14

Programming and Extending SSIS

Out of the box, Microsoft provides a huge list of components for you in SSIS. When you find that none of them fits your job, you need to be able to create your own. Initially this can be a steep learning curve, but hopefully with the help of this chapter you will be able to overcome this. In this chapter you will focus on the pipeline — not because it is better than any other area of programmability within SSIS but because it will probably be the area where you have the most benefit to gain, and it does require a slightly greater level of understanding. It also allows you to see some of the really interesting things that Microsoft has done in SSIS. All forms of extensibility are well covered in the SQL Server documentation and samples, so don't forget to leverage those resources as well.

The *pipeline*, for all intents and purposes, is the way your data moves from A to B and how it is manipulated, if at all. You can find it on the Data Flow tab of your packages after you have dropped a Data Flow task into the Control Flow. There's no need to go into any more detail about where to find the pipeline in your package, as this has been covered elsewhere in this book.

Most people think that even the most timid or "code scared" DBAs, when faced with not having what they want in the SSIS box, will be able to do a very reasonable job of designing and building the component they need themselves. Because SSIS is now hosted in the Visual Studio shell, traditional programmers may be most at home here, but that's no reason for everyone else to be left out in the cold. Hopefully by the end of this chapter you'll see that it doesn't have to be that way.

The Sample Components

Three sample components will be defined in this section to demonstrate the main component types. The transform component will then be expanded in Chapter 15 to include a user interface. All code samples will be available on the Web site for this book, which you can find at `www.wrox.com`.

Component 1: Source Adapter

The source adapter needs to be able to do quite a few things in order to be able to present the data to the downstream components in the pipeline in a format that the next component understands and is expecting. Here is a list of what the component needs to do:

- Accept a Connection Manager
- Validate the Connection Manager (did it get the right type of Connection Manager?)
- Add output columns to the component for the downstream processes
- Connect to the data source
- Get the data from the data source
- Assign the correct parts of the data to the correct output columns
- Handle any data errors

This component is going to need to do quite a bit of work in order to present its data to the outside world. Stick with it and you'll see how easy this can be. Your aim in the source adapter is to be able to take a file with a custom format, read it, and present it to the downstream components. The file will look like this:

```
<START>
Name:
Age:
Married:
Salary:
<END>
```

As you can see, this is a nonstandard format that none of the source adapters out of the box could deal with.

Component 2: Transformation

The transform is where you are going to take data from a source, manipulate it, and then present the newly arranged data to the downstream components. This component performs the following tasks:

- Create input columns to accept the data from upstream
- Validate the data to see that it is how the component expects it
- Check the column properties because this transform will be changing them in place
- Handle somebody trying to change the metadata of the transform by adding or removing inputs and/or outputs

The requirement here is to take data from the source and reverse the contents. The quirk, though, is that the column properties must be set correctly, and you can only perform this operation on certain data types.

Component 3: Destination Adapter

The destination adapter will take the data received from the upstream component and write it to the destination. This component will need to do the following:

- Create an input that accepts the data
- Validate that the data is correct
- Accept a Connection Manager
- Validate the Connection Manager (did you get the right type of Connection Manager?)
- Connect to the data source
- Write data from the data source

The destination adapter is basically a reverse of the source adapter. When it receives the input rows, it needs to create a new file with data resembling that of the source file except that some of the data will be the opposite way around compared to when it started out in the pipeline.

The components you'll build are really quite simple, but the point is not their complexity, but how you use the methods in Microsoft's object model.

The Pipeline Component Methods

Components are normally described as having two distinct phases: design-time and runtime. When you implement a component, you *inherit* from the base class, `Microsoft.SqlServer.Dts.Pipeline.PipelineComponent`, and provide your own functionality by *overriding* the base methods, some of which are primarily design-time, others runtime. If you are using native code, then the divide between the runtime and design-time is clearer because they are implemented on different interfaces. Commentary on the methods has been divided into these two sections, but there are some exceptions, notably the connection-related methods; a section on Connection Time–related methods is included later on.

> *In programming terms, a class can inherit functionality from another class, termed the base class. If the base class provides a method, and the inheriting class wishes to change the functionality within this method, it can override the method. In effect, you replace the base method with your own. From within the overriding method, you can still access the base method, and call it explicitly if required, but any consumer of the new class will see only the overriding method.*

Design-Time

The following methods are explicitly implemented for design-time, overriding the `PipelineComponent` methods, although they will usually be called from within your overriding method. Not all of the methods have been listed, because for some there is little more to say, and others have been grouped together according to their area of function. Refer to the SQL Server documentation for a complete list.

There are some methods that have been described as verification methods, and these are a particularly interesting group. They provide minor functions such as adding a column or setting a property value,

and you could quite rightly think that there is little point in ever overriding them, as there isn't much to add to the base implementation. As mentioned, these are your verification methods, and code has been added for verification that the operation about to take place within the base class is allowed. The following sections expand on the types of checks you can do, and if you want to build a robust component, these are well worth looking into.

Another very good reason to implement these methods as described is actually to reduce code. These methods will be used by both a custom user interface (UI) and the built-in component editor, or Advanced Editor. If you raise an error saying that a change is not allowed, then both user interfaces can capture this and provide feedback to the user. Although a custom UI would not be expected to offer controls to perform blatantly inappropriate actions, the Advanced Editor is designed to offer all functionality, so you are protecting the integrity of your component regardless of the method used.

ProvideComponentProperties

This method is provided so you can set up your component. It is called when a component is first added to the Data Flow, and it initializes the component. It does not perform any column-level activity, as this is left to `ReinitializeMetadata`; when this method is invoked, there are generally no inputs to outputs to be manipulated anyway. The sorts of procedures you may want to set in here are listed below.

- Remove existing settings, such as inputs and outputs. This allows the component to be rebuilt and can be useful when things go wrong.
- Add inputs and outputs, ready for column work later on in the component lifetime. You may also define custom properties on them and specify related properties, such as linking them together for synchronous behavior.
- Define the connection requirements. By adding an item to the RuntimeConnectionCollection, you have a placeholder prepared for the Connection Manager at runtime, as well as informing the designer of this requirement.
- The component may have custom properties that are configurable by a user in addition to those you get for free from Microsoft. These will hold settings other than the column-related one that effect the overall component operation or behavior.

Validate

`Validate` is called numerous times during the lifetime of the component, both at design-time and at runtime, but the most interesting work is usually the result of a design-time call. As the name suggests, it validates that the content of the component is correct and will enable you to at least run the package. If the validation encounters a problem, then the return code used is important to determine any further actions, such as calling `ReinitializeMetadata`. The base class version of `Validate` performs its own checks in the component, and you will need to extend it further in order to cover your specific needs. `Validate` should not be used to change the component at all; it should only report the problems it finds.

ReinitializeMetaData

The `ReinitializeMetaData` method is where all the building work for your component is done. You add new columns, remove invalid columns, and generally build up the columns. It is called when the `Validate` method returns `VS_NEEDSNEWMETADATA`. It is also your opportunity in the component to do any repairs that need to be done, particularly around invalid columns as mentioned previously.

MapInputColumn and MapOutputColumn

These methods are used to create a relationship between an input/output column and an external metadata column. An external metadata column is an offline representation of an output or input column and can be used by downstream components to create an input. It allows you to validate and maintain columns even when the data source is not available. It is not required, but it makes the user experience better. If the component declares that it will be using External Metadata (IDTSComponentMetaData90 .ValidateExternalMetadata), then the user in the advanced UI will see upstream columns to the left and the external columns on the right; if you are validating your component against an output, you will see the checked listbox of columns.

Input and Output Verification Methods

There are several methods you can use to deal with inputs and outputs. The three functions you may need to perform are adding, deleting, and setting a custom property. The method names clearly indicate their functions:

- `InsertInput`
- `DeleteInput`
- `SetInputProperty`
- `InsertOutput`
- `DeleteOutput`
- `SetOutputProperty`

For most components, the inputs and outputs will have been configured during `ProvideComponent Properties`, so unless you expect a user to add additional inputs and outputs and fully support this, you should override these methods and fire an error to prevent this. Similarly, unless you support additions, you would also want to deny deletions by overriding the corresponding methods. Properties can be checked for validity during the `Set` methods as well.

Set Column Data Types

There are two methods used to set column data types: one for output columns and the other for external metadata columns. There is no input column equivalent, as the data types of input columns are determined by the upstream component.

- `SetOutputColumnDataTypeProperties`
- `SetExternalMetadataColumnDataTypeProperties`

These are verification methods that can be used to validate or prevent changes to a column. For example, in a source component, you would normally define the columns and their data types within `ReinitializeMetaData`. You could then override `SetOutputColumnDataTypeProperties`, and by comparing the method's supplied data types to the existing column, you could prevent data type changes but allow length changes.

There is quite a complex relationship between all of the parameters for these methods; please refer to SQL Server documentation for reference when using this method yourself.

PerformUpgrade

This method should allow you to take a new version of the component and update an existing version of the component on the destination machine.

RegisterEvents

This method allows you to register custom events in a pipeline component. You can therefore have an event fire on something happening at runtime in the package. This is then eligible to be logged in the package log.

RegisterLogEntries

This method decides which of the new custom events are going to be registered and selectable in the package log.

SetComponentProperty

In the `ProvideComponentProperties` method, you told the component about any custom properties that you would like to expose to the user of the component and perhaps allow them to set. This is a verification method, and here you can check what it is that the user has entered for which custom property on the component and ensure that the values are valid.

Set Column Properties

There are three column property methods, each allowing you to set a property for the relevant column type.

- `SetInputColumnProperty`
- `SetOutputColumnProperty`
- `SetExternalMetadataColumnProperty`

These are all verification methods and should be used accordingly. For example, you may set a column property during `ReinitializeMetaData`, and to prevent a user interfering with this, you could examine the property name and throw an exception if it is a restricted property, in effect making it read-only.

Similarly, if several properties are used in conjunction with each other at runtime to provide direction on the operation to be performed, you could enumerate all column properties to ensure that those related properties exist and have suitable values. You could assign a default value if a value is not present or raise an exception depending on the exact situation.

For an external metadata column, which will be mapped to an input or output column, any property set directly on this external metadata column can be cascaded down onto the corresponding Input or Output column through this overridden function.

SetUsageType

This method deals with the columns on inputs into the component. In a nutshell, you use it to select a column and to tell the component how you will treat each column. What you see coming into this method is the Virtual Input. What this means is that it is a representation of what is available for selection to be used by your component. These are the three possible usage types for a column:

- `DTSUsageType.UT_IGNORED` — The column will not be used by the component. What happens is that you will be removing from the InputColumnCollection this InputColumn. This differs from the other two usage types, which add a reference to the InputColumn to the InputColumnCollection if it does not exist already or you may be changing its Read/Write property.
- `DTSUsageType.UT_READONLY` — The column is read-only. The column is selected, and data can be read and used within the component but cannot be modified.
- `DTSUsageType.UT_READWRITE` — The column is selected, and you can both read and write or change the data within your component.

This is another of the verification methods, and you should use it to ensure that the columns selected are valid. For example, the Reverse String sample shown below can operate only on string columns, so you must check that the data type of the input column is DT_STR for string or DT_WSTR for Unicode strings. Similarly, the component performs an in-place change, so the usage type must be read/write. Setting it to read-only would cause an exception during execution when you tried to write the changed data back to the pipeline buffer. Therefore you want to validate the columns as they are selected to ensure that they meet the requirements for your component design.

On Path Attachment

There are three closely related path attachment methods, called when the named events occur, and the first two in particular can be used to improve the user experience:

- `OnInputPathAttached`
- `OnOutputPathAttached`

The reason these methods are here is to handle situations where the inputs or outputs are all identical and interchangeable, the multicast being the example, where you attach to the dangling output and another dangling output is created. You detach, and the output is deleted.

Runtime

Runtime, also known as execution-time, is when you actually work with the data, through the pipeline buffer, with columns and rows of data. The following methods are all about preparing the component, doing the job it was designed for, and then cleaning up afterward.

PrepareForExecute

This method is rather like the `PreExecute` method below and can be used for setting up anything in the component that you will need at runtime. The difference is that you do not have access to the Buffer Manager, so you cannot get your hands on the columns in either the output or the input at this stage. The distinction between the two is very fine apart from that, so usually you will end up using `PreExecute` exclusively, as you will need access to the Buffer Manager anyway.

PreExecute

`PreExecute` is called once and once only in the component, and it is the recommendation of Microsoft that you do as much preparation as possible for the execution of your component in this method. In this case, you'll use it to enumerate the columns, reading off values and properties, calling methods to get more information, and generally preparing by gathering all the information you require in advance. This is stored in a variable, making it faster to access multiple times rather than creating objects during the real execution for every row. This is the earliest point in the component that you will access the component's Buffer Manager, so you have the live context of columns, as opposed to the design-time representation. The live and design time representations of columns may not match. The design time may contain more information that you do not need at runtime. As mentioned, it is here that you do the Column Preparation for your component in this method, because it is called only once per component execution, unlike some of the other runtime methods, which are called multiple times.

PrimeOutput and ProcessInput

These two methods are dealt with together because they are so closely linked that to deal with them any other way would be disjointed. These two methods are essentially how the data flows through components. Sometimes you use only one of them, and sometimes you use both. There are some rules you can follow.

In a source adapter, the `ProcessInput` method is never called, and all of the work is done through `PrimeOutput`. In a destination adapter, it is the opposite way around. The `PrimeOutput` method is never called, and the whole of the work is done through the `ProcessInput` method.

Things are not quite that simple with a transform. There are two types of transforms, and the type of transform you are writing will dictate which method or indeed methods your component should call.

- **Synchronous:** `PrimeOutput` is not called and therefore all the work is done in the `ProcessInput` method. The buffer Lineage IDs remain the same. For a detailed explanation of buffers and Lineage IDs, please refer to Chapter 10.
- **Asynchronous:** Both methods are called here. The only difference really between a synchronous and an asynchronous component is that the asynchronous component does not reuse the input buffer. The `PrimeOutput` method hands the `ProcessInput` method a buffer to fill with its data.

PostExecute

This method would be where you clean up anything that you started in `PreExecute`. Although it can do this, it is not limited to just that. After reading the description of the Cleanup method in just a second, you're going to wonder about the difference between that and this method. The answer is, for this release, nothing. If you want to think about this logically, then PreExecute is married to PrepareForExecute.

Cleanup

As the method name suggests, this is called as the very last thing your component will do, and it is your chance to clean up whatever resources may be left. However, it is rarely used, like `PostExecute`.

DescribeRedirectedErrorCode

If you are using an error output and directing rows down there in case of errors, then you should expose this method to give more information about the error. When you direct a row to the error output, you specify an error code. This method will be called by the pipeline engine, passing in that error code, and it is expected to return a full error description string for the code specified. These two values are then included in the columns of the error output.

Connection Time

These two methods are called several times throughout the life cycle of a component, both at design-time and at runtime, and are used to manage connections within the component.

AcquireConnections

This method is called both in design and when the component executes. There is no explicit result, but the connection is normally validated and then cached in a member variable within the component for later use. At this stage, a connection should be open and ready to use.

ReleaseConnections

If you have any open connections, as set in the `AcquireConnections` method, then this is where they should be closed and released. If the connection was cached in a member variable, use that reference to issue any appropriate `Close` or `Dispose` methods. For some connections, such as a File Connection Manager, this may not be relevant as all that was returned was a file path string, but if you took this a stage further and opened a text stream or similar on the file, it should now be closed.

Building the Components

Now you can move on to actually building the components. These components are simple and demonstrate the most commonly used methods when building your own components. They also help give you an idea of what the composition of a component resembles, the order in which things happen, and which method is meant to do what. They will not implement all the available methods. The components have been built and they can be extended, so why not download them and give them a go? If you happen to break them, simply revert back to a previous good copy. No programmer gets things right the first time, so having the component break is part of the experience. Or at least that's what programmers tell themselves at two o'clock in the morning when they are still trying to figure out why the thing isn't doing what they asked. The component classes will be covered in the next sections. You will then be shown how to make sure your component appears in the correct folder, what to put in the AssemblyInfo file, how it gets registered in the GAC, and how to sign the assembly. This is common to all three components, so it will be dealt with as one also.

Preparation

In this section of the chapter, you'll go through the steps that are common to all the pipeline components. These are the basic sets of things you need to do before you fly into coding.

Start by opening Visual Studio 2005, and create a New Project, a Class Library project as shown in Figure 14-1.

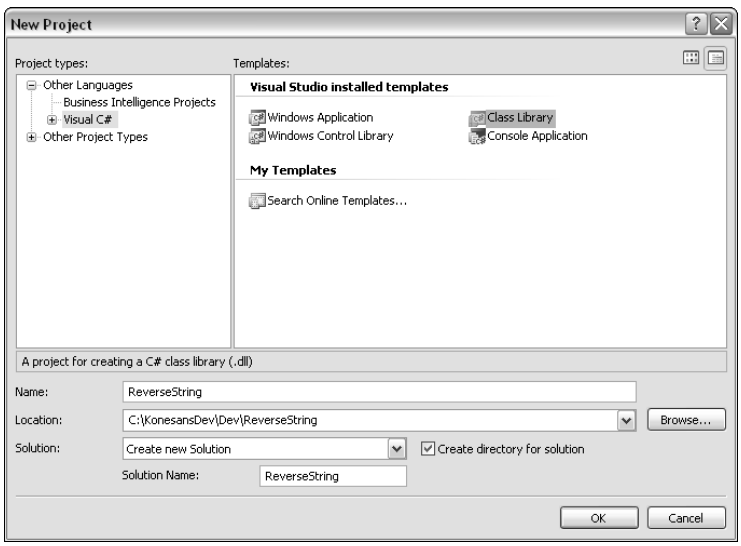


Figure 14-1

Now select the Add References option from the Project menu, and select the assemblies listed below, which are also illustrated in Figure 14-2.

- `Microsoft.SqlServer.DTSPipelineWrap`
- `Microsoft.Sqlserver.DTSRuntimeWrap`
- `Microsoft.Sqlserver.ManagedDTS`
- `Microsoft.SqlServer.PipelineHost`

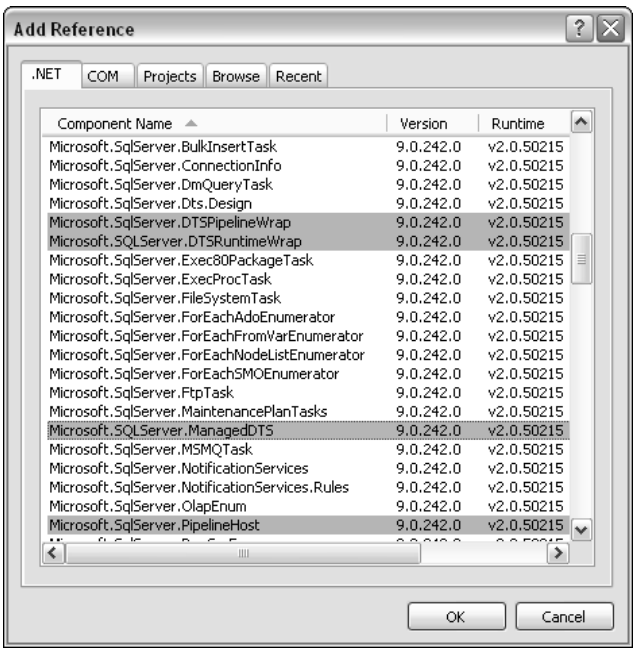


Figure 14-2

Once you have those set up, you can start to add the `using` directives. These are the directives:

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Text;
using System.Globalization;
using System.Runtime.InteropServices;
using Microsoft.SqlServer.Dts.Pipeline;
using Microsoft.SqlServer.Dts.Pipeline.Wrapper;
using Microsoft.SqlServer.Dts.Runtime.Wrapper;
using Microsoft.SqlServer.Dts.Runtime;

#endregion
```

The first stage in building a component is to inherit from the `PipelineComponent` base class and to decorate the class with `DtsPipelineComponent`. From this point on, you are officially working on a pipeline component.

```
namespace Konesans.Dts.Pipeline.ReverseString
{
   [DtsPipelineComponent(
      DisplayName = "ReverseString",
      ComponentType = ComponentType.Transform,
      IconResource =  "Konesans.Dts.Pipeline.ReverseString.ReverseString.ico")]
   public class ReverseString : PipelineComponent
   {
      ...
```

The `DtsPipelineComponent` attribute supplies design-time information about your component, and the first key property here is `ComponentType`. The three options — Source, Destination, or Transformation — reflect the three tabs within the SSIS designer Toolbox. This option determines which tab or grouping of components your component belongs to; it does not have any influence over the component behavior. The display name should be self-explanatory, and the `IconResource` is the reference to the icon in your project that will be shown to the user in both the Toolbox and when the component is dropped onto the package designer. This part of the code will be revisited later in the chapter when the attribute for the User Interface, which you'll be building later, is added.

Now type the following in the code window:

```
public override
```

Once you hit the spacebar after the word "override," you'll see a list of all the methods on the base class. You are now free to type away to your heart's content and code the component.

Once you've done that, though, the component will need to be built and it also needs a few other things to happen to it. If you are a seasoned developer, then this section will probably be old hat to you, but it's important for everybody to understand what needs to happen for the components to work. This is what needs to be covered:

- Provide a strong name key for signing the assembly.
- Set the build output location to the PipelineComponents folder.
- Use a post-build event to install the assembly into the global assembly cache (GAC).
- Set assembly-level attributes in the AssemblyInfo.cs file.

SSIS needs the GAC because it can execute in designer or agent, with different directories. Strong names are a consequence. The PipelineComponents folder allows the designer to discover the component and put it in the Toolbox. Assembly-level stuff is a consequence of the fact that the strong name, with version, is persisted in the package, making all your packages break if you rebuild the component unless you stop incrementing the version.

Probably the best way to go through the first three points is by way of screenshots. You can start by looking at how you sign the project. Right-click on your C# project and choose Properties from the context menu. You are not going to look at all of the tabs on the left-hand side of the screen, but you are going to look at the ones that are relevant to what you're doing here. Figure 14-3 shows the Application tab.

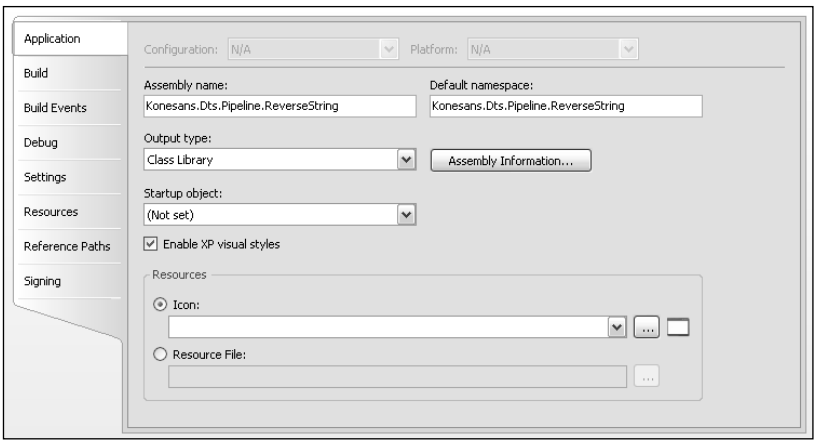


Figure 14-3

In this tab, the only thing you really need to do is change the assembly name to be the same as your default namespace.

On the Build tab, shown in Figure 14-4, you need to be concerned with the output path box toward the bottom of the dialog box. This tells the project that when it builds, the output should be placed in a certain folder. On your PC, this folder is:

```
C:\Program Files\Microsoft SQL Server\90\DTS\PipelineComponents
```

An alternative method of ensuring that the component assembly is automatically installed in the correct folder is to use a copy command to the post-build event command line, covered further below.

```
copy "$(TargetPath)" "%ProgramFiles%\Microsoft SQL
Server\90\DTS\PipelineComponents" /Y
```

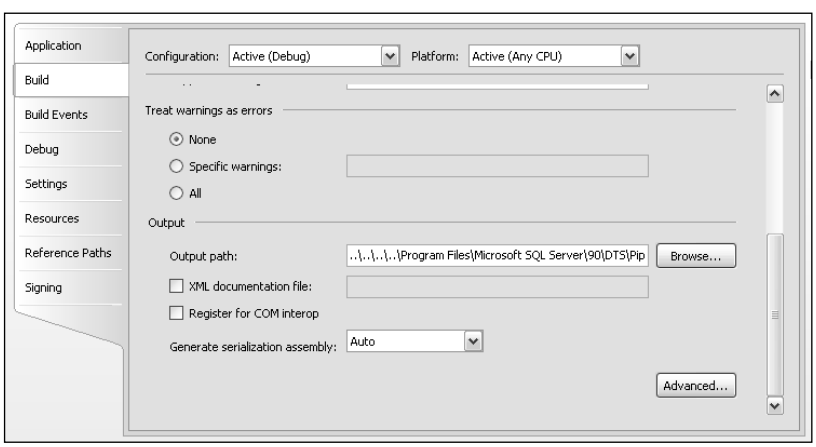


Figure 14-4

For the designer to use a component, it must be placed in a defined folder, and for the runtime engine to work correctly, it must be placed in the global assembly cache. So setting the build location and installing into the GAC are both required steps, which you can do manually, but it makes for faster development if you do it as part of the build process.

Some example build event commands are shown as follows and are illustrated in Figure 14-5:

```
"$(DevEnvDir)\..\..\SDK\v2.0\Bin\gacutil" /if "$(TargetPath)"
```

or

```
"C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin\Gacutil" /if
"$(TargetPath)"
```

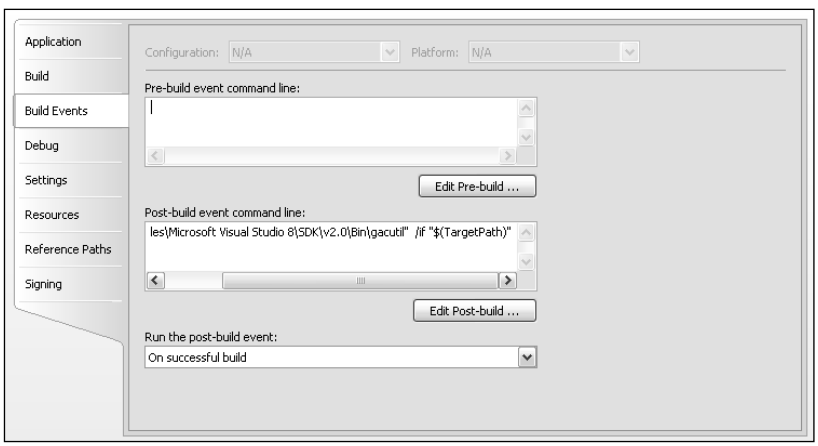


Figure 14-5

Because the assembly is to be installed in the GAC, you need to sign the assembly using a strong name key, which can be specified and created from the Signing page, as shown in Figure 14-6.

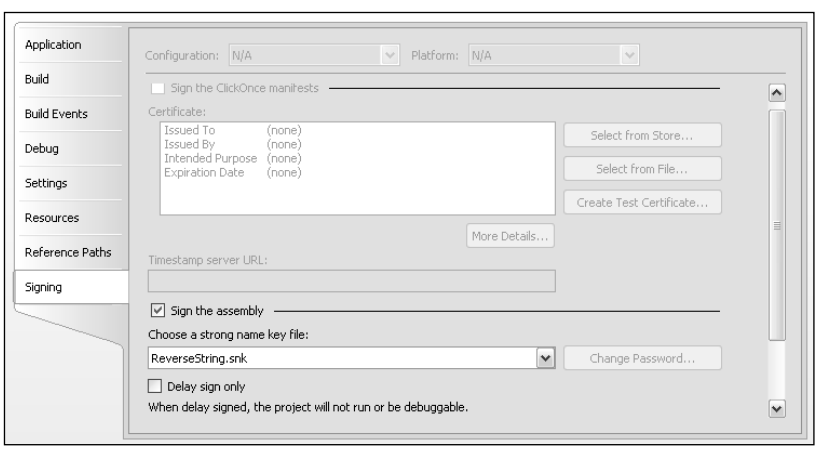


Figure 14-6

That is it as far as the project's properties are concerned, so now you can move on to looking at the AssemblyInfo file. While most assembly attributes can be set through the Assembly Information dialog box, available from the Application tab of Project Properties, shown previously in Figure 14-3, you require some additional settings. Shown below is the AssemblyInfo.cs file for the example project, which can be found under the Properties folder within the Solution Explorer of Visual Studio.

```
#region Using directives
using System;
using System.Security.Permissions;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

#endregion

[assembly: AssemblyTitle("ReverseString")]
[assembly: AssemblyDescription("Reversing String Transformation for SQL Server
Integration Services")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Konesans Ltd")]
[assembly: AssemblyProduct("Reverse String Transformation")]
[assembly: AssemblyCopyright("Copyright (c) 2004-2005 Konesans Ltd")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

[assembly: AssemblyVersion("1.1.0.0")]
[assembly: AssemblyFileVersion("1.1.0")]
[assembly: CLSCompliant(true)]
[assembly: PermissionSet(SecurityAction.RequestMinimum)]
[assembly: ComVisible(false)]
```

The first section of attributes listed represents primarily information, and you would change these to reflect your component and company, for example. The `AssemblyCulture` should be left blank unless you are experienced at working with localized assemblies and understand the implications of any change.

The `AssemblyVersion` attribute is also worth noting; as the version is fixed, it does not use the asterisk token to generate an automatically incrementing build number. The assembly version forms part of the fully qualified assembly name, which is how a package references a component under the covers. So if you changed the version for every build, you would have to rebuild your packages for every new version of the component. So that you can differentiate between versions, you should use `AssemblyFileVersion`, although you will need to manually update this.

The other attribute worth special note is `CLSCompliant`. Best practice dictates that the .Net classes and assemblies conform to the Command Language Specification (CLS), and compliance should be marked at the assembly level. Individual items of noncompliant code can then be decorated with the `CLSCompliant` attribute, marked as false. The completed samples all include this, and you can also refer to SQL Server documentation for guidance, as well as following the simple compiler warnings that are raised when this condition is not met.

Here is an example of how to deal with a method being noncompliant in your component.

```
[CLSCompliant(false)]
public override DTSValidationStatus Validate()
{
...
```

Building the Source Adapter

As mentioned earlier, the source adapter needs to be able to retrieve information from a file and present the data to the downstream component. The file is not your standard-looking file. The format is strange but consistent. When you design the destination adapter, you will write the contents of an upstream component to a file in a very similar format. After you have read this chapter, you may want to take the source adapter and alter it slightly so that it can read a file produced by the destination adapter.

The very first method to look at is `ProvideComponentProperties`. This gets called almost as soon as you drop the component onto the designer. Here is the method in full before you begin to break it down:

```
public override void ProvideComponentProperties()
{
   ComponentMetaData.RuntimeConnectionCollection.RemoveAll();
   RemoveAllInputsOutputsAndCustomProperties();

   ComponentMetaData.Name = "Professional SSIS Source Adapter";
   ComponentMetaData.Description = "Our first Source Adapter";
   ComponentMetaData.ContactInfo = "www.Konesans.com";

   IDTSRuntimeConnection90 rtc =
      ComponentMetaData.RuntimeConnectionCollection.New();
   rtc.Name = "File To Read";
   rtc.Description = "This is the file from which we want to read";

   IDTSOutput90 output = ComponentMetaData.OutputCollection.New();
   output.Name = "Component Output";
   output.Description = "This is what downstream Components will see";

   output.ExternalMetadataColumnCollection.IsUsed = true;

}
```

Now you can break down some of this code.

```
ComponentMetaData.RuntimeConnectionCollection.RemoveAll();
RemoveAllInputsOutputsAndCustomProperties();
```

The very first thing this code does is remove any runtime connections in the component, which you'll be adding back soon. You can also remove inputs, outputs, and custom properties. Basically your component is now a clean slate. This is not strictly required for this example; however, it's advantageous to follow this convention, as it prevents any unexpected situations that may arise in more complicated components.

```
ComponentMetaData.Name = "Professional SSIS Source Adapter";
ComponentMetaData.Description = "Our first Source Adapter";
ComponentMetaData.ContactInfo = "www.Konesans.com";
```

These three lines of code simply help to identify your component when you look in the property pages after adding it to the designer. The only property here that may not be obvious is `ContactInfo`, which simply identifies to the user the developer of the component. If a component throws a fatal error during loading or saving, for example — areas not influenced by the user-controlled settings — then the designer will show the contact information.

```
IDTSRuntimeConnection90 rtc = ComponentMetaData.RuntimeConnectionCollection.New();
rtc.Name = "File To Read";
rtc.Description = "This is the file from which we want to read";
```

Your component needs a runtime connection from which you can read and get the data. You removed any existing connections earlier in the method, so here is where you add it back. Simply give it a name and a description.

```
IDTSOutput90 output = ComponentMetaData.OutputCollection.New();
output.Name = "Component Output";
output.Description = "This is what downstream Components will see";
```

The way downstream components will see the data is to present it to them from an output in this component. Here you add a new output to the output collection and give it a name and a description. The final part of this component is to use `ExternalMetadataColumns`, which will allow you to view the structure of the data source with no connection.

```
output.ExternalMetadataColumnCollection.IsUsed = true;
```

Here, you tell the output you created earlier that it will use `ExternalMetaData` columns.

The next method to look at is the `AcquireConnections` method. In this method, you want to make sure that you have a runtime connection available and that it is the correct type. You then want to retrieve the file name from the file itself. Here is the method in full:

```
public override void AcquireConnections(object transaction)
{

   if (ComponentMetaData.RuntimeConnectionCollection["File To
Read"].ConnectionManager != null)
   {
      ConnectionManager cm =
Microsoft.SqlServer.Dts.Runtime.DtsConvert.ToConnectionManager(
ComponentMetaData.RuntimeConnectionCollection["File To Read"].ConnectionManager);

      if (cm.CreationName != "FILE")
      {
         throw new Exception("The Connection Manager is not a FILE Connection
Manager");
      }
      else
      {
         fil = (Microsoft.SqlServer.Dts.Runtime.DTSFileConnectionUsageType)
cm.Properties["FileUsageType"].GetValue(cm);

         if (_fil != DTSFileConnectionUsageType.FileExists)
         {
```

```
            throw new Exception("The type of FILE connection manager must be an
Existing File");
         }
         else
         {
            _filename = ComponentMetaData.RuntimeConnectionCollection["File To
Read"].ConnectionManager.AcquireConnection(transaction).ToString();
            if (_filename == null || _filename.Length == 0)
            {
               throw new Exception("Nothing returned when grabbing the filename");
            }
         }
      }
   }
}
```

This method covers a lot of ground and is really quite interesting. The first thing you want to do is find out if you can get a Connection Manager from the runtime connection collection of the component. The runtime connection was defined during `ProvideComponentProperties` earlier. If it is null, then the user has not provided a runtime connection.

```
if (ComponentMetaData.RuntimeConnectionCollection["File To Read"].ConnectionManager
!= null)
```

The next line of code is quite cool. What it does is convert the native Connection Manager object to a managed Connection Manager. You need the managed Connection Manager to find out what type it is and the properties.

```
ConnectionManager cm =
Microsoft.SqlServer.Dts.Runtime.DtsConvert.ToConnectionManager(
ComponentMetaData.RuntimeConnectionCollection["File To Read"].ConnectionManager);
```

Once you have the managed Connection Manager, you can start to look at some of its properties and make sure that it is what you want. All Connection Managers have a `CreationName` property. For this component, you want to make sure that the `CreationName` property is `FILE`, as highlighted below.

```
if (cm.CreationName != "FILE")
```

If the Creation Name is not `FILE`, then you send an exception back to the component.

```
throw new Exception("The type of FILE connection manager must be an Existing
File");
```

You've established that a connection has been specified and that it is the right type. The problem with the `FILE` Connection Manager is that it can still be the wrong type of connection. To find out if it is the right type, you will have to look at another of its properties, the `FileUsageType` property. This can return to you one of four values, defined by the `DTSFileConnectionUsageType` enumeration:

- `DTSFileConnectionUsageType.CreateFile` — The file does not yet exist and will be created by the component. If the file does exist, then you can raise an error, although you may also accept this and overwrite the file. Use this type for components that create new files.

- `DTSFileConnectionUsageType.FileExists` — The file exists, and you would be expected to raise an error if this is not the case.
- `DTSFileConnectionUsageType.CreateFolder` — The folder does not yet exist and will be created by the component. If the folder does exist, then you can decide how to handle this situation as with `CreateFile` above.
- `DTSFileConnectionUsageType.FolderExists` — The folder exists, and you would be expected to raise an error if this is not the case.

The type you want to check for in your component is `DTSFileConnectionUsageType.FileExists` and you do that like this, throwing an exception if the type is not what you want:

```
fil = (Microsoft.SqlServer.Dts.Runtime.DTSFileConnectionUsageType)cm.Properties
["FileUsageType"].GetValue(cm);

if (_fil != Microsoft.SqlServer.Dts.Runtime.DTSFileConnectionUsageType.FileExists)
{...}
```

You're nearly done checking your Connection Manager now. At this point, you need the file name so you can retrieve the file later on when you need to read it for data. You do that like this:

```
_filename = ComponentMetaData.RuntimeConnectionCollection
["File To Read"].ConnectionManager.AcquireConnection(transaction).ToString();
```

That concludes the `AcquireConnections` method, so you can now move straight on to the `Validate` method.

```
[CLSCompliant(false)]
public override DTSValidationStatus Validate()
{
   bool pbCancel = false;

   IDTSOutput90 output = ComponentMetaData.OutputCollection["Component Output"];

   if (ComponentMetaData.InputCollection.Count != 0)
   {
      ComponentMetaData.FireError(0, ComponentMetaData.Name, "Unexpected input
found. Source components do not support inputs.", "", 0, out pbCancel);
      return DTSValidationStatus.VS_ISCORRUPT;
   }

   if (ComponentMetaData.RuntimeConnectionCollection["File To
Read"].ConnectionManager == null)
   {
      ComponentMetaData.FireError(0, "Validate", "No Connection Manager
Specified.", "", 0, out pbCancel);
      return DTSValidationStatus.VS_ISBROKEN;
   }

   // Check for Output Columns, if not then force ReinitializeMetaData
   if (ComponentMetaData.OutputCollection["Component
       Output"].OutputColumnCollection.Count == 0)
   {
```

```
      ComponentMetaData.FireError(0, "Validate", "No output columns specified.
Making call to ReinitializeMetaData.", "", 0, out pbCancel);
      return DTSValidationStatus.VS_NEEDSNEWMETADATA;
   }

   //What about if we have output columns but we have no ExternalMetaData
   // columns?  Maybe somebody removed them through code.

   if (DoesEachOutputColumnHaveAMetaDataColumnAndDoDatatypesMatch(output.ID) ==
false)
   {
      ComponentMetaData.FireError(0, "Validate", "Output columns and metadata
columns are out of sync.  Making call to ReinitializeMetaData.", "", 0, out
pbCancel);
      return DTSValidationStatus.VS_NEEDSNEWMETADATA;
   }
   return base.Validate();
}
```

The first thing this method does is check for an input. If it has an input, it raises an error back to the component using the `FireError` method and returns `DTSValidationStatus.VS_ISCORRUPT`. This is a source adapter and there is no place for an input.

```
if (ComponentMetaData.InputCollection.Count != 0)
```

The next thing you do is check that the user has specified a Connection Manager for your component. If not, then you return back to the user a message indicating that a Connection Manager is required. Again, you do this through the `FireError` method. If there is no Connection Manager specified, then you tell the component it is broken. Remember that you do the validation of any Connection Manager that is specified in `AcquireConnections()`.

```
if (ComponentMetaData.RuntimeConnectionCollection["File To Read"].ConnectionManager
== null)
{
   ComponentMetaData.FireError(0, "Validate", "No Connection Manager Specified.",
"", 0, out pbCancel);
   return DTSValidationStatus.VS_ISBROKEN;
}
```

The next thing to do is check to see if the output has any columns. On the initial drop onto the designer, the output will have no columns. If this is the case, the `Validate()` method will return `DTSValidationStatus.VS_NEEDSNEWMETADATA`, which in turn calls `ReinitializeMetaData`. You will see later what happens in that method.

```
if (ComponentMetaData.OutputCollection["Component
Output"].OutputColumnCollection.Count == 0)
{
   ComponentMetaData.FireError(0, "Validate", "No output columns specified.  Making
call to ReinitializeMetaData.", "", 0, out pbCancel);
   return DTSValidationStatus.VS_NEEDSNEWMETADATA;
}
```

So if the output has output columns, then one of the things you want to check for is whether the output columns have an ExternalMetaDataColumn associated with them. You'll recall that in `ProvideComponentProperties` it was stated that you would use an `ExternalMetadataColumnCollection`. So for each output column, you need to make sure that there is an equivalent external metadata column and that the data type properties also match.

```
if (DoesEachOutputColumnHaveAMetaDataColumnAndDoDatatypesMatch(output.ID) == false)
{
   ComponentMetaData.FireError(0, "Validate", "Output columns and metadata columns
are out of sync.  Making call to ReinitializeMetaData.", "", 0, out pbCancel);
   return DTSValidationStatus.VS_NEEDSNEWMETADATA;
}
```

The rather long-named helper method here, `DoesEachOutputColumnHaveAMetaDataColumnAndDoDatatypesMatch`, accepts as a parameter the ID of an output, so you pass in the output's ID. There are two things that this method has to do. First, it has to check that each output column has an ExternalMetadataColumn associated with it, and second, it has to make sure that the two columns have the same column data type properties. Here is the method in full.

```
private bool DoesEachOutputColumnHaveAMetaDataColumnAndDoDatatypesMatch(int
outputID)
{

   IDTSOutput90 output =
ComponentMetaData.OutputCollection.GetObjectByID(outputID);
   IDTSExternalMetadataColumn90 mdc;
   bool rtnVal = true;

   foreach (IDTSOutputColumn90 col in output.OutputColumnCollection)
   {

      if (col.ExternalMetadataColumnID == 0)
      {
         rtnVal = false;
      }
      else
      {
         mdc =
         output.ExternalMetadataColumnCollection[col.ExternalMetadataColumnID];

         if (mdc.DataType != col.DataType || mdc.Length != col.Length ||
mdc.Precision != col.Precision || mdc.Scale != col.Scale || mdc.CodePage !=
col.CodePage)
         {
            rtnVal = false;
         }
      }
   }
   return rtnVal;
}
```

The first thing this method does is to translate the ID passed in as a parameter to the method into an output.

```
IDTSOutput90 output = ComponentMetaData.OutputCollection.GetObjectByID(outputID);
```

Once you have that, the code loops over the output columns in that output and asks if the `ExternalMetadataColumnID` associated with that output column has a value of 0 (that is, there is no value). If the code finds an instance of a value, then it sets the return value from the method to be false.

```
foreach (IDTSOutputColumn90 col in output.OutputColumnCollection)
{

   if (col.ExternalMetadataColumnID == 0)
   {
      rtnVal = false;
   }
...
```

If all output columns have a nonzero `ExternalMetadataColumnID`, then you move on to the second test:

```
mdc = output.ExternalMetadataColumnCollection[col.ExternalMetadataColumnID];

if (mdc.DataType != col.DataType || mdc.Length != col.Length || mdc.Precision !=
col.Precision || mdc.Scale != col.Scale || mdc.CodePage != col.CodePage)
{
   rtnVal = false;
}
```

In this part of the method, you are checking that all attributes of the output column's data type match those of the corresponding `ExternalMetadataColumn`. If they do not, then again you return false from the method, which causes the `Validate()` method to call `ReinitializeMetadata`. Notice how you are using the ID over a Name, since names can be changed by the end user.

`ReinitializeMetaData` is where a lot of the work happens in most components. In this component, it will fix up the output columns and the `ExternalMetadataColumns`. Here's the method:

```
public override void ReinitializeMetaData()
{
   IDTSOutput90 _profoutput = ComponentMetaData.OutputCollection["Component
Output"];

   if (_profoutput.ExternalMetadataColumnCollection.Count > 0)
   {
      _profoutput.ExternalMetadataColumnCollection.RemoveAll();
   }

   if (_profoutput.OutputColumnCollection.Count > 0)
   {
      _profoutput.OutputColumnCollection.RemoveAll();
   }

   CreateOutputAndMetaDataColumns(_profoutput);

}
```

This is a really simple way of doing things. Basically, you are going to remove all the ExternalMetaDataColumns and then remove the output columns. You will then add them back using the `CreateOutputAndMetaDataColumns` method.

As an exercise, you may want to see if you can work out which columns actually need fixing.

`CreateOutputAndMetaDataColumns` creates the output's output columns and the `ExternalMetaData` columns to go with them. This is very rigid, and it presumes that the file you get will be in one format only. There are actually two methods here, and the output you just created is passed in:

```
private void CreateOutputAndMetaDataColumns(IDTSOutput90 output)
{
   IDTSOutputColumn90 outName = output.OutputColumnCollection.New();
   outName.Name = "Name";
   outName.Description = "The Name value retieved from File";
   outName.SetDataTypeProperties(DataType.DT_STR, 50, 0, 0, 1252);
   CreateExternalMetaDataColumn(output.ExternalMetadataColumnCollection, outName);

   IDTSOutputColumn90 outAge = output.OutputColumnCollection.New();
   outAge.Name = "Age";
   outAge.Description = "The Age value retieved from File";
   outAge.SetDataTypeProperties(DataType.DT_I4, 0, 0, 0, 0);

   //Create an external metadata column to go alongside with it
   CreateExternalMetaDataColumn(output.ExternalMetadataColumnCollection, outAge);

   IDTSOutputColumn90 outMarried = output.OutputColumnCollection.New();
   outMarried.Name = "Married";
   outMarried.Description = "The Married value retieved from File";
   outMarried.SetDataTypeProperties(DataType.DT_BOOL, 0, 0, 0, 0);

   //Create an external metadata column to go alongside with it
   CreateExternalMetaDataColumn(output.ExternalMetadataColumnCollection,
      outMarried);

   IDTSOutputColumn90 outSalary = output.OutputColumnCollection.New();
   outSalary.Name = "Salary";
   outSalary.Description = "The Salary value retieved from File";
   outSalary.SetDataTypeProperties(DataType.DT_DECIMAL, 0, 0, 10, 0);

   //Create an external metadata column to go alongside with it
   CreateExternalMetaDataColumn(output.ExternalMetadataColumnCollection,
   outSalary);
}
```

This code follows the same path for every column you want to create, so you'll just look at one example here, as the rest are variations of the same code. In `CreateOutputAndMetaDataColumns`, you first need to create an output column and add it to the `outputcolumncollection` of the output, which is a parameter to the method. You give the column a name, a description, and a data type along with details about the data type.

`SetDataTypeProperties` *takes the name, the length, the precision, the scale, and the code page of that data type. A list of what is required for these fields can be found in Books Online.*

```
IDTSOutputColumn90 outName = output.OutputColumnCollection.New();
outName.Name = "Name";
outName.Description = "The Name value retieved from File";
outName.SetDataTypeProperties(DataType.DT_STR, 50, 0, 0, 1252);
```

You now look to create an `ExternalMetaDataColumn` for the `OutputColumn`, and you do that by calling the method `CreateExternalMetaDataColumn`. This method takes as parameters the `ExternalMetaDataColumnCollection` of the output and the `Column` for which you want to create an `ExternalMetaDataColumn`.

```
CreateExternalMetaDataColumn(output.ExternalMetadataColumnCollection, outName);
```

The first thing you do in the method is create a new `ExternalMetaDataColumn` in the `ExternalMetaDataColumnCollection` that was passed as a parameter. You then map the properties of the output column that was passed as a parameter to the new `ExternalMetaDataColumn`. Finally, you create the relationship between the two by assigning the ID of the `ExternalMetaDataColumn` to the `ExternalMetadataColumnID` property of the output column.

```
IDTSExternalMetadataColumn90 eColumn = externalCollection.New();
eColumn.Name = column.Name;
eColumn.DataType = column.DataType;
eColumn.Precision = column.Precision;
eColumn.Length = column.Length;
eColumn.Scale = column.Scale;
eColumn.CodePage = column.CodePage;
column.ExternalMetadataColumnID = eColumn.ID;
```

At this point, the base class will call the MapOutputColumn method. You can choose to override this method to decide if you want to allow the mapping to occur, but in this case you should choose to leave the base class to simply carry on.

Now you will move on to looking at the runtime methods. `PreExecute` is the usual place to start for most components, but it is done slightly different here. Normally you would enumerate the output columns and enter them into a struct so you could easily retrieve them later. You're not going to do that here, but you do this in the destination adapter, so you could port what you do there into this adapter as well. The only method you are interested in with this adapter is `PrimeOutput`. Here is the method in full:

```
public override void PrimeOutput(int outputs, int[] outputIDs, PipelineBuffer[]
buffers)
{
   ParseTheFileAndAddToBuffer(_filename, buffers[0]);
   buffers[0].SetEndOfRowset();
}
```

On the face of this method, it looks really easy, but as you can see, all the work is being done by the `ParseTheFileAndAddToBuffer` method. To that, you need to pass the file name you retrieved in `AcquireConnections`, and the buffer is `buffers[0]`, because there is only one buffer and the collec-

tion is zero-based. You'll look at the `ParseTheFileAndAddToBuffer` method in a moment, but the last thing you do in this method is call `SetEndOfRowset` on the buffer. This basically tells the downstream component that there are no more rows to be had from the adapter. Now you will look at the `ParseTheFileAndAddToBuffer` method in a bit more detail.

```
private void ParseTheFileAndAddToBuffer(string filename, PipelineBuffer buffer)
{
   TextReader tr = File.OpenText(filename);
   IDTSOutput90 output = ComponentMetaData.OutputCollection["Component Output"];
   IDTSOutputColumnCollection90 cols = output.OutputColumnCollection;
   IDTSOutputColumn90 col;

   string s = tr.ReadLine();
   int i = 0;

   while (s != null)
   {
      if (s.StartsWith("<START>"))
         buffer.AddRow();

      if (s.StartsWith("Name:"))
      {
         col = cols["Name"];
         i = BufferManager.FindColumnByLineageID(output.Buffer, col.LineageID);
         string value = s.Substring(5);
         buffer.SetString(i, value);
      }

      if (s.StartsWith("Age:"))
      {
         col = cols["Age"];
         i = BufferManager.FindColumnByLineageID(output.Buffer, col.LineageID);
         Int32 value;
         if (s.Substring(4).Trim() == "")
            value = 0;
         else
            value = Convert.ToInt32(s.Substring(4).Trim());

         buffer.SetInt32(i, value);
      }

      if (s.StartsWith("Married:"))
      {
         col = cols["Married"];
         bool value;
         i = BufferManager.FindColumnByLineageID(output.Buffer, col.LineageID);
         if (s.Substring(8).Trim() == "")
            value = true;
         else
            value = s.Substring(8).Trim() != "1" ? false : true;

         buffer.SetBoolean(i, value);
      }

      if (s.StartsWith("Salary:"))
```

```
        {
            col = cols["Salary"];
            Decimal value;
            i = BufferManager.FindColumnByLineageID(output.Buffer, col.LineageID);

            if (s.Substring(7).Trim() == "")
                value = 0M;
            else
                value = Convert.ToDecimal(s.Substring(8).Trim());

            buffer.SetDecimal(i, value);
        }
        s = tr.ReadLine();
    }
    tr.Close();
}
```

Because this is not a lesson in C# programming, you will simply describe the points relevant to SSIS programming in this component. You start off by getting references to the output columns collection in the component.

```
IDTSOutput90 output = ComponentMetaData.OutputCollection["Component Output"];
IDTSOutputColumnCollection90 cols = output.OutputColumnCollection;
IDTSOutputColumn90 col;
```

The `IDTSOutputColumn90` object will be used when you need a reference to particular columns. Now the problem with the file is that the columns in the file are in rows, and you need to pivot them into columns. The way to identify in the text file that you need to add a new row to the buffer is if when reading a line of text from the file it begins with the word `<START>`. You do that in this code here (remember that `s` is a line of text from the file):

```
if(s.StartsWith("<START>"))
    buffer.AddRow();
```

As you can see, you have added a row to the buffer. As you read lines in the file, you test the start of each line. This is important because you need to know this in order to be able to grab the right column from the output columns collection and assign it the value from the text file. The first column name you test for is the "Name" column.

```
if (s.StartsWith("Name:"))
{
    col = cols["Name"];
    i = BufferManager.FindColumnByLineageID(output.Buffer, col.LineageID);

    string value = s.Substring(5);
    buffer.SetString(i, value);
}
```

The first thing you do here is to check what the row begins with. In the example above, it is "Name:". Next, you set the `IDTSColumn90` variable column to reference the Name column in the `OutputColumnCollection`. You need to be able to locate the column in the buffer, and to do this you need to look at the Buffer Manager. This has a method called `FindColumnByLineageID`, which returns

the integer location of the column. You need this to assign a value to the column. To this method, you pass the output's buffer and the column's LineageID. Once you have that, you can use the `SetString` method on the buffer object to assign a value to the column by passing in the Buffer column index and the value you want to set the column to. You pretty much do the same with all the columns you want to set values for. The only variation is the method you call on the buffer object. The buffer object has a set`<datatype>` method for each of the possible data types. In this component, you need a `SetInt32`, a `SetBoolean`, and a `SetDecimal` method. They do not differ in structure from the `SetString` method at all.

Building the Transform

In this section, you will build the transform that is going to take data from the upstream source adapter. After reversing the strings, it will pass the data to the downstream component. In this example, the downstream component will be the destination adapter, which you'll be writing right after you're done with the transform. The component will need a few things during its lifetime, so you should have a look at those things now.

```
private ColumnInfo[] _inputColumnInfos;

const string ErrorInvalidUsageType = "Invalid UsageType for column '{0}'";
const string ErrorInvalidDataType = "Invalid DataType for column '{0}'";

CLSCompliant(false)]
public struct ColumnInfo
{
   public int bufferColumnIndex;
   public DTSRowDisposition columnDisposition;
   public int lineageID;
}
```

The structure or struct that you create here, called `ColumnInfo`, is something you use in various guises time and time again in your components. It is really useful for storing details about columns that you will need later in the component. In this component, you will store the `BufferColumnIndex`, which is basically where the column is in the buffer, so that you can retrieve the data. You'll store how the user wants the row to be treated in an error, and you'll also store the column's `LineageID`, which helps to retrieve the column from the `InputColumnCollection`.

Logically, it would make sense to code the component beginning with the design-time, followed by the runtime. The very first thing that happens when your component is dropped into the SSIS package designer surface is that it will make a call to `ProvideComponentProperties`. In this component, you want to set up an input and an output, and you also need to tell your component how it is going to handle data — as in whether it is a synchronous or an asynchronous transformation, as discussed earlier in the chapter. Just as you did with the source adapter, you'll look at the whole method first and then examine parts of the method in greater detail. Here is the method in full:

```
public override void ProvideComponentProperties()
{
   ComponentMetaData.UsesDispositions = true;

   ReverseStringInput = ComponentMetaData.InputCollection.New();
```

```
        ReverseStringInput.Name = "RSin";

        ReverseStringInput.ErrorRowDisposition = DTSRowDisposition.RD_FailComponent;

        ReverseStringOutput = ComponentMetaData.OutputCollection.New();
        ReverseStringOutput.Name = "RSout";

        ReverseStringOutput.SynchronousInputID = ReverseStringInput.ID;

        ReverseStringOutput.ExclusionGroup = 1;

        AddErrorOutput("RSErrors", ReverseStringInput.ID,
                       ReverseStringOutput.ExclusionGroup);


    }
```

Now to break it down. The very first thing you do is to tell the component to use dispositions.

```
ComponentMetaData.UsesDispositions = true;
```

In this case, this tells your component that it can expect an error output. Now you move on to adding an input to the component.

```
// Add a new Input, and name it.
ReverseStringInput = ComponentMetaData.InputCollection.New();
ReverseStringInput.Name = "RSin";

// If an error occurs during data movement, then the component will fail.
ReverseStringInput.ErrorRowDisposition = DTSRowDisposition.RD_FailComponent;

// Add a new Output, and name it.
ReverseStringOutput = ComponentMetaData.OutputCollection.New();
ReverseStringOutput.Name = "RSout";

// Link the Input and Output together for a synchronous behavior
ReverseStringOutput.SynchronousInputID = ReverseStringInput.ID;
```

This isn't too different from adding the input, except that you tell the component that this is a synchronous component by setting the `SynchronousInputID` on the output to the ID of the input you created earlier. If you were creating an asynchronous component, you would set the `SynchronousInputID` of the output to be 0, like this:

```
ReverseStringOutput.SynchronousInputID = 0
```

This tells SSIS to create a buffer for the output that is separate from the input buffer. This is not an asynchronous component, though; you will revisit some of the subtle differences later.

```
AddErrorOutput("RSErrors",
ReverseStringInput.ID,ReverseStringOutput.ExclusionGroup);

ReverseStringOutput.ExclusionGroup = 1;
```

`AddErrorOutput` creates a new output on the component and tags it as being an error output by setting the `IsErrorOut` property to true. To the method, you pass the name of the error output you want, the input's `ID` property, and the output's `ExclusionGroup`. An `ExclusionGroup` is needed when two outputs use the same synchronous input. Setting the exclusion group allows you to direct rows to the correct output later in the component using `DirectRow`.

That's it for `ProvideComponentProperties`. Now you'll move on to the `Validate` method. As mentioned earlier, this method is called on numerous occasions, and it is your opportunity within the component to check whether what has been specified by the user is allowable by the component.

Here is your completed `Validate` method:

```
[CLSCompliant(false)]
public override DTSValidationStatus Validate()
{
   bool Cancel;

   if (ComponentMetaData.AreInputColumnsValid == false)
      return DTSValidationStatus.VS_NEEDSNEWMETADATA;

   foreach (IDTSInputColumn90 inputColumn in
ComponentMetaData.InputCollection[0].InputColumnCollection)
   {
      if (inputColumn.UsageType != DTSUsageType.UT_READWRITE)
      {
         ComponentMetaData.FireError(0, inputColumn.IdentificationString,
String.Format(ErrorInvalidUsageType, inputColumn.Name), "", 0, out Cancel);
         return DTSValidationStatus.VS_ISBROKEN;
      }

      if (inputColumn.DataType != DataType.DT_STR && inputColumn.DataType !=
DataType.DT_WSTR)
      {
         ComponentMetaData.FireError(0, inputColumn.IdentificationString,
String.Format(ErrorInvalidDataType, inputColumn.Name), "", 0, out Cancel);
         return DTSValidationStatus.VS_ISBROKEN;
      }
   }

   return base.Validate();
}
```

This method will return a validation status to indicate the overall result and may cause subsequent methods to be called. Refer to the SQL Server documentation for a complete list of values (see `DTSValidationStatus`).

Now to break down the `Validate` method. A user can easily add and take away an input from the component at any stage and add it back. It may be the same one, or it may be a different one, presenting the component with an issue. When an input is added, the component will store the lineage IDs of the Input columns. If that input is removed and another is added, those lineage IDs may have changed because something like the query used to generate those columns may have changed; therefore you are presented with different columns, so you need to check to see if that has happened and if it has invalidated the lineage IDs. If it has, the component will call `ReinitializeMetaData`.

```
if (ComponentMetaData.AreInputColumnsValid == false)
{ return DTSValidationStatus.VS_NEEDSNEWMETADATA; }
```

The next thing you should check for is that each of the columns in the `InputColumnCollection` chosen for the component has been set to `READ WRITE`. This is because you will be altering them in place. If they are not set to `READ WRITE`, you need to feed that back by returning `VS_ISBROKEN`. You can invoke the `FireError` method on the component, which will result in a red cross on the component along with tooltip text indicating the exact error.

```
if (RSincol.UsageType != DTSUsageType.UT_READWRITE)
{
   ComponentMetaData.FireError(0, inputColumn.IdentificationString,
String.Format(ErrorInvalidUsageType, inputColumn.Name), "", 0, out Cancel);
   return DTSValidationStatus.VS_ISBROKEN;
}
```

The last thing you do in `Validate` is to check that the columns selected for the component have the correct data types.

```
if (inputColumn.DataType != DataType.DT_STR && inputColumn.DataType !=
DataType.DT_WSTR)
...
```

If the data type of the column is not one of those in the list, you again fire an error and set the return value to `VS_ISBROKEN`.

Now you will look at the workhorse method of so many of your components: ReinitializeMetaData. Here is the method in full:

```
public override void ReinitializeMetaData()
{
   if (!ComponentMetaData.AreInputColumnsValid)
   {
      ComponentMetaData.RemoveInvalidInputColumns();
   }

   base.ReinitializeMetaData();
}
```

Remember back in the `Validate` method mentioned earlier that if `Validate` returns `VS_NEEDSNEW METADATA`, then the component internally would call `ReinitializeMetaData`. The only time you do that for this component is when you have detected that the lineage IDs of the input columns are not quite as expected, that is to say, they do not exist on any upstream column and you want to remove them.

```
if (!ComponentMetaData.AreInputColumnsValid)
{
   ComponentMetaData.RemoveInvalidInputColumns();
}
```

You finish off by calling the base class's `ReinitializeMetaData` method as well. This method really can become the workhorse of your component. You can perform all kinds of triage on your component here and try to rescue the component from the user.

The `SetUsageType` method is called when the user is manipulating how the column on the input will be used by the component. In this component, this method validates the data type of the column and whether the user has set the column to be the correct usage type. The method returns an `IDTSInputColumn`, and this is the column being manipulated.

```
[CLSCompliant(false)]
public override IDTSInputColumn90 SetUsageType(int inputID, IDTSVirtualInput90
virtualInput, int lineageID, DTSUsageType usageType)
{
   IDTSVirtualInputColumn90 virtualInputColumn =
virtualInput.VirtualInputColumnCollection.GetVirtualInputColumnByLineageID(
lineageID);

   if (usageType == DTSUsageType.UT_READONLY)
      throw new Exception(String.Format(ErrorInvalidUsageType,
         virtualInputColumn.Name));

   if (usageType == DTSUsageType.UT_READWRITE)
   {
      if (virtualInputColumn.DataType != DataType.DT_STR &&
         virtualInputColumn.DataType != DataType.DT_WSTR)
      {
         throw new Exception(String.Format(ErrorInvalidDataType,
            virtualInputColumn.Name));
      }
   }

   return base.SetUsageType(inputID, virtualInput, lineageID, usageType);
}
```

The first thing the method does is get a reference to the column being changed, from the virtual input, which is the list of all upstream columns available.

You then perform the tests to ensure the column is suitable, before proceeding with the request through the base class. Note that this method looks a lot like the `Validate` method. The only real difference is that the `Validate` method obviously returned a different object but also reported errors back to the component. `Validate` uses the `FireError` method, but `SetUsageType` throws an exception; in `SetUsageType` you are checking against the `VirtualInput`, and in `Validate()` you check against the `Input90`. (We used to use FireError in here also, but we found that it wasn't as predictable on what got bubbled back to the user, and we were advised that the correct way would be to throw a new exception.) These are important, as this is one of the key verification methods you can use, allowing you to validate in real time the change that is made to your component and prevent it if necessary.

The `InsertOutput` method is the next design-time method you'll be looking at, and it is called when a user attempts to add an output to the component. In your component, you want to prohibit that, so if the user tries to add an output, you should throw an exception telling them it is not allowed.

```
[CLSCompliant(false)]
public override IDTSOutput90 InsertOutput(DTSInsertPlacement insertPlacement, int
outputID)
{
   throw new Exception("You cannot insert an output (" +
      outputID.ToString() + ")");
}
```

You do the same when the user tries to add an input to your component in the `InsertInput` method:

```
[CLSCompliant(false)]
public override IDTSInput90 InsertInput(DTSInsertPlacement insertPlacement, int
inputID)
{
   throw new Exception("You cannot insert an output (" +
      outputID.ToString() + ")");
}
```

Notice again how in both methods you throw an exception in order to tell the user that what they requested is not allowed.

If the component were asynchronous, you would need to add columns to the output yourself. There exists a choice of methods in which to do this. If you want to add an output column for every input column selected, then the `SetUsageType` method is probably the best place to do that. This is something about which Books Online agrees. Another method for doing this might be the `OnInputPathAttached`.

The final two methods you'll look at for the design-time methods are the opposite of the previous two. Instead of users trying to add an output or an input to your component, they are trying to remove one of them. You do not want to allow this either, so you can use the `DeleteOutput` and the `DeleteInput` methods to tell them. Here are the methods as implemented in your component.

First the `DeleteInput` Method:

```
[CLSCompliant(false)]
public override void DeleteInput(int inputID)
{
   throw new Exception("You cannot delete an input");
}
```

Now the `DeleteOutput` method:

```
[CLSCompliant(false)]
public override void DeleteOutput(int outputID)
{
   throw new Exception("You cannot delete an ouput");
}
```

That concludes the code for the design-time part of your transformation component. Now you will move on to the runtime methods.

The first runtime method you'll be using is the `PreExecute` method. As mentioned earlier, this is called once in your component's life, and it is where you typically do most of your setup using the struct mentioned at the top of this section. It is the first opportunity you get to access the Buffer Manager, providing access to columns within the buffer, which you will need in `ProcessInput` as well. Keep in mind that you will not be getting a call to `PrimeOutput`, as this is a synchronous component and `PrimeOutput` is not called in a synchronous component. Here is the `PreExecute` method in full:

```
public override void PreExecute()
{
   // Prepare array of column information. Processing requires
```

```
      // lineageID so we can do this once in advance.

      IDTSInput90 input = ComponentMetaData.InputCollection[0];
      _inputColumnInfos = new ColumnInfo[input.InputColumnCollection.Count];

      for (int x = 0; x < input.InputColumnCollection.Count; x++)
      {
         IDTSInputColumn90 column = input.InputColumnCollection[x];
         _inputColumnInfos[x] = new ColumnInfo();
         _inputColumnInfos[x].bufferColumnIndex =
   BufferManager.FindColumnByLineageID(input.Buffer, column.LineageID);
         _inputColumnInfos[x].columnDisposition = column.ErrorRowDisposition;
         _inputColumnInfos[x].lineageID = column.LineageID;
      }
   }
```

The first thing this method does is get a reference to the input collection. The collection is zero-based, and because you have only one input, you have used the indexer and not the name, though you could have used the name as well.

```
IDTSInput90 input = ComponentMetaData.InputCollection[0];
```

At the start of this section was a list of the things your component would need later. This included a struct that you were told you would use in various guises, and it also included an array of these structs. You now need to size the array, which you do next by setting the size of the array to the count of columns in the `InputColumnCollection` for your component.

```
_inputColumnInfos = new ColumnInfo[input.InputColumnCollection.Count];
```

Now you loop through the columns in the `InputColumnCollection`. For each of the columns, you create a new instance of a column and a new instance of the struct.

```
IDTSInputColumn90 column = input.InputColumnCollection[x];
_inputColumnInfos[x] = new ColumnInfo();
```

You then read from the column the details you require and store them in the `ColumnInfo` object. The first thing you want to retrieve is the location of the column in the buffer. You cannot simply do this by the order that you added them to the buffer. Though this would probably work, it is likely to catch you out at some point. You can find the column in the buffer by the use of a method called `FindColumnByLineageID` on the BufferManager object. This method takes the buffer and the LineageID of the column that you wish to find as arguments.

```
_inputColumnInfos[x].bufferColumnIndex =
BufferManager.FindColumnByLineageID(input.Buffer, column.LineageID);
```

You now need only two more details about the input column: the `LineageID` and the `ErrorRowDisposition`. Remember, `ErrorRowDisposition` tells the component how to treat an error.

```
_inputColumnInfos[x].columnDisposition = column.ErrorRowDisposition;
_inputColumnInfos[x].lineageID = column.LineageID;
```

When you start to build your own components, you will see that this method really becomes useful. You can use it to initialize any counters you may need or to open connections to data sources as well as anything else you think of.

The final method you are going to be looking at for this component is `ProcessInput`. Remember, this is a synchronous transform as dictated in `ProvideComponentProperties`, and this is the method in which the data is moved and manipulated. This method contains a lot of information that will help you understand the buffer and what to do with the columns in it when you receive them. It is called once for every buffer passed.

Here is the method in full:

```
public override void ProcessInput(int inputID, PipelineBuffer buffer)
{

   int errorOutputID = -1;
   int errorOutputIndex = -1;
   int GoodOutputId = -1;

   IDTSInput90 inp = ComponentMetaData.InputCollection.GetObjectByID(inputID);


   #region Output IDs
   GetErrorOutputInfo(ref errorOutputID, ref errorOutputIndex);
   // There is an error output defined
   errorOutputID = ComponentMetaData.OutputCollection["RSErrors"].ID;
   GoodOutputId = ComponentMetaData.OutputCollection["ReverseStringOutput"].ID;
   #endregion

   if (!buffer.EndOfRowset)
   {
      while (buffer.NextRow())
      {
         // Check if we have columns to process
         if (_inputColumnInfos.Length == 0)
         {
            // We do not have to have columns. This is a Sync component so the
            // rows will flow through regardless. Could expand Validate to check
            // for columns in the InputColumnCollection
            buffer.DirectRow(GoodOutputId);
         }
         else
         {
            try
            {
               for (int x = 0; x < _inputColumnInfos.Length; x++)
               {
                  ColumnInfo columnInfo = _inputColumnInfos[x];

                  if (!buffer.IsNull(columnInfo.bufferColumnIndex))
                  {
                     // Get value as character array
                     char[] chars =
```

```
buffer.GetString(columnInfo.bufferColumnIndex).ToString().ToCharArray();

                    // Reverse order of characters in array
                    Array.Reverse(chars);

                    // Reassemble reversed value as string
                    StringBuilder outputValue = new StringBuilder(chars.Length);
                    for (int i = 0; i < chars.Length; i++)
                    {
                       outputValue.Append(chars[i]);
                    }

                    // Set output value in buffer
                    buffer.SetString(columnInfo.bufferColumnIndex,
outputValue.ToString());
                 }
              }

              buffer.DirectRow(GoodOutputId);
           }
           catch(Exception ex)
           {
              switch (inp.ErrorRowDisposition)
              {

                 case DTSRowDisposition.RD_RedirectRow:
                    buffer.DirectErrorRow(errorOutputID, 0, buffer.CurrentRow);
                    break;
                 case DTSRowDisposition.RD_FailComponent:
                    throw new Exception("There was an error in your processing " +
ex.Message);
                                 case DTSRowDisposition.RD_IgnoreFailure:
                    buffer.DirectRow(GoodOutputId);
                    break;
              }
           }
        }
     }
  }
}
```

There is a lot going on in this method, so you should break it down to make it more manageable. The first thing you want to do is find out from the component the location of the error output, as shown here:

```
int errorOutputID = -1;
int errorOutputIndex = -1;
int GoodOutputId = -1;

#region Output IDs
GetErrorOutputInfo(ref errorOutputID, ref errorOutputIndex);

errorOutputID = ComponentMetaData.OutputCollection["RSErrors"].ID;
GoodOutputId = ComponentMetaData.OutputCollection["ReverseStringOutput"].ID;
#endregion
```

The method `GetErrorOutput` returns the output ID and the Index of the error output. Remember that you defined the error output in `ProvideComponentProperties`.

Because you could have many inputs to a component, you want to isolate the input for this component. You can do that by finding the output that is passed in to the method.

```
IDTSInput90 inp = ComponentMetaData.InputCollection.GetObjectByID(inputID);
```

You need this because you want to know what to do with the row if you encounter an issue. You gave a default value for the `ErrorRowDisposition` property of the input in `ProvideComponentProperties`, but this can be overridden in the UI.

The next thing you want to do is check that the upstream buffer has not called `SetEndOfRowset`, which would mean that it has no more rows to send. You then loop through the rows in the buffer like this.

```
if (!buffer.EndOfRowset)
{
   while (buffer.NextRow())
...
```

You then check to see if the user asked for any columns to be manipulated. Remember, this is a synchronous component, so all columns and rows are going to flow through even if you do not specify any columns for this component. Therefore, you tell the component that if there are no input columns selected, the row should be passed to the normal output. You do this by looking at the size of the array that holds the collection of `ColumnInfo` struct objects.

```
if (_inputColumnInfos.Length == 0)
{
   buffer.DirectRow(GoodOutputId);
}
```

If the length of the array is not zero, the user has asked the component to perform an operation on the column. In turn, you need to grab each of the ColumnInfo objects from the array so you can look at the data. Here you begin your loop through the columns, and for each column you create a new instance of the `ColumnInfo` struct.

```
for (int x = 0; x < _inputColumnInfos.Length; x++)
{
   ColumnInfo columnInfo = _inputColumnInfos[x];
...
```

You now have a reference to that column and are ready to start manipulating it. You first convert the column's data into an array of chars.

```
char[] chars =
buffer.GetString(columnInfo.bufferColumnIndex).ToString().ToCharArray();
```

The interesting part of this line is the method `Getstring()` on the buffer object. It returns the string data of the column and accepts as an argument the index of the column in the buffer. This is really easy, because you stored that reference earlier in the `PreExecute` method. Now that you have the char array, you can

perform some operations on the data. This code won't be shown in detail here because it is not particular to SSIS, and you will move straight on to where you reassign the changed data back to the column using the `SetString()` method on the buffer.

```
buffer.SetString(columnInfo.bufferColumnIndex, outputValue.ToString());
```

Again this method takes as one of the arguments the index of the column in the buffer. It also takes the string you want to assign to that column. You can see now why it was important to make sure that this column was read/write. If there was no error, you point the row to the good output buffer.

```
buffer.DirectRow(GoodOutputId);
```

If you encounter an error, you want to redirect this row to the correct output or alternatively throw an error. You do that in the `catch` block like this:

```
catch(Exception ex)
{
   switch (inp.ErrorRowDisposition)
   {

      case DTSRowDisposition.RD_RedirectRow:
         buffer.DirectErrorRow(errorOutputID, 0, buffer.CurrentRow);
         break;
      case DTSRowDisposition.RD_FailComponent:
         throw new Exception("There was an error in your processing " +
ex.Message);
      case DTSRowDisposition.RD_IgnoreFailure:
         buffer.DirectRow(GoodOutputId);
         break;
   }
}
```

The code is pretty self-explanatory. If the input was told to redirect the row to the error output, then you do that. If it was told to either fail the component, or the user did not specify anything, then you throw an exception. Otherwise the user is asked just to ignore the errors and allow the error row to flow down the normal output.

Now how could this have looked had it been an asynchronous transform? You would get a buffer from both `PrimeOutput` and `ProcessInput`. `ProcessInput` would contain the data and structure that came into the component, and `PrimeOutput` would contain the structure that the component expects to pass on. The trick here is to get the data from one buffer into the other. Here is one way you can approach it.

At the class level, create a variable of type `PipelineBuffer`, something like this:

```
PipelineBuffer _pipelinebuffer;
```

Now in PrimeOutput, assign the output buffer to this buffer:

```
public override void PrimeOutput(int outputs, int[] outputIDs, PipelineBuffer[]
buffers)
{
   _pipelinebuffer = buffers[0];
}
```

You now have a cached version of the buffer from `PrimeOutput`, and you go straight over to `ProcessInput`. Books Online has a great example of doing this in an asynchronous component: navigate to "asynchronous outputs."

> *Do not be afraid to look through Books Online. Microsoft has done a fantastic job of including content that helps with good, solid examples.*

Building the Destination Adapter

The requirement for the destination adapter is that it accepts an input from an upstream component of any description and converts it to a format similar to that seen in the source adapter. The component will accept a `FILE` Connection Manager, and as you have seen in earlier components, this involves a significant amount of validation. You also need to validate whether the component is structurally correct, and if it isn't, you need to correct things. The first thing you always need to do is declare some variables that will be used throughout the component. You also need to create the very valuable struct that is going to store the details of the columns, which will be needed in `PreExecute` and `ProcessInput`.

```
#region Variables
private ArrayList _columnInfos = new ArrayList();
private Microsoft.SqlServer.Dts.Runtime.DTSFileConnectionUsageType _fil;
private string _filename;
FileStream _fs;
StreamWriter _sw;
#endregion
```

You should quickly run through the meaning of these variables and when they will be needed. The `_columnInfos` variable will be used to store `ColumnInfo` objects, which describe the columns in the `InputColumnCollection`. The `_fil` variable will be used to validate the type of `FILE` Connection Manager the user has assigned to your component. `_filename` stores the name of the file that is retrieved from the `FILE` Connection Manager. The final two variables, `_fs` and `_sw`, are used when you write to the text file in `ProcessInput`. Now take a look at the `ColumnInfo` struct.

```
#region ColumnInfo
private struct ColumnInfo
{
   public int BufferColumnIndex;
   public string ColumnName;
}
#endregion
```

The struct will be used to store the index number of the column in the buffer and also to store the name of the column.

You will now move on to looking at the `ProvideComponentProperties` method, which is where you set up the component and prepare it for use by an SSIS package, as in the other two components. Here's the method in full.

```
public override void ProvideComponentProperties()
{

   ComponentMetaData.RuntimeConnectionCollection.RemoveAll();
```

```
    RemoveAllInputsOutputsAndCustomProperties();

    ComponentMetaData.Name = "Professional SSIS Destination Adapter";
    ComponentMetaData.Description = "Our first Destination Adapter";
    ComponentMetaData.ContactInfo = "www.Konesans.com";

    IDTSRuntimeConnection90 rtc =
ComponentMetaData.RuntimeConnectionCollection.New();
    rtc.Name = "File To Write";
    rtc.Description = "This is the file to which we want to write";

    IDTSInput90 input = ComponentMetaData.InputCollection.New();
    input.Name = "Component Input";
    input.Description = "This is what we see from the upstream component";
    input.HasSideEffects = true;
}
```

The first part of the method gets rid of any runtime Connection Managers that the component may have and removes any custom properties, inputs, and outputs that the component has. This makes the component a clean slate to which you can now add back anything it may need.

```
ComponentMetaData.RuntimeConnectionCollection.RemoveAll();
RemoveAllInputsOutputsAndCustomProperties();
```

The component requires one connection, as defined as follows:

```
IDTSRuntimeConnection90 rtc = ComponentMetaData.RuntimeConnectionCollection.New();
rtc.Name = "File To Write";
rtc.Description = "This is the file to which we want to write";
```

This piece of code gives the user the opportunity to specify a Connection Manager for the component. This will be the file to which you write the data from upstream.

```
IDTSInput90 input = ComponentMetaData.InputCollection.New();
input.Name = "Component Input";
input.Description = "This is what we see from the upstream component";
```

The next thing you do is add back the input. This is what the upstream component will connect to, and through which you will receive the data from the previous component. Now you need to make sure that the IDTSInput90 object of the component remains in the execution plan regardless of whether it is attached by making the `HasSideEffects` property true.

```
input.HasSideEffects = true;
```

Having finished with the `ProvideComponentProperties` method, you now move on to the `AcquireConnections` method. This method is not really any different from the `AcquireConnections` method you saw in the source adapter; the method is shown in full but is not described in detail. If you need to get the line-by-line details of what's happening, you can look back to the source adapter. The tasks this method accomplishes are the following:

- Check that the user has supplied a Connection Manager to the component
- Check that the Connection Manager is a `FILE` Connection Manager

- Make sure that the `FILE` Connection Manager has a `FileUsageType` property value of `DTSFileConnectionUsageType.FileExists`
- Get the file name from the Connection Manager

```
public override void AcquireConnections(object transaction)
{
   bool pbCancel = false;

   if (ComponentMetaData.RuntimeConnectionCollection["File To
Write"].ConnectionManager != null)
   {
      ConnectionManager cm =
Microsoft.SqlServer.Dts.Runtime.DtsConvert.ToConnectionManager(
ComponentMetaData.RuntimeConnectionCollection["File To Write"].ConnectionManager);

      if (cm.CreationName != "FILE")
      {
         ComponentMetaData.FireError(0, "Acquire Connections", "The Connection
Manager is not a FILE Connection Manager", "", 0, out pbCancel);
         throw new Exception("The Connection Manager is not a FILE Connection
Manager");
      }
      else
      {
         _fil =
(DTSFileConnectionUsageType)cm.Properties["FileUsageType"].GetValue(cm);

         if (_fil != DTSFileConnectionUsageType.FileExists)
         {
            ComponentMetaData.FireError(0, "Acquire Connections", "The type of FILE
connection manager must be an Existing File", "", 0, out pbCancel);
            throw new Exception("The type of FILE connection manager must be an
Existing File");

         }
         else
         {
            _filename = ComponentMetaData.RuntimeConnectionCollection["File To
Read"].ConnectionManager.AcquireConnection(transaction).ToString();

            if (_filename == null || _filename.Length == 0)
            {
               ComponentMetaData.FireError(0, "Acquire Connections", "Nothing
returned when grabbing the filename", "", 0, out pbCancel);
               throw new Exception("Nothing returned when grabbing the filename");
            }
         }
      }
   }
}
```

There is a lot of ground covered in the `AcquireConnections` method. A lot of this code is covered again in the `Validate` method, which you will visit now. The `Validate` method is also concerned that the input to the component is correct, and if it isn't, you'll try to fix what is wrong by calling `ReinitializeMetaData`. Here is the `Validate` method:

```
[CLSCompliant(false)]
public override DTSValidationStatus Validate()
{
   bool pbCancel = false;

   if (ComponentMetaData.OutputCollection.Count != 0)
   {
      ComponentMetaData.FireError(0, ComponentMetaData.Name, "Unexpected Output
found. Destination components do not support outputs.", "", 0, out pbCancel);
      return DTSValidationStatus.VS_ISCORRUPT;
   }

   if (ComponentMetaData.RuntimeConnectionCollection["File To
Write"].ConnectionManager == null)
   {
      ComponentMetaData.FireError(0, "Validate", "No Connection Manager returned",
"", 0, out pbCancel);
      return DTSValidationStatus.VS_ISCORRUPT;
   }

   if (ComponentMetaData.AreInputColumnsValid == false)
   {
      ComponentMetaData.InputCollection["Component
Input"].InputColumnCollection.RemoveAll();
      return DTSValidationStatus.VS_NEEDSNEWMETADATA;
   }

   return base.Validate();
}
```

The first check you do in the method is to make sure that the component has no outputs:

```
bool pbCancel = false;

if (ComponentMetaData.OutputCollection.Count != 0)
{
   ComponentMetaData.FireError(0, ComponentMetaData.Name, "Unexpected Output found.
Destination components do not support outputs.", "", 0, out pbCancel);
   return DTSValidationStatus.VS_ISCORRUPT;
}
```

You now want to check to make sure the user specified a Connection Manager. Remember that you are only validating the fact that a Connection Manager is specified, not whether it is a valid type. The extensive checking of the Connection Manager is done in `AcquireConnections()`.

```
if (ComponentMetaData.RuntimeConnectionCollection["File To
Write"].ConnectionManager == null)
{
   ComponentMetaData.FireError(0, "Validate", "No Connection Manager returned", "",
0, out pbCancel);
   return DTSValidationStatus.VS_ISCORRUPT;
}
```

The final thing you do in this method is to check that the input columns are valid. *Valid* in this instance means that the columns in the input collection reference existing columns in the upstream component. If this is not the case, you call the trusty `ReinitializeMetadata` method.

```
if (ComponentMetaData.AreInputColumnsValid == false)
{
   ComponentMetaData.InputCollection["Component
Input"].InputColumnCollection.RemoveAll();
   return DTSValidationStatus.VS_NEEDSNEWMETADATA;
}
```

The return value `DTSValidationStatus.VS_NEEDSNEWMETADATA` means that the component will now call `ReinitializeMetaData` to try to sort out the problems with the component. Here is that method in full.

```
public override void ReinitializeMetaData()
{
   IDTSInput90 _profinput = ComponentMetaData.InputCollection["Component Input"];
   _profinput.InputColumnCollection.RemoveAll();
   IDTSVirtualInput90 vInput = _profinput.GetVirtualInput();
   foreach (IDTSVirtualInputColumn90 vCol in vInput.VirtualInputColumnCollection)
   {
      this.SetUsageType(_profinput.ID, vInput, vCol.LineageID,
DTSUsageType.UT_READONLY);

   }
}
```

You will notice that the columns are blown away in ReinitializemetaData and built again from scratch. A better solution is to test what the invalid columns are and try to fix them. If you cannot fix them, you could remove them and then the user could reselect at leisure. Books Online has an example of doing this.

The `IDTSVirtualInput` and `IDTSVirtualInputColumnCollection` in this component need a little explanation. There is a subtle difference between these two objects and their input equivalents. The "virtual" objects are what your component could have as inputs — that is to say, they are upstream inputs and columns that present themselves as available to your component. The inputs themselves are what you have chosen for your component to have as inputs from the virtual object. In the `Reinitialize` method, you start by removing all existing input columns.

```
IDTSInput90 _profinput = ComponentMetaData.InputCollection["Component Input"];
_profinput.InputColumnCollection.RemoveAll();
```

You then get a reference to the input's virtual input.

```
IDTSVirtualInput90 vInput = _profinput.GetVirtualInput();
```

Now that you have the virtual input, you can add an input column to the component for every virtual input column you find.

```
foreach (IDTSVirtualInputColumn90 vCol in vInput.VirtualInputColumnCollection)
{
   this.SetUsageType(_profinput.ID, vInput, vCol.LineageID,
DTSUsageType.UT_READONLY);
}
```

The `SetUsageType` method simply adds an input column to the input column collection of the component or removes it depending on what your UsageType value is. When a user adds a connector from an upstream component that contains its output to this component and attaches it to this component's input, then the `OnInputAttached` is called. This method has been overridden in the component herein:

```
public override void OnInputPathAttached(int inputID)
{
   IDTSInput90 input = ComponentMetaData.InputCollection.GetObjectByID(inputID);
   IDTSVirtualInput90 vInput = input.GetVirtualInput();
   foreach (IDTSVirtualInputColumn90 vCol in vInput.VirtualInputColumnCollection)
   {
      this.SetUsageType(inputID, vInput, vCol.LineageID, DTSUsageType.UT_READONLY);
   }
}
```

This method is the same as the `ReinitializeMetaData` method except that you do not need to remove the input columns from the collection. This is because if the input is not mapped to the output of an upstream component, there can be no input columns.

You have now finished with the design-time methods for your component and can now move on to look at the runtime methods. You are going to be looking at only two methods: `PreExecute` and `ProcessInput`.

PreExecute is executed once and once only in this component, so you want to do as much preparation work as you can in this method. It is also the first opportunity in the component to access the Buffer Manager, which contains the columns. In this component, you use it for two things: getting the information about the component's input columns and storing essential details about them.

```
public override void PreExecute()
{
   IDTSInput90 input = ComponentMetaData.InputCollection["Component Input"];

   foreach (IDTSInputColumn90 inCol in input.InputColumnCollection)
   {
      ColumnInfo ci = new ColumnInfo();
      ci.BufferColumnIndex = BufferManager.FindColumnByLineageID(input.Buffer,
inCol.LineageID);
      ci.ColumnName = inCol.Name;
      _columnInfos.Add(ci);
   }

   // Open the file
   _fs = new FileStream(_filename, FileMode.Open, FileAccess.Write);
   _sw = new StreamWriter(_fs);
}
```

First, you get a reference to the component's input.

```
IDTSInput90 input = ComponentMetaData.InputCollection["Component Input"];
```

You now loop through the input's `InputColumnCollection`

```
foreach (IDTSInputColumn90 inCol in input.InputColumnCollection)
{
```

For each input column you find, you need to create a new instance of the `ColumnInfo` struct. You then assign to the struct values you can retrieve from the input column itself as well as the Buffer Manager. You assign these values to the struct and finally add them to the array that is holding all the `ColumnInfo` objects.

```
ColumnInfo ci = new ColumnInfo();
ci.BufferColumnIndex = BufferManager.FindColumnByLineageID(input.Buffer,
inCol.LineageID);
ci.ColumnName = inCol.Name;
_columnInfos.Add(ci);
```

Doing things this way will allow you to move more quickly through the `ProcessInput` method. The last thing you do in the `PreExecute` method is to get a reference to the file you want to write to.

```
_fs = new FileStream(_filename, FileMode.Open, FileAccess.Write);
_sw = new StreamWriter(_fs);
```

You will use this in the next method, `ProcessInput`. `ProcessInput` is where you are going to keep reading the rows that are coming from the upstream component. While there are rows, you will write those values to a file. This is a very simplistic view of what needs to be done, so you should have a look at how to make that happen.

```
public override void ProcessInput(int inputID, PipelineBuffer buffer)
{

   if (!buffer.EndOfRowset)
   {
      while (buffer.NextRow())
      {
         _sw.WriteLine("<START>");
         for (int i = 0; i < _columnInfos.Count; i++)
         {
            ColumnInfo ci = (ColumnInfo)_columnInfos[i];
            object o = buffer[ci.BufferColumnIndex];

            if (o == null)
            {
               _sw.WriteLine(ci.ColumnName + ":");
            }
            else
            {
            _sw.WriteLine(ci.ColumnName + ":" +
               buffer[ci.BufferColumnIndex].ToString());
            }
         }
         _sw.WriteLine("<END>");
      }
   }

   _sw.Close();
}
```

The first thing you do is check that the upstream component has not called `SetEndOfRowset`. You then check that there are still rows in the buffer.

```
if (!buffer.EndOfRowset)
{
   while (buffer.NextRow())
   {
...
```

You now need to loop through the array that is holding the collection of `ColumnInfo` objects that were populated in the `preExecute` method.

```
for (int i = 0; i < _columnInfos.Count; i++)
```

For each iteration, you create a new instance of the `ColumnInfo` object:

```
ColumnInfo ci = (ColumnInfo)_columnInfos[i];
```

You now need to retrieve from the buffer object the value of the column whose index you will pass in from the `ColumnInfo` object.

```
object o = buffer[ci.BufferColumnIndex];
```

If the value is not null, you write the value of the column and the column name to the text file. If the value is null, you write just the column name to the text file. Again, because you took the time to store these details in a `ColumnInfo` object earlier, the retrieval of these properties is easy.

```
if (o == null)
{
   _sw.WriteLine(ci.ColumnName + ":");
}
else
{
   _sw.WriteLine(ci.ColumnName + ":" + buffer[ci.BufferColumnIndex].ToString());
}
```

That concludes your look at the destination adapter. You are now going to look at how you get SSIS to recognize your components and what properties you need to assign to your components.

Debugging Components

Debugging components is a really great feature of SSIS. If you are a Visual Studio .NET developer, you should easily recognize the interface. If you're not familiar with Visual Studio, hopefully this section will allow you to become proficient in debugging your components.

There are two phases for debugging. The design-time can be debugged only while you're developing your package, so it makes sense that you will need to use BIDS to do this. The second experience, which is the runtime experience, is slightly different. You can still use BIDS, though, and when your package runs, the component will stop at breakpoints you designate. You need to set up a few things first, though. You can also use DTExec to fire the package straight from Visual Studio. The latter method saves you the cost of invoking another instance of Visual Studio.

The component you are going to debug is the Reverse String transform.

Design-Time

You will now jump straight in and start to debug the component at design-time. Open the component's design project and set a breakpoint at ProvideComponentProperties (breakpoints are discussed in Chapter 13). Now create a new SSIS project in BIDS. In the package, add a Data Flow task and double-click on it. If your component is not in the Toolbox already, add it now.

> *To add a component to the Toolbox, right-click on the Toolbox and select Choose Items from the context menu. When the Choose Toolbox Items dialog box appears, click the SSIS Data Flow Items tab and scroll down until you see the component. Check your new component and click OK. When you go back to the Toolbox, you should see your new component.*

You need to create a full pipeline in this package because you'll be using it later on when you debug the runtime, so get a Connection Manager and point it to the AdventureWorks database. Now add a source adapter to the design surface and configure it to use the Connection Manager you just created. It's now time to add your component to the designer. However, before you do that, you need to tell the component's design project to attach to the devenv.exe process you're working in so that it can receive the component's methods being fired. The way you do that is as follows. In the design project, select Tools ➪ Attach to Process. The Attach to Process dialog box opens (see Figure 14-7), which allows you to choose what you want to debug as well as which process.

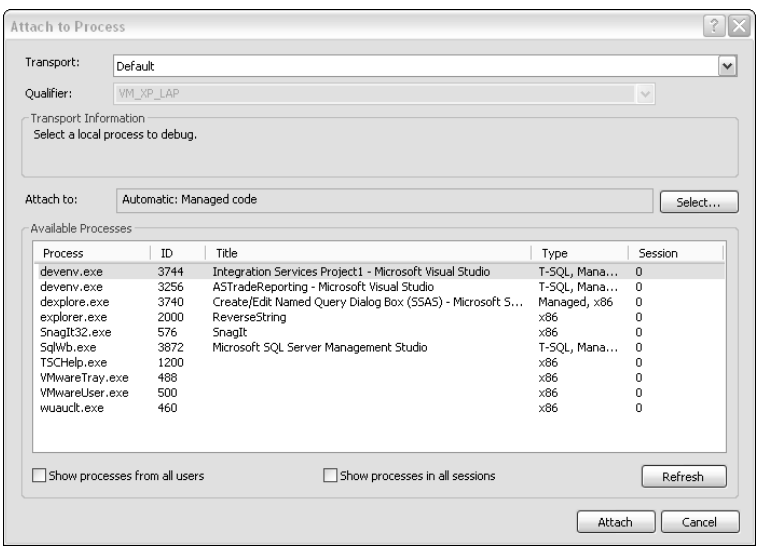
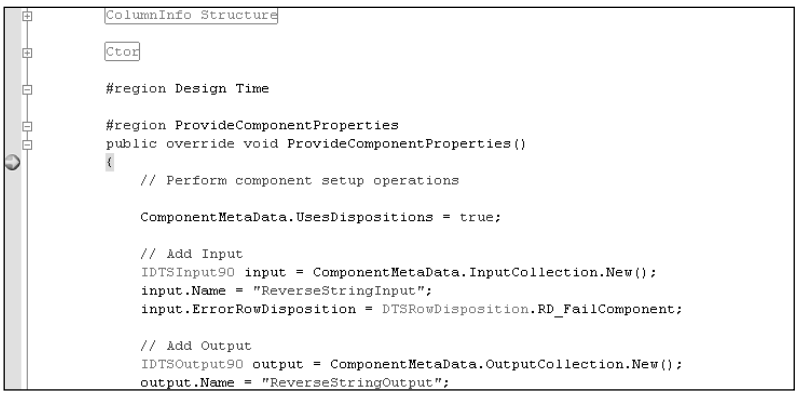


Figure 14-7

The process you're interested in is the package you're currently building. This shows up in the Available Processes list as Integration Services Project 1 – Microsoft Visual Studio (the name you see may differ). You can see just above this window a small box containing the words "Managed Code." This tells the debugger what you want to debug in the component. There are a number of options available, and if you click the Select button to the right of the label, you'll be able to see them. They are Managed, Native, and Script.

Highlight the process for your package and click Attach. If you look down now at the status bar in your component's design project, you should see a variety of debug symbols being loaded. Go back to the SSIS package and drop the ReverseString transform onto the design surface. Because one of the very first things a component does when it gets dropped into a package is call `ProvideComponentProperties`, you should immediately see your component break into the code in its design project, as shown in Figure 14-8.



```
ColumnInfo Structure

Ctor

#region Design Time

#region ProvideComponentProperties
public override void ProvideComponentProperties()
{
    // Perform component setup operations

    ComponentMetaData.UsesDispositions = true;

    // Add Input
    IDTSInput90 input = ComponentMetaData.InputCollection.New();
    input.Name = "ReverseStringInput";
    input.ErrorRowDisposition = DTSRowDisposition.RD_FailComponent;

    // Add Output
    IDTSOutput90 output = ComponentMetaData.OutputCollection.New();
    output.Name = "ReverseStringOutput";
```

Figure 14-8

As you can see, the breakpoint on ProvideComponentProperties in the component's design project has been hit. This is indicated by a yellow arrow inside the breakpoint red circle. You are now free to debug the component as you would with any other piece of managed code in Visual Studio.NET. If you're familiar with debugging, a number of windows appear at this point at the bottom of the IDE, things like "Locals," "Autos," and "Call Stack." These can help you get to the root of your problem, but you do not use them here. Don't be afraid of them.

Runtime

As promised, in this section you are going to look at two ways of debugging. As with design-time debugging, the first is through the BIDS designer. The other is by using DTExec and the package properties. Using BIDS is similar to the design-time method with a subtle variation.

You should now have a complete pipeline with the ReverseString transform in the middle. If you don't, quickly make up a pipeline like in Figure 14-9.

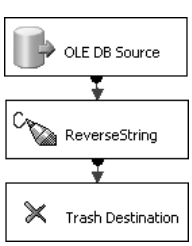


Figure 14-9

The Trash Destination you see terminating this pipeline is a simple destination adapter that requires no setup at all. Just drop it onto the design-sheet and go. It is used as a development aid, when you wish to test a partially completed workflow and you need a destination to quickly terminate the flow. The component is freely available for download from www.konesans.com or www.sqlis.com.

You then need to add a breakpoint to the Data Flow task that is hit when the Data Flow task hits the `OnPreExecute` event. You need to do this so that you can attach your debugger to the correct process at runtime. Right-click on the task itself and select Edit Breakpoints. The Set Breakpoints dialog box will appear, as shown in Figure 14-10.

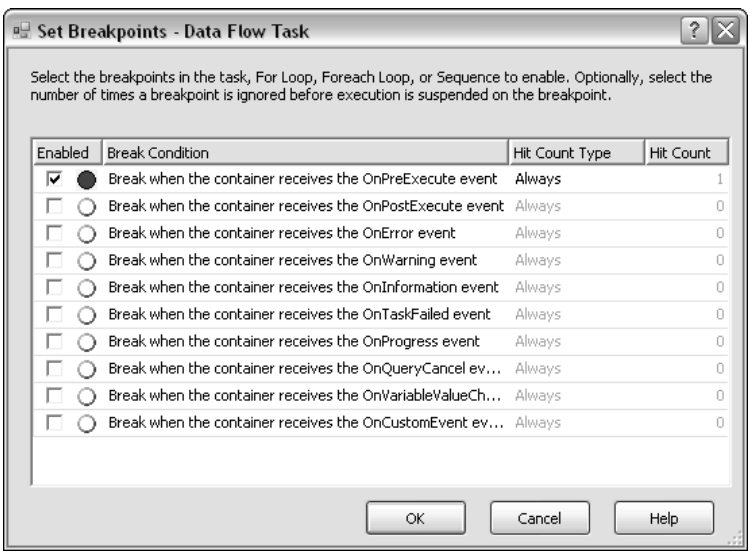


Figure 14-10

You are now ready to execute your package. Hit F5 and allow the breakpoint in the Data Flow task to be hit. When you hit the breakpoint, switch back to the component's design process and follow the steps detailed earlier when debugging the design-time in order to get to the screen where you chose what process to debug.

When you execute a package in the designer, it is not really the designer that is doing the work. It hands off the execution to a process called DtsDebugHost.exe. This is the package that you want to attach to, as shown in Figure 14-11.

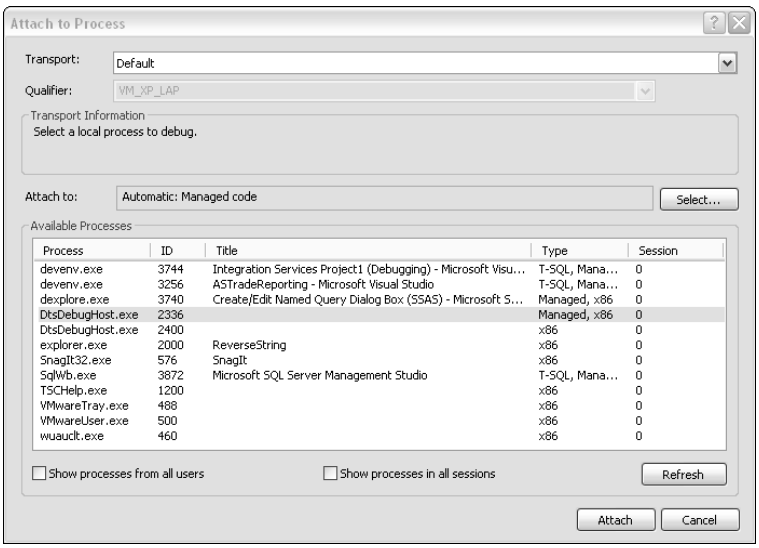


Figure 14-11

Click Attach and watch the debug symbols being loaded by the project. Before returning to the SSIS package, you need to set a breakpoint on one of the runtime methods used by your component, such as PreExecute. Now return to the SSIS project and press F5 again. This will release the package from its suspended state and allow the package to flow on. Now when the ReverseString component hits its `PreExecute` method, you should be able to debug what it is doing. In Figure 14-12, the user is checking to make sure that the LineageID of a column is being retrieved correctly and is ready to be used in the ProcessInput method that follows.

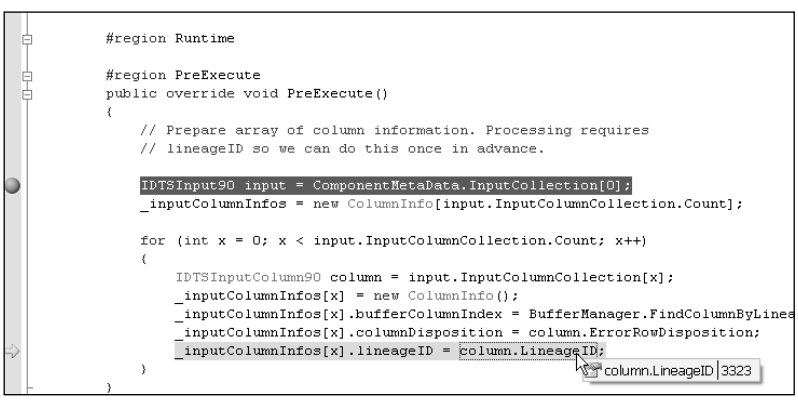


Figure 14-12

That concludes your look at the first method for debugging the runtime. The second method involves BIDS indirectly because you need to create a package like this one that you can call later. After that, you do not need BIDS at all. You do, however, still need the component's design project open. Open your project's properties and look at the Debug tab on the left, which should look similar to Figure 14-13.

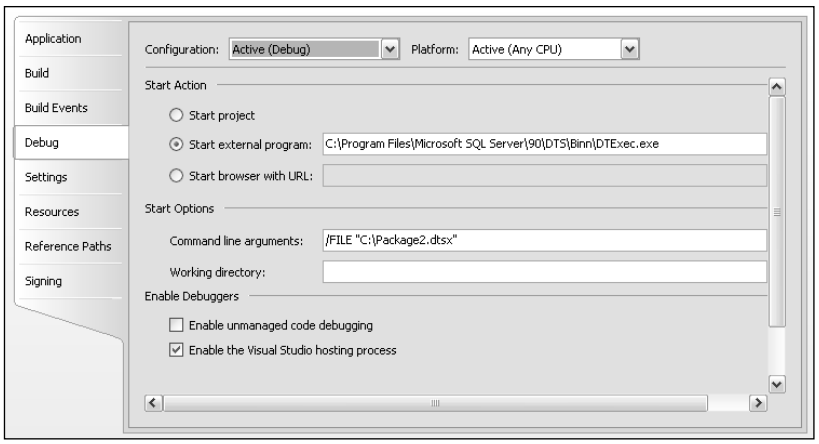


Figure 14-13

As you can see, you have said that you want to start an external program to debug. That program is DTExec, which is the new and more powerful version of DTSRun. On the command line, you will pass a parameter /FILE to DTExec. This tells DTExec the name and location of the package you just built. Make sure you still have a breakpoint set on PreExecute, and hit F5in your project. A DOS window will appear and you will see some messages fly past, which are the same messages you would see in the designer. Eventually you will get to your breakpoint, and it will break in exactly the same way as it did when you were using BIDS. So, why might you use one over the other? The most obvious answer is speed. It is much faster to get to where you want to debug your component using DTExec than it is doing the same in BIDS. The other advantage is that you do not need to have two tools open at the same time. You can focus on your component's design project and not have to worry about BIDS at all.

Summary

In this chapter, you have built pipeline components, but at the start, other types of components that you can build were alluded to. Although designing your own components isn't exactly like falling off a log, once you get a handle on what methods do what, when they are called, and what you can possibly use them for, you can certainly create them with only a moderate amount of knowledge in programming. The components you have designed are certainly very simple in nature, but hopefully this chapter will give you the confidence to experiment with some of your own unique requirements.