**[Reducing Coupling by Dynamic Loading of Assemblies](http://msdn.microsoft.com/en-us/library/cc984323.aspx)**

Danut Prisacaru  
Sean Payne

September 2008

Applies to:  
   Microsoft .NET Framework  
   Windows Presentation Foundation

**Summary:** This article’s foundation is a previous article (called “[Software Abstraction Layer](http://msdn2.microsoft.com/en-us/library/cc304370.aspx)”) that proposed a solution for the separation of concerns between logical and physical layers in an *n*-tier application. After we separate the layers, we have to look at how we can go even further and minimize the impact of changes in one layer if another layer changes—also, how we can increase the discipline in a team of developers and prevent them from taking shortcuts that will defeat the initial purpose of separating the components, so that they do not affect each other, as long as the interface between them is unchanged. This article tries to identify the points of coupling. It also proposes a solution that increases the separation of concerns, which allows the applications to be easier to change and deploy. Although the article makes its point by using Microsoft .NET Framework technologies and the C# language, the solutions can apply also to other platforms and languages.

Finally, we provide a proof-of-concept prototype that illustrates the points that we are trying to make in this article. (9 printed pages)

**Contents**

[Scenario](http://msdn.microsoft.com/en-us/library/cc984323.aspx#Scenario)  
[Existing Solutions](http://msdn.microsoft.com/en-us/library/cc984323.aspx#Existing)  
[Our Solution](http://msdn.microsoft.com/en-us/library/cc984323.aspx#OurSolution)  
[Main Application](http://msdn.microsoft.com/en-us/library/cc984323.aspx#MainApplication)  
[Modules Dependency](http://msdn.microsoft.com/en-us/library/cc984323.aspx#ModulesDependency)  
[Non-UI Modules](http://msdn.microsoft.com/en-us/library/cc984323.aspx#NonUIModules)  
[Real-World Applications](http://msdn.microsoft.com/en-us/library/cc984323.aspx#RealWorldApps)  
[Proof-of-Concept Prototype](http://msdn.microsoft.com/en-us/library/cc984323.aspx#ProofOfConcept)

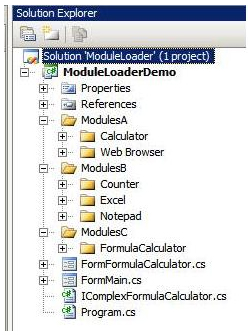
**Scenario**

Any complex application is made of more than one file (.exe). Usually, it is composed of more than one package, DLL, assembly, and so on. These assemblies attach to the application, either statically at build time or dynamically at run time. Making the link at run time gives more freedom for changing the internal structure of each module. While this is better than the static link, we think that the solution that we provide will make it even easier for maintenance.

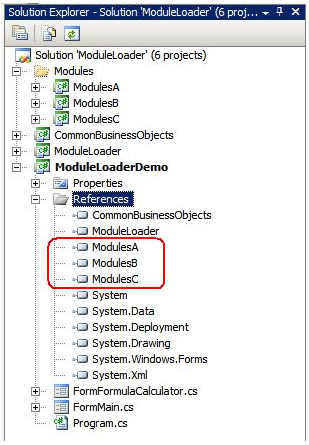
Let us say that we have a Smart Client application that is built on Microsoft .NET Framework 2.0. The application is quite complex; it has many menus, toolbar buttons, and so on. Each menu item will launch a form; some forms must be singleton; one instance of the form is allowed, other forms can have more than one instance on the screen; some forms are modal, some are modeless. For the sake of simplification, we will consider our application a Multiple Document Interface (MDI) application.

We also have some non–user-interface (UI) modules that are called by the UI to perform certain tasks.

Figure 1 illustrates the monolithic application; Figure 2 shows the same application with some decoupling, each module being in its own assembly:



**Figure 1. The monolithic application—one executable that has everything**



**Figure 2. The decoupled application—each module in its own assembly**

The main Microsoft Visual Studio solution will have references to each module assembly. With this approach, the developer will “see” all of the namespaces that are located in the modules. While classes can be well designed, “free” access into all of the namespaces, public classes, and members can increase coupling and even lead to circular dependencies.

If we launch the form from the main window, we will have code like the following:

MyForm form = new MyForm();

Form.Show();

 It looks fine, but what we have here is a coupling with the technology that is used—in this case, Windows Forms. If we decide to move this module to Windows Presentation Foundation (WPF), the preceding code will change to the following code:

MyWindow win = new MyWindow();

win.Show();

 It does not look like extensive changes. What the code does not show is that we had to add to our Windows Forms application references to WPF assemblies: **PresentationCore**, **PresentationFramework**, and **WindowsBase**.

This means that the main module code will have to change and be released again. The changes in the main module will trigger also more testing of this module. We want to minimize and isolate the changes, because we want to reduce the testing, and we also want to release only the modified module.

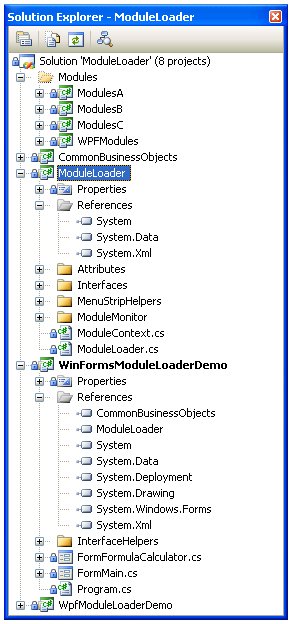
**Existing Solutions**

Some applications achieve a certain level of decoupling by using configuration files, either in the format of old .ini files or in XML. While they improve the flexibility, the configuration files can be difficult to maintain if the application has many different groups of users that have needs to access different modules.

**Our Solution**

A well-designed object-oriented (OO) application is made out of many small classes/objects that work together in a democratic way. This makes the application easy to maintain, because there is no central controller or main container. A good OO design avoids a big, fat base class from which many other classes have to inherit. Our solution implements a very light module loader that looks for specific attributes and interfaces in the assemblies that it finds on the disk. One assembly can contain one or more classes that “speak” the module-loader language. Each class is in charge of communicating to the module loader the information about the loadable module. Then, the module loader will call the module that must be loaded and pass the responsibility to it. This way, each loadable module will decide how it will load: as a singleton modeless form, a form that allows multiple instances to exist, a modal dialog, and so on.

Figure 3 shows that neither the main application nor the module loader has any references to the module assemblies:



**Figure 3. The dynamic loader application has no references to the modules or WPF libraries. ModuleLoader has no references to WinForms libraries, WPF libraries, or any modules.**

We decided to separate the loadable modules from the classes that contain information about the modules; we did that for decoupling. The loadable modules that implement a UI function might have to be refactored to move from Windows Forms to WPF, for example. If that is the case, the information class does not have to change; only the UI module is required to be changed.

The main application will have only a loader module and will “see” only the interfaces that each module has to implement.

During the application startup, the loader will look into a predefined folder and get the list of the files that have the “.dll” extension. Then, by using reflection, each file is inspected. The loader looks for specific C# interfaces and class attributes.

We decide to separate the process of loading a module from the interfaces/classes that launch these modules. This separation allows the modules to be replaced with new ones, eventually using different technologies, without changing the interface/loader class.

Imagine that you have a UI form that is written in Windows Forms. If you decide to replace this with WPF, the changes to the loader class are minimal.

In our prototype, the main .exe file is an MDI that is based on Windows Forms. One of the assemblies contains a WPF Window. None of the other modules or the main application contains any reference to any WPF libraries, and the WPF module does not contain any reference to any Windows Forms–related assemblies.

**Main Application**

The main application has code that initializes the Module Loader and the Library Monitor. The rest of the work is done by these two components.

**Module Loader**

The Module Loader is the one that does its work when the application initializes. It looks into the predefined folder (for example, the Modules folder) and enumerates and loads all of the existing assemblies, looking for the types that implement the **ILoadableModule** interface. Then, it inspects these types and gets the data from the **MenuInformation** attribute that is building the menu.

 Type[] assemblyTypes = asm.GetTypes();

foreach (Type module in assemblyTypes)

{

    if (null != module.GetInterface(typeof(ILoadableModule).Name))

    {

        String strModuleName =

strFile.Substring(strFile.LastIndexOf('\\') + 1);

        if (false == \_mapFilesLoaders.ContainsKey(strModuleName))

        {

            \_mapFilesLoaders.Add(strModuleName, new List<Type>());

        }

        // update our list of types

        \_mapFilesLoaders[strModuleName].Add(module);

        // this is a loadable module, return it to caller.

        yield return module;

    }

}

**Listing 1. This code checks if modules are of type ILoadableModule**

public void LoadModules(String strMask)

{

    foreach (Type loader in

GetLoadableModules(ModuleLoader.ModulesDirectory, strMask))

    {

        MenuInformationAttribute[] menuInfo = loader.GetCustomAttributes(

typeof(MenuInformationAttribute), false)

as MenuInformationAttribute[];

        if (null != menuInfo && 1 == menuInfo.Length)

        {

            Object item = \_menuCreator.CreateMenuFromAttribute(

menuInfo[0], loader);

            if (null != item)

            {

                \_owner.CreatedObject(item, loader);

            }

        }

        else

        {

            \_owner.CreatedObject(null, loader);

        }

    }

}

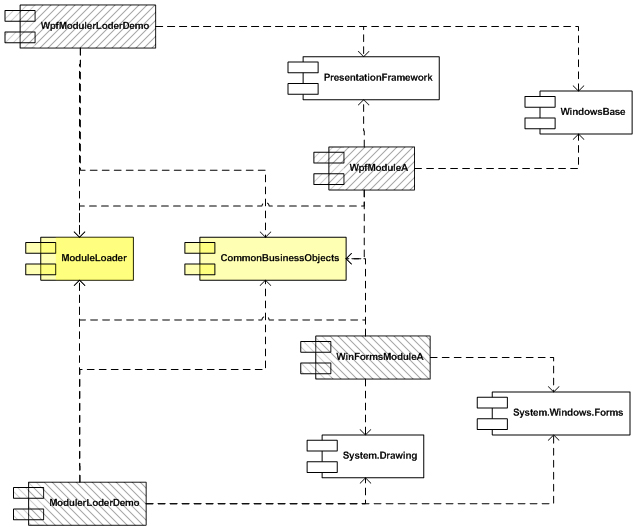
**Listing 2. For each module loaded by the previous method, we look at the class’s attributes to build the proper menu items.**

**Library Monitor**

The Library Monitor is constantly watching the predefined Modules folder and looks for any changes that happen to the assemblies; as soon as it detects that one or more assemblies were changed, it invokes the Module Loader.

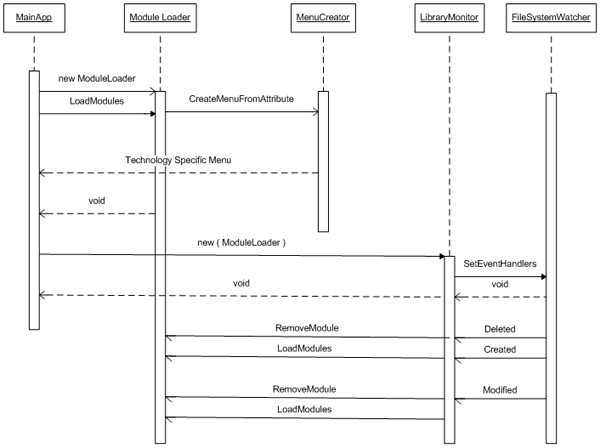
**Modules Dependency**

Figure 4 shows the dependencies among modules. You should be able to see clearly that the assemblies are not directly referenced by the main application; therefore, it is easy to remove assemblies, add new ones, or replace an older version of an assembly with a newer one.



**Figure 4. Modules dependency**

Figure 5 shows the sequence diagram and the order of operations for both the Module Loader and the Library Monitor:



**Figure 5. Sequence diagram**

**Non-UI Modules**

The scenario can become even more “dynamic” by allowing a “hot” drop of new or changed assemblies. Imagine that you have a module that implements some algorithms for calculating data that is required by the business. The business rules have changed, and we must update the module to support the new formulae. If the module is part of a back-end server that must run 24/7, stopping the server to release a new version of the module will affect the business.

With dynamic loading of assemblies, that is not necessary. We can just drop the new assembly over the old one, and the solution that we provide simply will reload the new version in memory. Because the old modules do not get unloaded physically in memory, thread synchronization is not an issue; threads which are in the code when the library is changed will finish their execution, but any new threads will instead enter into the new code, even as the old code is being executed. The preceding scenario assumes that the interface between the calculation module and the rest of the system/application did not change, but only the implementation of the module changed.

If unloading of the assemblies is required due to resource constraints, they would instead have to be loaded into their own application domains and accessed via proxies. However, this topic is beyond the scope of the current article.

**Real-World Applications**

The solution that we proposed was implemented in our application, which also fully implements the [software abstraction layer](http://msdn2.microsoft.com/en-us/library/cc304370.aspx). This solution allowed us to release a major change to one module in one week. This is how long it took both development and testing to implement and test the complex changes to that module. The release to the user’s computer consisted of just the changed assembly. That provided an unprecedented time-to-market reaction from our team that was very welcomed by our business partners and clients.

Adding new modules can be done simply without making changes to the main application. This avoids turning the main window/form into a big class with spaghetti code, as it happens in many applications. The same holds true when we must remove a module. Creating a new application from scratch is not the biggest challenge of a project. The most expensive part is maintenance, and we believe that our solution increases decoupling and reduces the maintenance cost, because it eliminates or reduces to a minimum the number of changes that are required in the main executable.

**Proof-of-Concept Prototype**

We provide a proof-of-concept prototype that shows the points that we make in this article. The main application is an MDI container that is built on Windows Forms UI. Several of our modules launch different external applications or display a form. One module will launch a WPF window.

The prototype can be found [here](http://www.codeplex.com/DLA/) on the CodePlex Web site.

<http://www.codeplex.com/DLA>

**About the authors**

**Danut** **Prisacaru** is an application architect who works in the financial industry. He has more than 17 years of experience in developing software applications. In the last few years, he has architected, designed, and implemented several two-tier and three-tier applications that use object-oriented design principles, .NET Framework, Microsoft SQL Server, Smart Client applications, and service-oriented architecture (SOA). Danut likes to find solutions that can make an application easy to maintain. Also, he likes the world of object-oriented design, and is passionate about taking business needs, analyzing them, and turning them into solutions that can improve the efficiency of software applications. Danut can be reached at [danut@computer.org](mailto:danut@computer.org).

**Sean Payne** is a software developer who works in the financial industry. He has spent the last two years developing two-tiered and three-tiered applications that use .NET Framework and Microsoft SQL Server. He enjoys improving the performance and end-usability of applications, to make them faster and more reliable for clients. Sean can be reached at [spayne713@gmail.com](mailto:spayne713@gmail.com).

<http://msdn.microsoft.com/en-us/library/cc984323.aspx>