2. Create a new WPF project for the host application in this directory. It doesn't matter what you name the project, but you must place it in the top-level directory you created in step 1 (for example, c:\AddInTest\HostApplication).

3. Add a new class library project for each pipeline component, and place them all in the same solution. At a bare minimum, you'll need to create a project for one add-in (for example, c:\AddInTest\MyAddIn), one add-in view (c:\AddInTest\MyAddInView), one add-in side adapter (c:\AddInTest\MyAddInAdapter), one host view (c:\AddInTest\HostView), and one host-side adapter (c:\AddInTest\HostAdapter). Figure 32-4 shows an example from the downloadable code for this chapter, which you'll consider in the following sections. It includes an application (named HostApplication) and two add-ins (named FadeImageAddIn and NegativeImageAddIn).

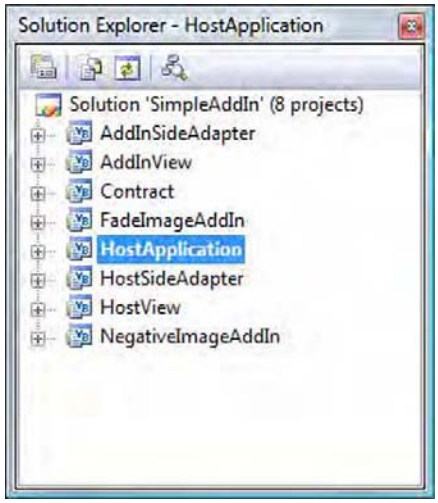


***Figure 32-4.*** *A solution that uses the add-in pipeline*

---

■ **Note** Technically, it doesn't matter what project names and directory names you use when you create the pipeline components. The required folder structure, which you learned about in the previous section, will be created when you build the application (provided you configure your project settings properly, as described in the following two steps). However, to simplify the configuration process, it's strongly recommended that you create all the project directories in the top-level directory you established in step 1.

---

4. Now you need to create a build directory inside the top-level directory. This is where your application and all the pipeline components will be placed once they're compiled. It's common to name this directory Output (as in c:\AddInTest\Output).

5. As you design the various pipeline components, you'll modify the build path of each one so that the component is placed in the right subdirectory. For example, your add-in adapter should be compiled to a directory like c:\AddInTest\Output\AddInSideAdapters. To modify the build path, double-click the Properties node in the Solution Explorer. Then, click the Build tab. In the Output section (at the bottom), you'll find a text box named Output Path. You need to use a relative output path that travels one level up the directory tree and then uses the Output directory. For example, the output path for an add-in adapter would be ..\Output\AddInSideAdapters. As you build each component in the following sections, you'll learn which build path to use. Figure 32-5 shows a preview of the final result, based on the solution shown in Figure 32-4.

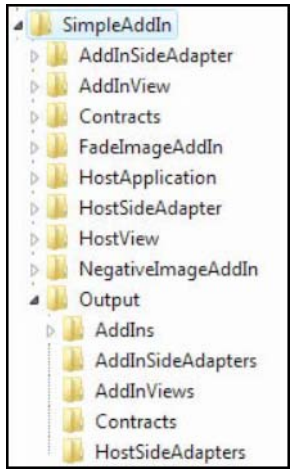


***Figure 32-5.*** *The folder structure for a solution that uses the add-in pipeline*

There's one more consideration when developing with the add-in model in Visual Studio: references. Some pipeline components need to reference other pipeline components. However, you don't want the referenced assembly to be copied with the assembly that contains the reference. Instead, you rely on the add-in model's directory system.

To prevent a referenced assembly from being copied, you need to select the assembly in the Solution Explorer (it appears under the References node). Then, set Copy Local to False in the Properties window. As you build each component in the following sections, you'll learn which references to add.

---

■ **Tip** Correctly configuring an add-in project can take a bit of work. To start off on the right foot, you can use the add-in example that's discussed in this chapter, which is available with the downloadable code for this book.

---

# An Application That Uses Add-Ins

In the following sections, you'll create an application that uses the add-in model to support different ways of processing a picture (Figure 32-6). When the application starts, it lists all the add-ins that are currently present. The user can then select one of the add-ins from the list and use it to modify the currently displayed picture.


***Figure 32-6.*** *An application that uses add-ins to manipulate a picture*

## The Contract

The starting point for defining the add-in pipeline for your application is to create a contract assembly. The contract assembly defines two things:

- The interfaces that determine how the host will interact with the add-in and how the add-in will interact with the host.
- Custom types that you use to exchange information between the host and add-in. These types must be serializable.

The example shown in Figure 32-6 uses an exceedingly simple contract. Plug-ins provide a method named ProcessImageBytes() that accepts a byte array with image data, modifies it, and returns the modified byte array. Here's the contract that defines this method:

```
[AddInContract]
public interface IImageProcessorContract : IContract
{
    byte[] ProcessImageBytes(byte[] pixels);
}
```

When creating a contract, you must derive from the IContract interface, and you must decorate the class with the AddInContract attribute. Both the interface and the attribute are found in the System.AddIn.Contract namespace. To have access to them in your contract assembly, you must add a reference to the System.AddIn.Contract.dll assembly.

Because the image-processing example doesn't use custom types to transmit data (just ordinary byte arrays), no types are defined in the contract assembly. Byte arrays can be transmitted between the host application and add-in because arrays and bytes are serializable.

The only additional step you need is to configure the build directory. The contract assembly must be placed in the Contracts subdirectory of the add-in root, which means you can use an output path of ..\Output\Contracts in the current example.

---

■ **Note** In this example, the interfaces are kept as simple as possible to avoid clouding the code with extra details. In a more realistic image-processing scenario, you might include a method that returns a list of configurable parameters that affect how the add-in processes the image. Each add-in would have its own parameters. For example, a filter that darkens a picture might include an Intensity setting, a filter that skews a picture might have an Angle setting, and so on. The host application could then supply these parameters when calling the ProcessImageBytes() method.

---

## The Add-in View

The add-in view provides an abstract class that mirrors the contract assembly and is used on the add-in side. Creating this class is easy:

```
[AddInBase]
public abstract class ImageProcessorAddInView
{
    public abstract byte[] ProcessImageBytes(byte[] pixels);
}
```

Notice that the add-in view class must be decorated with the AddInBase attribute. This attribute is found in the System.AddIn.Pipeline namespace. The add-in view assembly requires a reference to the System.AddIn.dll assembly in order to access it.

The add-in view assembly must be placed in the AddInViews subdirectory of the add-in root, which means you can use an output path of ..\Output\AddInViews in the current example.

## The Add-In

The add-in view is an abstract class that doesn't provide any functionality. To create a usable add-in, you need a concrete class that derives from the abstract view class. This class can then add the code that actually does the work (in this case, processing the image).

The following add-in inverts the color values to create an effect that's similar to a photo negative. Here's the complete code:

```
[AddIn("Negative Image Processor", Version = "1.0.0.0",
 Publisher = "Imaginomics",
 Description = "Inverts colors to look like a photo negative")]
public class NegativeImageProcessor : AddInView.ImageProcessorAddInView
```

```
{
    public override byte[] ProcessImageBytes(byte[] pixels)
    {
        for (int i = 0; i < pixels.Length - 2; i++)
        {
            // Assuming 24-bit, color, each pixel has three bytes of data.
            pixels[i] = (byte)(255 - pixels[i]);
            pixels[i + 1] = (byte)(255 - pixels[i + 1]);
            pixels[i + 2] = (byte)(255 - pixels[i + 2]);
        }
        return pixels;
    }
}
```

---

■ **Note** In this example, the byte array is passed into the ProcessImageBytes() method through a parameter, modified directly, and then passed back to the calling code as the return value. However, when you call ProcessImageBytes() from a different application domain, this behavior isn't as simple as it seems. The add-in infrastructure actually makes a *copy* of the original byte array and passes that copy to the add-in's application domain. Once the byte array is modified and has been returned from the method, the add-in infrastructure copies it back into the host's application domain. If ProcessImageBytes() didn't return the modified byte array in this way, the host would never see the changed picture data.

---

To create an add-in, you simply need to derive a class from the abstract view class and decorate it with the AddIn attribute. Additionally, you can use the properties of the AddIn attribute to supply an add-in name, version, publisher, and description, as done here. This information is made available to the host during add-in discovery.

The add-in assembly requires two references: one to the System.AddIn.dll assembly and one to the add-in view project. However, you must set the Copy Local property of the add-in view reference to False (as described earlier in the section "Preparing a Solution That Uses the Add-in Model"). That's because the add-in view isn't deployed with the add-in—instead, it's placed in the designated AddInViews subdirectory.

The add-in must be placed in its own subdirectory in the AddIns subdirectory of the add-in root. In the current example, you would use an output path like ..\Output\AddIns\NegativeImageAddIn.

## The Add-in Adapter

The current example has all the add-in functionality that you need, but there's still a gap between the add-in and the contract. Although the add-in view is modeled after the contract, it doesn't implement the contract interface that's used for communication between the application and the add-in.

The missing ingredient is the add-in adapter. It implements the contract interface. When a method is called in the contract interface, it calls the corresponding method in the add-in view. Here's the code for the most straightforward add-in adapter that you can create:

```
[AddInAdapter]
public class ImageProcessorViewToContractAdapter :
  ContractBase, Contract.IImageProcessorContract
```

```
{
    private AddInView.ImageProcessorAddInView view;

    public ImageProcessorViewToContractAdapter(
      AddInView.ImageProcessorAddInView view)
    {
        this.view = view;
    }

    public byte[] ProcessImageBytes(byte[] pixels)
    {
        return view.ProcessImageBytes(pixels);
    }
}
```

All add-in adapters must derive from ContractBase (from the System.AddIn.Pipeline namespace). ContractBase derives from MarshalByRefObject, which allows the adapter to be called over an application domain boundary. All add-in adapters must also be decorated with the AddInAdapter attribute (from the System.AddIn.Pipeline namespace). Furthermore, the add-in adapter must include a constructor that receives an instance of the appropriate view as an argument. When the add-in infrastructure creates the add-in adapter, it automatically uses this constructor and passes in the add-in itself. (Remember, the add-in derives from the abstract add-in view class expected by the constructor.) Your code simply needs to store this view for later use.

The add-in adapter requires three references: one to System.AddIn.dll, one to System.AddIn.Contract.dll, and one to the contract project. You must set the Copy Local property of the contract reference to False (as described earlier in the section "Preparing a Solution That Uses the Add-in Model").

The add-in adapter assembly must be placed in the AddInSideAdapters subdirectory of the add-in root, which means you can use an output path of ..\Output\AddInSideAdapters in the current example.

## The Host View

The next step is to build the host side of the add-in pipeline. The host interacts with the host view. Like the add-in view, the host view is an abstract class that closely mirrors the contract interface. The only difference is that it doesn't require any attributes.

```
public abstract class ImageProcessorHostView
{
    public abstract byte[] ProcessImageBytes(byte[] pixels);
}
```

The host view assembly must be deployed along with the host application. You can adjust the output path manually (for example, so the host view assembly is placed in the ..\Output folder in the current example). Or, when you add the host view reference to the host application, you can leave the Copy Local property set to True. This way, the host view will be copied automatically to the same output directory as the host application.

## The Host Adapter

The host-side adapter derives from the host view. It receives an object that implements the contract, which it can then use when its methods are called. This is the same forwarding process that the add-in adapter uses, but in reverse. In this example, when the host application calls the ProcessImageBytes()

method of the host view, it's actually calling ProcessImageBytes() in the host adapter. The host adapter calls ProcessImageBytes() on the contract interface (which is then forwarded across the application boundary and transformed into a method call on the add-in adapter).

Here's the complete code for the host adapter:

```
[HostAdapter]
public class ImageProcessorContractToViewHostAdapter :
 HostView.ImageProcessorHostView
{
    private Contract.IImageProcessorContract contract;
    private ContractHandle contractHandle;

    public ImageProcessorContractToViewHostAdapter(
      Contract.IImageProcessorContract contract)
    {
        this.contract = contract;
        contractHandle = new ContractHandle(contract);
    }

    public override byte[] ProcessImageBytes(byte[] pixels)
    {
        return contract.ProcessImageBytes(pixels);
    }
}
```

You'll notice the host adapter actually uses two member fields. It stores a reference to the current contract object, and it stores a reference to a System.AddIns.Pipeline.ContractHandle object. The ContractHandle object manages the lifetime of the add-in. If the host adapter doesn't create a ContractHandle object (and keep a reference to it), the add-in will be released immediately after the constructor code ends. When the host application attempts to use the add-in, it will receive an AppDomainUnloadedException.

The host adapter project needs references to System.Add.dll and System.AddIn.Contract.dll. It also needs references to the contract assembly and the host view assembly (both of which must have Copy Local set to false). The output path is the HostSideAdapters subdirectory in the add-in root (in this example it's ..\Output\HostSideAdapters).

## The Host

Now that the infrastructure is in place, the final step is to create the application that uses the add-in model. Although any type of executable .NET application could be a host, this example uses a WPF application.

The host needs just one reference that points to the host view project. The host view is the entry point to the add-in pipeline. In fact, now that you've done the heavy lifting in implementing the pipeline, the host doesn't need to worry about how it's managed. It simply needs to find the available add-ins, activate the ones it wants to use, and then call the methods that are exposed by the host view.

The first step—finding the available add-ins—is called *discovery*. It works through the static methods of the System.AddIn.Hosting.AddInStore class. To load add-ins, you simply supply the add-in root path and call AddInStore.Update(), as shown here:

```
// In this example, the path where the application is running
// is also the add-in root.
string path = Environment.CurrentDirectory;
```

```
AddInStore.Update(path);
```

After calling Update(), the add-in system will create two files with cached information. A file named PipelineSegments.store will be placed in the add-in root. This file includes information about the different views and adapters. A file named AddIns.store will be placed in the AddIns subdirectory, with information about all the available add-ins. If new views, adapters, or add-ins are added, you can update these files by calling AddInStore.Update() again. (This method returns quite quickly if there are no new add-ins or pipeline components.) If there is reason to expect that there is a problem with existing add-in files, you can call AddInStore.Rebuild() instead, which always rebuilds the add-in files from scratch.

Once you've created the cache files, you can search for the specific add-ins. You can use the FindAddIn() method to find a single specific add-in, or you can use the FindAddIns() method to find all the add-ins that match a specified host view. The FindAddIns() method returns a collection of tokens, each of which is an instance of the System.AddIn.Hosting.AddInToken class.

```
IList<AddInToken> tokens = AddInStore.FindAddIns(
  typeof(HostView.ImageProcessorHostView), path);
lstAddIns.ItemsSource = tokens;
```

You can get information about the add-in through a few key properties (Name, Description, Publisher, and Version). In the image-processing application (shown in Figure 32-6), the token list is bound to a ListBox control, and some basic information is shown about each add-in using the following data template:

```
<ListBox Name="lstAddIns" Margin="3">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Margin="3,3,0,8" HorizontalAlignment="Stretch">
        <TextBlock Text="{Binding Path=Name}" FontWeight="Bold" />
        <TextBlock Text="{Binding Path=Publisher}" />
        <TextBlock Text="{Binding Path=Description}"
         FontSize="10" FontStyle="Italic" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

You can create an instance of the add-in by calling the AddInToken.Activate<T> method. In the current application, the user clicks the Go button to activate an add-in. The information is then pulled out of the current image (which is shown in the window) and passed to the ProcessImageBytes() method of the host view. Here's how it works:

```
private void cmdProcessImage_Click(object sender, RoutedEventArgs e)
{
    // Copy the image information from the image to a byte array.
    BitmapSource source = (BitmapSource)img.Source;
    int stride = source.PixelWidth * source.Format.BitsPerPixel/8;
    stride = stride + (stride % 4) * 4;
    int arraySize = stride * source.PixelHeight *
      source.Format.BitsPerPixel / 8;
    byte[] originalPixels = new byte[arraySize];
    source.CopyPixels(originalPixels, stride, 0);

    // Get the selected add-in token.
```

```
    AddInToken token = (AddInToken)lstAddIns.SelectedItem;

    // Get the host view.
    HostView.ImageProcessorHostView addin =
      token.Activate<HostView.ImageProcessorHostView>(
        AddInSecurityLevel.Internet);

    // Use the add-in.
    byte[] changedPixels = addin.ProcessImageBytes(originalPixels);

    // Create a new BitmapSource with the changed image data, and display it.
    BitmapSource newSource = BitmapSource.Create(source.PixelWidth,
      source.PixelHeight, source.DpiX, source.DpiY, source.Format,
      source.Palette, changedPixels, stride);
    img.Source = newSource;
}
```

When you call the AddInToken.Activate<T> method, quite a few steps unfold behind the scenes:

1. A new application domain is created for the add-in. Alternatively, you can load the add-in into the application domain of the host application or into a completely separate process. However, the default is to place it in a distinct application domain in the current process, which usually gives the best compromise between stability and performance. You can also choose the level of permissions that are given to the new application domain. (In this example, they're limited to the Internet set of permissions, which is a heavily restricted permission set that's applied to code that's executed from the Web.)
2. The add-in assembly is loaded into the new application domain. The add-in is then instantiated through reflection, using its no-argument constructor. As you've already seen, the add-in derives from an abstract class in the add-in view assembly. As a result, loading the add-in also loads the add-in view assembly into the new application domain.
3. The add-in adapter is instantiated in the new application domain. The add-in is passed to the add-in adapter as a constructor argument. (The add-in is typed as the add-in view.)
4. The add-in adapter is made available to the host's application domain (through a remoting proxy). However, it's typed as the contract that it implements.
5. In the host application domain, the host adapter is instantiated. The add-in adapter is passed to the host adapter through its constructor.
6. The host adapter is returned to the host application (typed as the host view). The application can now call the methods of the host view to interact with the add-in through the add-in pipeline.

There are other overloads for the Activate<T> method that allow you to supply a custom permission set (to fine-tune security), a specific application domain (which is useful if you want to run several add-ins in the same application domain), and an outside process (which allows you to host the add-in in a completely separate EXE application for even greater isolation). All of these examples are illustrated in the Visual Studio help.

This code completes the example. The host application can now discover its add-ins, activate them, and interact with them through the host view.

### ADD-IN LIFETIME

You don't need to manage the lifetime of your add-ins by hand. Instead, the add-in system will automatically release an add-in and shut down its application domain. In the previous example, the add-in is released when the variable that points to the host view goes out of scope. If you want to keep the same add-in active for a longer time, you could assign it to a member variable in the window class.

In some situations, you might want more control over the add-in lifetime. The add-in model gives the host application the ability to shut down an add-in automatically using the AddInController class (from the System.AddIn.Hosting namespace), which tracks all the currently active add-ins. The AddInControls provides a static method named GetAddInController(), which accepts a host view and returns an AddInController for that add-in. You can then use the AddInController.Shutdown() method to end it, as shown here:

```
AddInController controller = AddInController.GetAddInController(addin);
controller.Shutdown();
```

At this point, the adapters will be disposed, the add-in will be released, and the add-in's application domain will be shut down if it doesn't contain any other add-ins.

## Adding More Add-Ins

Using the same add-in view, it's possible to create an unlimited number of distinct add-ins. In this example there are two, which process images in two different ways. The second add-in uses a crude algorithm to darken the picture by removing part of the color from random pixels:

```
[AddIn("Fade Image Processor", Version = "1.0.0.0", Publisher = "SupraImage",
 Description = "Darkens the picture")]
public class FadeImageProcessor : AddInView.ImageProcessorAddInView
{
    public override byte[] ProcessImageBytes(byte[] pixels)
    {
        Random rand = new Random();
        int offset = rand.Next(0, 10);
        for (int i = 0; i < pixels.Length - 1 - offset; i++)
        {
            if ((i + offset) % 5 == 0)
            {
                pixels[i] = 0;
            }
        }
        return pixels;
    }
}
```

In the current example, this add-in builds to the output path ..\Output\AddIns\FadeImageAddIn. There's no need to create additional views or adapters. Once you deploy this add-in (and then call the Rebuild() or Update() method of the AddInStore class), your host application will find both add-ins.

# Interacting with the Host

In the current example, the host is in complete control of the add-in. However, the relationship is often reversed. A common example is an add-in that drives an area of application functionality. This is particularly common with visual add-ins (the subject of the next section), such as custom toolbars. Often, this process of allowing the add-in to call the host is called *automation*.

From a conceptual standpoint, automation is quite straightforward. The add-in simply needs a reference to an object in the host's application domain, which it can manipulate through a separate interface. However, the add-in system's emphasis on versioning flexibility makes the implementation of this technique a bit more complicated. A single host interface is not enough, because it tightly binds the host and the add-in together. Instead, you'll need to implement a pipeline with views and adapters.

To see this challenge, consider the slightly updated version of the image-processing application, which is shown in Figure 32-7. It features a progress bar at the bottom of the window that's updated as the add-in processes the image data.


***Figure 32-7.*** *An add-in that reports progress*

■ **Tip** The rest of this section explores the changes you need to make to the image processor to support host automation. To see how these pieces fit together and examine the full code, download the code samples for this chapter.

For this application to work, the add-in needs a way to pass progress information to the host while it works. The first step in implementing this solution is to create the interface that defines how the add-in can interact with the host. This interface should be placed in the contract assembly (or in a separate assembly in the Contracts folder).

Here's the interface that describes how the add-in should be allowed to report progress, by calling a method named ReportProgress() in the host application:

```
public interface IHostObjectContract : IContract
{
    void ReportProgress(int progressPercent);
}
```

As with the add-in interface, the host interface must inherit from IContract. Unlike the add-in interface, the host interface does not use the AddInContract attribute, because it isn't implemented by an add-in.

The next step is to create the add-in view and host view. As when designing an add-in, you simply need an abstract class that closely corresponds to the interface you're using. To use the IHostObjectContract interface shown earlier, you simply need to add the following class definition to both the add-in view and host view projects.

```
public abstract class HostObject
{
    public abstract void ReportProgress(int progressPercent);
}
```

Notice that the class definition does not use the AddInBase attribute in either project.

The actual implementation of the ReportProgress() method is in the host application. It needs a class that derives from the HostObject class (in the host view assembly). Here's a slightly simplified example that uses the percentage to update a ProgressBar control:

```
public class AutomationHost : HostView.HostObject
{
    private ProgressBar progressBar;

    public Host(ProgressBar progressBar)
    {
        this.progressBar = progressBar;
    }

    public override void ReportProgress(int progressPercent)
    {
        progressBar.Value = progressPercent;
    }
}
```

You now have a mechanism that the add-in can use to send progress information to the host application. However, there's one problem—the add-in doesn't have any way to get a reference to the HostObject. This problem doesn't occur when the host application is using an add-in, because it has a discovery feature that it can use to search for add-ins. There's no comparable service for add-ins to locate their host.

The solution is for the host application to pass the HostObject reference to the add-in. Typically, this step will be performed when the add-in is first activated. By convention, the method that the host application uses to pass this reference is often called Initialize().

Here's the updated contract for image processor add-ins:

```
[AddInContract]
public interface IImageProcessorContract : IContract
```

```
{
    byte[] ProcessImageBytes(byte[] pixels);
    void Initialize(IHostObjectContract hostObj);
}
```

When Initialize() is called, the add-in will simply store the reference for later use. It can then call the ReportProgress() method whenever is appropriate, as shown here:

```
[AddIn]
public class NegativeImageProcessor : AddInView.ImageProcessorAddInView
{
    private AddInView.HostObject host;
    public override void Initialize(AddInView.HostObject hostObj)
    {
        host = hostObj;
    }

    public override byte[] ProcessImageBytes(byte[] pixels)
    {
        int iteration = pixels.Length / 100;

        for (int i = 0; i < pixels.Length - 2; i++)
        {
            pixels[i] = (byte)(255 - pixels[i]);
            pixels[i + 1] = (byte)(255 - pixels[i + 1]);
            pixels[i + 2] = (byte)(255 - pixels[i + 2]);

            if (i % iteration == 0)
                host.ReportProgress(i / iteration);
        }
        return pixels;
    }
}
```

So far, the code hasn't posed any real challenges. However, the last piece—the adapters—is a bit more complicated. Now that you've added the Initialize() method to the add-in contract, you need to also add it to the host view and add-in view. However, the signature of the method can't match the contract interface. That's because the Initialize() method in the interface expects an IHostObjectContract as an argument. The views, which are not linked the contract in any way, have no knowledge of the IHostObjectContract. Instead, they use the abstract HostObject class that was described earlier:

```
public abstract class ImageProcessorHostView
{
    public abstract byte[] ProcessImageBytes(byte[] pixels);

    public abstract void Initialize(HostObject host);
}
```

The adapters are the tricky part. They need to bridge the gap between the abstract HostObject view classes and the IHostObjectContract interface.

For example, consider the ImageProcessorContractToViewHostAdapter on the host side. It derives from the abstract ImageProcessorHostView class, and as a result it implements the version of Initialize() that receives a HostObject instance. This Initialize() method needs to convert this view to the contract and then call the IHostObjectContract.Initialize() method.

The trick is to create an adapter that performs this transformation (much like the adapter that performs the same transformation with the add-in view and add-in interface). The following code shows the new HostObjectViewToContractHostAdapter that does the work and the Initialize() method that uses it to make the jump from the view class to the contract interface:

```
public class HostObjectViewToContractHostAdapter : ContractBase,
  Contract.IHostObjectContract
{
    private HostView.HostObject view;

    public HostObjectViewToContractHostAdapter(HostView.HostObject view)
    {
        this.view = view;
    }

    public void ReportProgress(int progressPercent)
    {
        view.ReportProgress(progressPercent);
    }
}

[HostAdapter]
public class ImageProcessorContractToViewHostAdapter :
  HostView.ImageProcessorHostView
{
    private Contract.IImageProcessorContract contract;
    private ContractHandle contractHandle;

    ...

    public override void Initialize(HostView.HostObject host)
    {
        HostObjectViewToContractHostAdapter hostAdapter =
          new HostObjectViewToContractHostAdapter(host);
        contract.Initialize(hostAdapter);
    }
}
```

A similar transformation takes place in the add-in adapter, but in reverse. Here, the ImageProcessorViewToContractAdapter implements the IImageProcessorContract interface. It needs to take the IHostObjectContract object that it receives in its version of the Initialize() method and then convert the contract to a view. Next, it can pass the call along by calling the Initialize() method in the view. Here's the code:

```
[AddInAdapter]
public class ImageProcessorViewToContractAdapter : ContractBase,
  Contract.IImageProcessorContract
{
    private AddInView.ImageProcessorAddInView view;
```

```
    ...

    public void Initialize(Contract.IHostObjectContract hostObj)
    {
        view.Initialize(new HostObjectContractToViewAddInAdapter(hostObj));
    }
}

public class HostObjectContractToViewAddInAdapter : AddInView.HostObject
{
    private Contract.IHostObjectContract contract;
    private ContractHandle handle;

    public HostObjectContractToViewAddInAdapter(
      Contract.IHostObjectContract contract)
    {
        this.contract = contract;
        this.handle = new ContractHandle(contract);
    }

    public override void ReportProgress(int progressPercent)
    {
        contract.ReportProgress(progressPercent);
    }
}
```

Now, when the host calls Initialize() on an add-in, it can flow through the host adapter (ImageProcessorContractToViewHostAdapter) and the add-in adapter (ImageProcessorViewToContractAdapter), before being called on the add-in. When the add-in calls the ReportProgress() method, it flows through similar steps, but in reverse. First it flows through the add-in adapter (HostObjectContractToViewAddInAdapter), and then it passes to the host adapter (HostObjectViewToContractHostAdapter).

This walk-through completes the example—sort of. The problem is that the host application calls the ProcessImageBytes() method on the main user interface thread. As a result, the user interface is effectively locked up. Although the calls to ReportProgress() are handled and the progress bar is updated, the window isn't refreshed until the operation is complete.

A far better approach is to perform the time-consuming call to ProcessImageBytes() on a background thread, either by creating a Thread object by hand or by using the BackgroundWorker. Then, when the user interface needs to be updated (when ReportProgress() is called and when the final image is returned), you must use the Dispatcher.BeginInvoke() method to marshal the call back to the user interface thread. All of these techniques were demonstrated earlier in this chapter. To see the threading code in action in this example, refer to the downloadable code for this chapter.

# Visual Add-Ins

Considering that the WPF is a display technology, you've probably started to wonder whether there's a way to have an add-in generate a user interface. This isn't a small challenge. The problem is that the user interface elements in WPF aren't serializable. Thus, they can't be passed between the host application and the add-in.

Fortunately, the designers of the add-in system created a sophisticated workaround. The solution is to allow WPF applications to display user interface content that's hosted in separate application domains. In other words, your host application can display controls that are actually running in the

application domain of an add-in. If you interact with these controls (clicking them, typing in them, and so on), the events are fired in the add-in's application domain. If you need to pass information from the add-in to the application, or vice versa, you use the contract interfaces, as you've explored in the previous sections.

Figure 32-8 shows this technique in action in a modified version of the image-processing application. When an add-in is selected, the host application asks the add-in to provide a control with suitable content. That control is then displayed at the bottom of the window.


***Figure 32-8.*** *A visual add-in*

In this example, the negative image add-in has been selected. It provides a user control that wraps an Image control (with a preview of the effect) and a Slider control. As the slider is adjusted, the intensity of the effect is changed, and the preview is updated. (The update process is sluggish, because of the poorly optimized image-processing code. Much better algorithms could be used, possibly incorporating unsafe code blocks for maximum performance.)

Although the plumbing that makes this work is fairly sophisticated, it's surprisingly easy to use. The key ingredient is the INativeHandleContract interface from the System.AddIn.Contract namespace. It allows a window handle to be passed between an add-in and the host application.

Here's the revised IImageProcessorContract from the contract assembly. It replaces the ProcessImageBytes() method with a GetVisual() method that accepts similar image data but returns a chunk of user interface:

```
[AddInContract]
public interface IImageProcessorContract : IContract
{
    INativeHandleContract GetVisual(Stream imageStream);
}
```

You don't use the INativeHandlerContract in the view classes, because it isn't directly usable in your WPF applications. Instead, you use the type you expect to see—a FrameworkElement. Here's the host view:

```
public abstract class ImageProcessorHostView
{
    public abstract FrameworkElement GetVisual(Stream imageStream);
}
```

And here's the nearly identical add-in view:

```
[AddInBase]
public abstract class ImageProcessorAddInView
{
    public abstract FrameworkElement GetVisual(Stream imageStream);
}
```

This example is surprisingly similar to the automation challenge in the previous section. Once again, you have a different type being passed in the contract from the one that's used in the views. And once again, you need to use the adapters to perform the contract-to-view and view-to-contract conversion. However, this time the work is done for you by a specialized class called FrameworkElementAdapters.

FrameworkElementAdapters is found in the System.AddIn.Pipeline namespace, but it's actually part of WPF, and it's part of the System.Windows.Presentation.dll assembly. The FrameworkElementAdapters class provides two static methods that perform the conversion work: ContractToViewAdapter() and ViewToContractAdapter().

Here's how the FrameworkElementAdapters.ContractToViewAdapter() method bridges the gap in the host adapter:

```
[HostAdapter]
public class ImageProcessorContractToViewHostAdapter :
  HostView.ImageProcessorHostView
{
    private Contract.IImageProcessorContract contract;
    private ContractHandle contractHandle;
    ...

    public override FrameworkElement GetVisual(Stream imageStream)
    {
        return FrameworkElementAdapters.ContractToViewAdapter(
          contract.GetVisual(imageStream));
    }
}
```

And here's how the FrameworkElementAdapters.ViewToContractAdapter() method bridges the gap in the add-in adapter:

```
[AddInAdapter]
public class ImageProcessorViewToContractAdapter : ContractBase,
  Contract.IImageProcessorContract
{
    private AddInView.ImageProcessorAddInView view;
    ...
```

```
    public INativeHandleContract GetVisual(Stream imageStream)
    {
        return FrameworkElementAdapters.ViewToContractAdapter(
          view.GetVisual(imageStream));
    }
}
```

Now the final detail is to implement the GetVisual() method in the add-in. In the negative image processor, a new user control named ImagePreview is created. The image data is passed to the ImagePreview control, which sets up the preview image and handles slider clicks. (The user control code is beside the point for this example, but you can see the full details by downloading the samples for this chapter.)

```
[AddIn]
public class NegativeImageProcessor : AddInView.ImageProcessorAddInView
{
    public override FrameworkElement GetVisual(System.IO.Stream imageStream)
    {
        return new ImagePreview(imageStream);
    }
}
```

Now that you've seen how to return a user interface object from an add-in, there's no limit to what type of content you can generate. The basic infrastructure—the INativeHandleContract interface and the FrameworkElementAdapters class—remains the same.

# The Last Word

In this chapter, you dove into the deeply layered add-in model. You learned how its pipeline works, why it works the way it does, and how to create basic add-ins that support host automation and provide visual content.

There's quite a bit more you can learn about the add-in model. If you plan to make add-ins a key part of a professional application, you'll want to take a closer look at specialized versioning and hosting scenarios and deployment, best practices for dealing with unhandled add-in exceptions, and how to allow more complex interactions between the host and add-in and between separate add-ins. To get the details, you'll need to visit the team blog for the Microsoft developers who created the add-in system at `http://blogs.msdn.com/clraddins`. You may also be interested in Jason He's blog (`http://blogs.msdn.com/zifengh`). He is a member of the add-in team who has written about his experience adapting Paint.NET to use the add-in model.

CHAPTER 33

# ClickOnce Deployment

Sooner or later, you'll want to unleash your WPF applications on the world. Although you can use dozens of different ways to transfer an application from your development computer to an end user's desktop, most WPF applications use one of the following deployment strategies:

- **Run in the browser.** If you create a page-based WPF application, you can run it right in the browser. You don't need to install anything. However, your application needs to be able to function with a very limited set of privileges. (For example, you won't be allowed to access arbitrary files, use the Windows registry, pop up new windows, and so on.) You learned about this approach in Chapter 24.
- **Deploy via the browser.** WPF applications integrate closely with the ClickOnce setup feature, which allows users to launch a setup program from a browser page. Best of all, applications that are installed through ClickOnce can be configured to check for updates automatically. On the negative side, you have little ability to customize your setup and no way to perform system configuration tasks (such as modifying the Windows registry, creating a database, and so on).
- **Deploy via a traditional setup program.** This approach still lives on in the WPF world. If you choose this option, it's up to you whether you want to create a full-fledged Microsoft Installer (MSI) setup or a more streamlined (but more limited) ClickOnce setup. Once you've built your setup, you can choose to distribute it by placing it on a CD, in an e-mail attachment, on a network share, and so on.

In this chapter, you'll focus on the second approach: deploying your application with the ClickOnce deployment model.

---

**What's New** Although ClickOnce was designed as a lightweight deployment technology that wouldn't compete with full-featured setup programs, every version adds to its capabilities. In .NET 4, a ClickOnce setup can create a desktop icon and register file types, as you'll see in this chapter. Additionally, .NET now installs a Firefox add-in called the Microsoft .NET Framework Assistant, which allows users to run ClickOnce setups from Firefox as well as Internet Explorer (unless the user removes the add-in, of course).

---

# Understanding Application Deployment

Although it's technically possible to move a .NET application from one computer to another just by copying the folder that contains it, professional applications often require a few more frills. For example, you might need to add multiple shortcuts to the Start menu, add registry settings, and set up additional resources (such as a custom event log or a database). To get these features, you need to create a custom setup program.

You have many options for creating a setup program. You can use a retail product like InstallShield, or you can create an MSI setup using the Setup Project template in Visual Studio. Traditional setup programs give you a familiar setup wizard, with plenty of features for transferring files and performing a variety of configuration actions.

Your other choice is to use the ClickOnce deployment system that's closely integrated in WPF. ClickOnce has plenty of limitations (most of them by design), but it offers two important advantages:

- Support for installing from a browser page (which can be hosted on an internal network or placed on the Web).
- Support for automatically downloading and installing updates.

These two features might not be enough to entice developers to give up the features of a full-fledged setup program. But if you need a simple, lightweight deployment that works over the Web and supports automatic updates, ClickOnce is perfect.

## CLICKONCE AND PARTIAL TRUST

Ordinary WPF applications require full trust because your application needs unmanaged code permission to create a WPF window. That means installing a stand-alone WPF application using ClickOnce presents the same security roadblock as installing any type of application from the Web—namely, the web browser will present a security warning. If the user goes ahead, the installed application will have the ability to do anything that the current user can do.

ClickOnce works differently with older Windows Forms applications. Windows Forms applications can be configured to use partial trust and then deployed using ClickOnce. In a best-case scenario, this means users can install a partial-trust Windows Forms application through ClickOnce without a security prompt or permission elevation.

It might seem like the Windows Forms approach is better, but WPF *does* have a way to combine partial-trust programming and a ClickOnce-based install experience. The trick is to use the XBAP model described in Chapter 24. In this situation, your application runs in the browser; therefore, it doesn't need to create any windows, and it doesn't need unmanaged code permission. Even better, because the application is accessed through a URL (and then cached locally), the user always runs the latest, most up-to-date version. Behind the scenes, the automatic XBAP download uses the same ClickOnce technology that you'll use in this chapter.

XBAPs aren't covered in this chapter. For more information about XBAPs and partial-trust programming, refer to Chapter 24.