CHAPTER 3

Developing CLR Database Objects

IN THIS CHAPTER

Understanding CLR and SQL Server 2005 Database Engine
Creating CLR Database Objects
Debugging CLR Database Objects

The integration of the .NET Framework's Common Language Runtime (CLR) with SQL Server 2005 is arguably the most significant new development featured in the SQL Server 2005 release. The integration of the CLR brings with it a whole host of new capabilities, including the capability to create database objects using any of the .NET-compatible languages, including C#, Visual Basic, and managed C++. In this chapter you'll learn about how Microsoft has implemented the new .NET CLR integration with SQL Server as well as see how to create CLR database objects.

Understanding CLR and SQL Server 2005 Database Engine

The integration of the CLR with SQL Server extends the capability of SQL Server in several important ways. While T-SQL, the existing data access and manipulation language, is well suited for set-oriented data access operations, it also has limitations. Designed more than a decade ago, T-SQL is a procedural language, not an object-oriented language. The integration of the CLR with SQL Server 2005 brings with it the ability to create database objects using modern object-oriented languages like VB.NET and C#. While these languages do not have the same strong set-oriented nature as T-SQL, they do support complex logic, have better computation capabilities, provide access to external resources, facilitate code reuse, and have a first-class development environment that provides much more power than the old Query Analyzer.

The integration of the .NET CLR with SQL Server 2005 enables the development of stored procedures, user-defined functions, triggers, aggregates, and user-defined types using any of the .NET languages. The integration of the .NET CLR with SQL Server 2005 is more than just skin deep. In fact, the SQL Server 2005 database engine hosts the CLR in-process. Using a set of APIs, the SQL Server engine performs all of the memory management for hosted CLR programs.

The managed code accesses the database using ADO.NET in conjunction with the new SQL Server .NET Data Provider. A new SQL Server object called an *assembly* is the unit of deployment for .NET objects with the database. To create CLR database objects, you must first create a DLL using Visual Studio 2005. Then you import that DLL into SQL Server as an assembly. Finally, you link that assembly to a database object such as a stored procedure or a trigger. In the next section you'll get a more detailed look at how you actually use the new CLR features found in SQL Server 2005.

CLR Architecture

The .NET Framework CLR is very tightly integrated with the SQL Server 2005 database engine. In fact, the SQL Server database engine hosts the CLR. This tight level of integration gives SQL Server 2005 several distinct advantages over the .NET integration that's provided by DB2 and Oracle. You can see an overview of the SQL Server 2005 database engine and CLR integration in Figure 3-1.

As you can see in Figure 3-1, the CLR is hosted within the SQL Server database engine. A SQL Server database uses a special API or hosting layer to communicate with the CLR and interface the CLR with the Windows operating system.

Hosting the CLR within the SQL Server database gives the SQL Server database engine the ability to control several important aspects of the CLR, including

- Memory management
- Threading
- Garbage collection

The DB2 and Oracle implementation both use the CLR as an external process, which means that the CLR and the database engine both compete for system resources. SQL Server 2005's in-process hosting of the CLR provides several important advantages over the external implementation used by Oracle or DB2. First, in-process hosting enables SQL Server to control the execution of the CLR, putting

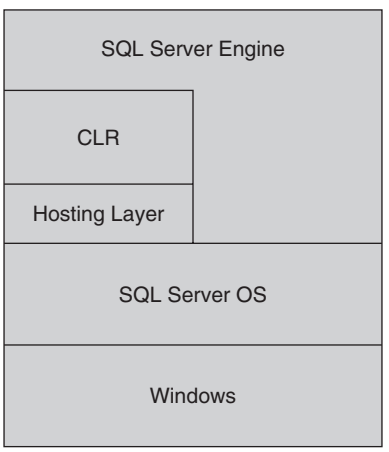

Figure 3-1 *The SQL Server CLR database architecture*

essential functions such as memory management, garbage collection, and threading under the control of the SQL Server database engine. In an external implementation the CLR will manage these things independently. The database engine has a better view of the system requirements as a whole and can manage memory and threads better than the CLR can do on its own. In the end, hosting the CLR in-process will provide better performance and scalability.

Enabling CLR Support

By default, the CLR support in the SQL Server database engine is turned off. This ensures that update installations of SQL Server do not unintentionally introduce new functionality without the explicit involvement of the administrator. To enable SQL Server's CLR support, you need to use the advanced options of SQL Server's sp_configure system stored procedure, as shown in the following listing:

```
sp_configure 'show advanced options', 1
GO
RECONFIGURE
GO
sp_configure 'clr enabled', 1
GO
RECONFIGURE
GO
```

CLR Database Object Components

To create .NET database objects, you start by writing managed code in any one of the .NET languages, such as VB, C#, or Managed C++, and compile it into a .NET DLL (dynamic link library). The most common way to do this would be to use Visual Studio 2005 to create a new SQL Server project and then build that project, which creates the DLL. Alternatively, you create the .NET code using your editor of choice and then compiling the code into a .NET DLL using the .NET Framework SDK. ADO.NET is the middleware that connects the CLR DLL to the SQL Server database. Once the .NET DLL has been created, you need to register that DLL with SQL Server, creating a new SQL Server database object called an assembly. The *assembly* essentially encapsulates the .NET DLL. You then create a new database object such as a stored procedure or a trigger that points to the SQL Server assembly. You can see an overview of the process to create a CLR database object in Figure 3-2.

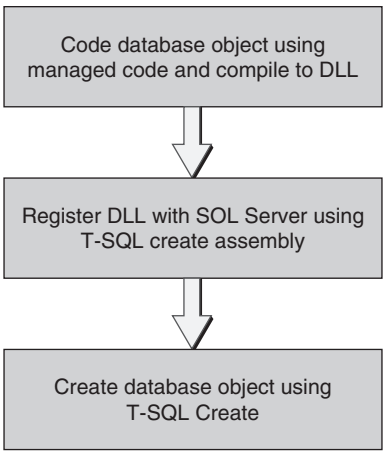

Figure 3-2 *Creating CLR database objects*

SQL Server .NET Data Provider

If you're familiar with ADO.NET, you may wonder exactly how CLR database objects connect to the database. After all, ADO.NET makes its database connection using client-based .NET data providers such as the .NET Framework Data Provider for SQL Server, which connects using networked libraries. While that's great for a client application, going through the system's networking support for a database call isn't the most efficient mode for code that's running directly on the server. To address this issue, Microsoft created the new SQL Server .NET Data Provider. The SQL Server .NET Data Provider establishes an in-memory connection to the SQL Server database.

Assemblies

After the coding for the CLR object has been completed, you can use that code to create a SQL Server assembly. If you're using Visual Studio 2005, then you can simply select the Deploy option, which will take care of both creating the SQL Server assembly as well as creating the target database object.

If you're not using Visual Studio 2005 or you want to perform the deployment process manually, then you need to copy the .NET DLL to a common storage location of your choice. Then, using SQL Server Management Studio, you can execute a T-SQL CREATE ASSEMBLY statement that references the location of the .NET DLL, as you can see in the following listing:

```
CREATE ASSEMBLY MyCLRDLL
FROM '\\SERVERNAME\CodeLibrary\MyCLRDLL.dll'
```

The CREATE ASSEMBLY command takes a parameter that contains the path to the DLL that will be loaded into SQL Server. This can be a local path, but more often it will be a path to a networked file share. When the CREATE ASSEMBLY is executed, the DLL is copied into the master database.

If an assembly is updated or becomes deprecated, then you can remove the assembly using the DROP ASSEMBLY command as follows:

```
DROP ASSEMBLY MyCLRDLL
```

Because assemblies are stored in the database, when the source code for that assembly is modified and the assembly is recompiled, the assembly must first be dropped from the database using the DROP ASSEMBLY command and then reloaded using the CREATE ASSEMBLY command before the updates will be reflected in the SQL Server database objects.

You can use the sys.assemblies view to view the assemblies that have been added to SQL Server 2005 as shown here:

```
SELECT * FROM sys.assemblies
```

Since assemblies are created using external files, you may also want to view the files that were used to create those assemblies. You can do that using the sys.assembly_files view as shown here:

```
SELECT * FROM sys.assembly_files
```

Creating CLR Database Objects

After the SQL Server assembly is created, you can then use SQL Server Management Studio to execute a T-SQL CREATE PROCEDURE, CREATE TRIGGER, CREATE FUNCTION, CREATE TYPE, or CREATE AGGREGATE statement that uses the EXTERNAL NAME clause to point to the assembly that you created earlier.

When the assembly is created, the DLL is copied into the target SQL Server database and the assembly is registered. The following code illustrates creating the MyCLRProc stored procedure that uses the MyCLRDLL assembly:

```
CREATE PROCEDURE MyCLRProc
AS EXTERNAL NAME
MyCLRDLL.StoredProcedures.MyCLRProc
```

The EXTERNAL NAME clause is new to SQL Server 2005. Here the EXTERNAL NAME clause specifies that the stored procedure MyCLRProc will

be created using a .SQL Server assembly. The DLL that is encapsulated in the SQL Server assembly can contain multiple classes and methods; the EXTERNAL NAME statement uses the following syntax to identify the correct class and method to use from the assembly:

```
Assembly Name.ClassName.MethodName
```

In the case of the preceding example, the registered assembly is named MyCLRDLL. The class within the assembly is StoredProcedures, and the method within that class that will be executed is MyCLRProc.

Specific examples showing how you actually go about creating a new managed code project with Visual Studio 2005 are presented in the next section.

Creating CLR Database Objects

The preceding section presented an overview of the process along with some example manual CLR database object creation steps to help you better understand the creation and deployment process for CLR database objects. However, while it's possible to create CLR database objects manually, that's definitely not the most productive method. The Visual Studio 2005 Professional, Enterprise, and Team System Editions all have tools that help create CLR database objects as well as deploy and debug them. In the next part of this chapter you'll see how to create each of the new CLR database objects using Visual Studio 2005.

NOTE

The creation of SQL Server projects is supported in Visual Studio 2005 Professional Edition and higher. It is not present in Visual Studio Standard Edition or the earlier releases of Visual Studio.

CLR Stored Procedures

Stored procedures are one of the most common database objects that you'll want to create using one of the managed .NET languages. One of the best uses for CLR stored procedures is to replace existing extended stored procedures. T-SQL is only able to access database resources. In order to access external system resources, Microsoft has provided support in SQL Server for a feature known as extended stored procedures. *Extended stored procedures* are unmanaged DLLs that run in the SQL Server process space and can basically do anything a standard executable program can do, including

accessing system resources that are external to the database, such as reading and writing to the file system, reading and writing to the Registry, and accessing the network. However, because extended stored procedures run in the same process space as the SQL Server database engine, bugs, memory violations, and memory leaks in the extended stored procedure could potentially affect the SQL Server database engine. CLR stored procedures solve this problem because they are implemented as managed code and run within the confines of the CLR. Another good candidate for CLR stored procedures is to replace existing T-SQL stored procedures that contain complex logic and embody business rules that are difficult to express in T-SQL. CLR stored procedures can take advantage of the built-in functionality provided by the classes in the .NET Framework, making it relatively easy to add functionality such as complex mathematical expressions or data encryption. Plus, since CLR stored procedure are compiled rather than interpreted like T-SQL, they can provide a significant performance advantage for code that's executed multiple times. However, CLR stored procedures are not intended to be used as a replacement for T-SQL stored procedures. T-SQL stored procedures are still best for data-centric procedures.

To create a CLR stored procedure in Visual Studio 2005, first select the New | Project option and then select the SQL Server Project template as is shown in Figure 3-3.

Give your project a name and click OK to create the project. In this example you can see that I've used the name usp_ImportFile for my stored procedure. This stored

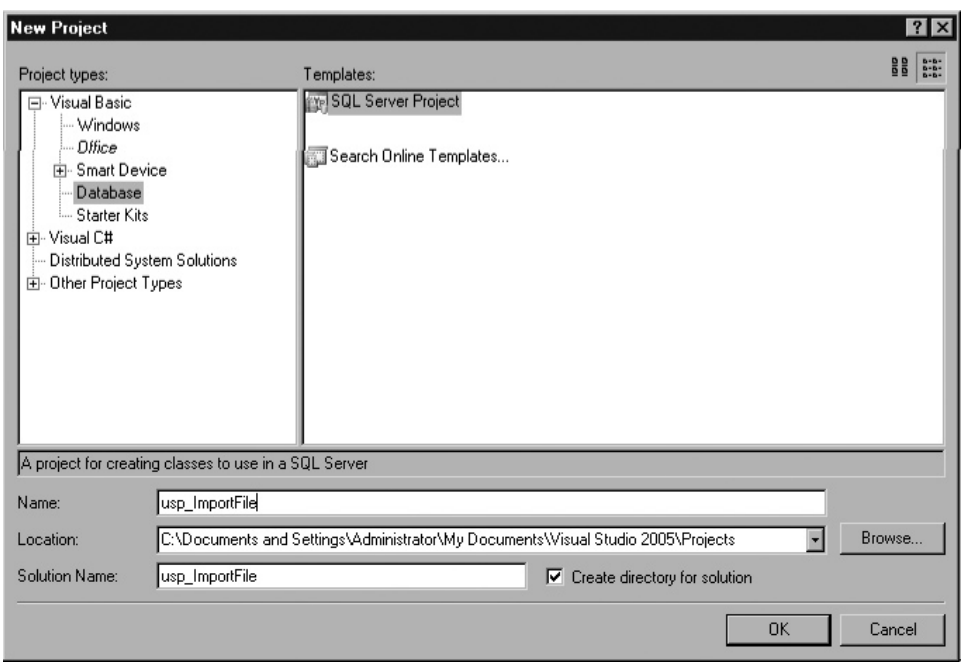

Figure 3-3 *Creating a new SQL Server stored procedure project*

procedure shows how you can replace an extended stored procedure with a CLR stored procedure. In this case the CLR stored procedure will read the contents of a file and store it in a SQL Server column. After naming the project, click OK. Before Visual Studio generates the project code, it displays the New Database Reference dialog that you can see in Figure 3-4.

Visual Studio 2005 uses the New Database Reference dialog to create a connection to your SQL Server 2005 system. That connection will be used to both debug and deploy the finished project. Drop down the Server Name box and select the name of the SQL Server that you want to use with this project. Then select the type of

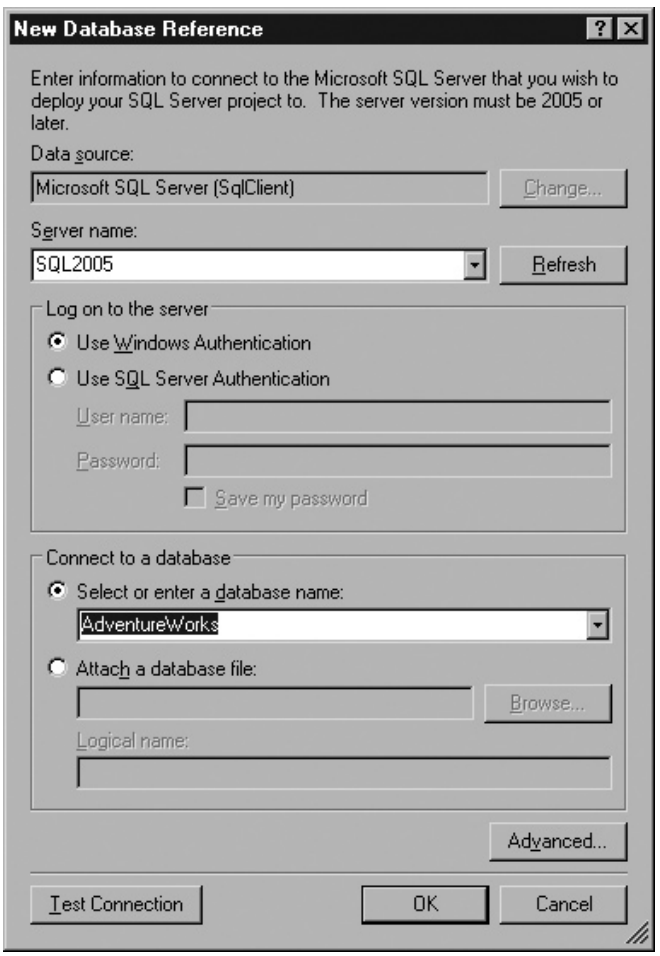

Figure 3-4 *The New Database Reference dialog*

authentication that you want to use and the database where the CLR stored procedure will be deployed. In Figure 3-4 you can see that I've selected the SQL Server system named SQL2005. The project will connect using Windows authentication, and the stored procedure will be deployed to the AdventureWorks database. You can verify the connection properties by clicking the Test Connection button. Once the connection properties are set up the way you want, click OK. All of the required references will automatically be added to your SQL Server project, and Visual Studio 2005 will generate a SQL Server starter project.

NOTE

While Visual Studio 2005 lets you group multiple stored procedures, triggers, and other CLR database objects in a single DLL, it's really better to create each CLR database object as a separate DLL. This gives you more granular control in managing and later updating the individual database objects.

Next, to create the CLR stored procedure, you can select the Project | Add Stored Procedure option to display the Visual Studio installed templates dialog that's shown in Figure 3-5.

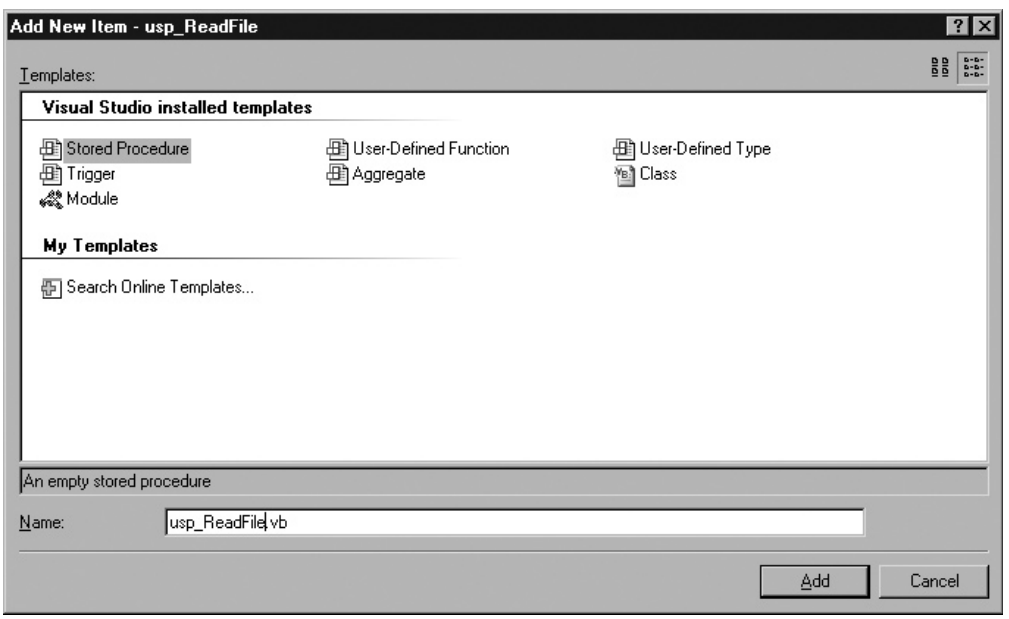

Figure 3-5 *Adding a CLR stored procedure*

From the Add New Item dialog, select the Stored Procedure option from the list of templates displayed in the Templates list and then provide the name of the stored procedure in the Name field that you can see at the bottom of the screen. Here you can see that the stored procedure will be created using the source file usp_ImportFile.vb. Visual Studio 2005 will add a new class to your project for the stored procedure. The generated class file is named after your stored procedure name and will include all of the required import directives as well as the starter code for the stored procedure. You can see the SQL Server CLR stored procedure template in Figure 3-6.

By default the SQL Server .NET Data Provider is added as a reference, along with an include statement for its System.Data.SqlServer namespace. Plus, you can see the System.Data reference, which provides support for ADO.NET and its data-oriented objects such as the DataSet and the System.Data.SqlTypes namespace that provides support for the SQL Server data types.

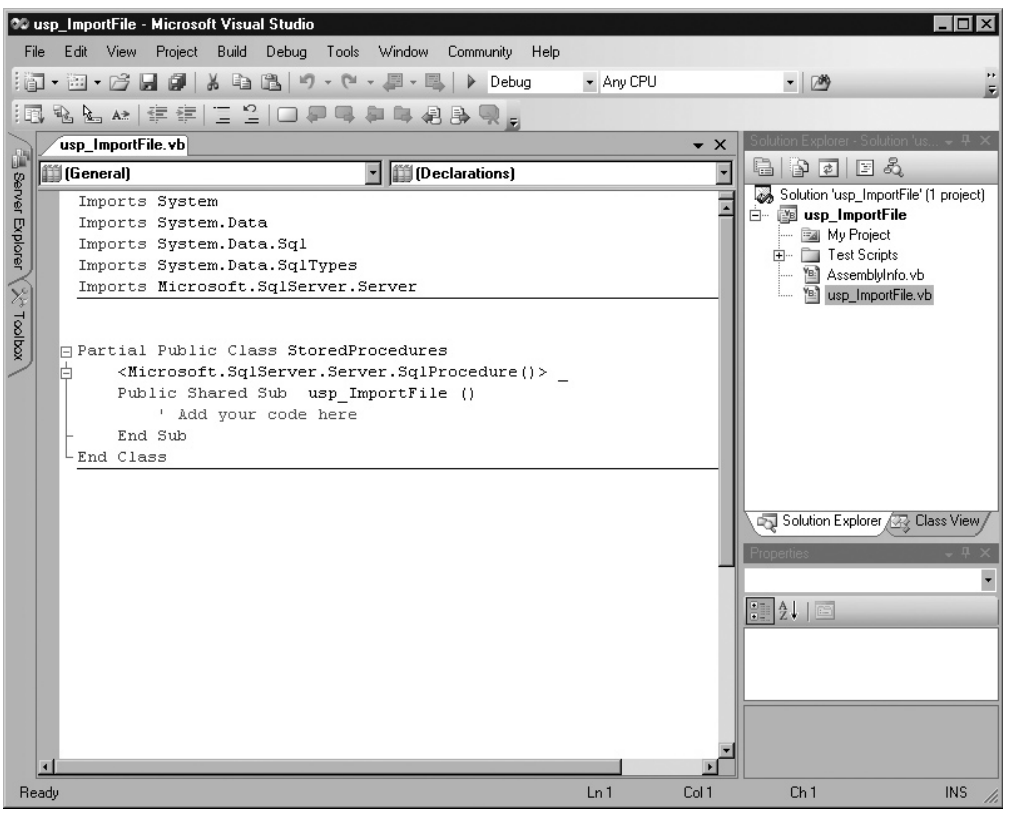

Figure 3-6 *The CLR stored procedure template*

It's up to you to fill in the rest of the code that makes the stored procedure work. The following example illustrates the source code required to create a simple CLR stored procedure that imports the contents of a file into a varchar or text column:

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.IO

Partial Public Class StoredProcedures
    <Microsoft.SqlServer.Server.SqlProcedure()> _
    Public Shared Sub usp_ImportFile _
    (ByVal sInputFile As String, ByRef sColumn As String)
        Dim sContents As String
        Try
            Dim stmReader As New StreamReader(sInputFile)
            sContents = stmReader.ReadToEnd()
            stmReader.Close()
            sColumn = sContents
        Catch ex As Exception
            Dim sp As SqlPipe = SqlContext.Pipe()
            sp.Send(ex.Message)
        End Try
    End Sub
End Class
```

The first important point to note in this code is the directive that imports the Microsoft.SqlServer.Server namespace. This enables the usp_ImportFile project to use the SQL Server .NET Data Provider without always needing to reference the fully qualified name. The second thing to notice is the <Microsoft.SqlServer.Server.SqlProcedure()> attribute that precedes the method name; it tells the compiler this method will be exposed as a SQL Server stored procedure. Next, you can see that the default class name for this stored procedure is set to StoredProcedures. This class contains a shared method named usp_ImportFile that accepts two parameters: a string that specifies the name of the file that will be imported and a second input parameter that specifies the name of a column that will contain the contents of the file. For C#, the method must be defined as static. For VB.NET code, the method would need to be defined as Shared.

Inside the usp_ImportFile method, a new string object named sContents is declared that will contain the contents of the file. Next, a Try-Catch loop is used to

capture any errors that may occur during the file import process. Within the Try-Catch loop a new StreamReader named stmReader is created that will be used to read the file from the operating system. The name of the file that will be read is passed into the StreamReader's instantiation call. Then the stmReader's ReadToEnd method is used to read the entire contents of the file into the sContent string variable. After the contents of the file have been read, the stmReader StreamReader is closed and the contents of the sContents variable are assigned to the SQL Server column.

If any errors occur while the input file is being read, then the code in the Catch portion of the Try-Catch structure is executed. Within the Catch block a SqlPipe object named sp is created and then used to send those errors back to the caller of the stored procedure. This code block uses the SqlPipe object, which represents a conduit that passes information between the CLR and the calling code. Here, the SqlPipe object enables the stored procedure to pass error information to the external caller.

Setting the Stored Procedure Security

At this point the code is finished for the stored procedure, but because of security concerns, it still can't execute. By default SQL Server CLR objects can only access database resources, and they cannot access external resources. In the case of the usp_ImportFile example, the stored procedure needs to access the file system, so the default security settings need to be changed. To enable external access, you need to open the project's properties and click the Database tab. Then in the Permissions Level drop-down you need to change the value from Safe to External. More information about the CLR security options is presented later in this chapter.

Deploying the Stored Procedure

After the CLR stored procedure source code has been compiled into an assembly, you can then add that assembly to the database and create the CLR stored procedure. You can do this in two ways. If you're using Visual Studio 2005 to create the SQL Server CLR database objects, then you can interactively deploy the CLR stored procedure directly from Visual Studio. To deploy the stored procedure to SQL Server, select the Build | Deploy Solution option from the Visual Studio menu.

You can perform the deployment manually as was shown in the earlier section "Creating CLR Database Objects". To do this, you essentially need to move the compiled DLL to a directory or file share where it can be accessed by SQL Server. Then run the CREATE ASSEMBLY statement to register the DLL and copy it into the database.

```
create assembly usp_ImportFile
from 'C:\temp\usp_ImportFile.dll'
WITH PERMISSION_SET = EXTERNAL
```

The CREATE ASSEMBLY statement copies the contents of the usp_ImportFile.dll file in the c:\temp directory into the SQL Server database. The WITH PERMISSION SET clause is used to specify that this assembly can access resources that are external to the SQL Server database. That's needed here because the stored procedure reads an external file.

```
CREATE PROCEDURE usp_ImportFile
 @filename nvarchar(1024),
 @columnname nvarchar(1024) OUT
AS
EXTERNAL NAME usp_ImportFile.[usp_ImportFile.StoredProcedures]
.usp_ImportFile
```

The CREATE PROCEDURE statement is used to create a new SQL Server stored procedure that uses the CLR assembly. This CLR stored procedure uses two parameters. The first is an input parameter, and the second is an output parameter. The EXTERNAL NAME clause uses a three-part name to identify the target method in the DLL. The first part of the name refers to the assembly name. The second part refers to the class. If the class is part of a namespace, as is the case here, then the namespace must preface the class name and both should be enclosed in brackets. Finally, the third part of the name identifies the method that will be executed.

Using the Stored Procedure

After the CLR stored procedure has been created, it can be called exactly like any T-SQL stored procedure, as the following example illustrates:

```
DECLARE @myColumn ntext
EXEC usp_ImportFile 'c:\temp\testfile.txt' @myColumn
```

User-Defined Functions

Creating .NET-based *user-defined functions (UDFs)* is another new feature that's enabled by the integration of the .NET CLR. User-defined functions that return scalar types must return a .NET data type that can be implicitly converted to a SQL Server data type. Scalar functions written with the .NET Framework can significantly outperform T-SQL in certain scenarios because unlike T-SQL functions, .NET functions are created using compiled code. User-defined functions can also return table types, in which case the function must return a result set.

To create a UDF using Visual Studio 2005, select the New | Project option and then select the SQL Server Project template as shown in Figure 3-7.

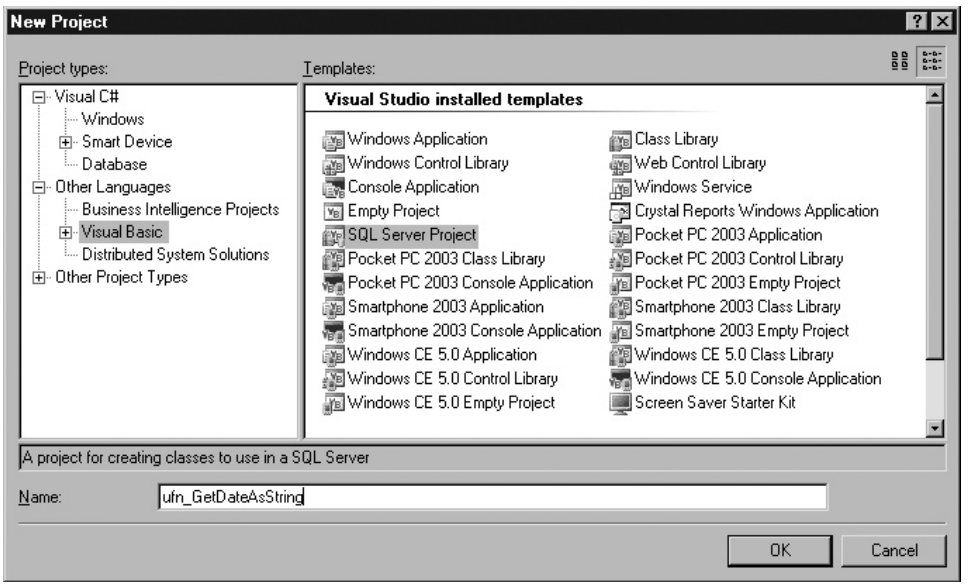

Figure 3-7 *Creating a new SQL Server UDF project*

As in the Stored Procedure example that was presented earlier, first give your project a name and click OK to create the project. In the example shown in Figure 3-7, you can see that I've used the name ufn_GetDateAsString for my user-defined function. This function returns a string value containing the system date and time. After naming the project, click OK to display the New Database Reference dialog for the CLR Function project, which will resemble the one shown in Figure 3-8.

NOTE

The Add Database Reference dialog is shown instead of the New Database Reference dialog when a database reference has already been created. This would be the case if you created the ufn_GetDateAsString function immediately after the usp_ImportFile project.

The New Database Reference dialog defines the connection between your Visual Studio project and SQL Server. The project will connect to the SQL Server system named sql2005, and the function will be deployed to the AdventureWorks database.

Once the Visual Studio project has been created and the connection has been defined, you use the Project | Add Function menu option to display the Add New Item dialog that you can see in Figure 3-9.

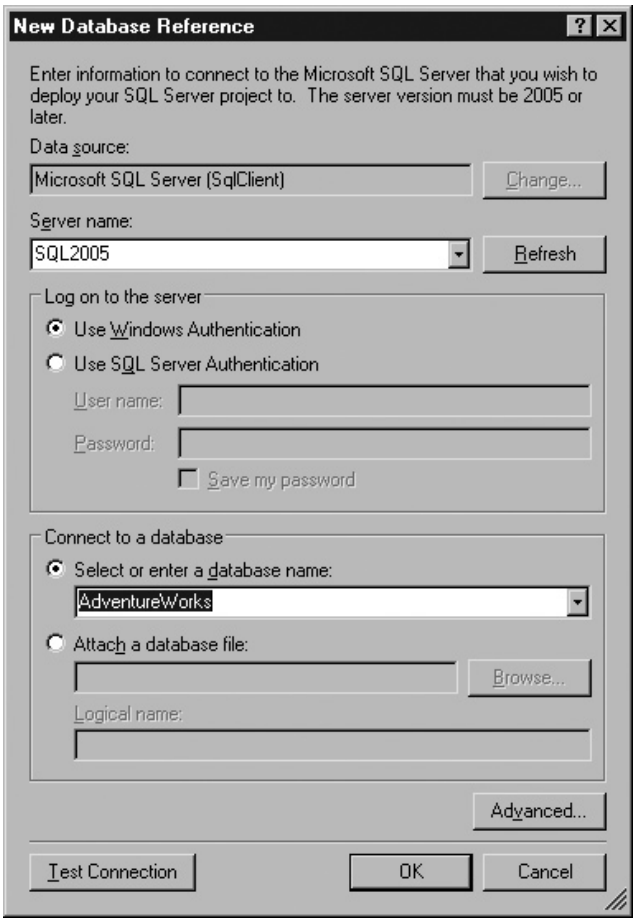

Figure 3-8 *The New Database Reference dialog*

Visual Studio uses the SQL Server Function project template to create a starter project that includes the reference to the SQL Server .NET Data Provider and a basic function wrapper for your source code. It's up to you to fill in the rest of the code. The following code listing shows the completed CLR function, ufn_GetDateAsString, that performs a basic date-to-string conversion:

```
Imports System
Imports System.Data
Imports System.Data.Sql
```

```
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server

Partial Public Class UserDefinedFunctions
    <Microsoft.SqlServer.Server.SqlFunction()> _
    Public Shared Function ufn_GetDateAsString() As SqlString
        Dim dtDataTime As New DateTime
        Return dtDataTime.ToString()
    End Function
End Class
```

Here, the Microsoft.SqlServer.Server namespace is not needed, as this particular function does not perform any data access. Next, Visual Studio 2005 generated the UserDefinedFunctions class to contain all of the methods that this assembly will expose as UDFs. You can also see that the <Microsoft.SqlServer.Server.SqlFunction()> attribute is used to identify the ufn_GetDateAsString method as a UDF. The code in this simple example just converts the system date to a string data type that's returned to the caller.

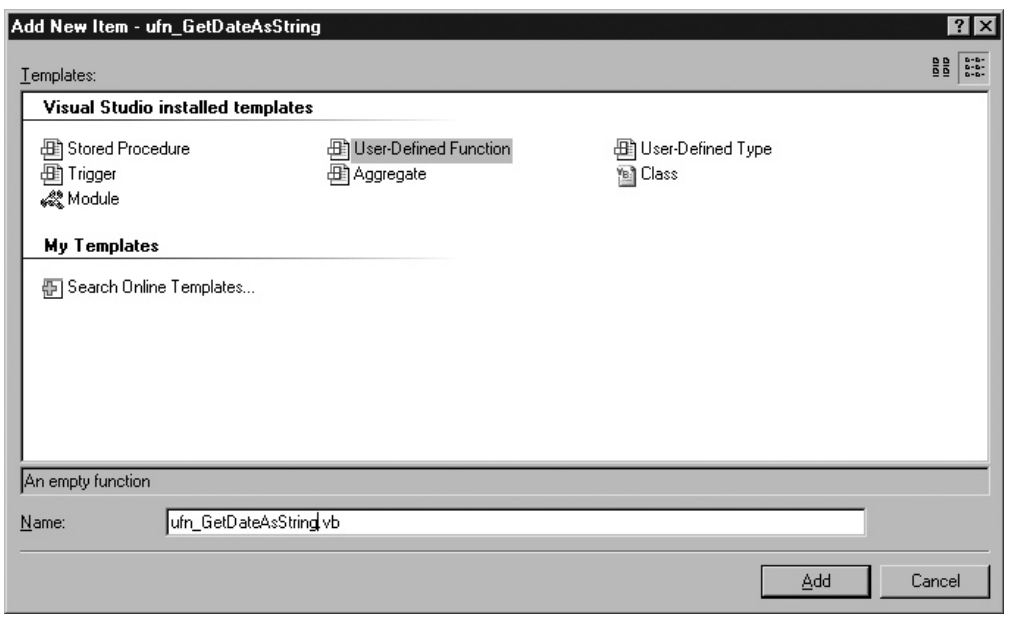

Figure 3-9 *Adding a CLR user-defined function*

Deploying the Function

To create the function in a SQL Server database, the assembly must first be created, as you saw in the stored procedure example. Then if you're using Visual Studio 2005, you can simply select the Build | Deploy Solution option and you're done.

If you're doing this manually, you'll need to copy the ufn_GetDataAsString.dll file to a location that's accessible by the SQL Server system and then create the assembly, followed by the function. The following CREATE ASSEMBLY statement can be used to copy the contents of ufn_GetDateAsString.dll into the SQL Server database:

```
CREATE ASSEMBLY ufn_GetDataAsString
FROM '\\MyFileShare\Code Library\ufn_GetDataAsString.dll'
```

The CREATE FUNCTION statement is then used to create a new SQL Server function that executes the appropriate method in the assembly. The following listing illustrates how the CREATE FUNCTION statement can create a .CLR user-defined function:

```
CREATE FUNCTION ufn_GetDateAsString()
RETURNS nvarchar(256)
EXTERNAL NAME
ufn_GetDateAsString.UserDefinedFunctions.ufn_GetDateAsString
```

For user-defined functions, the CREATE FUNCTION statement has been extended with the EXTERNAL NAME clause, which essentially links the user-defined function name to the appropriate method in the .NET assembly. In this example, the ufn_GetDateAsString function is using the assembly named ufn_GetDateAsString. Within that assembly, it's using the UserDefinedFunctions class and the ufn_GetDateAsString method within that class.

Using the Function

After the function has been created, it can be called like a regular SQL Server function. You can see how to execute the GetDateAsString function in the following example:

```
SELECT dbo.GetDateAsString()
```

Triggers

In addition to stored procedures and user-defined functions, the new .NET integration capabilities found in SQL Server 2005 also provide the ability to create CLR triggers. To create a trigger using Visual Studio 2005, you start your project as you saw in the

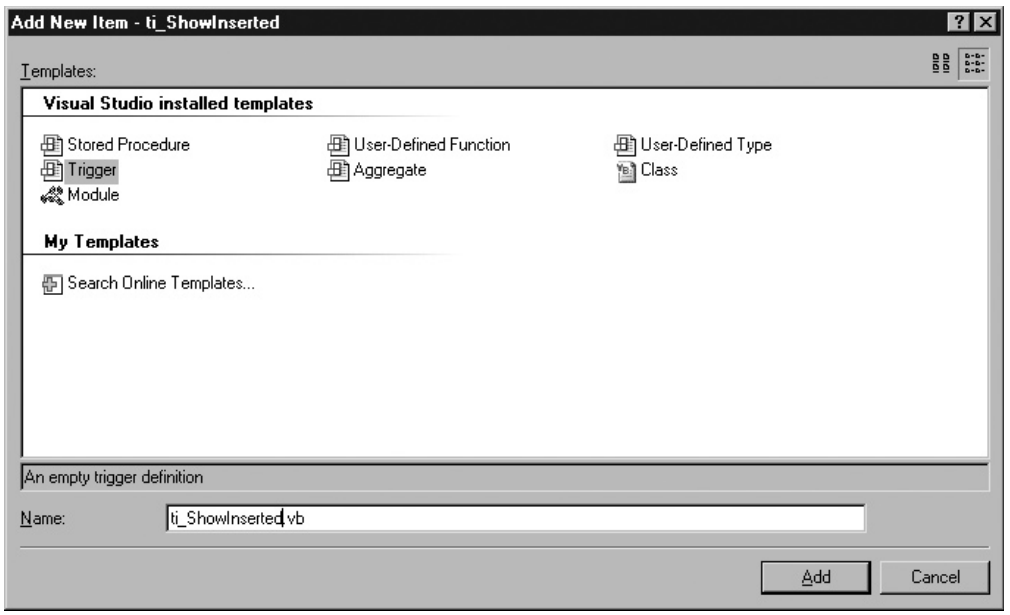

Figure 3-10 *Adding a CLR trigger*

earlier examples. To create a trigger using Visual Studio 2005, select the New | Project option, give your project a name, and click OK to create the project. For this project, I used the name ti_ShowInserted for my trigger. This trigger essentially retrieves the values of the row being inserted in a table and displays them. After naming the project and clicking OK, I filled out the New Database Reference dialog using the same values that were shown in the previous examples. Next, I used the Project | Add Trigger menu option that you can see in Figure 3-10 to create a starter project for the CLR trigger.

As you saw in the earlier example of CLR database objects, you select the Trigger option from the list of templates and then provide the name of the trigger in the name prompt. Visual Studio 2005 will generate a starter project file that you can add your code to. The starter project includes the appropriate import directives as well as generating a class, in this case appropriately named Triggers, and a method named ti_ShowInserted with its appropriate method attribute. The following code listing shows the completed code for the CLR trigger named ti_ShowInserted:

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlTypes
```

```
Imports Microsoft.SqlServer.Server
Imports System.Data.SqlClient

Partial Public Class Triggers
    ' Enter existing table or view for the target and uncomment
      the attribute line
    <Microsoft.SqlServer.Server.SqlTrigger(Name:="ti_ShowInserted", _
        Target:="Person.ContactType", Event:="FOR INSERT")> _
    Public Shared Sub ti_ShowInserted()
        Dim oTriggerContext As SqlTriggerContext = _
          SqlContext.TriggerContext
        Dim sPipe As SqlPipe = SqlContext.Pipe
        If oTriggerContext.TriggerAction = TriggerAction.Insert Then
            Dim oConn As New SqlConnection("context connection=true")
            oConn.Open()
            Dim oCmd As New SqlCommand("Select * from inserted", oConn)
            sPipe.ExecuteAndSend(oCmd)
        End If
    End Sub
End Class
```

The example CLR trigger displays the contents of the data that is used for an insert action that's performed on the Person.ContactTypes table in the Adventureworks database. The first thing to notice in this code listing is the Attribute for the ti_ ShowInserted subroutine (the code enclosed within the < > markers). The Attribute is used to name the trigger and identify the table the trigger will be applied to as well as the event that will cause the trigger to fire. When the Visual Studio 2005 trigger template initially generates this Attribute, it is prefaced by a comment symbol—essentially making the line a comment. This is because the trigger template doesn't know how or where you want the trigger to be used. In order for Visual Studio 2005 to deploy the trigger, you need to uncomment the Attribute line and then fill in the appropriate properties. The following table lists the properties used by the Visual Studio 2005 trigger template:

Property Name	Description
Name	The name the trigger will use on the target SQL Server system.
Target	The name of the table that the trigger will be applied to.
Event	The action that will fire the trigger. The following trigger events are supported: FOR INSERT, FOR UPDATE, FOR DELETE, AFTER INSERT, AFTER UPDATE, AFTER DELETE, INSTEAD OF INSERT, INSTEAD OF UPDATE, INSTEAD OF DELETE

In this example, the resulting trigger will be named ti_ShowInserted. It will be applied to the table named Person.ContactType, which is in the AdventureWorks database, and the trigger will only be fired for an insert operation.

The primary code for the trigger is found within the ti_ShowInserted subroutine. This code example makes use of another new ADO.NET object: SqlTriggerContext. The SqlTriggerContext object provides information about the trigger action that's fired and the columns that are affected. The SqlTriggerContext object is always instantiated by the SqlContext object. Generally, the SqlContext object provides information about the caller's context. Specifically, in this case, the SqlContext object enables the code to access the virtual table that's created during the execution of the trigger. This virtual table stores the data that caused the trigger to fire.

Next, a SqlPipe object is created. The SqlPipe object enables the trigger to communicate with the external caller, in this case to pass the inserted data values to the caller. The TriggerAction property of the SqlContext object is used to determine if the trigger action was an insert operation. Using the TriggerAction property is quite straightforward. It supports the following values:

TriggerAction Value	Description
TriggerAction.Insert	An insert operation was performed.
TriggerAction.Update	An update action was performed.
TriggerAction.Delete	A delete action was performed.

If the TriggerAction property equals TriggerAction.Insert, then an insert was performed and the contents of the virtual trigger table are retrieved and sent to the caller using the SqlPipe object's Execute method. In order to retrieve the contents of the virtual table, a SqlConnection object and a SqlCommand object are needed. These objects come from the System.Data.SqlClient namespace. You should note that when used with server-side programming, the Connection String used by the SqlConnection object must be set to the value of "context Connection=true". Then a SqlCommand object named oCmd is instantiated that uses the statement "Select * from inserted" to retrieve all of the rows and columns from the virtual table that contains the inserted values. Finally, the ExecuteAndSend method of SqlPipe object is used to execute the command and send the results back to the caller.

Deploying the Trigger

Once the code has been created, you can either deploy it to the database using the Visual Studio 2005 Build | Deploy solution option or manually drop and re-create the assembly and any dependent objects you saw in UDF examples earlier in this chapter.

To manually deploy the code, you'd need to copy ti_ShowInserted.dll to the SQL Server system or to a share that's accessible to the SQL Server system and then execute the following T-SQL Server commands:

```
Use AdventureWorks

create assembly ti_showinserted
from 'C:\temp\ti_ShowInserted.dll'
go

CREATE TRIGGER ti_ShowInserted
ON Person.ContactType
FOR INSERT
AS EXTERNAL NAME ti_ShowInserted.[ti_ShowInserted.Triggers].ti_ShowInserted
go
```

This example assumes that ti_ShowInsert.dll was copied into the c:\temp directory on the SQL Server system. First, the Create Assembly statement is used to copy the DLL into the SQL Server database and then the Create Trigger statement is used with the As External Name clause to create a trigger named ti_ShowInserted and attach it to the Person.ContactTypes table. As in the earlier examples, the As External Name clause identifies the assembly using a three-part name: *asssembly.class.method.* Pay particular attention to the class portion of this name. For triggers you must bracket the class name and include the namespace just before the class name. In this example, the assembly is named ti_ShowInserted. The Namespace is ti_ShowInserted. The class is named Triggers, and the method is named ti_ShowInserted.

Using the Trigger

After the CLR trigger has been deployed, it will be fired for every insert operation that's performed on the base table. For example, the following INSERT statement will add a row to the Person.ContactType table, which will cause the CLR trigger to fire:

```
INSERT INTO Person.ContactType VALUES(102, 'The Big Boss',
  '2005-05-17 00:00:00.000')
```

The example trigger, ti_ShowInserted, performs a select statement on the inserted row value. Then it uses the SqlPipe object to send the results back to the caller. In this example the trigger will send the contents of the inserted row values back to the caller:

```
ContactTypeID Name                                               ModifiedDate
------------- -------------------------------------------------- --------------
21            The Big Boss                                       2005-05-17
00:00:00.000
(1 row(s) affected)
(1 row(s) affected)
```

User-Defined Types

Another important new feature in SQL Server 2005 that is enabled by the integration of the .NET CLR is the ability to create true *user-defined types (UDTs).* Using UDTs, you can extend the raw types provided by SQL Server and add data types that are specialized to your application or environment.

In the following example you'll see how to create a UDT that represents a gender code: either M for male or F for female. While you could store this data in a standard one-byte character field, using a UDT ensures that the field will accept only these two values with no additional need for triggers, constraints, or other data validation techniques.

To create a UDT using Visual Studio 2005, select the New | Project option, give your project a name, and click OK to create the project. For this project I used the name of Gender for the new UDT. After naming the project and clicking OK, I filled out the New Database Reference dialog using the required connection values to deploy the project to the appropriate SQL Server system and database. Next, I used the Project | Add User-Defined Type option to display the Add New Item dialog that you can see in Figure 3-11.

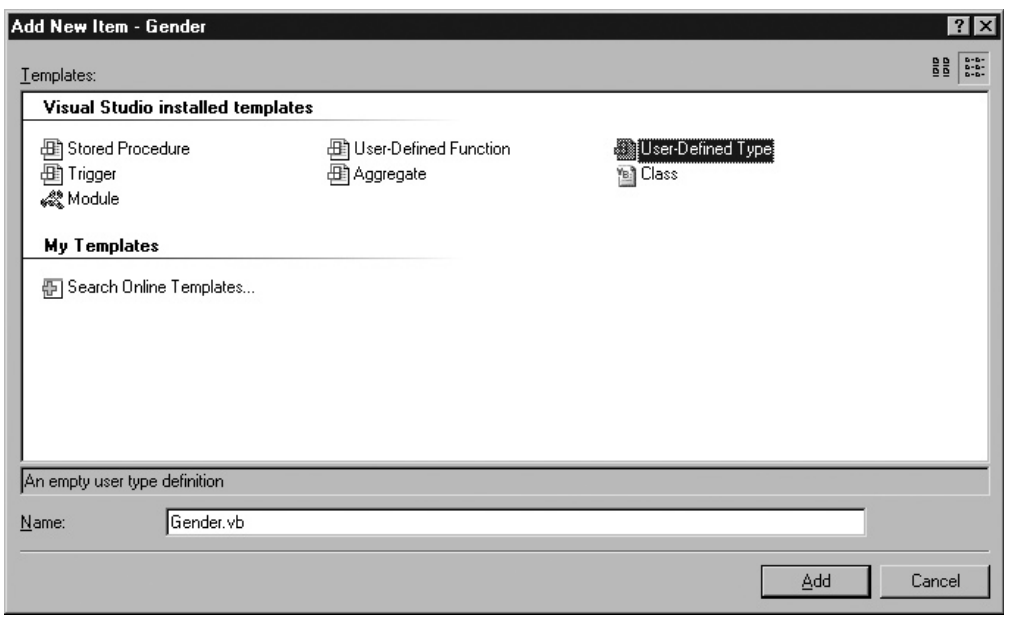

Figure 3-11 *Creating a .NET SQL Server UDT*

Method	Description
IsNull	This required method is used to indicate if the object is nullable. SQL Server 2005 requires all UDTs to implement nullability, so this method must always return true.
Parse	This required method accepts a string parameter and stores it as a UDT.
ToString	This required method converts the contents of the UDT to a string.
Default constructor	This required method creates a new instance of the UDT.

Table 3-1 *Required UDT Methods*

Select User-Defined Type from the list of SQL Server templates. Enter the name that you want to assign to the class and then click Open to have Visual Studio generate a starter project file for the UDT. The starter project file implements the four methods that SQL Server 2005 requires for all UDTs. These methods are needed to fulfill the SQL Server UDT contract requirements—it's up to you to add the code to make the UDT perform meaningful actions. The four required UDT methods are listed in Table 3-1.

You can see the completed Gender class that is used to implement a UDT for M (male) and F (female) codes in this listing:

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.IO

<Serializable()> _
<Microsoft.SqlServer.Server.SqlUserDefinedType _
 (Format.UserDefined, _
  IsFixedLength:=True, MaxByteSize:=2)> _
Public Structure Gender
    Implements INullable, IBinarySerialize

    Public Sub Read(ByVal r As BinaryReader) _
        Implements IBinarySerialize.Read
        m_value = r.ReadString.ToString()
    End Sub
```

```
 Public Sub Write(ByVal w As BinaryWriter) _
     Implements IBinarySerialize.Write
     w.Write(m_value.ToString())
 End Sub

Public Overrides Function ToString() As String
     If m_value.IsNull = False Then
         Return m_value.Value
     Else
         Return Nothing
     End If
 End Function

 Public ReadOnly Property IsNull() As Boolean _
    Implements INullable.IsNull
     Get
         If m_value.IsNull = True Then
             Return True
         Else
             Return False
         End If
     End Get
 End Property

 Public Shared ReadOnly Property Null() As Gender
     Get
         Dim h As Gender = New Gender
         h.m_Null = True
         Return h
     End Get
 End Property

 Public Shared Function Parse(ByVal s As SqlString) As Gender
     If s.IsNull Then
         Return Null
     End If

     Dim u As Gender = New Gender
     u.Value = s
     Return u
 End Function
```

```
    ' Create a Value Property
    Public Property Value() As SqlString
        Get
            Return m_value
        End Get

        Set(ByVal value As SqlString)
            If (value = "M" Or value = "F") Then
                m_value = value
            Else
                Throw New ArgumentException _
                  ("Gender data type must be M or F")
            End If
        End Set
    End Property

    ' Private members
    Private m_Null As Boolean
    Private m_value As SqlString

End Structure
```

To create a UDT, the code must adhere to certain conventions. The class's attributes must be serializable, the class must implement the INullable interface, and the class name must be set to the name of the UDT. You can optionally add the IComparable interface. In this example, Gender is the class name. Near the bottom of the listing you can see where a private string variable named m_value is declared to hold the value of the data type.

Like the other CLR database objects, the Attribute plays an important part in the construction of the CLR UDT. The SQL Server UDT Attribute accepts the property values shown in Table 3-2.

The first thing to notice in the code is the use of the INullable and IBinarySerialize interfaces. The INullable interface is required for all UDTs. The IBinarySerialize interface is required for UDTs that use the Format.UserDefined attribute. Because this example uses a String data type, the Format.UserDefined attribute is required, which means that this UDT also needs code to handle the serialization of the UDT. In practical terms, this means that the class must implement the IBinarySerialize Read and Write methods, which you can see in the following section of code.

At first it may seem a bit intimidating to use the IBinarySerialize interfaces, but as you can see in the Read and Write subroutines, it's actually pretty simple. The Read subroutine simply uses the ReadString method to assign a value to the UDT's

Property	Description
Format.Native	SQL Server automatically handles the serialization of the UDT. The Format.Native value can only be used for UDTs that contain fixed-sized data types. The following data types are supported: bool, byte, sbyte, short, ushort, int, uint, long, ulong, float, double, SqlByte, SqlInt16, SqlInt32, SqlInt64, SqlDateTime, SqlSingle, SqlDouble, SqlMoney. If this property is used, the MaxByteSize property cannot be used.
Format.UserDefined	The UDT class is responsible for serializing the UDT. The format. UserDefined value must be used for variable-length data types like String and SQLString. If this value is used, the UDT must implement the IBinarySerialize interface and the Read and Write routines. If this property is used, the MaxByteSize property must also be specified.
MaxByteSize	Specifies the maximum size of the UDT in bytes.
IsFixedLength	A Boolean value that determines if all instances of this type are the same length.
IsByteOrdered	A Boolean value that determines how SQL Server performs binary comparisons on the UDT.
ValidationMethodName	The name of the method used to validate instances of this type.
Name	The name of the UDT.

Table 3-2 *UDT Attribute Properties*

m_value variable (which contains the UDT's value). Likewise, the Write subroutine uses the Write method to serialize the contents of the m_value variable.

The ToString method checks to see if the contents of the m_value variable are null. If so, then the string "null" is returned. Otherwise, the m_value's ToString method returns the string value of the contents.

The next section of code defines the IsNull property. This property's get method checks the contents of the m_value variable and returns the value of true if m_value is null. Otherwise, the get method returns the value of false. Next, you can see the Null method, which was generated by the template to fulfill the UDT's requirement for nullability.

The Parse method accepts a string argument, which it stores in the object's Value property. You can see the definition for the Value property a bit lower down in the code. The Parse method must be declared as static, or if you're using VB.NET, it must be a Shared property.

The Value property is specific to this implementation. In this example, the Value property is used to store and retrieve the value of the UDT. It's also responsible for

editing the allowable values. In the set method, you can see that only the values of M or F are permitted. Attempting to use any other values causes an exception to be thrown that informs the caller that the "Gender data type must be M or F".

Deploying the UDT

Very much like a CLR stored procedure or function, the UDT is compiled into a DLL after the code is completed. That DLL is then imported as a SQL Server assembly using the CREATE ASSEMBLY and CREATE TYPE statements or by simply using the Visual Studio 2005 Deploy option. You can see the T-SQL code to manually create the CLR UDT in the following listing:

```
create assembly Gender
from 'C:\temp\Gender.dll'
go

CREATE TYPE Gender
EXTERNAL NAME Gender.[Gender.Gender]
go
```

This listing assumes that gender.dll has been copied into the c:\temp that's on the SQL Server system. One thing to notice in the CREATE TYPE statement is the class parameter. As in the earlier CLR examples, the first part of the External Name clause specifies the assembly that will be used. In the case of a UDT, the second part of the name identifies the namespace and class. In the Gender example, the Namespace was Gender and the UDT's class was also named Gender.

Using the UDT

Once the UDT is created, you can use it in T-SQL much like SQL Server's native data types. However, since UDTs contain methods and properties, there are differences. The following example shows how the Gender UDT can be used as a variable and how its Value property can be accessed:

```
DECLARE @mf Gender
SET @mf='N'
PRINT @mf.Value
```

In this listing the UDT variable is declared using the standard T-SQL DECLARE statement, and the SET statement is used to attempt to assign the value of N to the UDT's Value property. Because N isn't a valid value, the following error is generated:

```
 .Net SqlClient Data Provider: Msg 6522, Level 16, State 1, Line 2
A CLR error occurred during execution of 'Gender':
System.ArgumentException: Gender data type must be M or F
at Gender.set_Value(SqlString value)
```

Just as UDTs can be used as variables, they can also be used to create columns. The following listing illustrates creating a table that uses the Gender UDT:

```
CREATE TABLE MyContacts
(ContactID int,
FirstName varchar(25),
LastName varchar(25),
MaleFemale Gender)
```

While creating columns with the UDT type is the same as when using a native data type, assigning values to the UDT is a bit different than the standard column assignment. Complex UDTs can contain multiple values. In that case you need to assign the values to the UDT's members. You can access the UDT's members by prefixing them with the (.) symbol. In this case, since the UDT uses a simple value, you can assign values to it exactly as you can any of the built-in data types. This example shows how to insert a row into the example MyContacts table that contains the Gender UDT:

```
INSERT INTO MyContacts VALUES(1, 'Michael', 'Otey', 'M')
```

To retrieve the contents of the UDT using the SELECT statement, you need to use the UDT.Member notation as shown here when referencing a UDT column:

```
SELECT ContactID, LastName, MaleFemale.Value FROM MyContacts
```

To see the UDTs that have been created for a database, you can query the sys.Types view as shown here:

```
SELECT * FROM sys.Types
```

Aggregates

The CLR aggregate is another new type of .NET database object that was introduced in SQL Server 2005. Essentially, a *user-defined aggregate* is an extensibility function that enables you to aggregate values over a group during the processing of a query. SQL Server has always provided a basic set of aggregation functions like MIN, MAX, and SUM that you can use over a query. User-defined aggregates enable you

to extend this group of aggregate functions with your own custom aggregations. One really handy use for CLR aggregates is to enable the creation of aggregate functions for CLR UDTs. Like native aggregation functions, user-defined aggregates allow you to execute calculations on a set of values and return a single value. When you create a CLR aggregate, you supply the logic that will perform the aggregation. In this section you'll see how to create a simple aggregate that calculates the maximum variance for a set of numbers.

To create an aggregate using Visual Studio 2005, select the New | Project option, give your project a name, and click OK to create the project. This example uses the name of MaxVariance. After naming the project and clicking OK, complete the New Database Reference dialog using the required connection values for your SQL Server system and database. Next, to create the aggregate I used the Project | Add Aggregate option to display the Add New Item dialog that you can see in Figure 3-12.

Select Aggregate from the list of SQL Server templates and then enter the name for the class and click OK. As you can see in Figure 3-12, I used the name MaxVariance. Visual Studio will generate a starter project for the aggregate class. Much as with a UDT, the template for a SQL Server CLR aggregate implements four methods that SQL Server 2005 requires for all CLR aggregates. The four required methods are listed in Table 3-3.

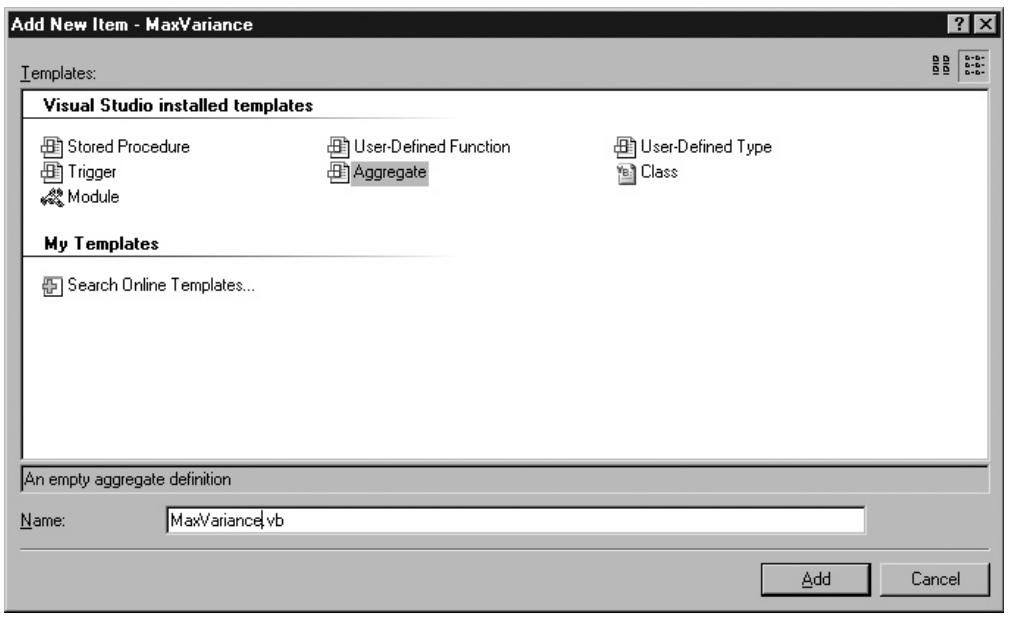

Figure 3-12 *Creating a CLR aggregate*

Method	Description
Init	This required method initializes the object. It is invoked once for each aggregation.
Accumulate	This required method is invoked once for each item in the set being aggregated.
Merge	This required method is invoked when the server executes a query using parallelism. This method is used to merge the data from the different parallel instances together.
Terminate	This required method returns the results of the aggregation. It is invoked once after all of the items have been processed.

Table 3-3 *Required Aggregate Methods*

You can see the code to implement the MaxVariance aggregate in the following listing:

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server

<Serializable()> _
<SqlUserDefinedAggregate(Format.Native)> _
Public Structure MaxVariance

    Public Sub Init()
        m_LowValue = 999999999
        m_HighValue = -999999999
    End Sub

    Public Sub Accumulate(ByVal value As Integer)
        If (value > m_HighValue)
            m_HighValue = value
        End If
        If (value < m_LowValue)
            m_LowValue = value
        End If
    End Sub

    Public Sub Merge(ByVal Group as MaxVariance)
        If (Group.GetHighValue() > m_HighValue)
            m_HighValue = Group.GetHighValue()
        End If
```

```
        If (Group.GetLowValue() < m_LowValue)
            m_LowValue = Group.GetLowValue()
        End If
    End Sub

    Public Function Terminate() As Integer
        return m_HighValue - m_LowValue
    End Function

    ' Helper methods
    Private Function GetLowValue() As Integer
        return m_LowValue
    End Function

    Private Function GetHighValue() As Integer
        return m_HighValue
    End Function

    ' This is a place-holder field member
    Private m_LowValue As Integer
    Private m_HighValue As Integer

End Structure
```

At the top of this listing you can see the standard set of Imports statements used by CLR objects, followed by the serialization attribute that's required by CLR aggregate objects. After that, in the Init method the two variables, m_LowValue and m_HighValue, are assigned high and low values, ensuring that they will be assigned values from the list. These two variables are declared near the bottom of the listing, and they serve to hold the minimum and maximum values that are encountered by the aggregate routine. The Init method is called one time only—when the object is first initialized.

While the Init method is called just once, the Accumulate method is called once for each row in the result set. In this example, the Accumulate method compares the incoming value with the values stored in the m_HighValue and m_LowValue variables. If the incoming value is higher than the current high value, it is stored in the m_HighValue variable. If the value is lower than the value of m_LowValue, it is stored in m_LowValue. Otherwise, no action is performed by the Accumulate method.

NOTE

Because aggregates are serialized, you need to be aware of the total storage requirements for some uses. The aggregate's value is serialized following each invocation of the Accumulate method, and it cannot exceed the maximum column size of 8000 bytes.

The Merge method is used when the aggregate is processed in parallel, which typically won't be the case for most queries. If the Merge is called, its job is to import the current aggregation values from the parallel instance. You can see here that it does that using two helper methods that essentially export the values in the m_HighValue and m_LowValue variables. These values are compared to the existing values, and if they are higher or lower, they will replace the current values in m_HighValue and m_LowValue.

The Terminate method is called once after all of the results have been processed. For this example, the Terminate method simply subtracts the lowest value found from the highest value found and returns the difference to the caller.

Deploying the Aggregate

After compiling the class into a DLL, you can import the DLL as a SQL Server assembly using either the Visual Studio 2005 Deploy option or manually using the CREATE ASSEMBLY statement and CREATE AGGREGATE statement as is shown in the following listing:

```
create assembly MaxVariance
from 'C:\temp\MaxVariance.dll'
go

CREATE AGGREGATE MaxVariance (@maXVar int)
RETURNS Int
EXTERNAL NAME MaxVariance.[MaxVariance.MaxVariance]
go
```

Like the earlier examples, this listing assumes that maxvariance.dll has been copied into the c:\temp directory on the local SQL Server system. In the CREATE AGGREGATE statement and the EXTERNAL NAME clause the first part of the name specifies the assembly that will be used, and the second part of the name identifies the namespace and class. Here all of these values are named MaxVariance.

Using the Aggregate

You can use the aggregate just like SQL Server's built-in aggregate functions. One small difference is that the UDAGG needs to be prefixed with the schema name to allow the system to locate it. The following line illustrates using the MaxVariance Aggregate:

```
SELECT dbo.MaxVariance(MinQty) FROM Sales.SpecialOffer
```

The result of this statement will show the difference between the high and low values found in the Sales.SpecialOffer column as is shown here:

```
-----------
61

(1 row(s) affected)
```

Debugging CLR Database Objects

One of the coolest features found in the integration of the .NET Framework, Visual Studio 2005, and SQL Server 2005 is the ability to debug the CLR database objects that you create. This tight level of integration sets SQL Server way ahead of competing database products like Oracle and DB2 that offer the ability to create stored procedures and functions using .NET code. While the other database products provide for the creation of these objects, they do not support the ability to provide integrated debugging. Visual Studio 2005 enables you to set breakpoints in your CLR database objects and then seamlessly step through your code and perform all of the debugging tasks that you would expects for a standard Windows or Web application, including the ability to set breakpoints, single-step through the code, inspect and change variables, and create watches—even between T-SQL and CLR code. Visual Studio 2005 automatically generates test scripts that are added to your projects. You can customize and use these test scripts to execute the CLR database objects that you create.

NOTE

You must compile and deploy the CLR database object before you can debug it.

To debug a SQL Server project using Visual Studio 2005, first open the project that you want to debug and then go to the Servers window and right-click the database connection. From the pop-up menu select the option Allow SQL/CLR Debugging as is shown in Figure 3-13.

Next, set up the script that you want to use to run the database object. Using the Solution window, open the Test Scripts folder and then the Test.sql file. You can set up multiple test scripts, but the Test.sql script is provided by default. If you want to change the script that Visual Studio 2005 uses to run the CLR database object, you simply right-click the desired script listed under the Test Scripts folder and select the Set As Default Debug Script option as is shown in Figure 3-14.

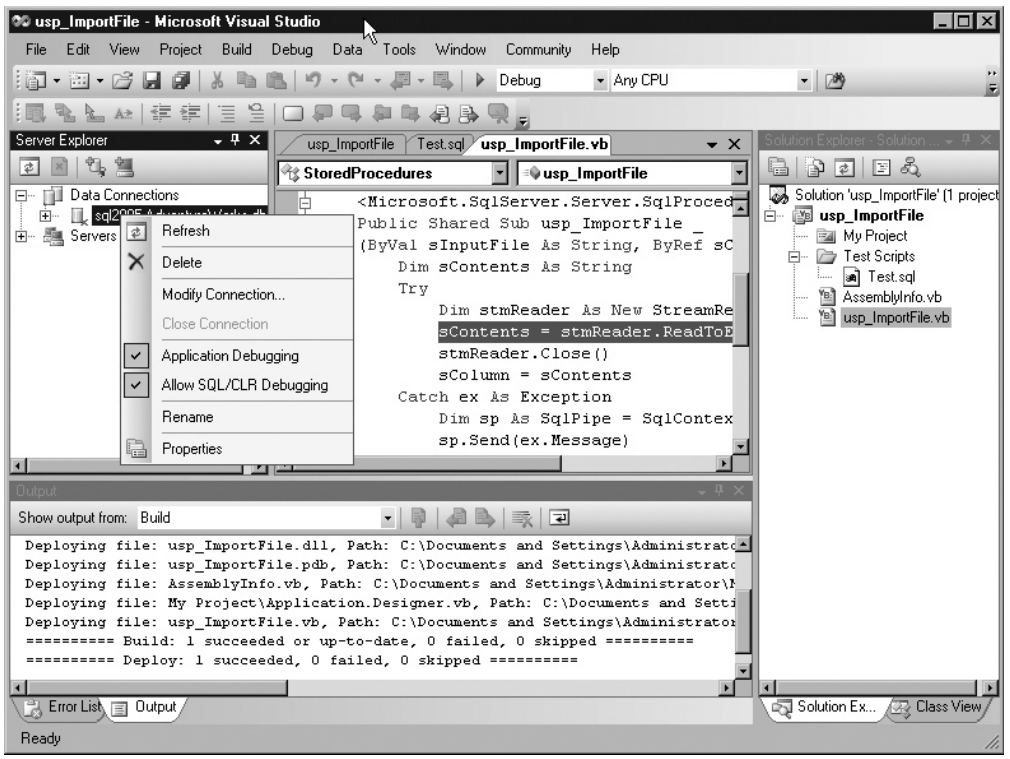

Figure 3-13 *Setting the Allow SQL/CLR Debugging option*

To use the default Test.sql script, open the file using the Visual Studio editor. Here you can see T-SQL boilerplate code for testing each of the different CLR database object types. Go to the section that you want and edit the code to execute the database object. You can see the test code for the usp_ImportFile stored procedure in the following listing:

```
-- Examples for queries that exercise different SQL objects
 -- implemented by this assembly
------------------------------------------------------------------------
-- Stored procedure
------------------------------------------------------------------------
declare @MyColumn varchar(30)
exec usp_ImportFile 'c:\temp\testfile.txt',@MyColumn
Select @MyColumn
```

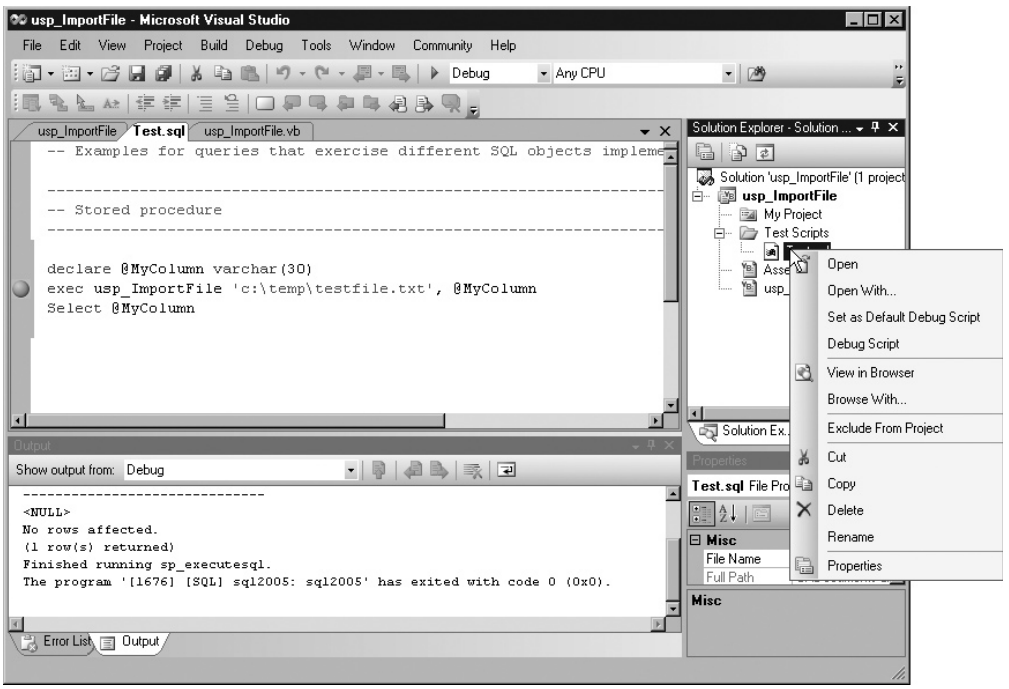

Figure 3-14 *Setting the default debug script*

When the test script is ready to go, use Visual Studio's Debug | Start option or simply press F5 to launch the Test.sql that will execute your CLR database object. You can see an example of using the Visual Studio 2005 debugger to step through a SQL Server project in Figure 3-15.

At this point you can step through the code, set new breakpoints, and change and inspect variables.

NOTE

Debugging should be performed on a development system, not on a production system. Using the SQLCRL debugger from Visual Studio causes all SQLCLR threads to stop, which prevents other CLR objects from running.

.NET Database Object Security

No discussion of the new CLR features would be complete without a description of the security issues associated with using .NET assemblies and the SQL Server CLR.

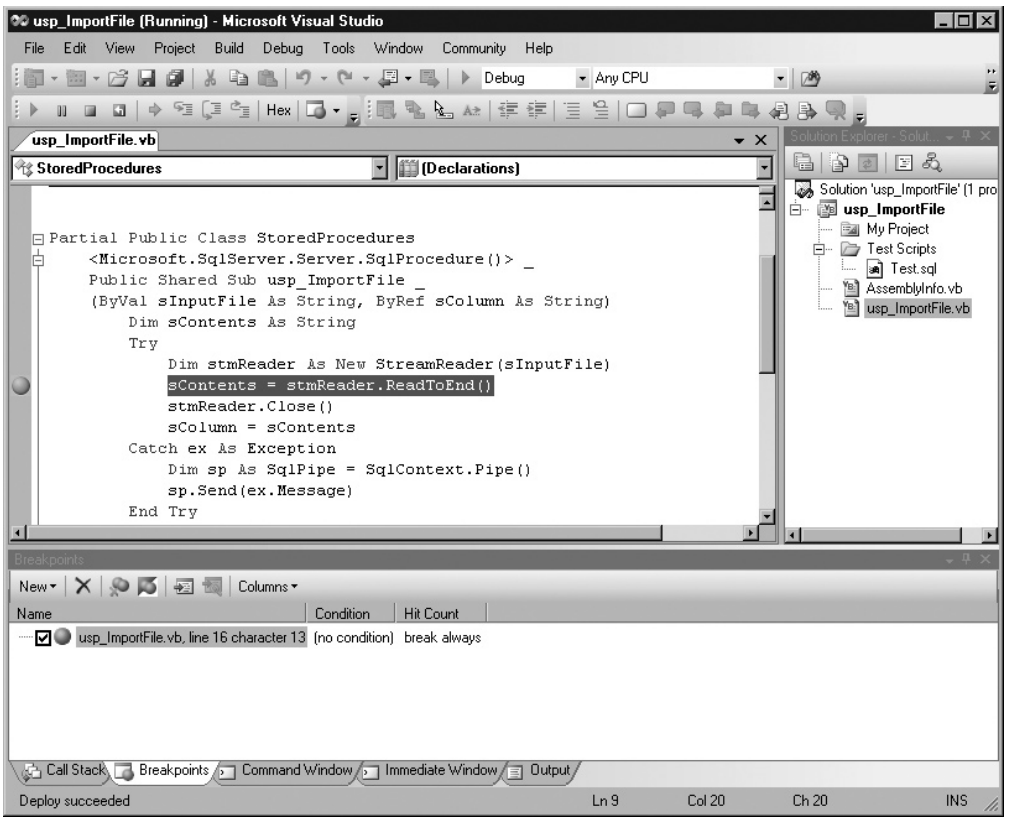

Figure 3-15 *Debugging Visual Studio 2005 SQL Server projects*

Unlike T-SQL, which doesn't have any native facilities for referencing resources outside the database, .NET assemblies are fully capable of accessing both system and network resources. Therefore, securing them is an important aspect of their development. With SQL Server 2005, Microsoft has integrated the user-based SQL Server security model with the permissions-based CLR security model. Following the SQL Server security model, users are able to access only database objects—including those created from .NET assemblies—to which they have user rights. The CLR security model extends this by providing control over the types of system resources that can be accessed by .NET code running on the server. CLR security permissions are specified at the time the assembly is created by using the WITH PERMISSION_SET clause of the CREATE ASSEMBLY statement. Table 3-4 summarizes the options for CLR database security permissions that can be applied to SQL Server database objects.

CRL Security	External Access Allowed	Calls to Unmanaged Code
SAFE	No external access	No calls to unmanaged code
EXTERNAL_ACCESS	External access permitted via management APIs	No calls to unmanaged code
UNSAFE	External access allowed	Calls to unmanaged code allowed

Table 3-4 *CLR Database Object Security Options*

Using the SAFE permission restricts all external access. The EXTERNAL_ ACCESS permission enables some external access of resources using managed APIs. SQL Server impersonates the caller in order to access external resources. You must have the new EXTERNAL_ACCESS permission in order to create objects with this permission set. The UNSAFE permission is basically an anything-goes type of permission. All system resources can be accessed, and calls to both managed and unmanaged code are allowed. Only system administrators can create objects with UNSAFE permissions.

In addition to using the CREATE ASSEMBLY statement, you can also set the CLR database object permission using the project properties as is shown in Figure 3-16.

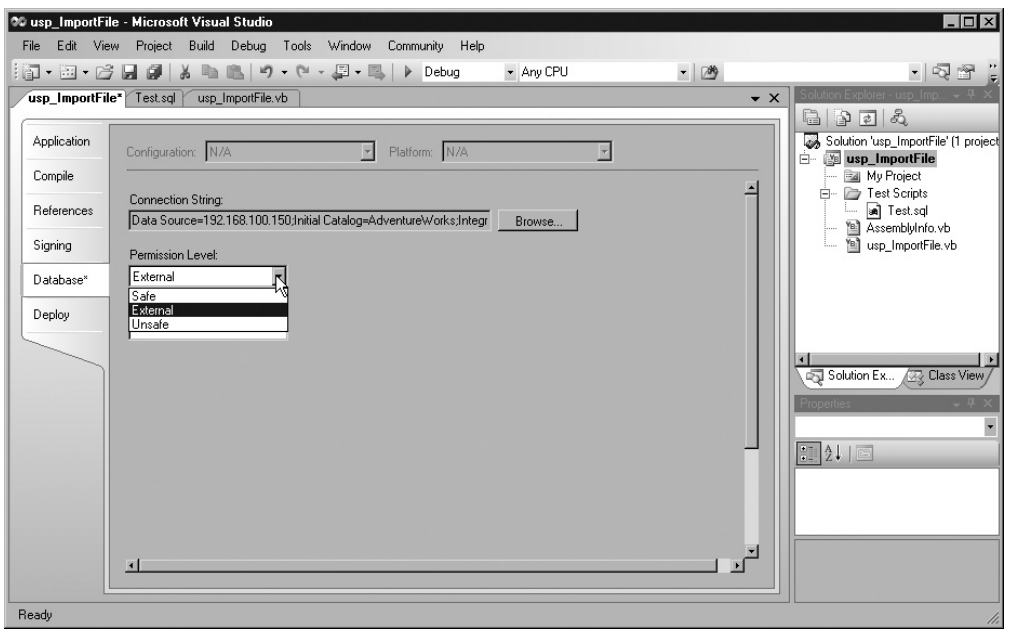

Figure 3-16 *Setting the CLR permission*

System View	Description
sys.objects	Contains all database objects. CLR database objects are identified in the typ_desc column.
sys.assemblies	Contains all of the assemblies in a database.
sys.assembly_files	Contains all of the filenames that were used to create the assemblies in a database.
sys.assembly_types	Contains all of the user-defined types that were added to a database.
sys.assembly_references	Contains all of the assembly references in a database.

Table 3-5 *System Views to Manage CLR Database Objects*

To interactively set the CLR permission level, open the project properties by selecting the Project | Properties option from the Visual Studio 2005 menu. Then open the Database tab and click the Permission Level drop-down. The project must be redeployed before the changes will take place.

Managing CLR Database Objects

As shown in Table 3-5, SQL Server 2005 provides system views that enable you to see the different CLR objects that are being used in the database.

Summary

Database objects created using the CLR are best suited for objects that replace extended stored procedures, require complex logic, or are potentially transportable between the database and the data tier of an application. They are not as well suited to raw data access and update functions as T-SQL. By taking advantage of CLR database objects, you can add a lot of power and flexibility to your database applications.