

50

Managed Add-In Framework

WHAT'S IN THIS CHAPTER?

- Architecture of the Managed Add-In Framework
- Defining a Contract
- Implementing a Pipeline
- Creating Add-Ins
- Hosting Add-Ins

This chapter describes in detail the Managed Add-In Framework (MAF) architecture and how to create add-ins and host add-ins using MAF. You can read about the architecture of MAF and what issues it solves by hosting add-ins, such as versioning, discovery, activation, and isolation. Then we discuss all the steps that are necessary to create the MAF pipeline that is used to connect the add-in with the host; how to create add-ins; and how to create a hosting application that uses add-ins.

MAF ARCHITECTURE

.NET 4 includes two technologies for creating and using add-ins. Probably you've already read Chapter 28, "Managed Extensibility Framework," where you saw how to create add-ins with the Managed Extensibility Framework (MEF). Applications using MEF can find add-ins during runtime from a directory or from within assemblies, and connect them using attributes. The difference from MAF is that MEF does not offer a separation of the add-in from the hosting application by using appdomains or different processes. This is where the MAF comes into play. However, to reach this, MAF adds higher complexity. If you would like to get the advantages of both MEF and MAF, you can combine these two technologies. Of course this also adds complexity.

When you create an application that allows you to add add-ins during runtime, you will need to deal with certain issues — for example, how to find the add-ins, and how to solve versioning issues so that the hosting application and the add-in can progress independently. In this section, you read about the issues of add-ins and how the architecture of MAF resolves them.

When you create a hosted application that dynamically loads assemblies that are added at a later time, there are several issues that must be dealt with, as shown in the table that follows.

ADD-INS ISSUES	DESCRIPTION
Discovery	How can new add-ins be found for the hosting application? There are several options. One way is to add information about add-ins to a configuration file. This has the disadvantage that the installation of new add-ins necessitates the changing of an existing configuration file. Another option is to just copy the assembly containing the add-in to a predefined directory and read information about the assembly with reflection. You can read more about reflection in Chapter 14, "Reflection."
Activation	With assemblies that are dynamically loaded, it is not possible to just use the new operator to create an instance. You can create such assemblies with the <code>Activator</code> class. Also, different activation options might apply if the add-in is loaded within a different application domain or a new process. Assemblies and application domains are described in Chapter 18, "Assemblies."
Isolation	An add-in can break the hosting application, as you've probably already seen with Internet Explorer crashes caused by various add-ins. Depending on the type of hosting application and how the add-ins are integrated, the add-in can be loaded within a different application domain or within a different process.
Lifetime	Cleaning up objects is the job of the garbage collector. However, the garbage collector cannot help here because add-ins might be active in a different application domain or a different process. Other ways to keep the object in memory are with reference count or leasing and sponsoring mechanisms.
Versioning	Versioning is a big issue with add-ins. Usually it should be possible for a new version of the host to still load old add-ins, and an old host should have the option to load new add-ins.

Now let's look at the architecture of MAF and how this framework resolves these issues. The design of MAF was influenced by these goals:

- It should be easy to develop add-ins.
- Finding add-ins during runtime should be performant.
- Developing hosts should be an easy process as well, but may not be as easy as developing add-ins.
- The add-in and the host application should progress independently.

MAF solves the issues using a pipeline that uses contracts in its center. We will discuss how MAF deals with discovery to find add-ins; how add-ins are activated; how they can be kept alive; and how you can do versioning.

Pipeline

The MAF architecture is based on a pipeline of seven assemblies. This pipeline resolves the versioning issues with add-ins. Because the assemblies from the pipeline have a very light dependency, it is possible that the contract, the hosting, and the add-in applications progress with new versions completely independent of one another.

Figure 50-1 shows the pipeline of the MAF architecture. In the center is the contract assembly. This assembly contains a contract interface that lists methods and properties that must be implemented by the add-in and can be called by the host. The left side of the contract is the host side, and the right is the add-in side. In the figure, you can see the dependencies between the assemblies. The host assembly, shown leftmost, does not have a real dependency to the contract assembly; the same is true of the add-in assembly. Both do not really implement the interface that is defined by the contract. Instead, they just have a reference to a view assembly. The host application references the host view; the add-in references the add-in view. The views contain abstract view classes that define methods and properties as defined by the contract.

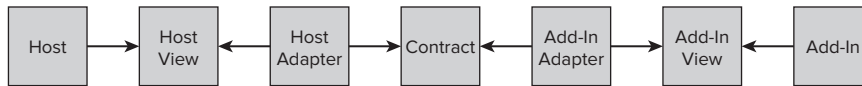


FIGURE 50-1

Figure 50-2 shows the relationship of the classes from the pipeline. The host class has an association with the abstract host view class and invokes its methods. The abstract host view class is implemented by the host adapter. Adapters make the connection between the views and the contract. The add-in adapter implements the methods and properties of the contract. This adapter contains a reference to the add-in view and forwards calls from the host side to the add-in view. The host adapter class defines a concrete class that derives from the abstract base class of the host view to implement the methods and properties. This adapter includes a reference to the contract to forward calls from the view to the contract.

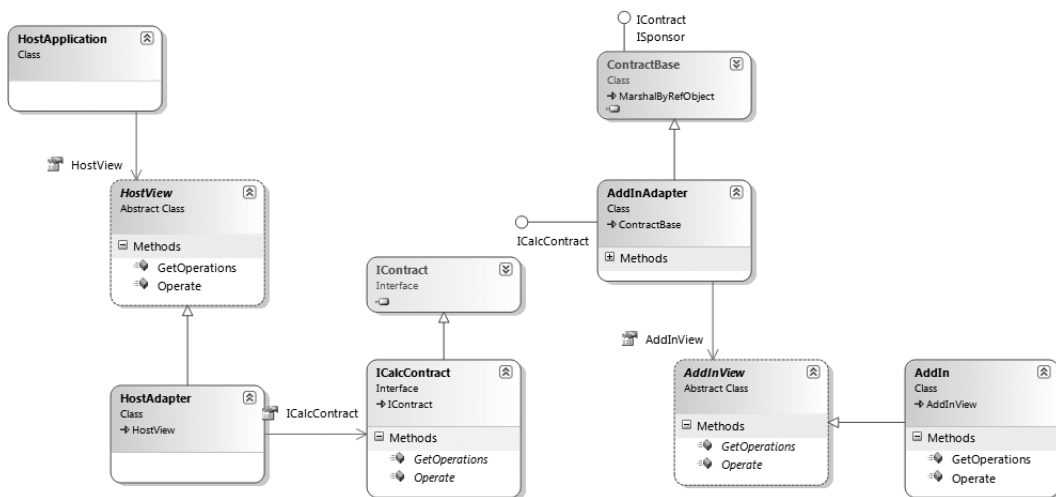


FIGURE 50-2

With this model it is possible that the add-in side and the host side emerge completely independent of each other. Just the mapping layer needs to adapt. For example, if a new version of the host is done that uses completely new methods and properties, the contract can still stay the same and only the adapter needs to change. It is also possible that a new contract is defined. Adapters can change, or several contracts can be used in parallel.

Discovery

How can new add-ins be found for the hosting application? The MAF architecture uses a predefined directory structure to find add-ins and the other assemblies of the pipeline. The components of the pipeline must be stored in these subdirectories:

- HostSideAdapters
- Contracts
- AddInSideAdapters
- AddInViews
- AddIns

All these directories with the exception of the AddIns directory directly contain the assembly of the specific part of the pipeline. The AddIns directory contains subdirectories for every add-in assembly. You can also store add-ins in directories that are completely independent of the other pipeline components.

The assemblies of the pipeline are not just loaded dynamically to get all the information about the add-in using reflection. With many add-ins, this would increase the startup time of the hosting application. Instead, MAF uses a cache with information about the pipeline components. The cache is created by the program installing the add-in or by the hosting application if the hosting application has write access to the directory of the pipeline.

The cache information about the pipeline components is created by invoking methods of the `AddInStore` class. The method `Update()` finds new add-ins that are not already listed with the store files. The `Rebuild()` method rebuilds the complete binary store file with information about the add-ins.

The following table lists the members of the `AddInStore` class.

ADDINSTORE MEMBERS	DESCRIPTION
<code>Rebuild()</code> <code>RebuildAddIns()</code>	The <code>Rebuild()</code> method rebuilds the cache for all components of the pipeline. If the add-ins are stored in a different directory, the method <code>RebuildAddIns()</code> can be used to rebuild the cache of the add-ins.
<code>Update()</code> <code>UpdateAddIns()</code>	While the <code>Rebuild()</code> method rebuilds the complete cache of the pipeline, the <code>Update</code> method just updates the cache with information about new pipeline components. The <code>UpdateAddIns()</code> method updates the cache of the add-ins only.
<code>FindAddIn()</code> <code>FindAddIns()</code>	These methods are used to find add-ins by using the cache. The method <code>FindAddIns()</code> returns a collection of all add-ins that match the host view. The <code>FindAddIn()</code> method returns a specific add-in.

Activation and Isolation

The `FindAddIns()` method of the `AddInStore` class returns a collection of `AddInToken` objects that represent an add-in. With the `AddInToken` class, you can access information about the add-in, such as its name, description, publisher, and version. You can activate the add-in by using the `Activate()` method. The following table lists properties and methods of the `AddInToken` class.

ADDINTOKEN MEMBERS	DESCRIPTION
<code>Name</code> <code>Publisher</code> <code>Version</code> <code>Description</code>	The <code>Name</code> , <code>Publisher</code> , <code>Version</code> , and <code>Description</code> properties of the <code>AddInToken</code> class return information about an add-in that was assigned to the add-in with the attribute <code>AddInAttribute</code> .
<code>AssemblyName</code>	<code>AssemblyName</code> returns the name of the assembly that contains the add-in.
<code>EnableDirectConnect</code>	With the property <code>EnableDirectConnect</code> , you can set a value to indicate that the host should directly connect to the add-in instead of using the components of the pipeline. This is only possible if the add-in and the host are running in the same application domain, and the types of the add-in view and the host view are the same. With this, it is still required that all components of the pipeline exist.
<code>QualificationData</code>	The add-in can mark appdomain and security requirements with the attribute <code>QualificationDataAttribute</code> . The add-in can list requirements for security and isolation requirements. For example, <code>[QualificationData("Isolation", "NewAppDomain")]</code> means that the add-in must be hosted in a new process. You can read this information from the <code>AddInToken</code> to activate the add-in with the specified requirements. In addition to appdomain and security requirements, you can use this attribute to pass custom information through the pipeline.
<code>Activate()</code>	The add-in is activated with the <code>Activate()</code> method. With parameters of this method, you can define if the add-in should be loaded inside a new application domain or a new process. You can also define what permissions the add-in gets.

One add-in can break the complete application. You may have seen Internet Explorer crash because of a failing add-in. Depending on the application type and the add-in type, you can avoid this by letting the add-in run within a different application domain or within a different process. MAF gives you several options here. You can activate the add-in in a new application domain or a new process. The new application domain might also have restricted permissions.

The `Activate()` method of the `AddInToken` class has several overloads where you can pass the environment into which the add-in should be loaded. The different options are listed in the following table.

PARAMETERS OF ADDINTOKEN.ACTIVATE()	DESCRIPTION
AppDomain	You can pass a new application domain into which the add-in should be loaded. This way, you can make it independent of the host application, and it can also be unloaded with the application domain.
AddInSecurityLevel	If the add-in should run with different security levels, you can pass a value of the <code>AddInSecurityLevel</code> enumeration. Possible values are <code>Internet</code> , <code>Intranet</code> , <code>FullTrust</code> , and <code>Host</code> .
PermissionSet	If the predefined security levels are not specific enough, you can assign a <code>PermissionSet</code> to the appdomain of the add-in.
AddInProcess	Add-ins can also run within a different process from that of the hosting application. You can pass a new <code>AddInProcess</code> to the <code>Activate()</code> method. The new process can shut down if all add-ins are unloaded, or it can keep running. This is an option that can be set with the property <code>KeepAlive</code> .
AddInEnvironment	Passing an <code>AddInEnvironment</code> object is another option for defining the application domain where the add-in should be loaded. With the constructor of <code>AddInEnvironment</code> , you can pass an <code>AppDomain</code> object. You can also get an existing <code>AddInEnvironment</code> of an add-in with the <code>AddInEnvironment</code> property of the <code>AddInController</code> class.

 *Application domains are explained in Chapter 18, “Assemblies.”*

The type of application may restrict the choices you have. WPF add-ins currently do not support crossing processes. With Windows Forms, it is not possible to have Windows controls connected across different application domains.

Let’s get into the steps of the pipeline when the `Activate()` method of an `AddInToken` is invoked:

1. The application domain is created with the permissions specified.
2. The assembly of the add-in is loaded into the new application domain with the `Assembly.LoadFrom()` method.
3. The default constructor of the add-in is invoked by using reflection. Because the add-in derives from the base class that is defined with the add-in view, the assembly of the view is loaded as well.
4. An instance of the add-in side adapter is constructed. The instance of the add-in is passed to the constructor of the adapter, so the adapter can connect the contract to the add-in. The add-in adapter derives from the base class `MarshalByRefObject`, so it can be invoked across application domains.
5. The activation code returns a proxy to the add-in side adapter to the application domain of the hosting application. Because the add-in adapter implements the contract interface, the proxy contains methods and properties of the contract interface.

6. An instance of the host side adapter is constructed in the application domain of the hosting application. The proxy of the add-in side adapter is passed to the constructor. The activation finds the type of host-side adapter from the add-in token.

The host-side adapter is returned to the hosting application.

Contracts

Contracts define the boundary between the host side and the add-in side of the MAF architecture. Contracts are defined with an interface that must be derived from the base interface `IContract`. The contract should be well thought out in that it supports flexible add-in scenarios as needed.

Contracts are not versionable and may not be changed so that previous add-in implementations can still run in newer hosts. New versions are created by defining a new contract.

There's some restriction on the types that you can use with the contract. The restriction exists because of versioning issues and also because application domains are crossed from the hosting application to the add-in. The types need to be safe and versionable, and able to pass it across the boundaries (application domain or cross-process) to pass it between hosts and add-ins.

Possible types that can be passed with a contract are:

- Primitive types
- Other contracts
- Serializable system types
- Simple serializable custom types. These custom types can consist of primitive types or contracts and may not have an implementation.

The members of the `IContract` interface are explained in the following table.

ICONTRACT MEMBERS	DESCRIPTION
<code>QueryContract()</code>	With <code>QueryContract()</code> it is possible to query a contract to verify if another contract is implemented as well. An add-in can support several contracts.
<code>RemoteToString()</code>	The parameter of <code>QueryContract()</code> requires a string representation of the contract. <code>RemoteToString()</code> returns a string representation of the current contract.
<code>AcquireLifetimeToken()</code> <code>RevokeLifetimeToken()</code>	The client invokes <code>AcquireLifetimeToken()</code> to keep a reference to the contract. <code>AcquireLifetimeToken()</code> increments a reference count. <code>RevokeLifetimeToken()</code> decrements a reference count.
<code>RemoteEquals()</code>	<code>RemoteEquals()</code> can be used to compare two contract references.

Contract interfaces are defined in the namespaces `System.AddIn.Contract`, `System.AddIn.Contract.Collections`, and `System.AddIn.Contract.Automation`. The following table lists contract interfaces that you can use with a contract.

CONTRACT	DESCRIPTION
<code>IListContract<T></code>	The <code>IListContract<T></code> can be used to return a list of contracts.
<code>IEnumeratorContract<T></code>	<code>IEnumeratorContract<T></code> is used to enumerate the elements of a <code>IListContract<T></code> .
<code>IServiceProviderContract</code>	An add-in can offer services for other add-ins. Add-ins that offer services are known as service providers and implement the interface <code>IServiceProviderContract</code> . With the method <code>QueryService()</code> an add-in implementing this interface can be queried for services offered.

CONTRACT	DESCRIPTION
IProfferServiceContract	IProfferServiceContract is the interface offered by a service provider in conjunction with IServiceProviderContract. IProfferServiceContract defines the methods ProfferService() and RevokeService(). ProfferService() adds an IServiceProviderContract to the services offered, RevokeService() removes it.
INativeHandleContract	This interface provides access to native window handles with the GetHandle() method. This contract is used with WPF hosts to use WPF add-ins.

Lifetime

How long does an add-in need to be loaded? How long is it used? When is it possible to unload the application domain? There are several options to resolve this. One option is to use reference counts. Every use of the add-in increments the reference count. If the reference count decrements to zero, the add-in can be unloaded. Another option is to use the garbage collector. If the garbage collector runs, and there's no more reference to an object, the object is the target of garbage collection. .NET Remoting is using a leasing mechanism and a sponsor to keep objects alive. As soon as the leasing time ends, sponsors are asked if the object should stay alive.

 *Application domains are explained in Chapter 18, “Assemblies.”*

With add-ins, there's a specific issue for unloading add-ins because they can run in different application domains and also in different processes. The garbage collector cannot work across different processes. MAF is using a mixed model for lifetime management. Within a single application domain, garbage collection is used. Within the pipeline, an implicit sponsorship is used, but reference counting is available from the outside to control the sponsor.

Let's consider a scenario where the add-in is loaded into a different application domain. Within the host application, the garbage collector cleans up the host view and the host side adapter when the reference is not needed anymore. For the add-in side, the contract defines the methods AcquireLifetimeToken() and RevokeLifetimeToken() to increment and decrement the reference count of the sponsor. These methods do not just increment and decrement a value, which could lead to the release of an object too early if one party called the revoke method too often. Instead, AcquireLifetimeToken() returns an identifier for the lifetime token, and this identifier must be used to invoke the RevokeLifetimeToken() method. So these methods are always called in pairs.

Usually you do not have to deal with invoking the AcquireLifetimeToken() and RevokeLifetimeToken() methods. Instead, you can use the ContractHandle class, which invokes AcquireLifetimeToken() in the constructor and RevokeLifetimeToken() in the finalizer.

 *The finalizer is explained in Chapter 13, “Memory Management and Pointers.”*

In scenarios where the add-in is loaded in a new application domain, it is possible to get rid of the loaded code when the add-in is not needed anymore. MAF uses a simple model to designate one add-in as the owner of the application domain to unload the application domain if the add-in is not needed anymore. An add-in is the owner of the application domain if the application domain is created when the add-in is activated. The application domain is not unloaded automatically if it was created previously.

The class `ContractHandle` is used in the host-side adapter to add a reference count to the add-in. The members of this class are explained in the following table.

CONTRACTHANDLE MEMBERS	DESCRIPTION
<code>Contract</code>	In the construction of the <code>ContractHandle</code> class, an object implementing <code>IContract</code> can be assigned to keep a reference to it. The <code>Contract</code> property returns this object.
<code>Dispose()</code>	The <code>Dispose()</code> method can be called instead of waiting for the garbage collector to do the finalization to revoke the lifetime token.
<code>AppDomainOwner()</code>	<code>AppDomainOwner()</code> is a static method of the <code>ContractHandle</code> class that returns the add-in adapter if it owns the application domain that is passed with the method.
<code>ContractOwnsAppDomain()</code>	With the static method <code>ContractOwnsAppDomain()</code> , you can verify if the specified contract is an owner of the application domain. Thus, the application domain gets unloaded when the contract is disposed of.

Versioning

Versioning is a very big issue with add-ins. The host application is developed further, as are the add-ins. One requirement for an add-in is that it should be possible that a new version of the host application can still load old versions of add-ins. This should work in the other direction as well: older hosts should run newer versions of add-ins. But what if the contract changes?

`System.AddIn` is completely independent from the implementation of the host application and add-ins. This is done with a pipeline concept that consists of seven parts.

ADD-IN SAMPLE

Let's start a simple sample of a hosting application that can load calculator add-ins. The add-ins can support different calculation operations that are offered by other add-ins.

You need to create a solution with six library projects and one console application. The projects of the sample application are listed in the following table. The table lists the assemblies that need to be referenced. With the references to the other projects within the solution, you need to set the property `Copy Local` to `False`, so that the assembly does not get copied. One exception is the `HostApp` console project, which needs a reference to the `HostView` project. This assembly needs to be copied so that it can be found from the host application. Also, you need to change the output path of the generated assemblies so that the assemblies are copied to the correct directories of the pipeline.

PROJECT	REFERENCES	OUTPUT PATH	DESCRIPTION
<code>CalcContract</code>	<code>System.AddIn</code> , <code>Contract</code>	<code>.\Pipeline\Contracts\</code>	This assembly contains the contract for communication with the add-in. The contract is defined with an interface.
<code>CalcView</code>	<code>System.AddIn</code>	<code>.\Pipeline\AddInViews\</code>	The <code>CalcView</code> assembly contains an abstract class that is referenced by the add-in. This is the add-in side of the contract.
<code>CalcAddIn</code>	<code>System.AddIn</code> <code>CalcView</code>	<code>.\Pipeline\AddIns\CalcAddIn\</code>	<code>CalcAddIn</code> is the add-in project that references the add-in view assembly. This assembly contains the implementation of the add-in.

PROJECT	REFERENCES	OUTPUT PATH	DESCRIPTION
CalcAddInAdapter	System.AddIn System.AddIn. Contract CalcView CalcContract	.\Pipeline\ AddInSideAdapters\	CalcAddInAdapter connects the add-in view and the contract assembly and maps the contract to the add-in view.
HostView			The assembly containing the abstract class of the host view does not need to reference any add-in assembly and also does not have a reference to another project in the solution.
HostAdapter	System.AddIn System.AddIn. Contract HostView CalcContract	.\Pipeline\ HostSideAdapters\	The host adapter maps the host view to the contract. Thus, it needs to reference both of these projects.
HostApp	System.AddIn HostView		The hosting application activates the add-in.

Add-In Contract

Let's start by implementing the contract assembly. Contract assemblies contain a contract interface that defines the protocol for communication between the host and the add-in.

With the following code, you can see the contract defined for the calculator sample application. The application defines a contract with the methods `GetOperations()` and `Operate()`. `GetOperations()` returns a list of mathematical operations supported by the calculator add-in. An operation is defined by the interface `IOperationContract` that is a contract by itself. `IOperationContract` defines the read-only properties `Name` and `NumberOperands`.

The `Operate()` method invokes the operation within the add-in and requires an operation defined by the `IOperation` interface and the operands with a `double` array. With this contract it is possible that the add-in supports any operations that require any number of `double` operands and returns one `double`. The attribute `AddInContract` is used by the `AddInStore` to build the cache. The `AddInContract` attribute marks the class as an add-in contract interface.



The contract used here functions similarly to the contract for the add-ins in Chapter 28, "Managed Extensibility Framework." However, MEF contracts don't have the same requirements that MAF contracts have. MAF contracts are very restrictive because of the appdomain and process boundaries between the host and the add-in. However, the MEF architecture does not use different appdomains.



Available for
download on
Wrox.com

```
using System.AddIn.Contract;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    [AddInContract]
    public interface ICalculatorContract: IContract
    {
        IListContract<IOperationContract> GetOperations();
        double Operate(IOperationContract operation, double[] operands);
    }
}
```

```

public interface IOperationContract: IContract
{
    string Name { get; }
    int NumberOperands { get; }
}

```

code snippet CalcContract/ICalculatorContract.cs

Calculator Add-In View

The add-in view redefines the contract as it is seen by the add-in. The contract defined the interfaces `ICalculatorContract` and `IOperationContract`. For this, the add-in view defines the abstract class `Calculator` and the concrete class `Operation`.

With `Operation`, a specific implementation is not required by every add-in. Instead, the class is already implemented with the add-in view assembly. This class describes an operation for mathematical calculations with the `Name` and `NumberOperands` properties.

The abstract class `Calculator` defines the methods that need to be implemented by the add-ins. While the contract defines parameters and return types that need to be passed across appdomain and process boundaries, that's not the case with the add-in view. Here, you can use types, which make it easy to write add-ins for the add-in developer. The `GetOperations()` method returns `ICollection<Operation>` instead of `ICollection<IOperationContract>`, as you've seen with the contract assembly.

The `AddInBase` attribute identifies the class as an add-in view for the store.



```

using System.AddIn.Pipeline;
using System.Collections.Generic;
namespace Wrox.ProCSharp.MAF
{
    [AddInBase]
    public abstract class Calculator
    {
        public abstract ICollection<Operation> GetOperations();
        public abstract double Operate(Operation operation, double[] operand);
    }
    public class Operation
    {
        public string Name { get; set; }
        public int NumberOperands { get; set; }
    }
}

```

code snippet CalcView/CalculatorView.cs

Calculator Add-In Adapter

The add-in adapter maps the contract to the add-in view. This assembly has references to both the contract and the add-in view assemblies. The implementation of the adapter needs to map the method `ICollection<IOperationContract> GetOperations()` from the contract to the view method `ICollection<Operation> GetOperations()`.

The assembly includes the classes `OperationViewToContractAddInAdapter` and `CalculatorViewToContractAddInAdapter`. These classes implement the interfaces `IOperationContract` and `ICalculatorContract`. The methods of the base interface `IContract` can be implemented by deriving it from the base class `ContractBase`. This class offers a default implementation. `OperationViewToContractAddInAdapter` implements the other members of the `IOperationContract` interface and just forwards the calls to the `Operation` view that is assigned in the constructor.

The class `OperationViewToContractAddInAdapter` also contains static helper methods `ViewToContractAdapter()` and `ContractToViewAdapter()` that map `Operation` to `IOperationContract` and the other way around.



```
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    internal class OperationViewToContractAddInAdapter: ContractBase,
        IOperationContract
    {
        private Operation view;
        public OperationViewToContractAddInAdapter(Operation view)
        {
            this.view = view;
        }
        public string Name
        {
            get { return view.Name; }
        }
        public int NumberOperands
        {
            get { return view.NumberOperands; }
        }
        public static IOperationContract ViewToContractAdapter(Operation view)
        {
            return new OperationViewToContractAddInAdapter(view);
        }
        public static Operation ContractToViewAdapter(
            IOperationContract contract)
        {
            return (contract as OperationViewToContractAddInAdapter).view;
        }
    }
}
```

code snippet CalcAddInAdapter/OperationViewToContractAddInAdapter.cs

The class `CalculatorViewToContractAddInAdapter` is very similar to `OperationViewToContractAddInAdapter`: It derives from `ContractBase` to inherit a default implementation of the `IContract` interface, and it implements a contract interface. This time the `ICalculatorContract` interface is implemented with the `GetOperations()` and `Operate()` methods.

The `Operate()` method of the adapter invokes the `Operate()` method of the view class `Calculator`, where `IOperationContract` must be converted to `Operation`. This is done with the static helper method `ContractToViewAdapter()`, which is defined with the `OperationViewToContractAddInAdapter` class.

The implementation of the `GetOperations` method needs to convert the collection `ICollection<IOperationContract>` to `ICollection<Operation>`. For such collection conversions, the class `CollectionAdapters` defines the conversion methods `ToICollection()` and `ToICollectionContract()`. Here, the method `ToICollectionContract()` is used for the conversion.

The attribute `AddInAdapter` identifies the class as an add-in-side adapter for the add-in store.



```
using System.AddIn.Contract;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    [AddInAdapter]
    internal class CalculatorViewToContractAddInAdapter: ContractBase,
        ICalculatorContract
    {
        private Calculator view;
```

```

public CalculatorViewToContractAddInAdapter(Calculator view)
{
    this.view = view;
}
public IListContract<IOperationContract> GetOperations()
{
    return CollectionAdapters.ToIListContract<Operation,
        IOperationContract>(view.GetOperations(),
            OperationViewToContractAddInAdapter.ViewToContractAdapter,
            OperationViewToContractAddInAdapter.ContractToViewAdapter);
}
public double Operate(IOperationContract operation, double[] operands)
{
    return view.Operate(
        OperationViewToContractAddInAdapter.ContractToViewAdapter(
            operation), operands);
}
}
}

```

code snippet CalcAddInAdapter/CalculatorViewToContractAddInAdapter.cs



Because the adapter classes are invoked by .NET reflection, it is possible that the internal access modifier is used with these classes. As these classes are an implementation detail, it's a good idea to use the internal access modifier.

Calculator Add-In

The add-in now contains the implementation of the functionality. The add-in is implemented by the class `CalculatorV1`. The add-in assembly has a dependency on the add-in view assembly, as it needs to implement the abstract `Calculator` class.

The attribute `AddIn` marks the class as an add-in for the add-in store, and adds publisher, version, and description information. On the host side, this information can be accessed from the `AddInToken`.

`CalculatorV1` returns a list of supported operations in the method `GetOperations()`. `Operate()` calculates the operands based on the operation.



```

using System;
using System.AddIn;
using System.Collections.Generic;
namespace Wrox.ProCSharp.MAF
{
    [AddIn("CalculatorAddIn", Publisher="Wrox Press", Version="1.0.0.0",
        Description="Sample AddIn")]
    public class CalculatorV1: Calculator
    {
        private List<Operation> operations;
        public CalculatorV1()
        {
            operations = new List<Operation>();
            operations.Add(new Operation() { Name = "+", NumberOperands = 2 });
            operations.Add(new Operation() { Name = "-", NumberOperands = 2 });
            operations.Add(new Operation() { Name = "/", NumberOperands = 2 });
            operations.Add(new Operation() { Name = "**", NumberOperands = 2 });
        }
        public override IList<Operation> GetOperations()
        {
            return operations;
        }
    }
}

```

```

    }
    public override double Operate(Operation operation, double[] operand)
    {
        switch (operation.Name)
        {
            case "+":
                return operand[0] + operand[1];
            case "-":
                return operand[0] - operand[1];
            case "/":
                return operand[0] / operand[1];
            case "*":
                return operand[0] * operand[1];
            default:
                throw new InvalidOperationException(
                    String.Format("invalid operation {0}", operation.Name));
        }
    }
}

```

code snippet CalcAddIn/Calculator.cs

Calculator Host View

Let's continue with the host view of the host side. Like the add-in view, the host view defines an abstract class with methods similar to the contract. However, the methods defined here are invoked by the host application.

Both the classes `Calculator` and `Operation` are abstract, as the members are implemented by the host adapter. The classes here just need to define the interface to be used by the host application.



```

using System.Collections.Generic;
namespace Wrox.ProCSharp.MAF
{
    public abstract class Calculator
    {
        public abstract IList<Operation> GetOperations();
        public abstract double Operate(Operation operation,
            params double[] operand);
    }
    public abstract class Operation
    {
        public abstract string Name { get; }
        public abstract int NumberOperands { get; }
    }
}

```

code snippet HostView/CalculatorHostView.cs

Calculator Host Adapter

The host adapter assembly references the host view and the contract to map the view to the contract. The class `OperationContractToViewHostAdapter` implements the members of the abstract `Operation` class. The class `CalculatorContractToViewHostAdapter` implements the members of the abstract `Calculator` class.

With `OperationContractToViewHostAdapter`, the reference to the contract is assigned in the constructor. The adapter class also contains a `ContractHandle` instance that adds a lifetime reference to the contract, so that the add-in stays loaded as long it is needed by the hosting application.



```
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    internal class OperationContractToViewHostAdapter: Operation
    {
        private ContractHandle handle;
        public IOperationContract Contract { get; private set; }
        public OperationContractToViewHostAdapter(IOperationContract contract)
        {
            this.Contract = contract;
            handle = new ContractHandle(contract);
        }
        public override string Name
        {
            get
            {
                return Contract.Name;
            }
        }
        public override int NumberOperands
        {
            get
            {
                return Contract.NumberOperands;
            }
        }
    }
}
internal static class OperationHostAdapters
{
    internal static IOperationContract ViewToContractAdapter(Operation view)
    {
        return ((OperationContractToViewHostAdapter)view).Contract;
    }
    internal static Operation ContractToViewAdapter(
        IOperationContract contract)
    {
        return new OperationContractToViewHostAdapter(contract);
    }
}
```

code snippet HostAdapter/OperationContractToViewHostAdapter.cs .

The class `CalculatorContractToViewHostAdapter` implements the methods of the abstract host view `Calculator` class and forwards the call to the contract. Again, you can see the `ContractHandle` holding the reference to the contract, which is similar to the adapter from the add-in-side type conversions. This time the type conversions are just in the other direction from the add-in adapters.

The attribute `HostAdapter` marks the class as an adapter that needs to be installed in the `HostSideAdapters` directory.



```
using System.Collections.Generic;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    [HostAdapter]
    internal class CalculatorContractToViewHostAdapter: Calculator
    {
        private ICalculatorContract contract;
        private ContractHandle handle;
        public CalculatorContractToViewHostAdapter(ICalculatorContract contract)
        {
            this.contract = contract;
        }
    }
}
```

```

        handle = new ContractHandle(contract);
    }
    public override IList<Operation> GetOperations()
    {
        return CollectionAdapters.ToIList<IOperationContract, Operation>(
            contract.GetOperations(),
            OperationHostAdapters.ContractToViewAdapter,
            OperationHostAdapters.ViewToContractAdapter);
    }
    public override double Operate(Operation operation, double[] operands)
    {
        return contract.Operate(OperationHostAdapters.ViewToContractAdapter(
            operation), operands);
    }
}
}

```

code snippet HostAdapter/CalculatorContractToViewHostAdapter.cs

Calculator Host

The sample host application uses the WPF technology. You can see the user interface of this application in Figure 50-3. On top is the list of available add-ins. On the left, the operations of the active add-in are shown. As you select the operation that should be invoked, operands are shown. After entering the values for the operands, the operation of the add-in can be invoked.

The buttons on the bottom row are used to rebuild and update the add-in store, and to exit the application.

The XAML code that follows shows the tree of the user interface. With the `ListBox` elements, different styles with item templates are used to give a specific representation of the list of add-ins, the list of operations, and the list of operands.

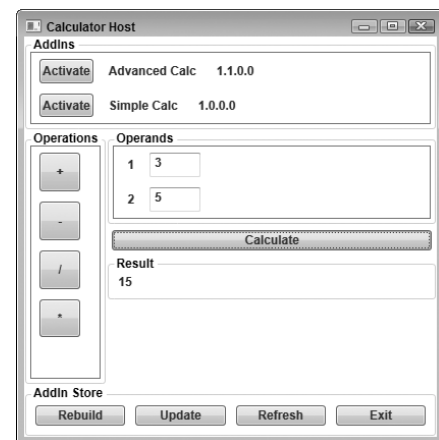


FIGURE 50-3



You can read information about item templates in Chapter 35, “Core WPF.”



```

<DockPanel>
    <GroupBox Header="AddIn Store" DockPanel.Dock="Bottom">
        <UniformGrid Columns="4">
            <Button x:Name="rebuildStore" Click="RebuildStore"
                Margin="5">Rebuild</Button>
            <Button x:Name="updateStore" Click="UpdateStore"
                Margin="5">Update</Button>
            <Button x:Name="refresh" Click="RefreshAddIns"
                Margin="5">Refresh</Button>
            <Button x:Name="exit" Click="App_Exit" Margin="5">Exit</Button>
        </UniformGrid>
    </GroupBox>
    <GroupBox Header="AddIns" DockPanel.Dock="Top">
        <ListBox x:Name="listAddIns" ItemsSource="{Binding}"
            Style="{StaticResource listAddInsStyle}" />
    </GroupBox>
    <GroupBox DockPanel.Dock="Left" Header="Operations">
        <ListBox x:Name="listOperations" ItemsSource="{Binding}"
            Style="{StaticResource listOperationsStyle}" />
    </GroupBox>
</DockPanel>

```

```

<StackPanel DockPanel.Dock="Right" Orientation="Vertical">
    <GroupBox Header="Operands">
        <ListBox x:Name="listOperands" ItemsSource="{Binding}"
            Style="{StaticResource listOperandsStyle}">
        </ListBox>
    </GroupBox>
    <Button x:Name="buttonCalculate" Click="Calculate" IsEnabled="False"
        Margin="5">Calculate</Button>
    <GroupBox DockPanel.Dock="Bottom" Header="Result">
        <Label x:Name="labelResult" />
    </GroupBox>
</StackPanel>
</DockPanel>

```

code snippet HostAppWPF/CalculatorHostWindow.xaml

In the code-behind, the `FindAddIns()` method is invoked in the constructor of the Window. `FindAddIns()` uses the `AddInStore` class to get a collection of `AddInToken` objects and pass them to the `DataContext` property of the `ListBox listAddIns` for display. The first parameter of the `AddInStore.FindAddIns()` method passes the abstract `Calculator` class, which is defined by the host view to find all add-ins from the store that apply to the contract. The second parameter passes the directory of the pipeline that is read from the application configuration file. When you run the sample application from the Wrox download site (<http://www.wrox.com>), you have to change the directory in the application configuration file to match your directory structure.



```

using System;
using System.AddIn.Hosting;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Windows;
using Wrox.ProCSharp.MAF.Properties;
namespace Wrox.ProCSharp.MAF
{
    public partial class CalculatorHostWindow: Window
    {
        private Calculator activeAddIn = null;
        private Operation currentOperation = null;
        public CalculatorHostWindow()
        {
            InitializeComponent();
            FindAddIns();
        }
        void FindAddIns()
        {
            try
            {
                this.listAddIns.DataContext =
                    AddInStore.FindAddIns(typeof(Calculator),
                        Settings.Default.PipelinePath);
            }
            catch (DirectoryNotFoundException ex)
            {
                MessageBox.Show("Verify the pipeline directory in the " +
                    "config file");
                Application.Current.Shutdown();
            }
        }
        //...
    }
}

```

code snippet HostAppWPF/CalculatorHostWindow.xaml.cs

To update the cache of the add-in store, the `UpdateStore()` and `RebuildStore()` methods are mapped to the `Click` events of the `Update` and `Rebuild` buttons. Within the implementation of these methods, the `Rebuild()` or `Update()` methods of the `AddInStore` class are used. These methods return a string array of warnings if assemblies are stored in the wrong directories. Because of the complexity of the pipeline structure, there's a good chance that the first time you may not get the project configuration completely right for copying the assemblies to the correct directories. Reading the returned information from these methods, you will get a clear explanation about what's wrong. For example, the message "No usable `AddInAdapter` parts could be found in assembly `Pipeline\AddInSideAdapters\CalcView.dll`" gives you a hint that the assembly `CalcView` is stored in the wrong directory.

```
private void UpdateStore(object sender, RoutedEventArgs e)
{
    string[] messages = AddInStore.Update(Settings.Default.PipelinePath);
    if (messages.Length != 0)
    {
        MessageBox.Show(string.Join("\n", messages),
            "AddInStore Warnings", MessageBoxButton.OK,
            MessageBoxImage.Warning);
    }
}

private void RebuildStore(object sender, RoutedEventArgs e)
{
    string[] messages =
        AddInStore.Rebuild(Settings.Default.PipelinePath);
    if (messages.Length != 0)
    {
        MessageBox.Show(string.Join("\n", messages),
            "AddInStore Warnings", MessageBoxButton.OK,
            MessageBoxImage.Warning);
    }
}
```

In Figure 50-3, you can see an `Activate` button beside the available add-in. Clicking this button invokes the handler method `ActivateAddIn()`. With this implementation, the add-in is activated by using the `Activate()` method of the `AddInToken` class. Here, the add-in is loaded inside a new process that is created with the `AddInProcess` class. This class starts the process `AddInProcess32.exe`. Setting the `KeepAlive` property of the process to `false`, the process is stopped as soon as the last add-in reference is garbage collected. The parameter `AddInSecurityLevel.Internet` leads to an add-in running with restricted permissions. The last statement of `ActivateAddIn()` invokes the `ListOperations()` method, which in turn invokes the `GetOperations()` method of the add-in. `GetOperations()` assigns the returned list to the data context of the `ListBox listOperations` for displaying all operations.

```
private void ActivateAddIn(object sender, RoutedEventArgs e)
{
    FrameworkElement el = sender as FrameworkElement;
    Trace.Assert(el != null, "ActivateAddIn invoked from the wrong " +
        "control type");

    AddInToken addIn = el.Tag as AddInToken;
    Trace.Assert(el.Tag != null, String.Format(
        "An AddInToken must be assigned to the Tag property " +
        "of the control {0}", el.Name);
    AddInProcess process = new AddInProcess();
    process.KeepAlive = false;

    activeAddIn = addIn.Activate<Calculator>(process,
        AddInSecurityLevel.Internet);
    ListOperations();
}
```

```

void ListOperations()
{
    this.listOperations.DataContext = activeAddIn.GetOperations();
}

```

After the add-in is activated and the list of operations is displayed in the UI, the user can select an operation. The Click event of the Button shown in the Operations category is assigned to the handler method `OperationSelected()`. In the implementation, the `Operation` object that is assigned to the `Tag` property of the Button is retrieved to get the number of operands needed with the operation. To allow the user to add values to the operands, an array of `OperandUI` objects is bound to the `ListBox` `listOperands`.

```

private void OperationSelected(object sender, RoutedEventArgs e)
{
    FrameworkElement el = sender as FrameworkElement;
    Trace.Assert(el != null, "OperationSelected invoked from " +
        "the wrong control type");

    Operation op = el.Tag as Operation;
    Trace.Assert(el.Tag != null, String.Format(
        "An AddInToken must be assigned to the Tag property " +
        "of the control {0}", el.Name);
    currentOperation = op;
    ListOperands(new double[op.NumberOperands]);
}
private class OperandUI
{
    public int Index { get; set; }
    public double Value { get; set; }
}
void ListOperands(double[] operands)
{
    this.listOperands.DataContext =
        operands.Select((operand, index) =>
            new OperandUI()
            { Index = index + 1, Value = operand }).ToArray();
}

```

The `Calculate()` method is invoked with the Click event of the Calculate button. Here, the operands are retrieved from the UI, the operation and operands are passed to the `Operate()` method of the add-in, and the result is shown with the content of a label.

```

private void Calculate(object sender, RoutedEventArgs e)
{
    OperandUI[] operandsUI = (OperandUI[])this.listOperands.DataContext;
    double[] operands = operandsUI.Select(opui => opui.Value).ToArray();
    labelResult.Content = activeAddIn.Operate(currentOperation,
        operands);
}

```

Additional Add-Ins

The hard work is now done. The pipeline components and the host application are created. The pipeline is now working, yet it's an easy task to add other add-ins to the host application, such as the Advanced Calculator add-in shown in the following code segment.



```

[AddIn("Advanced Calc", Publisher = "Wrox Press", Version = "1.1.0.0",
    Description = "Another AddIn Sample")]
public class AdvancedCalculatorV1: Calculator

```

code snippet AdvancedCalcAddIn/AdvancedCalculator.cs

SUMMARY

In this chapter, you've learned the concepts of a new .NET 3.5 technology: the Managed Add-In Framework.

MAF uses a pipeline concept to create complete independence between the hosting and add-in assemblies. A clearly defined contract separates the host view from the add-in view. Adapters make it possible for both sides to change independently of each other.

The next chapter provides information on how to use .NET components with .NET Enterprise Services.

