D

Introducing MAF and MEF Frameworks

At times, you need to use some additional services of an application to accomplish your task. For instance, you are working on an application that handles the sales data of your company. While entering the records, you need to use the currency converter service of the application to convert the currency of one country to another. Now, since the currency converter service is not a part of the original application but added afterwards to serve some specific purpose, it is called an add-in service. The difference between an add-in and a host application is that a host application is complete in itself and can run on its own; whereas, an add-in can only work when loaded in a host application. The .NET Framework 3.5 has introduced a programming model, Managed Add-in Framework (MAF), to create add-ins and load them in their host applications. The communication between a host application and its add-ins is popularly known as the pipeline communication.

A drawback of using MAF is that you cannot add new add-ins or components without modifying the source code of the original host application, which is a complex and tedious task. In addition, MAF requires setting up the complex add-in pipeline even in simple applications to facilitate communication between the host application and add-ins. To overcome this problem, the .NET Framework 4.0 releases a new framework called as Managed Extensibility Framework (MEF). MEF allows you to create lightweight and extensible applications. This framework provides an interface and the application programming interface (API) to work with add-ins. The interface provided by MEF allows you to implement an add-in; whereas, the API allows the add-in to interact with the host application. As the add-ins implemented in a MEF Framework do not communicate with each another, there is no need of setting the complex pipeline communication among them.

In this appendix, you learn about the MAF and MEF frameworks. The appendix starts with exploring MAF, where you learn to discover, activate, and add qualification data to add-ins. Then, you learn about MEF and its architecture, including contracts, exports, imports, catalog, and composition container.

Let's start by exploring MAF.

Exploring Managed AddIn Framework

MAF is a set of assemblies that are required to provide extensible applications or add-ins for a host application. For instance, an e-mail application can use various add-ins or pre-defined services, such as spell checking, currency converter, or virus-scanner. MAF enables the communication between a host application and add-ins through a pipeline. This communication pipeline, also called the add-in pipeline, is a series of segments that exchange data between an add-in and its host application.

Figure D.1 shows how a host application communicates with an add-in through a pipeline:

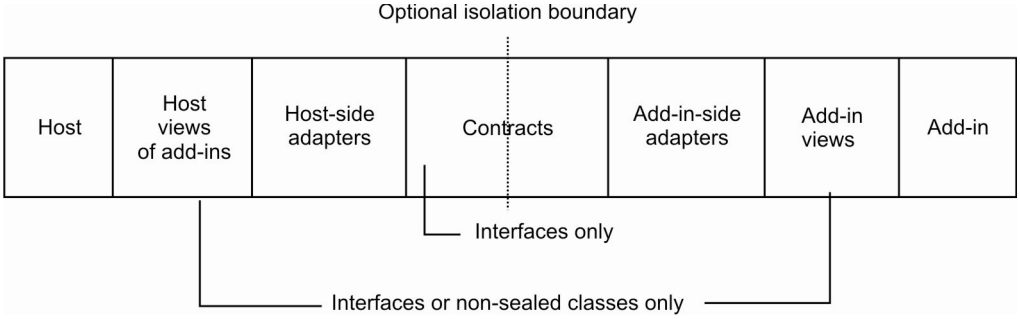

Figure D.1: Displaying Pipeline Communication between a Host Application and an Add-in

The various segments of a pipeline communication, shown in Figure D.1 are listed in Table D.1:

Table D.1: Segments Involved in a Pipeline Communication

Pipeline Segment	Description
Host	Refers to the application assembly that loads or creates instances of an add-in.
Host view of the add-in	Refers to the view of a host application, which is an abstract base class or interface that provides object types and methods to communicate with add-ins.
Host-side adapter	Refers to the segments that are used to convert a non- contract or non-serializable type to a contract or serializable type. This is because a type must be a contract or a serializable type, if it has to be passed through the isolation boundary.
Contract	Refers to a interface that contains virtual methods to exchange types between a host application and add-in.
Add-in-side adapter	Refers to the segments that are responsible to convert the types of an add-in application from a non- contract or non-serializable type to a contract or serializable type.
Add-in view	Refers to the view of an add-in application, which is an abstract base class or interface that provides object types and methods to communicate with the host application.
Add-in	Refers to an application that performs a service for a host application.

Let's understand how to handle add-ins in MAF in the following sections

- Exploring the required types for add-ins
- Discovering pipeline segments and add-ins
- Activating add-ins
- Adding qualification data to add-ins

Exploring the Required Types for Add-ins

To create add-in applications in MAF of .NET, you need to use classes and interfaces defined in the following namespaces:

- **System.AddIn**—Provides types to identify an object as an add-in
- **System.AddIn.Hosting**—Provides types to discover, register, activate, and control add-ins
- **System.AddIn.Pipeline**—Provides types to create pipeline communication between a host application and an add-in
- **System.AddIn.Contract**—Provides types to establish communication between two independent components, such as a host application and an add-in

Table D.2 describes the class provided by the System.AddIn namespace:

Table D.2: Class Contained in the System.AddIn Namespace	
Class	**Description**
AddInAttribute	Helps you to identify an object as an add-in

Table D.3 describes the classes provided by the System.AddIn.Hosting namespace:

Table D.3: Classes Contained in the System.AddIn.Hosting Namespace	
Class	**Description**
AddInController	Enables you to access an add-in to perform a variety of tasks, such as to shut down an add-in or to mark an add-in for future activation
AddInEnvironment	Enables you to access the application domain and the add-ins running in it
AddInProcess	Provides an external process for running add-ins besides a host application
AddInSegmentDirectoryNotFound Exception	Raises an exception when a segment directory is missing from the pipeline directory structure
AddInStore	Contains the information of available add-ins and pipeline segments
AddInToken	Represents an add-in that can be activated by a host application
InvalidPipelineStoreException	Raises an exception when a directory is not found and the user does not have permission to access the pipeline root path or an add-in path

Table D.4 describes the structure provided by the System.AddIn.Hosting namespace:

Table D.4: Structure Contained in the System.AddIn.Hosting Namespace	
Structure	**Description**
QualificationDataItem	Represents the data of a pipeline segment that is used by a host application

Table D.5 describes the enumerations provided by the System.AddIn.Hosting namespace:

Table D.5: Enumerations Contained in the System.AddIn.Hosting Namespace	
Enumeration	**Description**
AddInSecurityLevel	Specifies a security trust level that is assigned to an application domain in which an add-in is loaded
AddInSegmentType	Specifies the type of a pipeline segment
PipelineStoreLocation	Specifies the location where the pipeline store is located
Platform	Specifies the bits-per-word that can be executed in an add-in

Table D.6 describes the classes provided by the System.AddIn.Pipeline namespace:

Table D.6: Classes Contained in the System.AddIn.Pipeline Namespace	
Class	**Description**
AddInAdapterAttribute	Represents an object as an add-in-side adapter segment of the pipeline
AddInBaseAttribute	Represents an object as an add-in view segment of the pipeline
AddInContractAttribute	Represents an object as a add-in contract segment of the pipeline
CollectionAdapters	Represents the adapters that are used to pass collections between a host application and add-ins
ContractAdapter	Represents the adapter that provides methods to access add-ins
ContractBase	Implements the members of the IContract interface

Table D.6: Classes Contained in the System.AddIn.Pipeline Namespace	
Class	**Description**
ContractHandle	Provides the methods to handle the lifetime of an add-in
FrameworkElementAdapters	Represents the adapters that are used by Windows Presentation Foundation User Interfaces (WPF UIs) between host applications and add-ins
HostAdapterAttribute	Represents an object as a host-side adapter segment of the pipeline
QualificationDataAttribute	Represents the data that is specified by an add-in

Table D.7 describes the structures provided by the System.AddIn.Contract namespace:

Table D.7: Structures Contained in the System.AddIn.Contract Namespace	
Structure	**Description**
RemoteArgument	Helps in creating the instances of a type that can be passed across a process or an application domain
SerializableObjectData	Allows you to serialize the information of an object

Table D.8 describes the interfaces provided by the System.AddIn.Contract namespace:

Table D.8: Interfaces Contained in the System.AddIn.Contract Namespace	
Interface	**Description**
IContract	Represents the base interface that is implemented by a contract, which is used for communication between two independent components
IEnumeratorContract<T>	Represents the interface that enumerates the elements in the IListContract<T> collection
IExecutorExtensionContract	Represents the interface that is implemented by a host application to extend add-in executors
IListContract<T>	Represents a generic list of types that is used by a contract for communication between a host application and an add-in
INativeHandleContract	Represents a contract that controls a window using native code
IProfferServiceContract	Represents a contract for the components that use custom service
ISerializableObjectContract	Represents a contract that provides the information of a serializable object
IServiceProviderContract	Represents a contract that provides a service to a component

Table D.9 describes the enumeration provided by the System.AddIn.Contract namespace:

Table D.9: Enumeration Contained in the System.AddIn.Contract Namespace	
Enumeration	**Description**
RemoteArgumentKind	Specifies the type of argument taken by an RemoteArgument object

Discovering Pipeline Segments and Add-ins

In MAF, discovering is the process in which a host application searches for the appropriate pipeline segments and add-in services. .NET Framework creates a specific directory structure, known as the pipeline directory, to store all the pipeline segments and add-ins. The host application needs to access the pipeline directory structure to discover the required pipeline segments and add-ins.

Figure D.2 shows the pipeline directory structure:

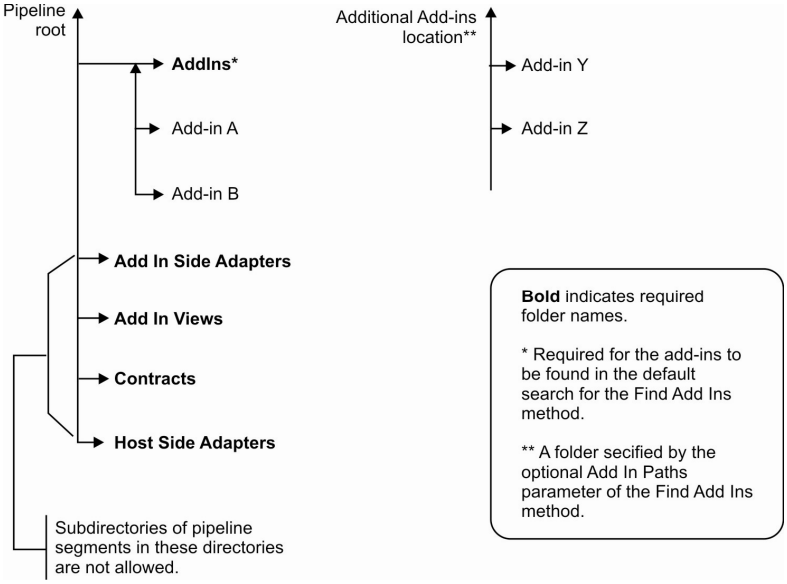

Figure D.2: Displaying the Pipeline Directory Structure

Table D.10 describes various types of directories, shown in Figure D.2, that are used to store pipeline segments and add-ins:

Table D.10: List of Directories Containing Add-ins

Directory	Description
Pipeline root	Contains the subdirectories of pipeline segments. You can use this directory by specifying the location as ..\Pipeline.
AddIns	Contains one or more subdirectories to store add-ins. The name of this directory must be specified as AddIns. You can use this directory by specifying the location as ..\Pipeline\AddIns.
AddInSideAdapters	Stores the add-in-side adapters assembly. The name of this directory must be specified as AddInSideAdapters. You can use this directory by specifying the location as ..\Pipeline\AddInSideAdapters.
AddInViews	Stores the add-in views assembly. The name of this directory must be specified as AddInViews. You can use this directory by specifying the location as ..\Pipeline\AddInViews.
Contracts	Stores the contracts assembly. The name of this directory must be specified as Contracts. You can use this directory by specifying the location as ..\Pipeline\Contracts.
HostSideAdapters	Stores the host-side adapters assembly. The name of this directory must be specified as HostSideAdapters. You can use this directory by specifying the location as ..\Pipeline\HostSideAdapters.

A host application can discover add-ins by performing the following tasks:

- Registering the information of add-ins and their pipeline segments in cache files
- Searching add-ins in the cache for a specific host view

Registering Add-ins and their Pipeline Segments in Cache Files

The cache files, AddIns.store and PipelineSegments.store, are used to register the information of valid add-ins and their pipelines segments, respectively. The AddIns.store file is stored in the directory containing one or more

add-in subdirectories. This directory can exist either in the pipeline directory structure or at some other location. The PipelineSegments.store file is stored in the root directory of the pipeline directory structure.

Table D.11 describes the methods of the AddInStore class that are required to register add-ins and their pipeline segments in cache:

Table D.11: Required Methods to Register Add-ins and their Pipeline Segments:

Registration method	Description
Rebuild()	Allows you to rebuild the pipeline segments cache whenever any new segment is added in the pipeline. This method also rebuilds the add-ins cache, when add-ins exist in the pipeline directory structure.
RebuildAddIns()	Allows you to rebuild the add-ins cache, when add-ins exist outside the pipeline directory structure.
Update()	Allows you to update the pipeline segments cache for any new addition of pipeline segment. This method also updates the add-ins cache, when add-ins exist in the pipeline directory structure. Note that if there are no new segments or add-ins, the Update() method validates only the cache.
UpdateAddIns()	Refers to a method that updates the add-ins cache, when add-ins exist outside the pipeline directory structure. Note that if there are no new add-ins, the UpdateAddIns() method validates only the cache.

NOTE

The Rebuild() and Update() methods are overloaded and can take a parameter of either the root directory of the pipeline directory structure or a value of the PipelineStoreLocation enumeration.

Searching Add-ins in the Cache for a Specified Host View

The FindAddIns() method is used to search an add-in that matches with a specified host view of the add-in. This method takes a parameter of either the root directory of the pipeline directory structure or a value of the PipelineStoreLocation enumeration to find the files that are created in cache by registration methods. The FindAddIns() method returns the objects of the IList<T> class. These objects provide the information about the add-ins and the pipeline segments associated with them. Each object of the IList<T> class is described with the help of the AddInToken class and identified as an add-in by using the AddInAttribute class. Each add-in must provide its basic information, which includes its name, description, publisher, and version.

The following code snippet shows how to build the cache files and find add-ins from them:

```
    String pipeRoot = Environment.CurrentDirectory;
    string[] warnings = AddInStore.Update(pipeRoot);
    foreach (string warning in warnings)
    {
        Console.WriteLine(warning);
    }
    Collection<AddInToken> tokens =

AddInStore.FindAddIns(typeof(Converter),PipelineStoreLocation.ApplicationBase);
```

In the preceding code snippet, it is assumed that the path of an add-in is the pipeline directory structure root directory. The cache files are updated by using the Update() method for any new addition of add-ins or pipeline segments. Then, the add-ins of the Converter type are searched by using the FindAddIns() method.

Activating Add-ins

After discovering the add-ins from cache, a host application can activate them by using the objects of the AddInToken class. The Activate() method of the AddInToken class is used to activate an add-in by specifying the following options:

- An application domain in which an add-in has to be loaded

- The security trust level or permission set that needs to be applied to the application domain in which an add-in has to be loaded
- The external process in which an add-in has to be activated

The Activate() method returns the host view of an add-in, and then the host application can invoke the methods defined in the contract. This method is overloaded and can take a parameter to specify a security level or to create an application domain for the add-in. When the Activate() method creates an application domain for an add-in, the related settings for the application domain are stored in the addinassemblyname.config file. You should note that the application domain of an add-in can be same or different from a host application. In addition, a single application domain can have host application along with one or more add-ins.

NOTE

In general, a host application and an add-in application run in separate application domains. In other words, add-ins operate in isolated contexts from the host application as well as from other add-ins. This isolation prevents conflicts among add-ins.

A host application can also control the life of an add-in by using the AddInController class. For instance, a host application can set the shut down time of an add-in or can specify the add-ins that can be activated in future. A host application also allows the garbage collector to claim for an add-in which is no longer in use.

The following code snippet shows how to activate an add-in:

```
AddInToken selectedAddin = ChooseAddIn(tokens);
Converter CnvtAddIn = selectedAddin.Activate<Converter>(AddInSecurityLevel.Internet);
RunConverter(CnvtAddIn);
```

In the preceding code snippet, an add-in is activated with a specified security level in an automatically created application domain. The add-in is activated by using the AddInToken object in a new application domain with the Internet trust level. Finally, the add-in is run by using a user defined method, RunConverter.

Adding Qualification Data to Add-ins

The data that helps to differentiate an add-in from other add-ins is termed as the qualification data. This includes general information about an add-in, such as the name of the add-in and its version. The following code snippet shows how to assign qualification data to an add-in:

```
[AddIn("Converter Add-in",Version="2.0.0.0")]
[QualificationData("Isolation", "NewAppDomain")]
public class MyAddIn : Converter2
{
....
....
....
}
```

Let's now discuss about MEF in detail.

Exploring Managed Extensibility Framework

In .NET Framework 4.0, MEF is a library that helps you to create lightweight and extensible applications. It allows you to discover and use extensible applications without modifying any configuration. You can use MEF in client applications based on Windows Forms or WPF as well as in server applications that use ASP.NET. MEF is basically an extensible framework that is used to compose applications from a set of loosely-coupled parts discovered and evolved at runtime. It also helps to develop reusable applications from reusable components that can be dynamically discovered at runtime by the application itself.

Unlike MAF, in MEF, you do not need to explicitly register a component on a host application. However, MEF allows you to implicitly discover a component by the host application.

Figure D.3 shows the architecture of MEF:

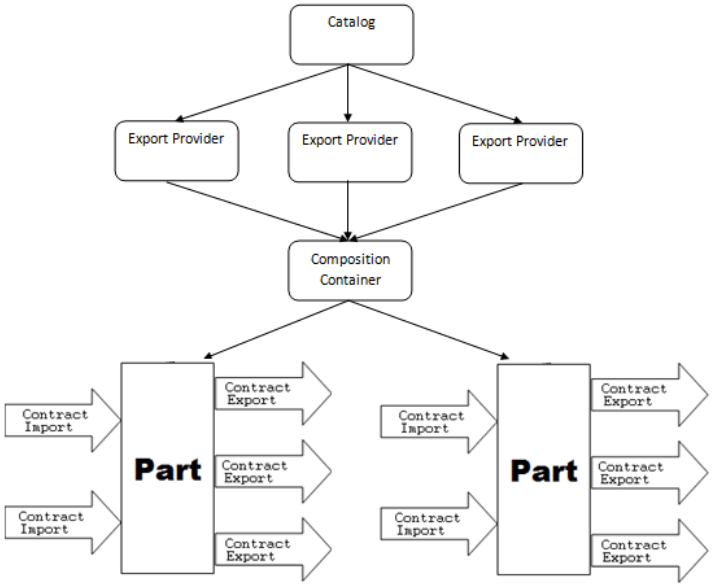

Figure D.3: Displaying the Architecture of MEF

In Figure D.3, you can see that the architecture of MEF depends on various components, which includes composable parts, exports, imports, contracts, catalog, and composition container. A composable part, also called a part, has to manage an object instance of a given type, and depends on some dependencies and capabilities. These dependencies and capabilities are called as imports and exports, respectively. The composition container can be defined as a container that composes parts by using their exports and imports together. It interacts with catalogs to provide composable parts that act as extensions for an application.

Let's now learn about the components of MEF in the following sections:

- Composable parts and contracts
- Exports and imports
- Catalogs, export providers, and composition container

Composable Parts and Contracts

A composable part is required to export or import services with other composable parts. The System.ComponentModel.Composition.Import and System.ComponentModel.Composition.Export attributes are required to import and export the services of the composable parts, respectively. Contract is a random string, which is used by a composable part to interact between an import and an export. Every export contains a contract; whereas, every import declares the required contract.

Exports and Imports

Exports are services provided by a composable part. Therefore, when a part provides a service, it is said that the part exports it. Note that a composable part can have one or more exports. In MEF, exports are declared by using the [System.ComponentModel.Composition.ExportAttribute] attribute. There are four types of exports of a composable part, which are given as follows:

- **ComposablePart**—Refers to an export, in which a composable part is exported itself by using the [System.ComponentModel.Composition.ExportAttribute] attribute.
- **Property**—Refers to an export, in which the properties, for instance data types, of a composable part are exported.

- **Method**—Refers to an export, in which the methods of a composable part are exported. Methods are exported as delegates that are specified in an export contract.
- **Inherited**—Refers to an export in which exports of base classes or interfaces are exported by using the System.ComponentModel.Composition.InheritedExportAttribute attribute.

Imports refer to the services that are used or consumed by a composable part. This means that whenever a service is consumed by a part, it is said that the part has imported it. Note that a composable part can import a single service at a particular instance of time. In MEF, imports are declared by using the [System.ComponentModel.Composition.ImportAttribute] attribute. There are three types of imports of a composable part, which are given as follows:

- **Property**—Refers to an import in which the properties values of a composable part are imported
- **Constructor parameter**—Refers to an import in which constructor's arguments are imported by using the [System.ComponentModel.Composition.ImportingConstructorAttribute] attribute
- **Field**—Refers to an import in which the values of contracts are imported directly to fields

Catalogs, Export Providers and Composition Container

In MEF, catalog refers to an object that helps you to discover the required composable parts from different resources. The following is the list of various types of catalogs:

- **Assembly catalog**—Discovers exports in the assembly of a composable part.
- **Directory catalog**—Discovers exports in all the assemblies of a directory.
- **Aggregate catalog**—Refers to the aggregation of more than one catalog. If the Assembly catalog and the Directory catalog are not able to meet the requirements individually, then a combination of catalogs is implemented in the application, which is known as the Aggregate catalog.
- **Type catalog**—Discovers exports in a particular set of types.

In MEF, export providers retrieve the exports from catalog and pass these exports to a composition container. This composition container is used to match an import to its corresponding export. You can create a composition container by using the CompositionContainer class. The CompositionContainer class performs composition in parts and also uses catalogs to find out which parts are available for extensions. The following code snippet shows how to declare an object of the CompositionContainer class:

```
    private CompositionContainer _container;
```

The preceding code snippet shows the declaration of the _container object of the CompositionContainer class.

With this, we have come to the end of this appendix.