Managed Extensibility Framework (MEF)

.NET Framework 4.5 Other Versions

This topic provides an overview of the Managed Extensibility Framework introduced in the .NET Framework 4.

This topic contains the following sections.

What is MEF?

The Managed Extensibility Framework or MEF is a library for creating lightweight, extensible applications. It allows application developers to discover and use extensions with no configuration required. It also lets extension developers easily encapsulate code and avoid fragile hard dependencies. MEF not only allows extensions to be reused within applications, but across applications as well.

The Problem of Extensibility

Imagine that you are the architect of a large application that must provide support for extensibility. Your application has to include a potentially large number of smaller components, and is responsible for creating and running them.

The simplest approach to the problem is to include the components as source code in your application, and call them directly from your code. This has a number of obvious drawbacks. Most importantly, you cannot add new components without modifying the source code, a restriction that might be acceptable in, for example, a Web application, but is unworkable in a client application. Equally problematic, you may not have access to the source code for the components, because they might be developed by third parties, and for the same reason you cannot allow them to access yours.

A slightly more sophisticated approach would be to provide an extension point or interface, to permit decoupling between the application and its components. Under this model, you might provide an interface that a component can implement, and an API to enable it to interact with your application. This solves the problem of requiring source code access, but it still has its own difficulties.

Because the application lacks any capacity for discovering components on its own, it must still be explicitly told which components are available and should be loaded. This is typically accomplished by explicitly registering the available components in a configuration file. This means that assuring that the components are correct becomes a maintenance issue, particularly if it is the end user and not the developer who is expected to do the updating.

In addition, components are incapable of communicating with one another, except through the rigidly defined channels of the application itself. If the application architect has not anticipated the need for a particular communication, it is usually impossible.

Finally, the component developers must accept a hard dependency on what assembly contains the interface they implement. This makes it difficult for a component to be used in more than one application, and can also create problems when you create a test framework for components.

What MEF Provides

Instead of this explicit registration of available components, MEF provides a way to discover them implicitly, via *composition*. A MEF component, called a *part*, declaratively specifies both its dependencies (known as *imports*) and what capabilities (known as *exports*) it makes available. When a part is created, the MEF composition engine satisfies its imports with what is available from other parts.

This approach solves the problems discussed in the previous section. Because MEF parts declaratively specify their capabilities, they are discoverable at runtime, which means an application can make use of parts without either hard-coded references or fragile configuration files. MEF allows applications to discover and examine parts by their metadata, without instantiating them or even loading their assemblies. As a result, there is no need to carefully specify when and how extensions should be loaded.

In addition to its provided exports, a part can specify its imports, which will be filled by other parts. This makes communication among parts not only possible, but easy, and allows for good factoring of code. For example, services common to many components can be factored into a separate part and easily modified or replaced.

Because the MEF model requires no hard dependency on a particular application assembly, it allows extensions to be reused from application to application. This also makes it easy to develop a test harness, independent of the application, to test extension components.

An extensible application written by using MEF declares an import that can be filled by extension components, and may also declare exports in order to expose application services to extensions. Each extension component declares an export, and may also declare imports. In this way, extension components themselves are automatically extensible.

Where Is MEF Available?

MEF is an integral part of the .NET Framework 4, and is available wherever the .NET Framework is used. You can use MEF in your client applications, whether they use Windows Forms, WPF, or any other technology, or in server applications that use ASP.NET.

MEF and MAF

Previous versions of the .NET Framework introduced the Managed Add-in Framework (MAF), designed to allow applications to isolate and manage extensions. The focus of MAF is slightly higher-level than MEF, concentrating on extension isolation and assembly loading and unloading, while MEF's focus is on discoverability, extensibility, and portability. The two frameworks interoperate smoothly, and a single application can take advantage of both.

SimpleCalculator: An Example Application

The simplest way to see what MEF can do is to build a simple MEF application. In this example, you build a very simple calculator named SimpleCalculator. The goal of SimpleCalculator is to create a console application that accepts basic arithmetic commands, in the form "5+3" or "6-2", and returns the correct answers. Using MEF, you will be able to add new operators without changing the application code.

To download the complete code for this example, see the SimpleCalculator sample.

Note

The purpose of SimpleCalculator is to demonstrate the concepts and syntax of MEF, rather than to necessarily provide a realistic scenario for its use. Many of the applications that would benefit most from the power of MEF are more complex than SimpleCalculator. For more extensive examples, see the Managed Extensibility Framework on Codeplex.

To start, in Visual Studio 2010, create a new Console Application project named **SimpleCalculator**. Add a reference to the System.ComponentModel.Composition assembly, where MEF resides. Open Module1.vb or Program.cs and add **Imports** or **using** statements for System.ComponentModel.Composition and System.ComponentModel.Composition.Hosting. These two namespaces contain MEF types you will need to develop an extensible application. In Visual Basic, add the `Public` keyword to the line that declares the `Module1` module.

Composition Container and Catalogs

The core of the MEF composition model is the *composition container*, which contains all the parts available and performs composition. (That is, the matching up of imports to exports.) The most common type of composition container is CompositionContainer, and you will use this for SimpleCalculator.

In Visual Basic, in Module1.vb, add a public class named `Program`. Then add the following line to the `Program` class in Module1.vb or Program.cs:

C# VB

```
private CompositionContainer _container;
```

In order to discover the parts available to it, the composition containers makes use of a *catalog*. A catalog is an object that makes available parts discovered from some source. MEF provides catalogs to discover parts from a provided type, an assembly, or a directory. Application developers can easily create new catalogs to discover parts from other sources, such as a Web service.

Add the following constructor to the `Program` class:

C# VB

```
private Program()
{
    //An aggregate catalog that combines multiple catalogs
    var catalog = new AggregateCatalog();
    //Adds all the parts found in the same assembly as the Program class
    catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));

    //Create the CompositionContainer with the parts in the catalog
    _container = new CompositionContainer(catalog);

    //Fill the imports of this object
    try
```

```
        {
            this._container.ComposeParts(this);
        }
        catch (CompositionException compositionException)
        {
            Console.WriteLine(compositionException.ToString());
       }
    }
```

The call to ComposeParts tells the composition container to compose a specific set of parts, in this case the current instance of `Program`. At this point, however, nothing will happen, since `Program` has no imports to fill.

Imports and Exports with Attributes

First, you have `Program` import a calculator. This allows the separation of user interface concerns, such as the console input and output that will go into `Program`, from the logic of the calculator.

Add the following code to the `Program` class:

C# VB

```
[Import(typeof(ICalculator))]
public ICalculator calculator;
```

Notice that the declaration of the `calculator` object is not unusual, but that it is decorated with the ImportAttribute attribute. This attribute declares something to be an import; that is, it will be filled by the composition engine when the object is composed.

Every import has a *contract*, which determines what exports it will be matched with. The contract can be an explicitly specified string, or it can be automatically generated by MEF from a given type, in this case the interface `ICalculator`. Any export declared with a matching contract will fulfill this import. Note that while the type of the `calculator` object is in fact `ICalculator`, this is not required. The contract is independent from the type of the importing object. (In this case, you could leave out the `typeof(ICalculator)`. MEF will automatically assume the contract to be based on the type of the import unless you specify it explicitly.)

Add this very simple interface to the module or `SimpleCalculator` namespace:

C# VB

```
public interface ICalculator
{
    String Calculate(String input);
}
```

Now that you have defined `ICalculator`, you need a class that implements it. Add the following class to the module or `SimpleCalculator` namespace:

C# VB

```
[Export(typeof(ICalculator))]
class MySimpleCalculator : ICalculator
{

}
```

Here is the export that will match the import in `Program`. In order for the export to match the import, the export must have the same contract. Exporting under a contract based on `typeof(MySimpleCalculator)` would produce a mismatch, and the import would not be filled; the contract needs to match exactly.

Since the composition container will be populated with all the parts available in this assembly, the MySimpleCalculator part will be available. When the constructor for Program performs composition on the Program object, its import will be filled with a MySimpleCalculator object, which will be created for that purpose.

The user interface layer (Program) does not need to know anything else. You can therefore fill in the rest of the user interface logic in the Main method.

Add the following code to the Main method:

C# VB

```
static void Main(string[] args)
{
    Program p = new Program(); //Composition is performed in the constructor
    String s;
    Console.WriteLine("Enter Command:");
    while (true)
    {
        s = Console.ReadLine();
        Console.WriteLine(p.calculator.Calculate(s));
    }
}
```

This code simply reads a line of input and calls the Calculate function of ICalculator on the result, which it writes back to the console. That is all the code you need in Program. All the rest of the work will happen in the parts.

Further Imports and ImportMany

In order for SimpleCalculator to be extensible, it needs to import a list of operations. An ordinary ImportAttribute attribute is filled by one and only one ExportAttribute. If more than one is available, the composition engine produces an error. To create an import that can be filled by any number of exports, you can use the ImportManyAttribute attribute.

Add the following operations property to the the MySimpleCalculator class:

C# VB

```
[ImportMany]
IEnumerable<Lazy<IOperation, IOperationData>> operations;
```

Lazy<T, TMetadata> is a type provided by MEF to hold indirect references to exports. Here, in addition to the exported object itself, you also get *export metadata*, or information that describes the exported object. Each Lazy<T, TMetadata> contains an IOperation object, representing an actual operation, and an IOperationData object, representing its metadata.

Add the following simple interfaces to the module or SimpleCalculator namespace:

C# VB

```
public interface IOperation
{
     int Operate(int left, int right);
}

public interface IOperationData
{
    Char Symbol { get; }
}
```

In this case, the metadata for each operation is the symbol that represents that operation, such as +, -, *, and so on. To make the addition operation available, add the following class to the module or `SimpleCalculator` namespace:

C# VB

```
[Export(typeof(IOperation))]
[ExportMetadata("Symbol", '+')]
class Add: IOperation
{
    public int Operate(int left, int right)
    {
        return left + right;
    }
}
```

The ExportAttribute attribute functions as it did before. The ExportMetadataAttribute attribute attaches metadata, in the form of a name-value pair, to that export. While the `Add` class implements `IOperation`, a class that implements `IOperationData` is not explicitly defined. Instead, a class is implicitly created by MEF with properties based on the names of the metadata provided. (This is one of several ways to access metadata in MEF.)

Composition in MEF is *recursive*. You explicitly composed the `Program` object, which imported an `ICalculator` that turned out to be of type `MySimpleCalculator`. `MySimpleCalculator`, in turn, imports a collection of `IOperation` objects, and that import will be filled when `MySimpleCalculator` is created, at the same time as the imports of `Program`. If the `Add` class declared a further import, that too would have to be filled, and so on. Any import left unfilled results in a composition error. (It is possible, however, to declare imports to be optional or to assign them default values.)

Calculator Logic

With these parts in place, all that remains is the calculator logic itself. Add the following code in the `MySimpleCalculator` class to implement the `Calculate` method:

C# VB

```
public String Calculate(String input)
{
    int left;
    int right;
    Char operation;
    int fn = FindFirstNonDigit(input); //finds the operator
    if (fn < 0) return "Could not parse command.";

    try
    {
        //separate out the operands
        left = int.Parse(input.Substring(0, fn));
        right = int.Parse(input.Substring(fn + 1));
    }
    catch
    {
        return "Could not parse command.";
    }

    operation = input[fn];

    foreach (Lazy<IOperation, IOperationData> i in operations)
    {
        if (i.Metadata.Symbol.Equals(operation)) return i.Value.Operate(left, right).ToString
    }
    return "Operation Not Found!";
}
```

The initial steps parse the input string into left and right operands and an operator character. In the `foreach` loop, every member of the `operations` collection is examined. These objects are of type Lazy<T, TMetadata>, and their metadata values and exported object can be accessed with the Metadata property and the Value property respectively. In this case, if the `Symbol` property of the `IOperationData` object is discovered to be a match, the calculator calls the `Operate` method of the `IOperation` object and returns the result.

To complete the calculator, you also need a helper method that returns the position of the first non-digit character in a string. Add the following helper method to the `MySimpleCalculator` class:

C# **VB**

```
private int FindFirstNonDigit(String s)
{
    for (int i = 0; i < s.Length; i++)
    {
        if (!(Char.IsDigit(s[i]))) return i;
    }
    return -1;
}
```

You should now be able to compile and run the project. In Visual Basic, make sure that you added the `Public` keyword to `Module1`. In the console window, type an addition operation, such as "5+3", and the calculator will return the results. Any other operator will result in the "Operation Not Found!" message.

Extending SimpleCalculator Using A New Class

Now that the calculator works, adding a new operation is easy. Add the following class to the module or `SimpleCalculator` namespace:

C# **VB**

```
[Export(typeof(IOperation))]
[ExportMetadata("Symbol", '-')]
class Subtract : IOperation
{
    public int Operate(int left, int right)
    {
        return left - right;
    }
}
```

Compile and run the project. Type a subtraction operation, such as "5-3". The calculator now supports subtraction as well as addition.

Extending SimpleCalculator Using A New Assembly

Adding classes to the source code is simple enough, but MEF provides the ability to look outside an application's own source for parts. To demonstrate this, you will need to modify SimpleCalculator to search a directory, as well as its own assembly, for parts, by adding a DirectoryCatalog.

Add a new directory named **Extensions** to the SimpleCalculator project. Make sure to add it at the project level, and not at the solution level. Then add a new Class Library project to the solution, named **ExtendedOperations**. The new project will compile into a separate assembly.

Open the Project Properties Designer for the ExtendedOperations project and click the **Compile** or **Build** tab. Change the **Build output path** or **Output path** to point to the Extensions directory in the SimpleCalculator project directory (..\SimpleCalculator\Extensions\).

In Module1.vb or Program.cs, add the following line to the `Program` constructor:

C# **VB**

```
catalog.Catalogs.Add(new DirectoryCatalog("C:\\SimpleCalculator\\SimpleCalculator\\Extensions
```

Replace the example path with the path to your Extensions directory. (This absolute path is for debugging purposes only. In a production application, you would use a relative path.) The DirectoryCatalog will now add any parts found in any assemblies in the Extensions directory to the composition container.

In the ExtendedOperations project, add references to SimpleCalculator and System.ComponentModel.Composition. In the ExtendedOperations class file, add an **Imports** or a **using** statement for System.ComponentModel.Composition. In Visual Basic, also add an **Imports** statement for SimpleCalculator. Then add the following class to the ExtendedOperations class file:

C# **VB**

```
[Export(typeof(SimpleCalculator.IOperation))]
[ExportMetadata("Symbol", '%')]
public class Mod : SimpleCalculator.IOperation
{
    public int Operate(int left, int right)
    {
        return left % right;
    }
}
```

Note that in order for the contract to match, the ExportAttribute attribute must have the same type as the ImportAttribute.

Compile and run the project. Test the new Mod (%) operator.

Conclusion

This topic covered the basic concepts of MEF.

- Parts, catalogs, and the composition container

  Parts and the composition container are the basic building blocks of a MEF application. A part is any object that imports or exports a value, up to and including itself. A catalog provides a collection of parts from a particular source. The composition container uses the parts provided by a catalog to perform composition, the binding of imports to exports.

- Imports and exports

  Imports and exports are the way by which components communicate. With an import, the component specifies a need for a particular value or object, and with an export it specifies the availability of a value. Each import is matched with a list of exports by way of its contract.

Where Do I Go Now?

To download the complete code for this example, see the SimpleCalculator sample.

For more information and code examples, see Managed Extensibility Framework. For a list of the MEF types, see the System.ComponentModel.Composition namespace.