extensions_ExtensionClicked(object sender, EventArgs e) {
    IExtender extender = ((ToolStripMenuItemEx)sender).Extension;
    MessageBox.Show((String)extender.Execute());
}

private Extensions extensions = null;
private void button1_Click(object sender, EventArgs e) {
    extensions = new Extensions();
    extensions.ExtensionClicked += new
Extensions.ExtensionClickedEventHandler(extensions_ExtensionClicked);
    try {
        extensions.populate(Application.StartupPath, false,
this.extensionsToolStripMenuItem);
    } catch (Exception ex) {
        MessageBox.Show(ex.Message);
    }
}
}

```

Running the application before clicking **button1**, you will see that the **extensions** item is empty. Clicking button 1 will do the following:

- Create a new **extensions** collection
- Set up an event handler
- Populate the **extensions** collection with all extensions in the **Application.Startup** path directory

If you look at the **extensions** menuItem, you will now see the **extensions**. Clicking on it will display the text from the **execute** method of the **extension**. Viola! you've just extended your application with functionality that didn't exist when you created it. Now anyone who creates a DLL with **extensions** can drop it into the application folder, and it will be included automatically in your application.

Some Development Notes

Because we want to dynamically load our extensions without referencing them in our main application (it would defeat our purpose), our return type from our **execute** method is of type **object**. This will allow us to return and use only types that your application knows about. A good candidate for a return object is an **XmlDocument** object. This would allow you to represent any data you want, as long as your main application can interpret it. You can get around this limitation by providing an object model with specific types that an extension is required to return. It's up to you.

Happy extending!

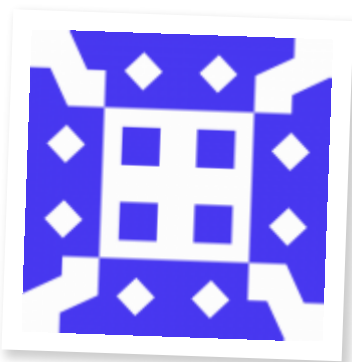
History

- 30th July, 2007: Initial post

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author



Greg Osborne


United States 

Visual Basic Developer since version 1.0

Java web developer

Currently developing in vb and c#

Comments and Discussions

 **2 messages** have been posted for this article Visit <https://www.codeproject.com/Articles/19805/A-Simple-Extension-Framework-in-C> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#)
[Advertise](#)
[Privacy](#)
[Cookies](#)
[Terms of Use](#)

Article Copyright 2007 by Greg Osborne
Everything else Copyright © [CodeProject](#), 1999-2021

Web04 2.8.20210901.1