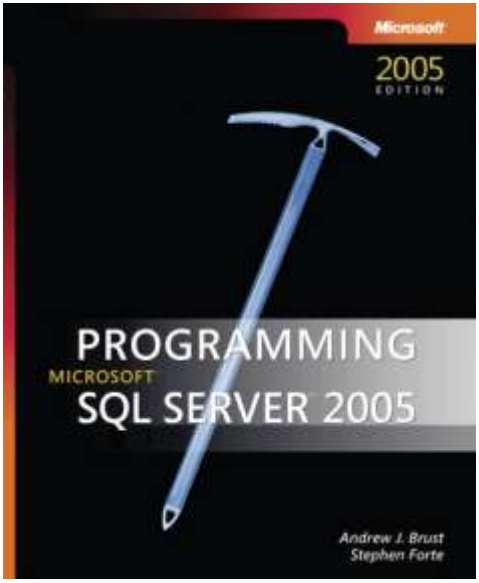

Programming Microsoft® SQL Server™ 2005

Andrew J. Brust,
Stephen Forte

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/9153.aspx

9780735619234
Publication Date: December 2004

Microsoft Press

Table of Contents

Part III Reporting and Business Intelligence

Chapter 3

An Overview of SQL CLR

–Andrew Brust

In this chapter:

The banner headline for Microsoft SQL Server 2005 is its integration of the Microsoft .NET common language runtime (CLR). This architectural enhancement means that SQL Server can use certain .NET classes as basic data types and can accommodate the use of .NET languages for the creation of stored procedures, triggers, and functions, and even user-defined aggregates.

Note Throughout this chapter, we will refer to the CLR integration in SQL Server as SQL CLR features, functionality, or integration, and we will refer to stored procedures, triggers, functions, aggregates, and user-defined types as the five basic SQL CLR entities.

Let's face facts: Transact SQL (T-SQL) is essentially a hack. Back when SQL Server was first developed, Microsoft and Sybase took SQL—a declarative, set-based language—and added variable declaration, conditional branching, looping, and parameterized subroutine logic to make it into a quasi-procedural language. Although extremely clever and useful, T-SQL lacked, and still lacks, many of the niceties of a full-fledged procedural programming language.

This shortcoming has forced some people to write T-SQL stored procedures that are overly complex and difficult to read. It has forced others to put logic in their middle-tier code that they would prefer to implement on the database. And it's even forced some people to abandon stored procedures altogether and use dynamic SQL in their applications, a practice that we do not endorse. Because of these workarounds to address T-SQL's procedural limitations, CLR integration is a welcome new feature in SQL Server, and it has caught the market's attention.

Meanwhile, T-SQL—and, one might argue, SQL itself—is vastly superior to procedural languages for querying and manipulating data. Its set-based syntax and implementation simply transcend the approach of procedurally iterating through rows of data. This is no less true in SQL Server 2005 than in previous versions of the product, and T-SQL has been greatly enhanced in this release (as detailed in Chapter 2), making it more valuable still.

So this places database application developers at a crossroads. We must simultaneously learn how the SQL CLR features work and develop a sophisticated, judicious sense of when to use T-SQL instead of the SQL CLR feature set. We must resist the temptation to completely ".NET-ify" our databases but learn to take advantage of the SQL CLR feature set where and when prudent. This chapter aims to help you learn to use SQL CLR features and to develop an instinct for their appropriate application.

In this chapter, you will learn:

- How to enable (or disable) SQL CLR integration on your SQL Server
- How SQL Server accommodates CLR code, through the loading of .NET assemblies
- How to use SQL Server 2005 and Visual Studio 2005 together to write SQL CLR code and deploy it, simply and quickly
- How to deploy SQL CLR code independently of Visual Studio, using T-SQL commands, with or without the help of SQL Server Management Studio
- How to create simple CLR stored procedures, triggers, functions, aggregates, and user-defined types, use them in your databases, and utilize them from T-SQL
- How both the standard SQL Server client provider and the new server-side library can be combined to implement SQL CLR functionality
- How SQL CLR security works and how to configure security permissions for your assemblies
- When to use SQL CLR functionality, and when to opt to use T-SQL instead

Getting Started: Enabling CLR Integration

Before you can learn how to use SQL CLR features, you need to know how to enable them. As with many new products in the Microsoft Windows Server System family, many advanced features of SQL Server 2005 are disabled by default. The reasoning behind this is sound: Each

additional feature that is enabled provides extra "surface area" for attacks on security or integrity of the product, and the added exposure is inexcusable if the feature goes unused.

The SQL CLR features of SQL Server 2005 are sophisticated and can be very useful, but they are also, technically, nonessential. It is possible to build high-performance databases and server-side programming logic without SQL CLR integration, so it is turned off by default.

Don't be discouraged, though: Turning on the feature is easy. Microsoft provides both a user-friendly GUI tool (aptly named the SQL Server Surface Area Configuration tool) and a system stored procedure for enabling or disabling SQL CLR integration. We'll cover both approaches.

To use the Surface Area Configuration tool, simply start it from the Configuration Tools subgroup in the Microsoft SQL Server 2005 programs group on the Windows start menu. Figure 3-1 shows the tool as it appears upon startup.

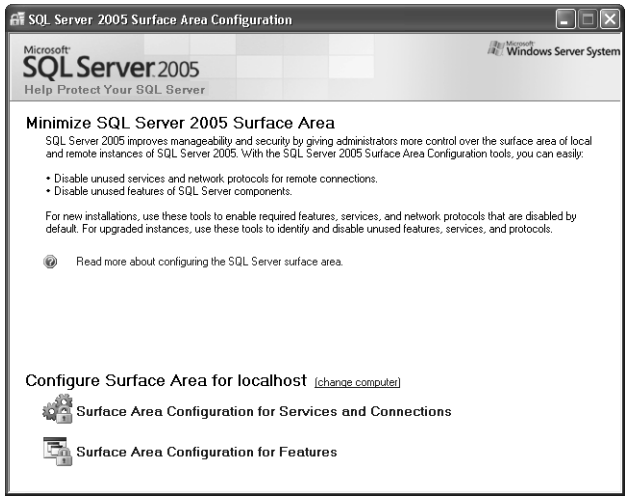

Figure 3-1 The SQL Server 2005 Surface Area Configuration tool

To configure CLR integration, click the Surface Area Configuration For Features link at the bottom of the form. After a short pause, the Surface Area Configuration For Features dialog box appears; a tree view-style list of features appears on the left, and the Ad Hoc Remote Queries feature is preselected. Click the CLR Integration node immediately below it, and you will see an Enable CLR Integration check box on the right of the form. (This is shown in Figure 3-2.) To enable SQL CLR features, make sure that the check box is checked, and click OK to close the Surface Area Configuration For Features window. (You can also clear the check box to disable SQL CLR integration.) Close the Surface Area Configuration tool by clicking its close box in the upper-right corner of the window.

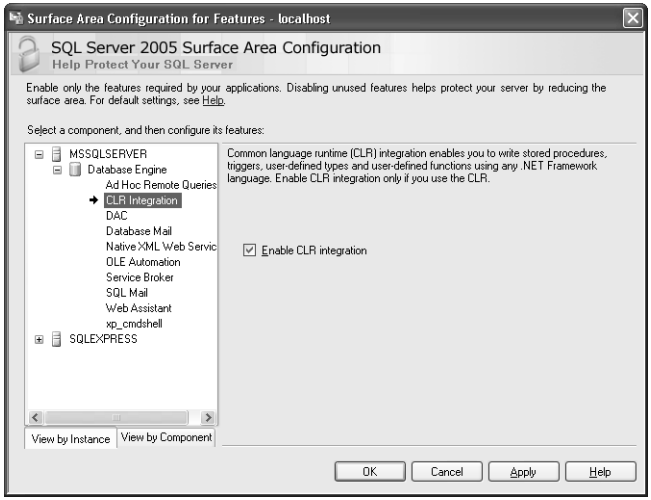

Figure 3-2 The Surface Area Configuration For Features window

If you'd prefer a command-line method for enabling or disabling SQL CLR functionality, open up SQL Server Management Studio and connect to the server you'd like to configure. Then, from a query window, type the following commands, and click the Execute button on the Management Studio SQL Editor toolbar.

```
sp_configure 'clr enabled', 1
GO
RECONFIGURE
GO
```

That's all there is to it! To disable SQL CLR integration, just use a value of 0, instead of 1, as the second parameter value in the *sp_configure* call.

Tip Don't forget that this will work from any tool that can connect to SQL Server, not just Management Studio. In fact, you could issue the previous command text from your own code using the ADO.NET *SqlCommand* object's *ExecuteNonQuery* method as long as your code can connect to your server and your sever can authenticate as a user in the sysadmin server role.

With SQL CLR integration enabled, you're ready to get started writing SQL CLR code. Before we dive in, we need to discuss Visual Studio/SQL Server integration and when to use it.

Visual Studio/SQL Server Integration

Visual Studio 2005 and SQL Server 2005 integrate tightly in a number of ways. It's important to realize, however, that the use of Visual Studio integration is completely optional and the use of T-SQL is a sufficient substitute. T-SQL has been enhanced with new DDL commands for

maintaining CLR assemblies, types, and aggregates, and its existing commands for stored procedures, triggers, and functions have been enhanced to recognize code within deployed assemblies. Visual Studio can execute those commands on your behalf. It can also make writing individual SQL CLR classes and functions easier.

Ultimately, we think all developers should be aware of both Visual Studio–assisted and more manual coding and deployment methods. You might decide to use one method most of the time, but in some situations you'll probably need the other, so we want to prepare you. As we cover each major area of SQL CLR programming, we will discuss deployment from both points of view. We'll cover some general points about Visual Studio integration now, and then we'll move on to cover SQL CLR development.

SQL Server Projects in Visual Studio

The combination of Visual Studio 2005 and SQL Server 2005 on the same development machine provides a special SQL Server Project type in Visual Studio and, within projects of that type, defined templates for the five basic SQL CLR entities. These templates inject specific code attributes and function stubs that allow you to create SQL CLR code easily. The attributes are used by Visual Studio to deploy your assembly and its stored procedures, triggers, and so on to your database. Some of them are also used by SQL Server to acknowledge and properly use your functions, user-defined types (UDTs), and aggregates.

To test out the new project type and templates, start Visual Studio 2005 and create a new project by using the File/New/Project... main menu option, the New Project toolbar button, the Ctrl+Shift+N keyboard accelerator, or the Create Project... hyperlink on the Visual Studio Start Page. In the New Project dialog box (Figure 3-3), select Database from the Project types tree view on the left (the Database node appears under the parent node for your programming language of choice; in Figure 3-3, the language is C#), and click the SQL Server Project icon in the Templates list on the right. Enter your own project name if you'd like, and click OK.

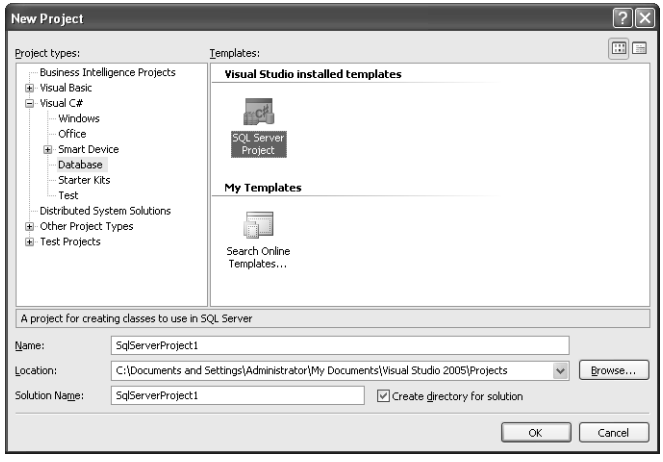

Figure 3-3 The Visual Studio 2005 New Project dialog box with the SQL Server project type selected

Next, the Add Database Reference dialog box appears (Figure 3-4).

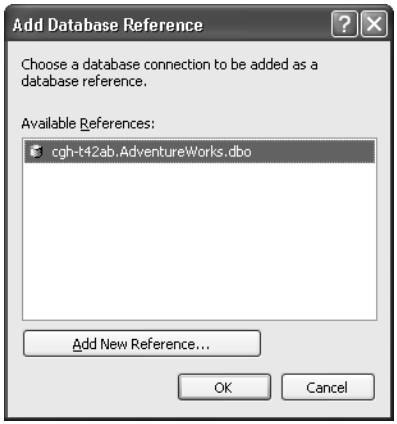

Figure 3-4 The Add Database Reference dialog box

Because Visual Studio provides automated deployment of your SQL CLR code, it must associate your project with a specific server and database via a database reference (connection). Any database connections that have already been defined in the Server Explorer window appear in this window, as does an Add New Reference button that allows you to define a new connection, if necessary. Pick an existing connection or define a new one, and then click OK. The project opens.

Note If no data connections have already been defined in the Server Explorer window, the New Database Reference dialog box will appear in place of the Add Database Reference dialog box. In the New Database Reference dialog box, you may specify server, login, and database details for a new database connection that will be used by your project as its database reference and added to the Server Explorer as a new data connection.

You can easily add preconfigured classes for the five basic SQL CLR entities to your project. You can do this in a number of ways: directly from the Project menu or from the Add submenu on the Server Explorer's project node shortcut menu (Figure 3-5).

You can also add the preconfigured classes from the Add New Item dialog box (Figure 3-6), which is available from the Project/Add New Item... option on the main menu, or the Add/New Item... option on the Solution Explorer project node's shortcut menu.

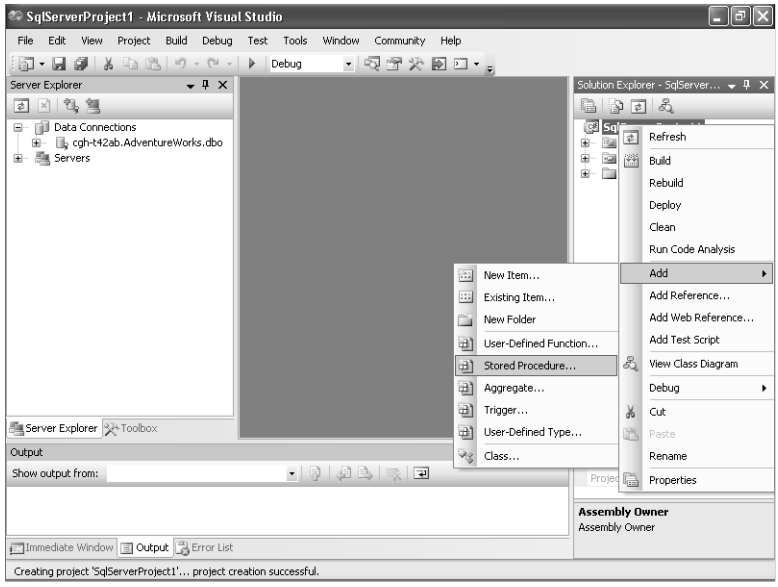

Figure 3-5 The Server Explorer project node shortcut menu and its Add submenu

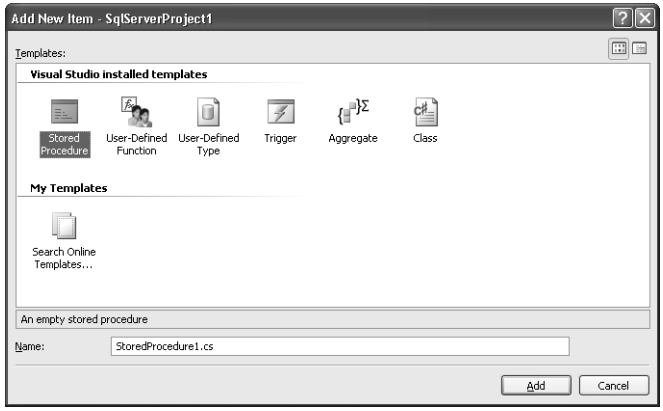

Figure 3-6 The Visual Studio SQL Server project Add New Item dialog box

Automated Deployment

Once opened, SQL Server projects add a Deploy option to the Visual Studio Build menu. In addition, the Play (Start Debugging) button and the Start Debugging, Start Without Debugging, and Step Over options on the Debug menu (and their keyboard shortcuts F5, Ctrl+F5, and F10, respectively) all deploy the project assembly in addition to performing their listed function.

Visual Studio can do a lot of deployment work for you. But as you'll learn, you can do so on your own and, in certain circumstances, have more precise control over the deployment process when you do so.

SQL CLR Code Attributes

A number of .NET code attributes are provided for SQL CLR developers; these are contained in the *Microsoft.SqlServer.Server* namespace. Many of them are inserted in your code when you use the various templates in the SQL Server project type, as is a *using* statement for the *Microsoft.SqlServer.Server* namespace itself. If you choose to develop code without these templates, you must add the appropriate attributes, and optionally the *using* statement, yourself. Although all these attributes are provided in the same namespace, some are used exclusively by Visual Studio and others are used by both Visual Studio and SQL Server.

Covering all SQL CLR attributes and their parameters would itself require an entire chapter, so our coverage will be intentionally selective. Specifically, we will provide coverage of the *SqlProcedure*, *SqlFunction*, *SqlTrigger*, *SqlUserDefinedAggregate*, and *SqlUserDefinedType* attributes. We will not cover the *SqlFacet* and *SqlMethod* attributes.

Just as certain attributes are not covered here, we cover only some of the parameters accepted by the attributes that we do cover. And in some cases, we cover only certain of the possible values that can be passed to these attributes. For example, *SqlFunction* accepts several parameters but the only ones we will cover are *Name*, *FillRowMethodName*, and *TableDefinition*. For *SqlUserDefinedAggregate* and *SqlUserDefinedType*, we will cover only a single value setting for the *Format* parameter, and will not cover the several other parameters those two attributes accept.

The coverage we provide will be more than sufficient for you to implement basic, intermediate, and certain advanced functionality with all the basic five SQL CLR entities. The attributes and parameters that we won't cover are useful mostly for optimizing your SQL CLR code, and they are well documented in SQL Server Books Online and articles on MSDN.

About the Sample Code

The sample .NET code for this chapter is provided in two versions. The primary material is supplied as a Visual Studio SQL Server project, accessible by opening the solution file Chapter03.sln in the Chapter03 subfolder of this chapter's VS sample code folder. We also supply the code as a standard Class Library project, accessible by opening the solution file Chapter03Manual.sln in the Chapter03Manual subfolder. The code in each project is virtually identical, although the Class Library project does not autodeploy when the various Build and Debug options are invoked in Visual Studio 2005. As we cover each SQL CLR feature, we'll discuss how automated deployment takes place from the SQL Server project and how command-driven deployment should be performed for the Class Library project.

> We'll also discuss executing test scripts from within Visual Studio for the SQL Server project and from SQL Server Management Studio for the Class Library project. As a companion to those discussions, we also provide a Management Studio project, accessible by opening Chapter03.ssmssln in this chapter's SSMS folder. This project consists of a number of SQL scripts used for testing the sample SQL CLR code and a script for cleaning up everything in the database created by the sample code and tests. The project also contains a script file called CreateObjects.sql, which deploys the Class Library assembly and the SQL CLR entities within it.

Your First SQL CLR Stored Procedure

Although SQL CLR programming can get quite complex and involved, it offers in reality a simple model that any .NET developer can use with high productivity in relatively short order. That's because the crux of SQL CLR functionality is nothing more than the ability of SQL Server 2005 to load .NET assemblies into your database and then to allow you to use the procedures, functions, and types within the assembly as you define your columns, views, stored procedures, triggers, and functions.

To give you a good understanding of SQL CLR integration, we must go through its features and techniques carefully. Before doing so, however, let's quickly go through an end-to-end scenario for creating and executing a SQL CLR stored procedure. This will make it easier for you to understand the individual features as we describe them.

Strictly speaking, any .NET class library assembly (in certain cases using appropriate .NET code attributes in its classes and functions) can be loaded into your database with a simple T-SQL command. To see how easily this works, start up Management Studio and open a query window using a connection to the AdventureWorks sample database. In the sample code folder for this chapter, confirm that the file Chapter03.dll is located in the VS\Chapter03Manual\Chapter03\bin\Debug subfolder. If the parent folder were C:\ProgrammingSQL2005\Chapter03, you would load the assembly into the AdventureWorks database with the following T-SQL command:

```
CREATE ASSEMBLY Chapter03
FROM 'C:\ProgrammingSQL2005\Chapter03\VS\Chapter03Manual\Chapter03\bin\Debug\Chapter03.dll'
```

There are other syntax options for the *CREATE ASSEMBLY* command, but for now we'll focus on the previous limited usage.

Functions in an assembly that reside within a class and perform local computational tasks and/or certain types of data access can be easily exposed as SQL Server stored procedures, triggers, or functions. As with conventional stored procedures, triggers, and functions, all it takes is a simple T-SQL *CREATE PROCEDURE*, *CREATE TRIGGER*, or *CREATE FUNCTION*

command to make this happen. We'll go through each of these options in this chapter, but let's cut to the chase and create a simple CLR stored procedure right now.

You can view the source code for the Chapter03 assembly by opening the solution file VS\Chapter03Manual\Chapter03Manual.sln in this chapter's sample code folder. Within the project, the file Sprocs.cs contains the following code:

```
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;

public partial class Sprocs
{
    public static void spContactsQuick()
    {
        SqlContext.Pipe.ExecuteAndSend(new SqlCommand("Select * from Person.Contact"));
    }
};
```

The code within the procedure is designed to connect to the database in which its assembly has been loaded (AdventureWorks), perform a *SELECT* * against the Person.Contact table, and use special server-side objects to send the data back to the client application. To make this CLR code available via SQL Server as a stored procedure, also called *spContactsQuick*, you simply execute the following command from the Management Studio query window you opened previously.

```
CREATE PROCEDURE spContactsQuick
AS EXTERNAL NAME
Chapter03.Sprocs.spContactsQuick
```

Important Be sure to enter the *Sprocs.spContactsQuick* portion of the command verbatim. This phrase is case-sensitive.

To test the SQL CLR stored procedure, run it from a Management Studio query window as you would any conventional stored procedure:

```
EXEC spContactsQuick
```

Or simply:

```
spContactsQuick
```

Management Studio should respond by displaying the contents of the Person.Contact table in the Results tab of the query window.

As you can see from this rather trivial example, writing a CLR stored procedure can be very easy and is a lot like writing client-side or middle-tier code that performs data access using

ADO.NET. The biggest differences involve the provision of a database connection and the fact that the data must be "piped" back to the client rather than loaded into a *SqlDataReader* and returned, manipulated, or displayed through a UI. In addition, the presence of the *SqlContext* object differentiates SQL CLR code from conventional .NET data access code. We'll cover the use of the *SqlContext* object and its *Pipe* property in the next section.

The bits of T-SQL and C# code just shown certainly don't tell the whole SQL CLR story. The use of the *ExecuteAndSend* method allowed us to skip over a number of otherwise important concepts. There are three ways to deploy assemblies, and you've seen only a simplified version of one of those ways. Security considerations must be taken into account, and we haven't even begun to look at triggers, functions, aggregates, or UDTs. So although the example showed how easy SQL CLR programming can be, we'll now take our time and show you the nooks and crannies.

CLR Stored Procedures and Server-Side Data Access

Our previous "quick and dirty" sample looked at CLR stored procedure development, but we need to cover that topic more thoroughly now. We've already covered the mechanics of writing and deploying a stored procedure, but let's back up a bit and try and understand how CLR stored procedures work from a conceptual standpoint.

SQL CLR stored procedure code runs in an instance of the .NET CLR that is hosted by SQL Server itself; it is not called as an external process, as COM-based extended stored procedures (XPs) would be. Because SQL CLR code runs in the context of the server, it treats objects in the database as native, *local* objects, more or less. Likewise, it must treat the client that calls it as *remote*. This contextual environment is, in effect, the opposite of that under which client and middle-tier ADO.NET code runs. This takes a little getting used to, but once you've mastered thinking about things this way, SQL CLR code becomes easy to write and understand.

Meanwhile, as .NET has no intrinsic way of accessing local objects on the server or transmitting data and messages to the client, you must use a special set of classes to perform these tasks. These classes are contained in the *Microsoft.SqlServer.Server* namespace.

Note As an aside, it is interesting and important to note that the *Microsoft.SqlServer.Server* namespace is actually supplied by the *System.Data* Framework assembly. This means that you don't need to worry about adding a reference to your project to use this namespace. The namespace's location within *System.Data* also further emphasizes the tight integration between .NET and SQL Server.

If you'd like, you can think of *Microsoft.SqlServer.Server* as a helper library for *System.Data.SqlClient*. It supplies the SQL CLR code attributes we already mentioned, a few enumerations, an exception class, an interface, and five classes: *SqlContext*, *SqlPipe*, *SqlTriggerContext*, *SqlMetaData*, and *SqlDataRecord*. We'll cover *SqlMetaData* and *SqlDataRecord* at the end of this

section, and we'll cover *SqlTriggerContext* when we discuss CLR triggers. We'll cover the *SqlContext* and *SqlPipe* objects right now.

At a high level, the *SqlContext* object, which is *static*, provides a handle to the server-side context in which your code runs. It also has a channel to the client through which you can return data and text: its *Pipe* property, which in turn provides access to a properly initiated *SqlPipe* object.

A *SqlPipe* object can send data and messages to the calling client though several methods: *Send*, *SendResultsStart*, *SendResultsRow*, *SendResultsEnd*, and *ExecuteAndSend*. In the previous code sample, we used the *SqlPipe* object's *ExecuteAndSend* method to implicitly open a connection, call *ExecuteReader* on an *SqlCommand* object that uses that connection, and transmit the contents of the resulting *SqlDataReader* back to the client. Although the implicit work done by *ExecuteAndSend* might have been convenient, it's important to avoid such shortcuts in our detailed discussion on SQL CLR programming.

In general, SQL CLR stored procedure code that queries tables in the database must open a connection to that database, use the *SqlCommand* object's *ExecuteReader* method to query the data, and then use one or a combination of the *Send* methods to send it back. The *Send* methods do not accept *DataSet* objects; they accept only *SqlDataReader* objects, strings, and/or special *SqlDataRecord* objects. Listing 3-1, which shows the implementation of the function *spContacts* from spTest.cs in the sample project, is a representative example of how this is done.

Listing 3-1 *spContacts* from spTest.cs

```
[SqlProcedure]
public static void spContacts()
{
    SqlConnection conn = new SqlConnection("context connection=true");
    SqlCommand cm = new SqlCommand("Select * from Person.Contact", conn);

    conn.Open();
    SqlDataReader dr = cm.ExecuteReader();
    SqlContext.Pipe.Send("Starting data dump");
    SqlContext.Pipe.Send(dr);
    SqlContext.Pipe.Send("Data dump complete");
    dr.Close();
    conn.Close();
}
```

For this code to work, we need to use both the *Microsoft.SqlServer.Server* and *System.Data.SqlClient* namespaces (and if you look in the sample project rather than Listing 3-1, you'll see that we have). This is because any conventional ADO.NET objects we might use, such as *SqlConnection*, *SqlCommand*, and *SqlDataReader*, are supplied to us from *System.Data.SqlClient*, just as they would be in a conventional client application or middle-tier assembly. As already discussed, we need the *Microsoft.SqlServer.Server* namespace in order to use objects such as *SqlContext* and

SqlPipe. The stored procedure template in Visual Studio SQL Server projects includes the *using* statement for *Microsoft.SqlServer.Server* and *System.Data.SqlClient* automatically.

Note Readers who worked with early beta versions of SQL Server 2005 might recall a *System.Data.SqlServer* library, which in effect supplied all conventional and server-side ADO.NET objects necessary to write SQL CLR code. This hybrid library was eliminated and replaced with the dual-library approach later in the beta process.

Although server-side code uses *SqlClient* objects, it does so in a specialized way. For example, notice that the *context connection=true* connection string passed to the *SqlConnection* object's constructor. This essentially instructs ADO.NET to open a new connection to the database in which the CLR assembly resides. Notice also the second call to the *SqlContext.Pipe* object's *Send* method. Here, the *SqlDataReader* parameter overload of the *SqlPipe* object's *Send* method is used to push the contents of the *SqlDataReader* back to the client. You can think of this method as performing a *while (dr.Read())* loop through the *SqlDataReader* and echoing out the values of each column for each iteration of the loop, but instead of having to do that work yourself, the *Send* method does it for you.

Before and after the *SqlDataReader* is piped, we use the *String* parameter overload of the *Send* method to send status messages to the client. When this stored procedure is run in Management Studio, the piped text appears on the Results tab of the query window when you use the Management Studio Results To Text option and on the Messages tab when you use the Results To Grid option.

The rest of the listing contains typical ADO.NET code, all of it using objects from the *SqlClient* provider. And that illustrates well the overall theme of SQL CLR programming: Do what you'd normally do from the client or middle tier, and use a few special helper objects to work within the context of SQL Server as you do so.

Piping Data with *SqlDataRecord* and *SqlMetaData*

We mentioned that the *SqlPipe* object's *Send* method can accept an object of type *SqlDataRecord*, and we mentioned previously that *Microsoft.SqlServer.Server* provides this object as well as an object called *SqlMetaData*. You can use these two objects together in a CLR stored procedure to return a result set one row at a time, instead of having to supply the *SqlPipe* object's *Send* method with an *SqlDataReader*. This allows (but does not require) you to inspect the data before sending it back to the client. Sending *SqlDataReader* objects prevents inspection of the data within the stored procedure because *SqlDataReader* objects are forward-only result set structures. Using the *ExecuteAndSend* method and an *SqlCommand* object has the same limitation.

The *SqlDataRecord* object permits .NET code to create an individual record/row to be returned to the calling client. Its constructor accepts an array of *SqlMetaData* objects, which in turn describe the metadata for each field/column in the record/row.

Listing 3-2, which shows the implementation of function *spContactCount* from spTest.cs in the sample project, illustrates how to use *SqlPipe.Send* together with *SqlDataRecord* and *SqlMetaData* objects to return a single-column, single-row result set from a stored procedure.

Listing 3-2 *spContactCount* from spTest.cs

```
[SqlProcedure()]
public static void spContactCount()
{
    SqlConnection conn = new SqlConnection("context connection=true");
    SqlCommand cm = new SqlCommand("Select Count(*) from Person.Contact", conn);
    SqlDataRecord drc = new SqlDataRecord(new SqlMetaData("ContactCount", SqlDbType.Int));

    conn.Open();
    drc.SetInt32(0, (Int32)cm.ExecuteScalar());
    SqlContext.Pipe.Send(drc);
    conn.Close();
}
```

The code declares variable *drc* as a *SqlDataRecord* object and passes its constructor a single *SqlMetaData* object. (Passing a single object rather than an array is permissible if the *SqlDataRecord* object will only have a single field/column.) The *SqlMetaData* object describes a column called *ContactCount* of type *SqlDbType.Int*.

Note The *SqlDbType* enumeration is contained within the *System.Data.SqlTypes* namespace. The SQL Server Stored Procedure template inserts a *using* statement for this namespace. If you are creating SQL CLR code without using this template, you should add the *using* statement yourself.

The rest of the code is rather straightforward. First, a context connection and command are opened and a *SELECT COUNT(*)* query is performed against the AdventureWorks Person.Contact table. Because the query returns a single scalar value, it is run using the *SqlCommand* object's *ExecuteScalar* method. Next, the value returned by *ExecuteScalar* is casted into an integer and that value is loaded into field/column 0 (the only one) of the *SqlDataRecord* object using its *SetInt32* method. The *SqlDataRecord* is then piped back to the client using the *SqlContext* object's *Send* method.

Note If we wanted to send back multiple *SqlDataRecord* objects, we would send the first one using the *SqlContext* object's *SendResultsStart* method and then send all subsequent *SqlDataRecord* objects using the *SendResultsRow* method. We would call the *SendResultEnd* method after all *SqlDataRecord*s had been sent.

Once the stored procedure has been deployed (the techniques for which we will discuss shortly), you can execute it from SQL Server Management Studio as you would any other stored procedure. Although the result is a single value, it is presented as a column and the

column name ContactCount is shown on the Results tab of the query window. Keep in mind that this *COUNT(*)* query result could have been returned without using the *SqlMetaData* and *SqlDataRecord* objects; the sample is provided to demonstrate the use of these objects as an alternative to piping *SqlDataReader* objects and text to the client.

CLR Stored Procedure Usage Guidelines

It's important to understand how to perform data access and retrieval in CLR stored procedures. As a .NET developer, you already know how to do more computational tasks within your code, so our samples illustrate server-side data access more than anything else. As proof-of-concept code, these samples are completely adequate.

Meanwhile, you should avoid writing CLR stored procedures that merely perform simple "CRUD" (Create, Retrieve, Update, and Delete) operations. Such tasks are better left to conventional T-SQL stored procedures, which typically perform these operations more efficiently than ADO.NET can. CLR stored procedures work well when you need to perform computation on your data and you need the expressiveness of a .NET language to do so (where such expressiveness is missing from T-SQL).

For example, implementing a "fuzzy search" using business logic embedded in .NET assemblies to determine which data has an affinity to other data is a good use of SQL CLR stored procedures. Regular-expression-based data validation in an update or insert stored procedure is another good application of SQL CLR integration. As a general rule, straight data access should be left to T-SQL. "Higher-valued" computations are good candidates for SQL CLR integration. We'll revisit the SQL CLR usage question at various points in this chapter.

Deployment

Before you can test your SQL CLR code, you must deploy the assembly containing it and register the individual functions that you want recognized as stored procedures. A number of deployment methods are at your disposal; we will pause to cover them now, before discussing testing of your stored procedures and the other four basic SQL CLR entities.

Deploying Your Assembly

As mentioned earlier, Visual Studio deploys the SQL Server project version of the sample code when you build, start, or step through the project or use the Build/Deploy function on Visual Studio's main menu. If you're working with the SQL Server project version of the samples, go ahead and use the Deploy option or one of the Start or Build options in Visual Studio now.

For deploying the Class Library project version, assuming C:\ProgrammingSQL2005\ Chapter03 as this chapter's sample code parent directory, you can execute the following T-SQL command

from within Management Studio:

```
CREATE ASSEMBLY Chapter03
AUTHORIZATION dbo
FROM 'C:\ProgrammingSQL2005\Chapter03\VS\Chapter03Manual\Chapter03\bin\Debug\Chapter03.dll'
WITH PERMISSION_SET = SAFE
GO
```

The *AUTHORIZATION* clause allows you to specify a name or role to which ownership of the assembly is assigned. The default authorization is that of the current user, and because you are most likely logged in as dbo for AdventureWorks, in this case the clause is unnecessary (which is why we omitted it from our previous example).

The meaning and effect of the *WITH PERMISSION_SET* clause are discussed at the end of this chapter. For now, just note that this clause allows you to specify the security permissions with which your assembly runs. As with the *AUTHORIZATION* clause, in this case the *WITH PERMISSION_SET* clause is technically unnecessary because *SAFE* is the default *PERMISSION_SET* value used when a *CREATE ASSEMBLY* command is executed.

If your assembly has dependencies on other assemblies, SQL Server looks to see if those assemblies have already been loaded into the database and, if so, confirms that their ownership is the same as that of the specified assembly. If the dependent assemblies have not yet been loaded into the database, SQL Server looks for them in the same folder as the specified assembly. If it finds all dependent assemblies in that location, it loads them and assigns them the same ownership as the primary assembly. If it does not find the dependent assemblies in that folder, the *CREATE ASSEMBLY* command will fail.

You can supply a string expression instead of a literal in the *FROM* clause, allowing for some interesting data-driven possibilities. For example, you could fetch an assembly path reference from a table in your database. It is also possible to supply a bitstream in the *FROM* clause instead of a file specification. You do this by specifying a varbinary literal value or expression (or a comma-delimited list of varbinary values or expressions, when dependent assemblies must be specified) that contains the actual binary content of your assembly (or assemblies). This allows the creation of a database, including any CLR assemblies it contains, to be completely scripted, without requiring distribution of actual assembly files. The binary stream can be embedded in the script itself or, using an expression, it can be fetched from a table in a database.

More Info See SQL Server Books Online for more information on this option.

In addition to using Visual Studio deployment and the *T-SQL CREATE ASSEMBLY* command, you can upload the assembly into your database interactively from Management Studio. Simply right-click the *servername*/AdventureWorks/Programmability/Assemblies node in the Object Explorer (where *servername* is the name of your server) and choose New Assembly... from the shortcut menu. The New Assembly dialog box, shown in Figure 3-7, appears.

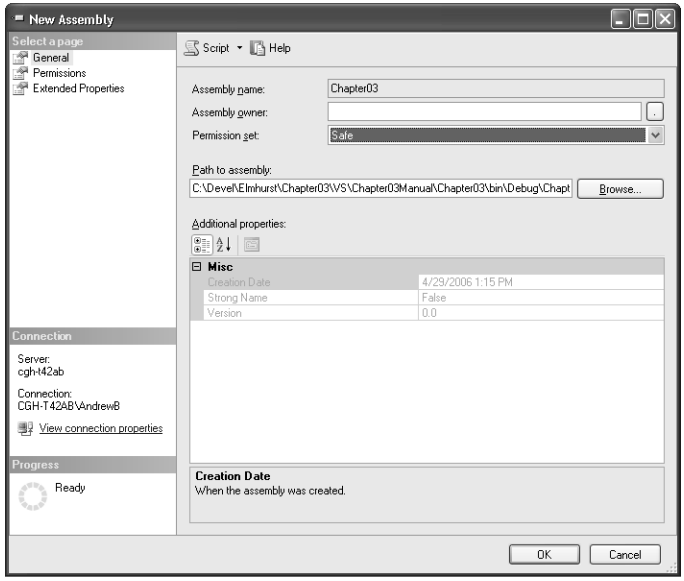

Figure 3-7 The Management Studio New Assembly dialog box

Type the assembly path and file name in the Path To Assembly text box, or use the Browse... button to specify it interactively. You can specify *AUTHORIZATION* and *WITH PERMISSION_SET* details in the Assembly Owner text box (using the ellipsis button, if necessary) and the Permission Set combo box, respectively.

Regardless of the deployment method you use, once your assembly has been added to your database, it becomes an integral part of that database and its underlying MDF file. This means if your database is backed up and restored, or xcopy deployed, any assemblies within it move along with the data itself and need not be manually added as a subsequent step.

Deploying Your Stored Procedures

In the SQL Server project version of the sample code, deployment of all the stored procedures is handled by Visual Studio when the assembly itself is deployed. This is due to the application of the *SqlProcedure* attribute to the functions in class *StoredProcedures* (found in file spTest.cs). The *SqlProcedure* attribute accepts an optional *Name* parameter, the value of which is the actual callable stored procedure name. If you do not supply a value for the *Name* parameter, the name of the .NET function is used as the stored procedure name.

The *SqlProcedure* attribute is used only by Visual Studio in SQL Server projects. Therefore, it has been removed from the source code in the Class Library project. Deploying the stored procedures from that version of the source code requires issuing a *CREATE PROCEDURE* T-SQL command using the new *EXTERNAL NAME* clause to specify the assembly, fully qualified class name specifier, and function name. For example, to load the Class Library version of *spContacts*, you would issue the following command.

```
CREATE PROCEDURE spContacts
AS EXTERNAL NAME Chapter03.StoredProcedures.spContacts
```

The preceding command specifies that function *spContacts*, in class *StoredProcedures*, in the loaded assembly with T-SQL name *Chapter03*, should be registered as a CLR stored procedure callable under the name *spContacts*.

Note All necessary *CREATE PROCEDURE* commands for the Class Library project version of the sample code are contained in the CreateObjects.sql script in the Management Studio project supplied with the sample code. You will need to run that script in order to execute the various SQL CLR entities implemented in the Class Library project.

Note that had the CLR stored procedure been written in Visual Basic .NET rather than C#, the class name specifier would change to *Chapter03.StoredProcedures*. This would necessitate a change to the deployment T-SQL code as follows:

```
CREATE PROCEDURE spContacts
AS EXTERNAL NAME Chapter03.[Chapter03.StoredProcedures].spContacts
```

In Visual Basic projects, the default namespace for a project itself defaults to the project name, as does the assembly name. The class within the project must be referenced using the default namespace as a prefix. Because the class specifier is a multipart dot-separated name, it must be enclosed within square brackets so that SQL Server can identify it as a single indivisible name. Because C# projects handle the default namespace setting a little differently, the namespace prefix is not used in the class specifier for C# assemblies.

One last point before we discuss how to test your now-deployed CLR stored procedures. It is important to realize that the class specifier and function name in the *EXTERNAL NAME* clause are *case-sensitive* and that this is true *even for assemblies developed in Visual Basic .NET*. Although this point perplexed us quite a bit at first, it does make sense in hindsight. SQL Server searches for your subs/functions within your assemblies, not within your source code. In other words, it's looking within Microsoft Intermediate Language (MSIL) code, not Visual Basic .NET or C# source code. Because MSIL is case-sensitive (it has to be, to support case-sensitive languages like C#), SQL Server must be as well as it searches for a specific class and sub/function.

The fact that SQL Server is not case sensitive by default (even though it once was) and that Visual Basic .NET is not a case-sensitive language is of no import! If you attempt to register a sub/function and you receive an error that it cannot be found within the assembly, double-check that the case usage in your command matches that of your source code.

Testing Your Stored Procedures

With your assembly and stored procedures now deployed, you're ready to run and test them. Typically, you should do this from Management Studio; however, Visual Studio SQL Server projects allow you to test your SQL CLR code from Visual Studio itself. When you create a Visual Studio SQL Server project, a folder called Test Scripts is created as a subdirectory in your source code directory. Within that subdirectory, Visual Studio creates a script file called Test.sql. If you look at that file, you will see that it contains commented instructions as well as commented sample T-SQL code for testing stored procedures, functions, and UDTs. It also contains an uncommented generic *SELECT* command that echoes a text literal to the caller.

Visual Studio connects to your database and runs this script immediately after your assembly is deployed, and the output from the script appears in Visual Studio's Output window. This allows you to execute any number of T-SQL commands directly from Visual Studio without having to switch to another tool. Although this approach is much less interactive than a Management Studio query window, it allows you to run quick tests against your code. It is especially useful for regression testing—that is, confirming that a new version of your assembly does not break older, critical functionality.

The file extension of the script must be .sql, but otherwise the name of the file is inconsequential. You can have multiple script files in the Test Scripts folder. To add a new one, right-click the Test Scripts folder node or the project node in the Solution Explorer window and select the Add Test Script option from the shortcut menu. Only one script can be active at one time, and as soon as you have more than one script, you must specify which one is active. To make a script active, simply right-click its node in the Solution Explorer window and select the Set As Default Debug Script option from its shortcut menu. When you do so, the node is displayed in bold. You may run or debug a script even if it is not the active script. To do so, right-click its node in the Solution Explorer window and select the Debug Script option from its shortcut menu.

Warning At press time, there appears to be an anomaly in the working of the test script facility and the Output window in Visual Studio 2005. Simply put, if your test script executes a query (whether it be a T-SQL *SELECT* command or a call to a stored procedure) that returns a column of type uniqueidentifier (GUID), the query's result set will not appear in the Output window, and execution of the test script might hang Visual Studio. For this reason, you should avoid calling the sample code CLR stored procedures *spContactsQuick* and *spContacts* (both of which perform a *SELECT * FROM Person.Contact* query and thus retrieve the rowguid column, which is of type uniqueidentifier) from your test script and instead test these procedures from SQL Server Management Studio, where the anomaly does not occur. You can safely call *spContactCount*, which simply performs a *SELECT COUNT(*) FROM Person.Contact* query, from your Visual Studio test script. Alternatively, you can modify *spContactsQuick* and/or *spContacts* to select specific columns from the Person.Contact table, making sure that *rowguid* is not one of them.

If you're working with the Class Library version of the sample code, you must test the stored procedures from Management Studio or another SQL Server query tool. Even if you are working with the SQL Server project version, you'll find that testing your SQL CLR code in Management Studio provides a richer experience and more flexibility.

The script file TestStoredProcs.sql in the Management Studio project supplied with the sample code will run both of our CLR stored procedures (*spContactCount* and *spContacts*). Open the file in Management Studio, and click the Execute button on the SQL Editor toolbar, choose the Query/Execute option on the main menu, or press F5. (You can also right-click the query window and select Execute from the shortcut menu.)

When the script runs, you should see the single-valued result of the *spContactCount* stored procedure appear first, as shown in Figure 3-8. Note that the column name ContactCount appears on the Results tab and recall that this is a direct result of your using the *SqlMetaData* object in the CLR code. Below the *spContactCount* result, you will see the results from the *spContacts* stored procedure come in. Because the Person.Contact table has almost 20,000 rows, these results might take some time to flow in.

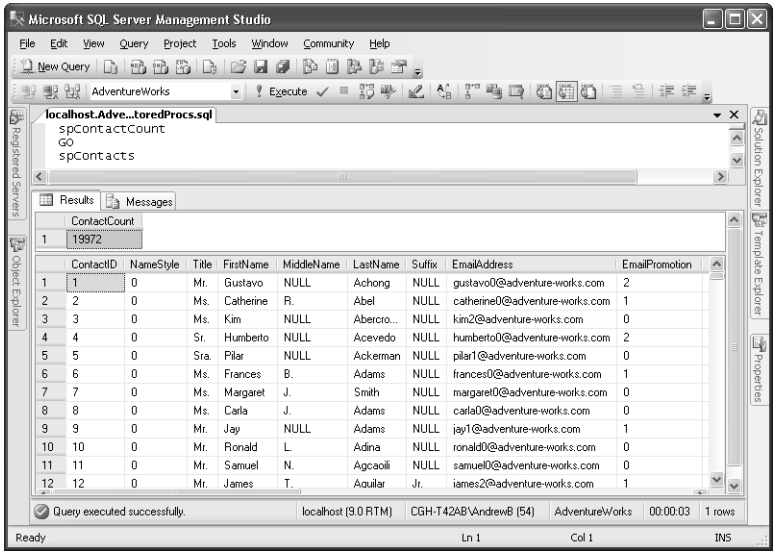

Figure 3-8 TestStoredProcs.sql script code and results

Even while the results are coming in, the "Starting data dump" status message should be visible on the Messages tab (or on the Results tab if you're using Management Studio's Results To Text option). Once all rows have been fetched, you should see the "Data dump complete" message appear as well. If you get impatient and want to abort the query before all rows have been fetched, you can use the Cancel Executing Query button on the SQL Editor toolbar or the Query/Cancel Executing Query option on the main menu; you can also use the Alt+Break keyboard shortcut.

We have yet to cover CLR functions, triggers, aggregates, and UDTs, but you have already learned most of the skills you need to develop SQL CLR code. You have learned how to create Visual Studio SQL Server projects and use its autodeployment and test script features. You have also learned how to develop SQL CLR code in standard Class Library projects and to use T-SQL commands and Management Studio to deploy the code for you. You've learned about the subtle differences between deploying C# code and Visual Basic .NET code, and we've covered the case-sensitive requirements of T-SQL-based deployment.

With all this under your belt, we can cover the remaining four basic SQL CLR entities relatively quickly.

CLR Functions

Let's take everything we've discussed about CLR stored procedures and deployment and apply it to CLR functions. As any programmer knows, a function is a lot like a procedure, except that it returns a value (or an object). Mainstream .NET functions typically return .NET types. SQL CLR functions, on the other hand, must return a *SqlType*. So to start with, we need to make sure our classes that implement SQL CLR functions import/use the *System.Data.SqlTypes* namespace. The SQL Server Project template for User Defined Functions contains the appropriate *using* code by default; you must add the code manually to standard Class Library class code.

Once the namespace is imported, you can write the functions themselves. In Visual Studio SQL Server Projects, they should be decorated with the *SqlFunction* attribute; this attribute accepts an optional name parameter that works identically to its *SqlProcedure* counterpart. In our sample code, we will not supply a value for this parameter. *SqlFunction* is used by Visual Studio SQL Server projects for deployment of your SQL CLR functions, but for scalar-valued functions in Class Library projects it is optional, so it appears in the Class Library sample code only for our table-valued function (described later).

Listing 3-3, which shows the code for function *fnHelloWorld* from fnTest.cs in the sample project, implements a simple "Hello World" function that returns a value of type *SqlString*.

Listing 3-3 *fnHelloWorld* from fnTest.cs

```
[SqlFunction()]
public static SqlString fnHelloWorld()
{
    return new SqlString("Hello World");
}
```

Notice that *SqlType* objects require explicit instantiation and constructor value passing; you cannot simply declare and assign values to them. The code in Listing 3-3 instantiates a *SqlString* object inline within the *return* statement to avoid variable declaration.

A function that returns a hardcoded value is of little practical use. Typically, functions are passed values and perform calculations on them, and they are often used from within T-SQL statements, in effect as extensions to the functions built into the T-SQL language itself. Listing 3-4, which shows the code for function *fnToCelsius* in fnTest.cs in the sample project, implements a Fahrenheit-to-Celsius conversion function.

Listing 3-4 *fnToCelsius* from fnTest.cs

```
[SqlFunction()]
public static SqlDecimal fnToCelsius(SqlInt16 Fahrenheit)
{
    return new SqlDecimal((((Int16)Fahrenheit) - 32) / 1.8);
}
```

The function accepts a Fahrenheit temperature (as a *SqlInt16*), converts it to Celsius, and returns it (as a *SqlDecimal*). Notice that the code casts the input parameter from a *SqlInt16* to a .NET *Int16*, applies a Fahrenheit-to-Celsius conversion formula, and passes the result to the constructor of a new *SqlDecimal* object.

Deployment of these functions is automatic in the Visual Studio SQL Server project version of our sample code. For the Class Library version, use the T-SQL *CREATE FUNCTION* command in a similar fashion to our use of the *CREATE PROCEDURE* command in the previous section, but include a data type specification for the return value. For example, to deploy the *fnHelloWorld* function, you would use this command:

```
CREATE FUNCTION fnHelloWorld()
RETURNS NVARCHAR(4000) WITH EXECUTE AS CALLER
AS EXTERNAL NAME Chapter03.UserDefinedFunctions.fnHelloWorld
```

Notice the use of data type *NVARCHAR(4000)* to correspond with the *SqlString* type used in the function's implementation. The *WITH EXECUTE AS CALLER* clause specifies that the SQL CLR function should execute under the caller's identity.

Tip You can enter the *CREATE FUNCTION* command yourself, but all such necessary commands for the sample code SQL CLR functions are contained in the CreateObjects.sql script file in the Management Studio project supplied with the sample code.

You can test these functions using the Visual Studio SQL Server project test script or in Management Studio. Use the following query in your test script or a Management Studio query window to test the two functions. (You can also run the TestScalarFunctions.sql script file in the Management Studio sample project.)

```
SELECT
 dbo.fnHelloWorld() AS HelloWorld,
 dbo.fnToCelsius(212) AS CelsiusTemp
```

T-SQL functions can return result sets as well as scalar values. Such functions are called *table-valued functions* (TVFs). Writing SQL CLR TVFs is possible, although you do so differently than you would CLR scalar-valued functions or CLR stored procedures. CLR TVFs must return a type that implements the .NET interface *IEnumerable*, and they must declare a "*FillRow*" method that interprets that type and converts an instance of the type to a table row.

Listing 3-5, which shows the code for functions *fnPortfolioTable* and *FillTickerRow* in fnTest.cs in the sample project, implements a TVF called *fnPortfolioTable*.

Listing 3-5 *fnPortfolioTable* and *FillTickerRow* from fnTest.cs

```
[SqlFunction(
    FillRowMethodName="FillTickerRow",
    TableDefinition="TickerSymbol nvarchar(5), Value decimal")]
public static System.Collections.IEnumerable fnPortfolioTable(SqlString TickersPacked)
{
    string[] TickerSymbols;
    object[] RowArr = new object[2];
    object[] CompoundArray = new object[3];
    char[] parms = new char[1];

    parms[0] = ';';
    TickerSymbols = TickersPacked.Value.Split(parms);

    RowArr[0] = TickerSymbols[0];
    RowArr[1] = 1;
    CompoundArray[0] = RowArr;

    RowArr = new object[2];
    RowArr[0] = TickerSymbols[1];
    RowArr[1] = 2;
    CompoundArray[1] = RowArr;

    RowArr = new object[2];
    RowArr[0] = TickerSymbols[2];
    RowArr[1] = 3;
    CompoundArray[2] = RowArr;

    return CompoundArray;
}

public static void FillTickerRow(object row, ref SqlString TickerSymbol, ref SqlDecimal
Value)
{
    object[] rowarr = (object[])row;
    TickerSymbol = new SqlString((string)rowarr[0]);
    Value = new SqlDecimal(decimal.Parse(rowarr[1].ToString()));
}
```

Rather than implementing its own *IEnumerable*-compatible type, *fnPortfolioTable* uses an array. This is perfectly legal because arrays implement *IEnumerable*. Function *fnPortfolioTable* accepts a semicolon-delimited list of stock ticker symbols and returns a table with each ticker symbol appearing in a separate row as column TickerSymbol and a value for the ticker as column Value. The structure of the returned table is declared in the *TableDefinition* parameter of the *SqlFunction* attribute in SQL Server projects and in the *CREATE FUNCTION* T-SQL command for Class Library projects. The assigned values are hardcoded, and only three rows are returned, regardless of how many ticker symbols are passed in. As with our other samples, this one is more useful as a teaching tool than as a practical application of TVFs.

Arrays are the name of the game here. First the *String.Split* method is used to crack the delimited ticker list into an array of single ticker strings. Then the TVF structures the data so that each element in the return value array (*CompoundArray*) is itself a two-element array storing a single ticker symbol and its value. The function code itself needs only to return *CompoundArray*. Next, the *FillTickerRow* function (named in the *FillRowMethodName* parameter of the *SqlFunction* attribute) takes each two-element array and converts its members to individual scalars that correspond positionally to the columns in the *TableDefinition* argument of the *SqlFunction* attribute.

Because the *FillRowMethodName* parameter of the *SqlFunction* attribute is required by SQL Server, we have decorated the Class Library version of function *fnPortfolioTable* with that attribute, supplying a value for that one parameter. In the SQL Server project version, we also supply a value for the *TableDefinition* parameter to enable autodeployment of the TVF.

As with the other functions, deployment of this function is performed by Visual Studio in the SQL Server project sample code. For the Class Library version, you can deploy the function using the following T-SQL command (also contained in the CreateObjects.sql script file):

```
CREATE FUNCTION fnPortfolioTable(@TickersPacked [NVARCHAR](4000))
RETURNS TABLE (
   TickerSymbol NVARCHAR(5),
   VALUE DECIMAL
)
WITH EXECUTE AS CALLER
AS EXTERNAL NAME Chapter03.UserDefinedFunctions.fnPortfolioTable
```

As with *fnHelloWorld*, we have mapped the *SqlString* data type to an *NVARCHAR(4000)*, this time for one of the input parameters. Because *fnPortfolioTable* is a TVF, its return type is declared as *TABLE*, with inline specifications for the table's definition.

Use the following query in your Visual Studio test script or a Management Studio query window to test the TVF (or run the TestTableValuedFunction.sql script file in the Management Studio sample project):

```
SELECT * FROM fnPortfolioTable('IBM;MSFT;SUN')
```

The following data should be returned:

```
TickerSymbol Value
------------ ------
IBM          1
MSFT         2
SUN          3
```

CLR Triggers

T-SQL triggers are really just stored procedures that are called by SQL Server at specific times and query values in the "inserted" and "deleted" pseudo-tables. SQL CLR triggers are similar to SQL CLR stored procedures, and they can be created for all data manipulation language (DML) actions (updates, inserts, and deletes).

SQL Server 2005 introduces the concept of data definition language (DDL) triggers, which handle actions such as *CREATE TABLE* and *ALTER PROCEDURE*. Like DML triggers, DDL triggers can be implemented in T-SQL or SQL CLR code. We will cover SQL CLR DML and DDL triggers in this section.

SQL CLR DML triggers, like their T-SQL counterparts, have access to the "inserted" and "deleted" pseudo-tables and must be declared as handling one or more specific events for a specific table or, under certain circumstances, a specific view. Also, they can make use of the *SqlTriggerContext* object (through the *SqlContext* object's *TriggerContext* property) to determine which particular event (update, insert, or delete) caused them to fire and which columns were updated.

Once you latch on to these concepts, writing SQL CLR DML triggers is really quite simple. Listing 3-6, which shows the code for function *trgUpdateContact* from trgTest.cs in the sample project, shows the SQL CLR code for DML trigger *trgUpdateContact*, which is designed to function as a *FOR UPDATE* trigger on the Person.Contact table in the AdventureWorks database.

Listing 3-6 *trgUpdateContact* from trgTest.cs

```
//[SqlTrigger(Target="Person.Contact", Event="for UPDATE")]
public static void trgUpdateContact()
{
    SqlTriggerContext TriggerContext = SqlContext.TriggerContext;
    String OldName = String.Empty;
    String NewName = String.Empty;
    String OldDate = String.Empty;
    String NewDate = String.Empty;
    SqlConnection conn = new SqlConnection("context connection=true");
    SqlCommand cmOld = new SqlCommand("SELECT FirstName, ModifiedDate from DELETED", conn);
    SqlCommand cmNew = new SqlCommand("SELECT FirstName, ModifiedDate from INSERTED", conn);
    conn.Open();
    SqlDataReader drOld = cmOld.ExecuteReader();
    if (drOld.Read())
```

```
    {
        OldName = (string)drOld[0];
        OldDate = drOld[1].ToString();
    }
    drOld.Close();
    SqlDataReader drNew = cmNew.ExecuteReader();
    if (drNew.Read())
    {
        NewName = (string)drNew[0];
        NewDate = drNew[1].ToString();
    }
    drNew.Close();
    conn.Close();
    SqlContext.Pipe.Send("Old Value of FirstName:" + OldName);
    SqlContext.Pipe.Send("New Value of FirstName:" + NewName);
    SqlContext.Pipe.Send("Old Value of ModifiedDate:" + OldDate);
    SqlContext.Pipe.Send("New Value of ModifiedDate:" + NewDate);
    for (int i = 0; i <= TriggerContext.ColumnCount - 1; i++)
    {
        SqlContext.Pipe.Send("Column " + i.ToString() + ": " + TriggerContext
.IsUpdatedColumn(i).ToString());
    }

}
```

This CLR DML trigger queries the "deleted" and "inserted" tables and echoes back the "before and after" values for the FirstName and ModifiedDate columns when a row is updated. It does so not by piping back *SqlDataReader* objects but by fetching values from them and echoing back the values as text using the *SqlPipe* object's *Send* method. The trigger code also uses the *TriggerContext.IsUpdatedColumn* method to echo back a list of all columns in the Person.Contact table and whether each was updated.

To deploy the trigger automatically, you would normally configure a *SqlTrigger* attribute and apply it to the .NET function that implements the trigger. Because DML triggers are applied to a target object (a table or a view) and an event (for example, "for update" or "instead of insert"), the *SqlTrigger* attribute has parameters for each of these pieces of information and you must supply values for both. The *SqlTrigger* attribute deploys only a single copy of the trigger, but you can use T-SQL to deploy the same code as a separate trigger for a different event and/or table. Each separate deployment of the same code is assigned a unique trigger name.

Unfortunately, a bug in Visual Studio prevents the *SqlTrigger* attribute from being used for target objects not in the dbo schema. (For example, our table, Person.Contact, is in the Person schema rather than the dbo schema.) This is because the value for the *Target* parameter is surrounded by square brackets when Visual Studio generates its T-SQL code (generating, for example, *[Person.Contact]*, which will cause an error). It is for this reason that the *SqlTrigger* attribute code is commented out in Listing 3-6. A workaround to this problem is available

through the use of pre-deployment and post-deployment scripts, which we will discuss shortly.

Important Although you might be tempted to work around the Visual Studio schema bug by supplying a *Target* value of *Person].[Contact* instead of *Person.Contact*, rest assured that this will not work. You may initiate a trace in SQL Server Profiler to observe the erroneous T-SQL generated by Visual Studio in either scenario.

Although Listing 3-6 does not demonstrate it, you can create a single piece of code that functions as both the update and insert trigger for a given table. You can then use the *TriggerContext* object's *TriggerAction* property to determine exactly what event caused the trigger to fire, and you can execute slightly different code accordingly. Should you wish to deploy such a CLR trigger using the *SqlTrigger* attribute, you would set its *Event* parameter to "FOR UPDATE, INSERT".

The T-SQL command to register a .NET function as a SQL CLR trigger for the update event only is as follows:

```
CREATE TRIGGER trgUpdateContact
ON Person.Contact
FOR UPDATE
AS EXTERNAL NAME Chapter03.Triggers.trgUpdateContact
```

Note All necessary *CREATE TRIGGER* commands for the Class Library project version of the sample code are contained in the CreateObjects.sql script in the Management Studio project supplied with the sample code.

Beyond using such T-SQL code in Management Studio, there is a way to execute this T-SQL command from Visual Studio, and thus work around the *SqlTrigger* non-dbo schema bug. An essentially undocumented feature of Visual Studio SQL Server projects is that they allow you to create two special T-SQL scripts that will run immediately before and immediately after the deployment of your assembly. To use this feature, simply create two scripts, named PreDeployScript.sql and PostDeployScript.sql, in the root folder (not the Test Scripts folder) of your project. Although not case-sensitive, the names must match verbatim.

Tip You can create the PreDeployScript.sql and PostDeployScript.sql scripts outside of Visual Studio and then add them to your project using Visual Studio's Add Existing Item... feature. You can also add them directly by right-clicking the project node or Test Scripts folder node in the Solution Explorer, choosing the Add Test Script option from the shortcut menu, renaming the new scripts, and dragging them out of the Test Scripts folder into the root folder of your project.

To use this feature to work around the *SqlTrigger* non-dbo schema bug, insert the preceding *CREATE TRIGGER* code in your PostDeployScript.sql file and insert the following T-SQL code into your PreDeployScript.sql:

```
IF EXISTS (SELECT * FROM sys.triggers WHERE object_id = OBJECT_ID(N'[Person].[trgUpdateC
ontact]'))
DROP TRIGGER Person.trgUpdateContact
```

Regardless of deployment technique, you can use the following query in your Visual Studio test script or a Management Studio query window to test the trigger (this T-SQL code can be found in the TestTriggers.sql script file in the Management Studio project):

```
UPDATE Person.Contact
SET    FirstName = 'Gustavoo'
WHERE  ContactId = 1
```

When you run the preceding query, you will notice that the trigger is actually run twice. This is because the AdventureWorks Person.Contact table already has a T-SQL update trigger, called *uContact*. Because *uContact* itself performs an update on the ModifiedDate column of Person.Contact, it implicitly invokes a second execution of *trgUpdateContact*. By looking at the output of *trgUpdateContact*, you can confirm that the FirstName column is updated on the first execution (by the test query) and the ModifiedDate column is modified on the second execution (by trigger *uContact*). The two executions' output might appear out of order, but the values of *ModifiedDate* will make the actual sequence clear.

If you place the *TriggerContext* object's *TriggerAction* property in a comparison statement, IntelliSense will show you that there is a wide array of enumerated constants that the property can be equal to, and that a majority of these values correspond to DDL triggers. This demonstrates clearly that SQL CLR code can be used for DDL and DML triggers alike.

In the case of DDL triggers, a wide array of environmental information might be desirable to determine exactly what event caused the trigger to fire, what system process ID (SPID) invoked it, what time the event fired, and other information specific to the event type such as the T-SQL command that caused the event. The *SqlTriggerContext* object's *EventData* property can be queried to fetch this information. The *EventData* property is of type *SqlXml*; therefore it, in turn, has a *CreateReader* method and a *Value* property that you can use to fetch the XML-formatted event data as an *XmlReader* object or a string, respectively.

The code in Listing 3-7, taken from function *trgCreateTable* in trgTest.cs in the sample project, shows the SQL CLR code for the DDL trigger *trgCreateTable* registered to fire for any *CREATE TABLE* command executed on the AdventureWorks database.

Listing 3-7 *trgCreateTable* from trgTest.cs

```
[SqlTrigger(Target = "DATABASE", Event = "FOR CREATE_TABLE")]
public static void trgCreateTable()
{
    SqlTriggerContext TriggerContext = SqlContext.TriggerContext;
    if (!(TriggerContext.EventData == null))
    {
        SqlContext.Pipe.Send("Event Data: " + TriggerContext.EventData.Value.ToString());
    }
}
```

The code interrogates the *Value* property of *SqlContext.TriggerContext.EventData*, casts it to a string, and pipes that string back to the client. Note that the *SqlTrigger* attribute is not commented out in this case because a schema prefix is not used in the *Target* parameter value. Thus, you can use attribute-based deployment in the SQL Server project or the following command for the Class Library version:

```
CREATE TRIGGER trgCreateTable
ON DATABASE
FOR CREATE_TABLE
AS EXTERNAL NAME Chapter03.Triggers.trgCreateTable
```

Use the following T-SQL DDL command in your Visual Studio test script or a Management Studio query window to test the DDL trigger. (You can find this code in the TestTriggers.sql script file in the sample Management Studio project.)

```
CREATE TABLE Test (low INT, high INT)
DROP TABLE Test
```

Your result should appear similar to the following:

```
<EVENT_INSTANCE>
  <EventType>CREATE_TABLE</EventType>
  <PostTime>2006-04-29T16:37:50.690</PostTime>
  <SPID>54</SPID>
  <ServerName>CGH-T42AB</ServerName>
  <LoginName>CGH-T42AB\AndrewB</LoginName>
  <UserName>dbo</UserName>
  <DatabaseName>AdventureWorks</DatabaseName>
  <SchemaName>dbo</SchemaName>
  <ObjectName>Test</ObjectName>
  <ObjectType>TABLE</ObjectType>
  <TSQLCommand>
     <SetOptions ANSI_NULLS="ON" ANSI_NULL_DEFAULT="ON" ANSI_PADDING="ON" QUOTED_IDENTIFIER=
"ON" ENCRYPTED="FALSE" />
   <CommandText>CREATE TABLE Test (low INT, high INT)</CommandText>
  </TSQLCommand>
</EVENT_INSTANCE>
```

Note The actual output would consist of continuous, unformatted text. We added the line breaks and indentation to make the *EventData* XML easier to read.

CLR Aggregates

T-SQL has a number of built-in aggregates, such as *SUM()*, *AVG()*, and *MAX()*, but that set of built-in functions is not always sufficient. Luckily, the SQL CLR features in SQL Server 2005 allow us to implement user-defined aggregates in .NET code and use them from T-SQL. User-defined aggregates can be implemented only in SQL CLR code; they have no T-SQL equivalent. Because aggregates tend to perform computation only, they provide an excellent use case for SQL CLR code. As it turns out, they are also quite easy to build.

At first, aggregates feel and look like functions because they accept and return values. In fact, if you use an aggregate in a non-data-querying T-SQL call (for example, *SELECT SUM(8)*), you are in fact treating the aggregate as if it were a function. The thing to remember is that the argument passed to an aggregate is typically a column, and so each discrete value for that column, for whichever *WHERE*, *HAVING*, *ORDER BY*, and/or *GROUP BY* scope applies, gets passed into the aggregate. It is the aggregate's job to update a variable, which eventually will be the return value, as each discrete value is passed to it.

CLR aggregates require you to apply the *SqlUserDefinedAggregate* attribute to them. The *SqlUserDefinedAggregate* attribute accepts a number of parameters, but all of them are optional except *Format*. In our example, we will use the value *Format.Native* for the *Format* parameter. For more advanced scenarios, you might want to study SQL Server Books Online to acquaint yourself with the other parameters this attribute accepts. Sticking with *Format.Native* for the *Format* parameter is sufficient for many scenarios.

Unlike the *SqlProcedure*, *SqlFunction*, and *SqlTrigger* attributes, the *SqlUserDefinedAggregate* attribute is required by SQL Server for your class to be eligible for use as an aggregate. Visual Studio SQL Server projects do use this attribute for deployment, and the attribute is included in the aggregate template, but it also must be used in generic Class Library project code in order for T-SQL registration of the aggregate to succeed.

Aggregate classes must have four methods: *Init*, *Accumulate*, *Merge*, and *Terminate*. The *Accumulate* method accepts a SQL type, the *Terminate* method returns one, and the *Merge* method accepts an object typed as the aggregate class itself.

The *Accumulate* method handles the processing of a discrete value into the aggregate value, and the *Terminate* method returns the final aggregated value after all discrete values have been processed. The *Init* method provides startup code, typically initializing a class-level private variable that will be used by the *Accumulate* method. The *Merge* method is called in a specific multi-threading scenario, which we will describe later on.

Just to be perfectly clear, your aggregate class will not implement an interface to supply these methods; you must create them to meet what we might term the "conventions" that are expected of SQL CLR aggregate classes (as opposed to a "contract" with which they must comply). When you develop your code in a Visual Studio 2005 SQL Server project, the Aggregate template includes stubs for these four methods as well as the proper application of the *SqlUserDefinedAggregate* attribute.

Creating your own aggregates is fairly straightforward, but thinking through aggregation logic can be a bit confusing at first. Imagine you want to create a special aggregate called *BakersDozen* that increments its accumulated value by 1 for every 12 units accumulated (much as a baker, in simpler times, would throw in a free 13th donut when you ordered 12). By using what you now know about CLR aggregates and combining that with integer division, you can implement a *BakersDozen* aggregate quite easily. Listing 3-8, the code from struct *BakersDozen* in aggTest.cs in the sample project, contains the entire implementation of the aggregate *BakersDozen*.

Listing 3-8 struct *BakersDozen* from aggTest.cs

```
[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate(Format.Native)]
public struct BakersDozen
{
    private SqlInt32 DonutCount;

    public void Init()
    {
        DonutCount = 0;
    }

    public void Accumulate(SqlInt32 Value)
    {
        DonutCount += Value + ((Int32)Value) / 12;
    }

    public void Merge(BakersDozen Group)
    {
        DonutCount += Group.DonutCount;
    }

    public SqlInt32 Terminate()
    {
        return DonutCount;
    }

}
```

The code here is fairly straightforward. The private variable *DonutCount* is used to track the *BakersDozen*-adjusted sum of items ordered, adding the actual items- ordered value and incrementing the running total by the integer quotient of the ordered value divided by 12. By this

logic, bonus items are added only when an individual value equals or exceeds a multiple of 12. Twelve includes a full dozen, and so would 13. Twenty-four includes two dozen, and so would 27. Two individual orders of 6 items each would not generate any bonus items because a minimum of 12 items must be ordered in a line item to qualify for a bonus.

To deploy the aggregate, use attribute-based deployment in the SQL Server project or the following command for the Class Library version:

```
CREATE AGGREGATE BakersDozen
        (@input int)
RETURNS int
EXTERNAL NAME Chapter03.BakersDozen
```

Notice that no method name is specified because the aggregate is implemented by an entire class rather than an individual function. Notice also that the return value data type must be declared as the data type of the values this aggregate function will process. The *@input* parameter acts as a placeholder, and its name is inconsequential. Note that aggregates can be built on SQL CLR types (covered in the next section) as well as SQL scalar types.

Note The preceding *CREATE AGGREGATE* command for the Class Library project version of the sample code is contained in the CreateObjects.sql script in the Management Studio project supplied with the sample code.

To see the aggregate work, first run the CreateTblAggregateTest.sql script file in the Management Studio sample project to create a table called AggregateTest with columns OrderItemId, OrderId, and ItemsOrdered and several rows of data, as shown here:

```
CREATE TABLE tblAggregateTest(
    [OrderItemId] [int] IDENTITY(1,1) NOT NULL,
    [OrderId] [int] NULL,
    [ItemsOrdered] [int] NOT NULL
)
GO

INSERT INTO tblAggregateTest VALUES (1,2)
INSERT INTO tblAggregateTest VALUES (1,4)
INSERT INTO tblAggregateTest VALUES (2,1)
INSERT INTO tblAggregateTest VALUES (2,12)
INSERT INTO tblAggregateTest VALUES (3,3)
INSERT INTO tblAggregateTest VALUES (3,2)
```

With such a table built, use the following T-SQL DDL command in your Visual Studio test script or a Management Studio query window to test the aggregate function:

```
SELECT
  OrderId,
  SUM(ItemsOrdered) AS SUM,
  dbo.BakersDozen(ItemsOrdered) AS BakersDozen
FROM tblAggregateTest
GROUP BY OrderId
```

For each distinct value in the OrderId column, this query effectively uses our CLR code under the following algorithm:

- Call *Init()*.
- Call *Accumulate* once for each row with the same *OrderId* value, passing it that row's value of the *ItemsOrdered* column.
- Call *Terminate* upon a change in the *OrderId* value to retrieve the aggregated value that the query will pipe to the client.

The results should be as follows:

```
OrderId     SUM         BakersDozen
----------- ----------- -----------
1           6           6
2           13          14
3           5           5
```

By including the built-in T-SQL aggregate *SUM* in our query, we can see how many bonus items were added. In this case, for OrderId 2, a single bonus item was added, due to one row in the table with the following values:

```
OrderItemId OrderId     ItemsOrdered
----------- ----------- ------------
4           2           12
```

All the other rows contain *ItemsOrdered* values of less than 12, so no bonus items were added for them.

Because SQL Server sometimes segments the work required to satisfy a query over multiple threads, the query processor might need to execute your aggregate function multiple times for a single query and then merge the results together. For your aggregate to work properly in this scenario, you must implement a *Merge* method.

The *Merge* method takes the result of one thread's aggregation and merges it into the current thread's aggregation. The calculation required to do this could be complicated for some aggregates; in our case, we simply added the *DonutCount* value from the secondary thread's aggregate (accessible via the *Group* input parameter) to our own. There is no need to add bonus

items because they would have been added in the individual *Accumulate* calls on the secondary thread. Simple addition is all that's required. An aggregate that calculated some type of average, or tracked the largest value in the data series supplied, for example, would require more complex merge code.

Don't forget that aggregates can be passed scalar values and can be used from T-SQL without referencing a table. Your aggregate must accommodate this scenario, even if it seems impractical. In the case of *BakersDozen*, single scalar values are easily handled. To see for yourself, try executing the following table-less T-SQL query:

```
SELECT dbo.BakersDozen(13)
```

You will see that it returns the value 14.

Note The TestAggregate.sql script file in the Management Studio project contains both aggregate-testing queries.

Aggregates are an excellent use of SQL CLR programming. Because they are passed data values to be processed, they typically perform only computational tasks and no data access of their own. They consist of compiled CLR code, so they perform well, and unlike stored procedures, triggers, and functions, they cannot be implemented at all in T-SQL. That said, you must still make your aggregate code, especially in the *Accumulate* method, as "lean and mean" as possible. Injecting your own code into the query processor's stream of work is an honor, a privilege, and a significant responsibility. Take that responsibility seriously, and make sure that your code is as low-impact as possible.

CLR Types

The last SQL CLR feature for us to explore is user-defined types (UDTs). This feature is perhaps the most interesting, yet also the most controversial. It's interesting because, technically, it allows for storage of objects in the database. It's controversial because it's prone to abuse. CLR types were not implemented to allow developers to create object-oriented databases; they were created to allow multi-value or multi-behavior data types to be stored, retrieved, and easily manipulated.

CLR types have an 8 KB size limit. They also have certain indexing limitations, and their entire value must be updated when any of their individual property/field values is updated.

Note More information on CLR user-defined types is available in the MSDN article "Using CLR Integration in SQL Server 2005" by Rathakrishnan, Kleinerman, et al. You can find this article online at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/sqlclrguidance.asp*.

CLR type methods must be static. You cannot, therefore, call methods from T-SQL as instance methods; instead, you must use a special *TypeName::MethodName()* syntax. You can implement properties as you would in any conventional class and read from them or write to them from T-SQL using a standard variable.property/column.property dot-separated syntax.

Listing 3-9, the code from struct *typPoint* in typTest.cs in the sample project, shows the implementation of *typPoint*, a CLR type that can be used to store Cartesian coordinates in the database.

Listing 3-9 struct *typPoint* from typTest.cs

```
[Serializable]
[SqlUserDefinedType(Format.Native)]
public struct typPoint : INullable
{
    private bool m_Null;
    private double m_x;
    private double m_y;

    public override string ToString()
    {
        if (this.IsNull)
            return "NULL";
        else
            return this.m_x + ":" + this.m_y;
    }

    public bool IsNull
    {
        get
        {
            return m_Null;
        }
    }

    public static typPoint Null
    {
        get
        {
            typPoint pt = new typPoint();
            pt.m_Null = true;
            return pt;
        }
    }

    public static typPoint Parse(SqlString s)
    {
        if (s.IsNull)
            return Null;
        else
```

```
        {
            typPoint pt = new typPoint();
            char[] parms = new char[1];
            parms[0] = ':';
            string str = (string)s;
            string[] xy = str.Split(parms);
            pt.X = double.Parse(xy[0]);
            pt.Y = double.Parse(xy[1]);
            return pt;
        }
    }

    public static double Sum(typPoint p )
    {
        return p.X + p.Y;
    }

    public double X
    {
        get { return m_x; }
        set { m_x = value; }
    }

    public double Y
    {
        get { return m_y; }
        set { m_y = value; }
    }

}
```

Through the class's *X* and *Y* properties, you can process coordinates in a single database column or variable. You can assign coordinate values to an instance of the type as a colon-delimited string (for example, *3:4*, by using the *Parse* method [implicitly]); you can read them back in the same format by using the *ToString* method. Once a value has been assigned, you can individually read or modify its X or Y portion by using the separate *X* and *Y* properties. The class implements the *INullable* interface and its *IsNull* property. The *Sum* method demonstrates how to expose a static member and allow it to access instance properties by accepting an instance of the CLR type of which it is a member.

Notice that the class is a struct and that the *Serializable* and *SqlUserDefinedType* attributes have been applied to it. As with the *SqlUserDefinedAggregate* attribute, *SqlUserDefinedType* is required by SQL Server and appears in the Class Library sample code as well as the SQL Server project version. As with the *SqlUserDefinedAggregate*, we simply assign a value of *Format.Native* to the *Format* parameter and leave the other parameters unused.

More Info You might want to study SQL Server Books Online for information on using other parameters for this attribute.

Listing 3-10, the code from struct *typBakersDozen* in typTest.cs in the sample project, re-implements the BakersDozen logic we used in our aggregate example, this time in a UDT.

Listing 3-10 struct *typBakersDozen* from typTest.cs

```
[Serializable]
[SqlUserDefinedType(Format.Native)]
public struct typBakersDozen : INullable
{
    private bool m_Null;
    private double m_RealQty;

    public override string ToString()
    {
        return (m_RealQty + (long)m_RealQty / 12).ToString();
    }

    public bool IsNull
    {
        get
        {
            return m_Null;
        }
    }

    public static typBakersDozen Null
    {
        get
        {
            typBakersDozen h = new typBakersDozen();
            h.m_Null = true;
            return h;
        }
    }

    public static typBakersDozen Parse(SqlString s)
    {
        if (s.IsNull)
            return Null;
        else
        {
            typBakersDozen u = new typBakersDozen();
            u.RealQty = double.Parse((string)s);
            return u;
        }
    }

    public static typBakersDozen ParseDouble(SqlDouble d)
    {
        if (d.IsNull)
            return Null;
        else
```

```
        {
            typBakersDozen u = new typBakersDozen();
            u.RealQty = (double)d;
            return u;
        }
    }

    public double RealQty
    {
        get { return m_RealQty; }
        set { m_RealQty = value; }
    }

    public double AdjustedQty
    {
        get
        {
            return (m_RealQty + (long)m_RealQty / 12);
        }
        set
        {
            if (value % 12 == 0)
                m_RealQty = value;
            else
                m_RealQty = value - (long)value / 13;
        }
    }

}
```

The *RealQty* and *AdjustedQty* properties allow the ordered quantity to be assigned a value and the adjusted quantity to be automatically calculated, or vice versa. The real quantity is the default "input" value, the adjusted quantity is the default "output" value of the type, and the *Parse* and *ToString* methods work accordingly. If the *AdjustedQty* property is assigned a value that is an even multiple of 12 (which would be invalid), that value is assigned to the *RealQty* property, forcing the *AdjustedQty* to be set to its passed value plus its integer quotient when divided by 12.

To deploy the UDTs, use attribute-based deployment for the SQL Server project. The script file CreateObjects.sql in the Management Studio project supplied with the sample code contains the T-SQL code necessary to deploy the Class Library versions of the UDTs. Here's the command that deploys typPoint:

```
CREATE TYPE typPoint
EXTERNAL NAME Chapter03.typPoint
```

The script file TestTypPoint.sql in the Management Studio project contains T-SQL code that tests *typPoint*. Run it and examine the results for an intimate understanding of how to work

with the type. The script file CreateTblPoint.sql creates a table with a column that is typed based on *typPoint*. Run it, and then run the script file TestTblPoint.sql to see how to manipulate tables that use SQL CLR UDTs.

The script file TestTypBakersDozen.sql contains T-SQL code that tests *typBakersDozen*. The *ParseDouble* method demonstrates how to implement a non-*SqlString* parse method. We named it *ParseDouble* because the *Parse* method itself cannot be overloaded. You must call *ParseDouble* explicitly as follows:

```
DECLARE @t AS dbo.typBakersDozen
SET @t = typBakersDozen::ParseDouble(12)
```

This is equivalent to using the default *Parse* method (implicitly) and assigning the string *12* as follows:

```
DECLARE @t AS dbo.typBakersDozen
SET @t = '12'
```

Notice that *typBakersDozen* essentially stores a value for the real quantity, and its properties are really just functions that accept or express that value in its native form or as an adjusted quantity. There is no backing variable for the *AdjustedQty* property; the *get* block of the *AdjustedQty* property merely applies a formula to the backing variable for *RealQty* and returns the result.

So *typBakersDozen* is not really an object with distinct properties, as *typPoint* is (with its admittedly simple ones). Because it merely implements a *SqlDouble* and adds some specialized functionality to it, *typBakersDozen* is a more appropriate implementation of CLR UDTs than is *typPoint*. In this vein, other good candidates for CLR UDTs include date-related types (for example, a type that stores a single underlying value for an annual quarter and accepts/presents its value as a calendar quarter or a fiscal quarter) and currency types (especially those with fixed rates of exchange between their native and converted values).

In general, you can think of CLR UDTs as "super scalars"—classes that wrap a scalar value and provide services and conversion functions for manipulating that scalar value and converting it among different interpretive formats or numbering systems. Do not think of SQL CLR UDTs as object-relational entities. This might seem counterintuitive, but consider the use of (de)serialization and the XML data type as more appropriate vehicles for storing objects in the database.

We have now investigated all five SQL CLR entities. Before we finish up, we need to discuss assembly security and ongoing maintenance of SQL CLR objects in your databases.

Security

Depending on the deployment method, you have numerous ways to specify what security level to grant a CLR assembly. All of them demand that you specify one of three permission sets:

- *Safe* Assembly can perform local data access and computational tasks only.
- *External_Access* Assembly can perform local data access and computational tasks and also access the network, the file system, the registry, and environment variables. Although *External_Access* is less restrictive than *Safe*, it still safeguards server stability.
- *Unsafe* Assembly has unrestricted permissions and can even call unmanaged code. This setting can significantly compromise SQL Server security; only members of the sysadmin role can create (load) unsafe assemblies.

When you deploy an assembly from Visual Studio, its security level is set to Safe by default. To change it, you can select the project node in the Solution Explorer window and set the Permission Level property in the Properties window by selecting Safe, External, or Unsafe from the combo box provided (Figure 3-9).

Figure 3-9 The Permission Level property and its options in the Visual Studio 2005 Properties window

Alternatively, you can right-click the project node in the Solution Explorer window and select Properties from the shortcut menu. You can also double-click the Properties node (or the My Project node in Visual Basic projects). Either action opens up the project properties designer. Select the designer's Database tab and then select a permission set from the Permission Level combo box (Figure 3-10). (The same three options are available here as in the Properties window.)

To specify an assembly's permission set using T-SQL, simply specify *SAFE*, *EXTERNAL_ACCESS*, or *UNSAFE* within the "*WITH PERMISSION_SET*" clause of the *CREATE ASSEMBLY* statement covered earlier in this chapter. Recall that our example used the default *SAFE* setting in this clause.

Finally, in the Management Studio New Assembly dialog box (shown earlier in Figure 3-7), you can select Safe, External Access, or Unsafe from the Permission Set combo box.

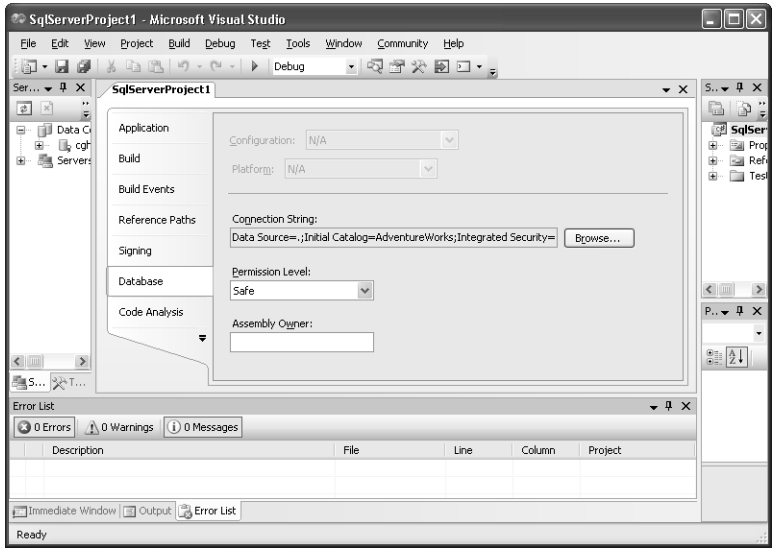

Figure 3-10 The Database tab of the Visual Studio project properties designer

Examining and Managing CLR Types in a Database

Once deployed, your SQL CLR stored procedures, functions, triggers, aggregates, and user-defined types and their dependencies might become difficult to keep track of in your head. Luckily, you can easily perform discovery on deployed CLR entities using the Management Studio UI. All CLR objects in a database can be found in Management Studio's Object Explorer window. To find them within the Object Explorer window's tree view, first navigate to the *servername*\Databases*databasename* node (where *servername* and *databasename* are the names of your server and database, respectively). Refer to Table 3-1 for the subnodes of this node that contain each CLR entity.

Table 3-1 Finding CLR Objects in Object Explorer

To view...	Look in...
Parent node for SQL CLR stored procedures, DDL triggers, functions, aggregates, and UDTs	Programmability (see Figure 3-11)
Assemblies	Programmability\Assemblies (see Figure 3-12)
Stored procedures	Programmability\Stored Procedures (see Figure 3-13)
Functions	Programmability\Functions\Scalar-Valued Functions and Programmability\Functions\Table-Valued Functions (see Figure 3-14)
Aggregates	Programmability\Functions\Aggregate Functions (see Figure 3-14)
DML triggers	Tables*tablename*\Triggers, where *tablename* is the name of the database table, including schema name, on which the trigger is defined (see Figure 3-15)

Table 3-1 Finding CLR Objects in Object Explorer

To view...	Look in...
DDL triggers	Programmability\Database Triggers (see Figure 3-16) (also *servername*\Server Objects\Triggers, where *servername* is the name of your server)
User-defined types	Programmability\Types\User-Defined Types (see Figure 3-17)

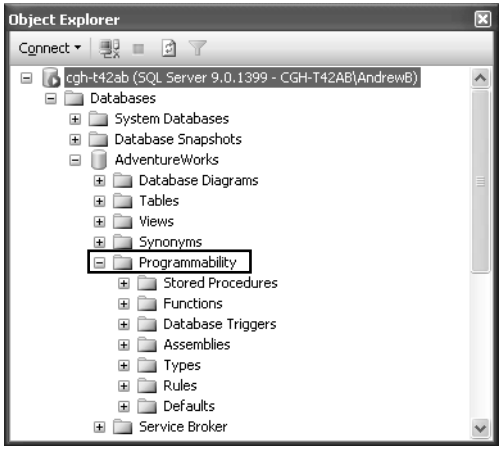

Figure 3-11 The SQL Server Management Studio Object Explorer window, with Programmability node highlighted

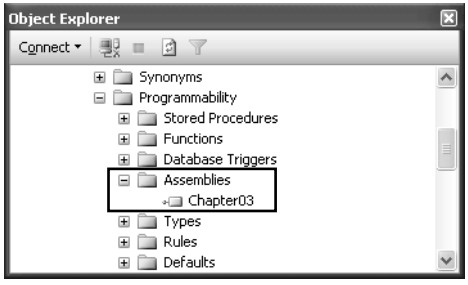

Figure 3-12 The Object Explorer window, with Assemblies node highlighted

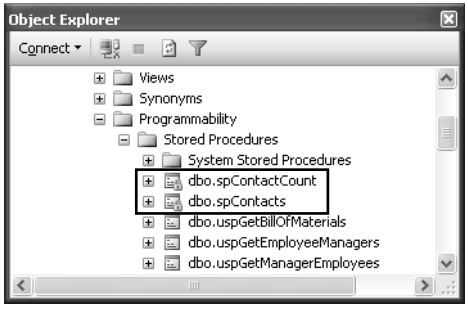

Figure 3-13 The Object Explorer window, with CLR stored procedures highlighted

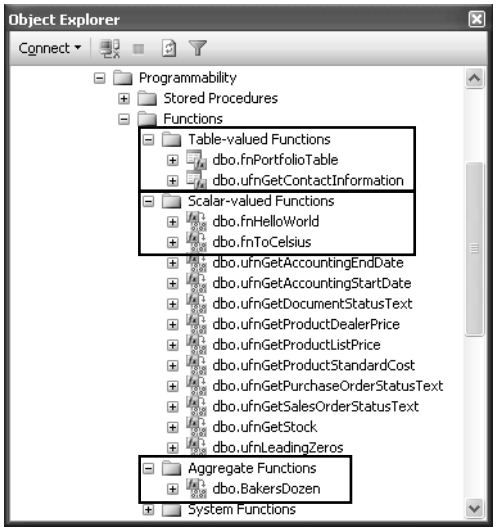

Figure 3-14 The Object Explorer window, with CLR table-valued, scalar-valued, and aggregate functions highlighted

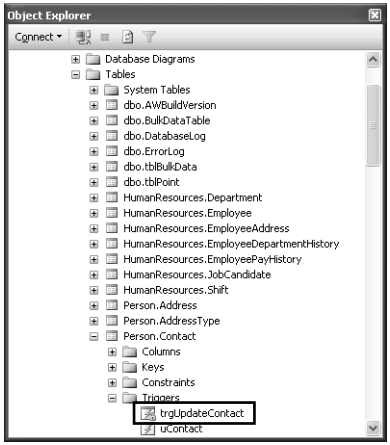

Figure 3-15 The Object Explorer window, with CLR DML trigger highlighted

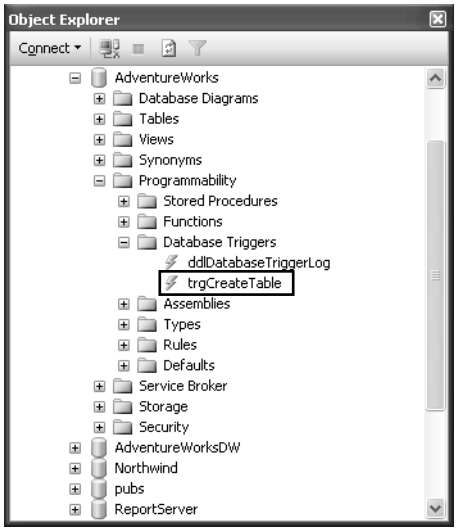

Figure 3-16 The Object Explorer window, with CLR DDL trigger highlighted

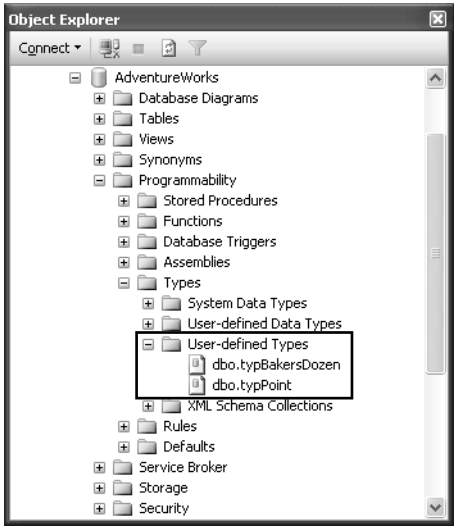

Figure 3-17 The Object Explorer window, with CLR UDTs highlighted

Bear in mind that you might need to use the Refresh shortcut menu option on the nodes listed in the table to see your CLR objects. If you've deployed or deleted any SQL CLR objects (as discussed shortly) since opening the Object Explorer's connection to your database, the tree view will be out of date and will have to be refreshed. Notice that the tree view icons for CLR stored procedures and CLR DML triggers differ slightly from their T-SQL counterparts; they have a small yellow padlock on the lower-right corner.

Once you've located a CLR entity in the Object Explorer window, you can right-click its tree view node and generate CREATE, DROP, and in some cases ALTER scripts for it by selecting

the Script *object type* As option from the shortcut menu (where *object type* is the SQL CLR object type selected). The script text can be inserted into a new query window, a file, or the clipboard.

For stored procedures, you can also generate *EXECUTE* scripts or, by selecting Execute Stored Procedure from the shortcut menu, execute it interactively and generate the corresponding script via Management Studio's Execute Procedure dialog box. This dialog box explicitly prompts you for all input parameters defined for the stored procedure.

In addition to generating scripts for your CLR entities, you can view their dependencies (either objects that are dependent on them or objects on which they depend). Just right-click the object and choose the View Dependencies option from the shortcut menu.

To remove your CLR objects, either in preparation for loading a new version of your assembly or to delete the objects permanently, you have several options. For Visual Studio SQL Server projects, redeploying your assembly causes Visual Studio to drop it and any SQL CLR objects within it that were previously deployed by Visual Studio. This means that new versions can be deployed from Visual Studio without any preparatory steps.

For Class Library projects, you must issue T-SQL *DROP* commands for each of your SQL CLR objects and then for the assembly itself. You must drop any dependent objects before you drop the SQL CLR entity. For example, you must drop tblPoint before dropping typPoint. You can write these *DROP* scripts by hand or generate them by using the Script *object type* As/DROP To shortcut menu options in the Management Studio Object Explorer window.

You can also use the Delete shortcut menu option on any SQL CLR object in the Management Studio Object Explorer window to drop an object. This option brings up the Delete Object dialog box (Figure 3-18).

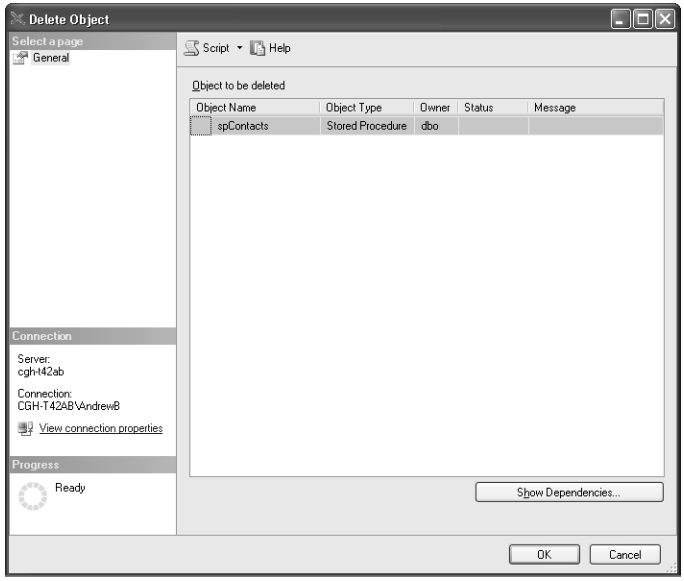

Figure 3-18 The Management Studio Delete Object dialog box

The script file Cleanup.sql in the Management Studio project provided with the sample code contains all the necessary *DROP* commands, in the proper order, for removing all traces of our Visual Studio SQL Server project or Class Library project from the AdventureWorks database. For the SQL Server project, run this script only if you want to permanently remove these objects. For the Class Library project, run it before you deploy an updated version of your assembly or if you want to permanently remove these objects.

SQL CLR objects, with the exception of DDL triggers, can also be viewed in Visual Studio 2005's Server Explorer window, as shown in Figure 3-19.

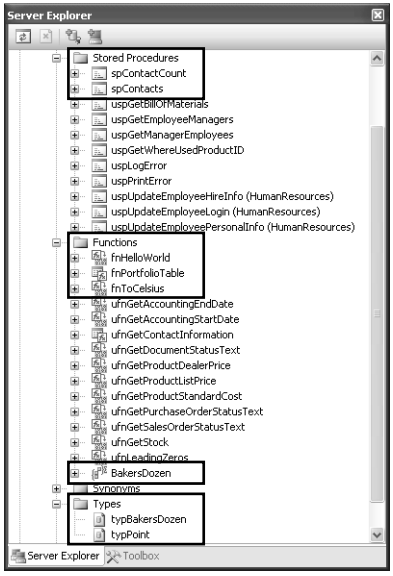

Figure 3-19 The Visual Studio Server Explorer window, with CLR stored procedures, functions, aggregates, and UDTs highlighted

You'll find most of the objects under their appropriate parent nodes within the data connection parent node: CLR stored procedures appear under the Stored Procedures node; scalar and table-valued functions, as well as aggregates, appear under the Functions node; the Types and Assemblies nodes contain their namesake objects; and DML triggers appear under the node of the table to which they belong.

You may also drill down on a particular assembly node and view a list of all its SQL CLR objects, as well as the source code files that make it up (see Figure 3-20).

For assemblies created from Visual Studio 2005 SQL Server projects, you may double-click on any SQL CLR object in the Server Explorer window to view its source code. (You may also do this by selecting the Open option from the SQL CLR object node's shortcut menu or the Data/Open option from Visual Studio's main menu while the node is selected.) If the assembly's project is open when you open the object's source, the code will be editable; if the project is not open, the source will be read-only.

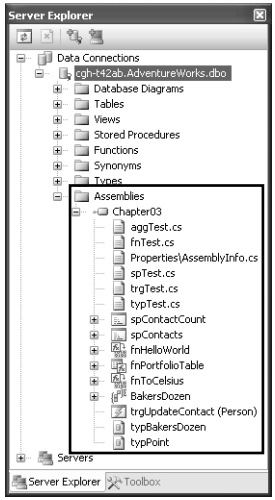

Figure 3-20 The Server Explorer window, with the Assemblies node and its child nodes highlighted

Caution Because *trgUpdateContact*, our SQL CLR DML trigger, was deployed via T-SQL in the PostDeployScript.sql script and not via the *SqlTrigger* attribute, its source cannot be browsed through the Server Explorer window connection's Tables\Contact (Person)\trgUpdateContact node or its Assemblies\Chapter03\trgUpdateContact (Person) node. You can, however, view its source through the Assemblies\Chapter03\trgTest.cs node.

Best Practices for SQL CLR Usage

Before we close this chapter, we'd like to summarize certain best practices for the appropriate use of SQL CLR programming.

The CLR integration in SQL Server 2005 is a powerful technology. In some cases, it allows you to do things you can't do practically in T-SQL (such as apply complex business logic in stored procedures or triggers), and in other cases it allows you to do things you can't do at all in T-SQL (such as create your own aggregate functions).

The fact remains, however, that set-based data selection and modification is much better handled by the declarative constructs in T-SQL than in the procedural constructs of .NET and the ADO.NET object model. SQL CLR functionality should be reserved for specific situations when the power of .NET as a calculation engine is required.

In general, functions and aggregates are great uses of SQL CLR integration. UDTs, if used as "super scalars" rather than objects per se, make good use of SQL CLR integration as well.

For stored procedures and triggers, we recommend that you start with the assumption that these should be written in T-SQL and write them using SQL CLR code only if a case can be made that they cannot be reasonably written otherwise. And before you make such a case,

consider that SQL CLR functions, aggregates, and UDTs can be used from within T-SQL stored procedures and triggers.

Summary

In this chapter, you've been exposed to the "mechanics" of developing the five basic SQL CLR entities and using them from T-SQL. You've seen how to take advantage of SQL Server 2005/Visual Studio 2005 integration as well as how to develop SQL CLR code in conventional Class Library assemblies and deploy them using T-SQL and SQL Server Management Studio. You've also been exposed to most of the SQL CLR .NET code attributes and their use in SQL Server projects and standard Class Library projects. You've gotten a sense of how to use Management Studio and the Visual Studio 2005 Server Explorer window as management tools for your SQL CLR objects, and we've discussed scenarios in which using SQL CLR integration is a good choice as well as scenarios in which T-SQL is the better choice.

This first part of the book essentially covers building databases, so in this chapter we intentionally kept our focus on using SQL CLR objects from T-SQL and Management Studio. In the previous chapter, we highlighted a number of enhancements to T-SQL that you can use in your database development process. The second part of the book covers developing applications that use your databases. In Chapter 8, we'll look at how to consume your SQL CLR objects in .NET applications by using ADO.NET, including within strongly typed *DataSet* objects, and optionally using Windows Forms and ASP.NET data binding. And in Chapter 9, we'll show you how to perform end-to-end debugging of client-side and SQL CLR code, as well as T-SQL code, in Visual Studio 2005. The sum total of the material from Chapters 3, 8, and 9 provides a rich resource for diving into SQL CLR development.

Chapter 14

Developing Desktop Applications with SQL Server Express Edition

–Rob Walters

In this chapter:

As time passes, database engines that perform basic relational database actions are becoming more of a commodity. With the advent of MySQL, Postgres, and a number of other open source database vendors, more options exist today for developers then ever before. The impact of these disruptive technologies can be seen in the free database offerings not only from Microsoft but also from Oracle (Oracle Express) and IBM (DB2 Express). Each of these relational database vendors wants you, the developer, to use their product so that one day your application might grow enough that you actually need to buy a license for an "upper-level" edition. Although these marketing tactics might be of little interest to us as developers, in reality we benefit tremendously from this competitive environment. Could you ever have imagined 10 years ago that companies like Microsoft would essentially give away a powerful developer tool like Visual Studio Express? Bundled with a rock-solid database like SQL Server, it solves a lot of development needs for a reasonable price: free.

This chapter covers SQL Server Express Edition. It goes into depth about installation, configuration, and management, and it explores Express Edition's features. It also compares and contrasts Express Edition with the Microsoft SQL Server Desktop Engine (MSDE). Finally, you will learn how Express Edition fits into the overall SQL Server product lineup.

What Is SQL Server Express Edition?

Express Edition is one of four SQL Server editions. The others are Workgroup, Standard, and Enterprise. Figure 14-1 shows the progression of editions from Express to Enterprise.

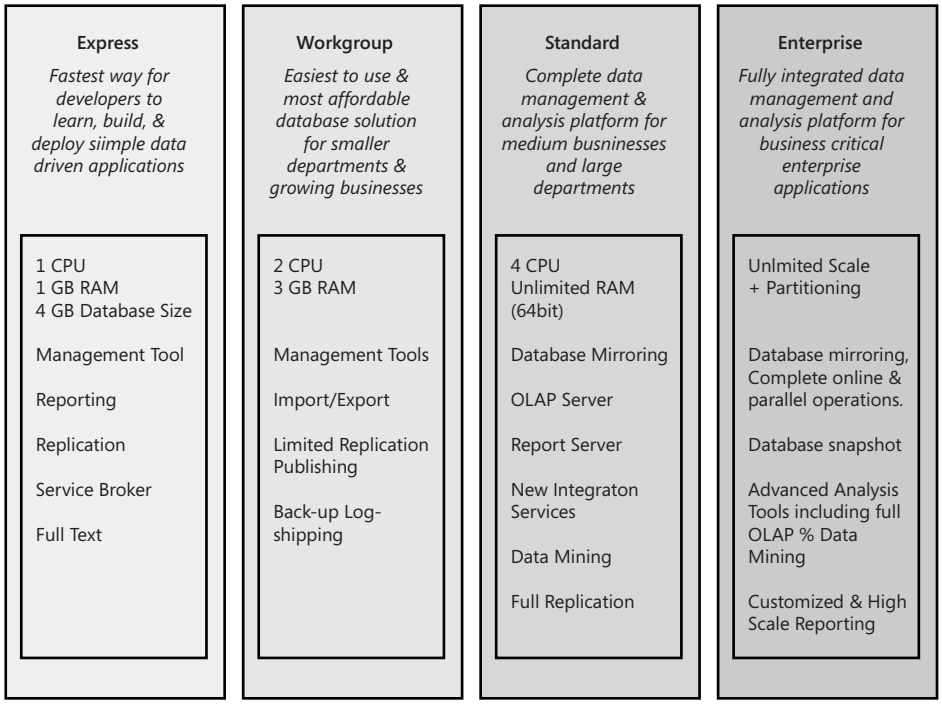

Figure 14-1 The four editions of SQL Server 2005

Some people might be wondering, "Where is Developer Edition?" The Developer and Evaluation editions are actually the same as Enterprise Edition, but with some licensing restrictions. Each SQL Server edition builds on the previous one's functionality. So Standard Edition, for example, has the same functionality as Express Edition and Workgroup Edition (such as management tools, the ability to import and export, and so on).

Note that the Express database engine itself is the same as those of the other editions. It differs only in the restrictions imposed on its use: 1 CPU, 1 GB of RAM, and 4 GB database file size. There are some interesting things to note about these restrictions. If you have a multi-core CPU, Express Edition will leverage all the cores. (After all, to the SQL engine it's still just one CPU.) If you exceed the database file size limit, you will get an error on the command that caused the size to exceed the limit and any transactions in that session will be rolled back. This is the same behavior you would see if you set a fixed size for a database, the database almost reached that size, and then you tried to insert one more row, causing it to exceed the limit. One of the restrictions that existed previously in MSDE was the workload governor. This "feature" intentionally decreased performance as the workload increased. You can forget about this in Express Edition—there is no such thing, so you can throw as much work at Express Edition as your box can handle (and as the CPU, RAM, and database size restrictions allow). Express Edition truly is the same database engine as all the other editions.

Licensing

Some in the developer community might be thinking about deploying Express Edition with their own applications. This is possible, legal, and royalty-free as long as you do not change any parts of the shipped bits of Express Edition. You must also register on Microsoft's Web site if you want to redistribute Express Edition. The URL is *http://go.microsoft.com/fwlink/?LinkId=64062*.

Later in this chapter, we will cover how to silently install Express Edition—this is the most likely way you will deploy Express Edition with your application. In addition to the database engine, Express Edition contains a few extra components such as SQL Server Management Studio Express and SQLCMD. All these tools are also redistributable as long as you don't change them. You cannot redistribute the tools alone, however—you must redistribute Express Edition in addition to the tools. You can, however, redistribute just the SQL Server Express Edition engine.

Feature Review

With the exception of the memory, database size, and CPU restrictions, SQL Server Express Edition contains all the basic relational database features seen in the other editions. For example, you can create common language runtime (CLR) stored procedures, store data as XML, and encrypt your data using symmetric keys. However, Microsoft did not give away the farm in this edition—if you want to use more "enterprise-like" features such as database mirroring or partitioning, you have to pay for the upper-level edition. The business intelligence tools (such as SQL Server Analysis Services) and extract, transform, and load (ETL) tools (such as SQL Server Integration Services, known as Data Transformation Services in SQL 2000) are also not available in Express Edition.

The lack of Integration Services in Express Edition causes some minor pain. For one thing, there is no Import And Export Wizard for moving a database nor any relatively easy alternative. The easiest workaround is to perform a detach, file copy, and attach. Alternatively, you can back up and then restore a database on the destination server. If you need just specific tables within a database, you might consider using the Bulk Copy Program (BCP), which ships with Express Edition as well as all the other editions of SQL Server.

Now let us focus on the database engine–specific features that are significant to SQL Server Express Edition users. Table 14-1 shows key database administration features for the SQL Server product as described by the folks in SQL Server marketing. This table includes a column that describes the availability of the feature in Express Edition.

Table 14-1 SQL Server Database Engine Features

Feature	Available in Express Edition?
Database Mirroring	No
Online Restore	No
Online Indexing Operations	Yes
Fast Recovery	Yes

Table 14-1 SQL Server Database Engine Features

Feature	Available in Express Edition?
Security Enhancements	Yes
New SQL Server Management Studio	Available as a scaled-down version called SQL Server Management Studio Express.
Dedicated Administrator Connection	Yes, but it is not enabled by default. To use it in Express Edition, you must start the service using the trace flag 7806. You do not need to specify this trace flag if you are using any other edition of SQL Server.
Snapshot Isolation	No
Data Partitioning	No
Replication Enhancements	Available but scaled down. Later in this chapter, see the "Replication in Express Edition" section for additional details.
SQL Server Agent	No. The workaround is to create Transact-SQL (T-SQL) scripts that are executed via SQLCMD through the Windows Task Scheduler.
SQL Mail and Database Mail	No
Mirrored Media Sets	No
Address Windowing Extensions (AWE)	No
Hot-add memory	No
Failover clustering	No
VIA Protocol support	No

Table 14-2 shows the key development features of SQL Server as well as their availability in Express Edition.

Table 14-2 SQL Server Database Development Features

Feature	Available in Express Edition?
.NET Framework Hosting	Yes. This feature is off by default in all SQL Server editions, so you must enable the CLR by selecting the Enable CLR check box in the SQL Server Surface Area Configuration tool.
XML Technologies	Yes
ADO.NET 2.0	Yes
T-SQL Enhancements	Yes
SQL Service Broker	Available but scaled down. Later in this chapter, see the "Service Broker in Express Edition" section for additional details.
Notification Services	No
Web Services	No. You cannot create HTTP endpoints in Express Edition, but it is still possible to create a Web service and use Express Edition—you just can't make Express Edition an HTTP listener.
Reporting Services	Yes. Later in this chapter, see the "Reporting Services in Express Edition" section for additional details.
Full-Text Search Enhancements	Yes. Later in this chapter, see the "Full Text in Express Edition" section for additional details.

Replication in Express Edition

Express Edition can serve as a Subscriber for all types of replication, providing a convenient way to distribute data to client applications that use SQL Server Express Edition. Note that this is a change in behavior from MSDE. If you are using MSDE for replication, MSDE can be either a Snapshot Publisher or a Merge Publisher. If you want to continue to use this functionality, you must upgrade to at least Workgroup Edition.

SQL Server Express Edition does not include the SQL Server Agent, which is typically used to run replication agents. If you use push subscription, replication agents run at the Distributor, which will be an instance of SQL Server 2005, so options are available for synchronizing. But if you use a pull subscription, in which agents run at the Subscriber, you must synchronize the subscription by using the Windows Synchronization Manager tool or do it programmatically using Replication Management Objects (RMO).

Windows Synchronization Manager is available with Microsoft Windows 2000 and later. If SQL Server is running on the same computer as Synchronization Manager, you can do the following:

- Synchronize a subscription.
- Reinitialize a subscription.
- Change the update mode of an updatable transactional subscription.

Service Broker in Express Edition

Service Broker is a new technology in SQL Server 2005 that helps developers create distributed applications that provide support for queuing and reliable messaging. Developers can compose applications from independent, self-contained components called services. Applications can then use messages to interact with these services and access their functionality. Service Broker uses TCP/IP to exchange messages between SQL Server instances, and it includes features to help prevent unauthorized access from the network and to encrypt messages sent over the network.

The SQL Server Express Edition database engine supports Service Broker only as a client. Express Edition can participate in a Service Broker messaging application only when a paid edition of SQL Server (Workgroup, Standard, or Enterprise) is part of the message chain. Express Edition can also send Service Broker messages to itself.

SQL Server 2005 Express Edition with Advanced Services

At the time SQL Server 2005 Express Edition shipped, Microsoft promised that it would soon release a version of Reporting Services as well as Full-Text Search capabilities for Express Edition. The time has come—with the release of Service Pack 1 (SP1) also comes the release of SQL Server Express Edition with Advanced Services, SQL Server Express Edition Toolkit, and SQL Server Management Studio Express.

Express Edition Download Page

SQL Server 2005 Express Edition with Advanced Services is available for download from the main SQL Server Express Edition page. The URL is *http://msdn.microsoft.com/vstudio/express/sql/download/*.

On the download page for Express Edition, you have a number of options, depending on exactly what features you need. The following are the available downloads:

- **SQL Server Express Edition with SP1 (SQLEXPR.exe)** This download gives you just the Express Edition database engine and connectivity tools. It does not include any management tools or any of the advanced services such as Full-Text Search or Reporting Services.
- **SQL Server Express Edition with Advanced Services (SQLEXPR_ADV.exe)** This download gives you the Express Edition database engine plus additional components that include SQL Server Management Studio Express, support for full-text catalogs, and support for viewing reports via Report Server. SQL Server Express Edition with Advanced Services also includes SP1. It's a bit tricky if you already have Express Edition installed from RTM: Installing this component is just like installing another instance of Express Edition. During the setup of Express Edition with Advances Services, if you choose not to upgrade your current Express Edition installation, you will end up with two installations of Express Edition on your box.
- **SQL Server Express Edition Toolkit (SQLEXPR_TOOLKIT.exe)** SQL Server 2005 Express Edition Toolkit provides tools and resources to manage SQL Server Express Edition and SQL Server Express Edition with Advanced Services. It also allows you to create reports using SQL Server 2005 Business Intelligence Development Studio. This download does not include the database engine; it includes only the management tools.
- **SQL Server Management Studio Express (SQLServer2005_SSMSEE.msi)** SQL Server Management Studio Express provides a graphical management tool for managing SQL Server 2005 Express Edition and SQL Server 2005 Express Edition with Advanced Services instances. Management Studio Express can also manage relational engine instances created by any edition of SQL Server 2005. It cannot manage Analysis Services, Integration Services, SQL Server 2005 Mobile Edition, Notification Services, Reporting Services, or SQL Server Agent. This download is just for the Management Studio Express tool; it does not contain the database engine or the reporting services designer.

In summary, if you want to install Express Edition with all the bells and whistles, just download and install SQL Server Express Edition with Advanced Services (SQLEXPR_ADV.exe) followed by SQL Server Express Edition Toolkit (SQLEXPR_TOOLKIT.exe). This will get you all the components that Express Edition has to offer.

Reporting Services in Express Edition

Reporting Services in SQL Server Express Edition is a server-based solution that enables the creation and management of reports derived from data stored in Express Edition. The full version of Reporting Services found in other editions provides many more features—such as the ability to schedule deployment of reports and create subscriptions—but even without these advanced features, Express Edition users have a powerful enough database engine and toolset to conquer many scenarios.

To use Reporting Services within Express Edition, you must have Microsoft Internet Information Services (IIS) installed before you install Express Edition.

Note If you installed the Microsoft .NET Framework before installing IIS, you might receive an error in the Express Edition setup program asking you to reinstall the .NET Framework. To avoid rolling back and reinstalling the .NET Framework, run the following on the command line and then click the Retry button.

Aspreg_iis.exe –i This .exe file is located in the .NET Framework folder under <installation drive>:\WINDOWS\Microsoft.NET\Framework\v2.0.50727.

Reporting Services can be considered a two-part installation. The Reporting Server service and Web-based administration page can be installed as an option with SQL Server Express Edition with Advanced Services (SQLEXPR_ADV.exe). If you want to actually create reports and do not want to modify XML using Notepad, you might want to also install BI Development Studio from the SQL Server Express Edition Toolkit (SQLEXPR_TOOLKIT.exe). BI Development Studio is a Visual Studio application that allows you to create reports using a full-featured wizard or by manually building reports using a designer surface (Figure 14-2).

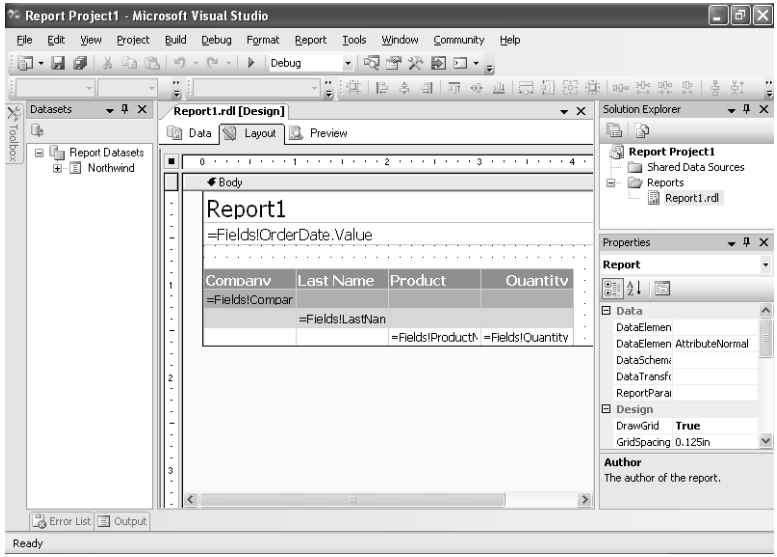

Figure 14-2 Report designer in SQL Server BI Development Studio

During the setup of SQL Server Express Edition with Advanced Services, if you opt to include Report Server, you are presented with some additional forms asking whether you want to configure the Report Server now or later. If you install the report server without configuring it, you can use the Reporting Services Configuration Manager to perform the same configuration work. You must configure the Report Server before you can deploy any reports (Figure 14-3).

Figure 14-3 Configure Report Server tool

Note If you instructed setup to automatically configure Report Server, you should know that it creates the Report Server Virtual Directory as ReportServer$SQLEXPRESS and it creates the Report Manager Virtual Directory as Reports$SQLEXPRESS. This is important because when you are deploying your report using the Report Designer, you must tell the designer to use ReportServer$SQLEXPRESS, not ReportServer, which is the default. This property is called the *TargetServerURL*, and you can change it by going to the Properties dialog box reached via the Project menu.

An in-depth discussion of Reporting Services is beyond the scope of this book, but a plethora of resources are available online, including free Report Packs to help you get started writing reports. Report Packs can be found at *http://www.microsoft.com/downloads/details .aspx?FamilyId=D81722CE-408C-4FB6-A429-2A7ECD62F674&displaylang=en*.

MSDN provides tutorials on Reporting Services as well as Webcasts and many other types of information. Go to this URL: *http://msdn.microsoft.com/sql/bi/reporting/*.

Full Text in Express Edition

The full-text feature within SQL Server Express Edition is no different than in any other edition of SQL Server. The main issue with full text in Express Edition is that the SQL Server

Management Studio Express tool does not have any fancy user interface to manage the full-text catalogs or anything related to full text. Thus, to use full text within Express Edition, you must brush up on your T-SQL skills. Luckily, Template Explorer within Management Studio Express helps you by providing templates for creating a full-text catalog, creating a full-text index, populating an index, and stopping population of an index.

To start playing with full text in Express Edition, we'll walk through creating a full-text index on the ProductName of the Products table in the Northwind database. First we must create our full-text catalog as shown here:

```
USE Northwind
GO
CREATE FULLTEXT CATALOG MyNorthwindCatalog AS DEFAULT
```

In SQL Server 2005, all user-defined databases are enabled for full text by default, so there is no need to explicitly call *sp_fulltext_database 'enable'*, as you do in previous versions of SQL Server. Next we will create the actual index:

```
CREATE FULLTEXT INDEX ON dbo.Products(ProductName) KEY INDEX PK_Products
```

Now we can use this index and query the Products table. Because I love gumbo, I can now quickly return all the gumbo products using the *CONTAINS* keyword, as shown here:

```
SELECT ProductName from Products where CONTAINS(ProductName,'Gumbo')
```

The result is as follows:

```
ProductName
----------------------
Chef Anton's Gumbo Mix
```

In SQL Server 2005, only columns of type *char*, *varchar*, *nchar*, *nvarchar*, *text*, *ntext*, *image*, *xml*, and *varbinary* can be indexed for full-text search. The previous example shows how to quickly get an index up and running. However, as with most multi-generational features in SQL Server, the capabilities of full text go far beyond what was discussed here. If you are interested in learning more about full-text searching within SQL Server 2005, go to *http://msdn2.microsoft.com/en-us/library/ms142547(SQL.90).aspx*.

Configuration

Every new database version seems to have even more features, more knobs to turn, and other things that users must learn about. SQL Server 2005 reflects a lot of effort put into improving the security of the product. One aspect of this effort was an "off by default" initiative, which basically required all nonessential features of SQL Server to be turned off. For example, on a default installation of SQL Server, the Browser service and Full-Text service are not running. Turning these features off reduces the surface area available for a malicious user.

The SQL Server Surface Area Configuration tool allows you to turn on and off these various features. This tool is installed as part of a database engine installation, so you have it regardless of which edition or components you install.

When the Surface Area Configuration tool is launched, you get two options: Surface Area Configuration for Services and Connections, and Surface Area Configuration for Features.

The Services and Connections dialog box, shown in Figure 14-4, shows a tree view list of SQL Server components installed on the server.

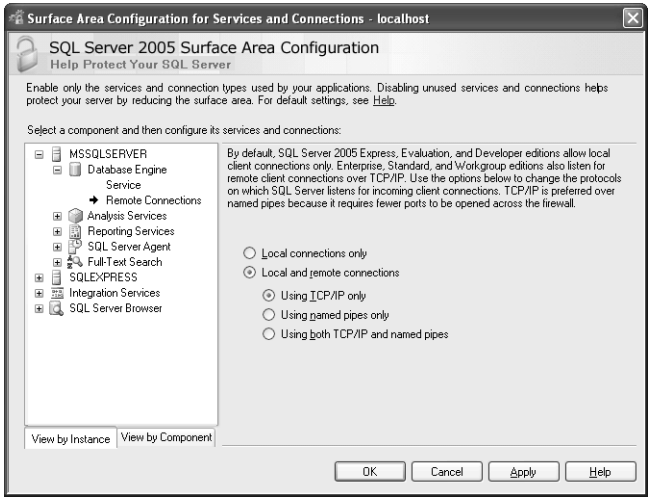

Figure 14-4 The Surface Area Configuration tool's remote connections panel

Under Database Engine you'll find two panels: Services allows the user to start, stop, and change the automatic startup settings of the service, and Remote Connections is perhaps one of the more important switches to remember. By default, in the Express, Evaluation, and Developer editions of SQL Server, remote connections are disabled. Thus, if you installed Express Edition and try to connect to it from another client machine, the connection will fail. To enable remote connections, you must choose which kind of remote connections to allow.

Note If you allow remote connections and are still having trouble connecting, check your firewall settings. You might have to add an exception for the sqlsvr.exe process or open a specific port for SQL Server.

The other important feature of this tool is the ability to configure which features should be on or off. Clicking on Surface Area Configuration for Features launches the dialog box shown in Figure 14-5.

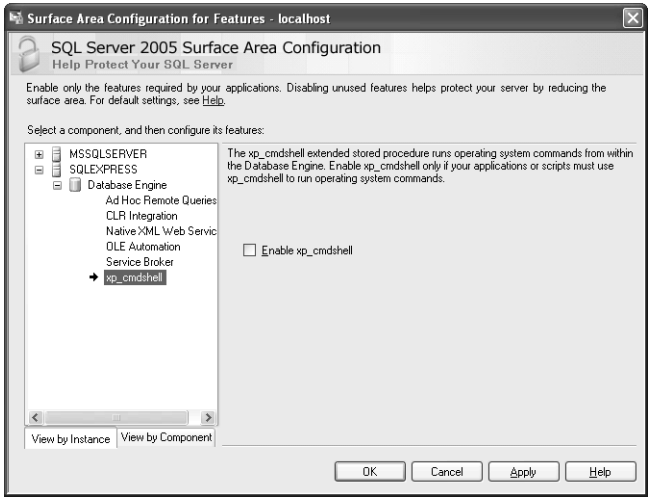

Figure 14-5 The Surface Area Configuration tool's features panel

Express Edition has a reduced set of features to enable or disable compared to other versions of SQL Server. This is because Express Edition doesn't have features such as SQL Mail, Database Mail, and SQL Server Agent. In Figure 14-5, you can see that the Surface Area Configuration tool allows you to enable and disable the use of the *xp_cmdshell* extended stored procedure. Other features that you can turn on and off include the CLR, ad hoc remote queries, and Service Broker.

You can programmatically turn on and off each of these options via the *sp_configure* stored procedure. To view these options, set the Show Advanced property on *sp_configure* first. There is also a new system view called *sys.configurations* that effectively shows the same information as *sp_configure* with the Show Advanced option set.

Working with SQL Server Express Edition

SQL Server Management Studio Express is a graphical user interface tool for managing SQL Server. It is a stripped-down version of its parent SQL Server Management Studio. The full version, which ships with all other editions of SQL Server, includes support for managing Analysis Services and Integration Services and offers much more functionality. This isn't to say that Management Studio Express is not useful; on the contrary, it supports all the basic administration functions, such as backup and restore, and it provides a rich text editor for creating and executing queries.

You can launch Management Studio Express from the Start menu. You first see a Connection dialog box. The default *SQLEXPRESS* instance name is automatically provided for you, so to connect to your Express Edition instance on your local machine, simply click the Connect button.

The GUI has three main parts: the toolbar, the Object Explorer tree view, and the document window, as shown in Figure 14-6. Visual Studio users will be most familiar with how the GUI works—in fact, SQL Server Management Studio is technically a Visual Studio application.

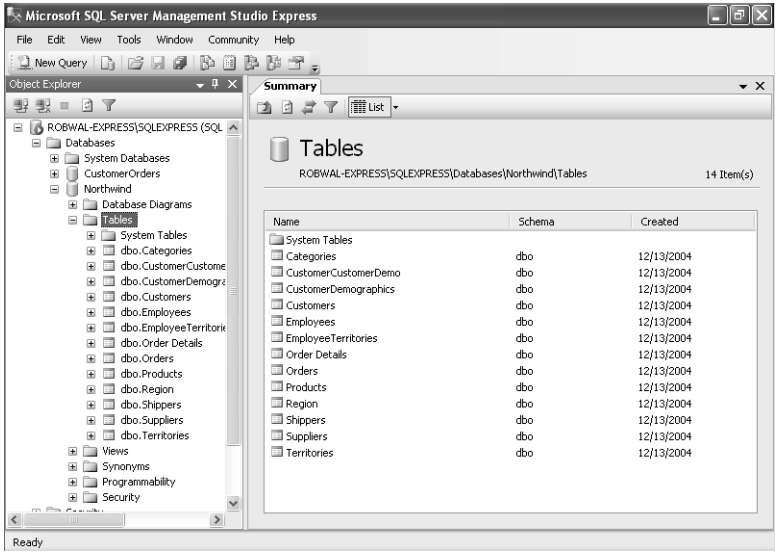

Figure 14-6 The SQL Server Management Studio Express tool

Figure 14-6 shows Management Studio Express connected to a server. The Object Explorer tree enumerates the Tables node of the Northwind database. One issue to note is that multi-select is not supported in Object Explorer in either Management Studio or Management Studio Express. To perform an action such as deleting multiple objects, you must use the Summary panel. By default, this panel shows information that depends on which object is selected in Object Explorer. You can close this panel without closing Management Studio Express. If this panel is not visible, you can always enable it by choosing Summary from the View menu or by pressing F7.

On the subject of multi-select, you can hold down the Ctrl key and select multiple objects in the grid and then right-click on the grid to see a shortcut menu of available options. This is the same behavior you would expect from any other Windows application. Another thing you will notice with shortcut menus and multi-select is that a lot more options are available when you select only one object. Because Object Explorer only supports selecting one item at a time, you get the same shortcut menu for both Object Explorer and the Summary grid if you have only one object selected.

You can use Management Studio Express to manage other editions of SQL Server, but the functionality is limited to what is available for Express Edition. (For example, when you connect to Workgroup Edition or Enterprise Edition, there is no way to manage Agent jobs.)

At this point, we are ready to discuss some of the more useful features within Management Studio Express. For this example, we will create a new database called CustomerOrders. Connect to your Express Edition server and select New Database from the shortcut menu of the Database Node in Object Explorer. This brings up the New Database dialog box (Figure 14-7).

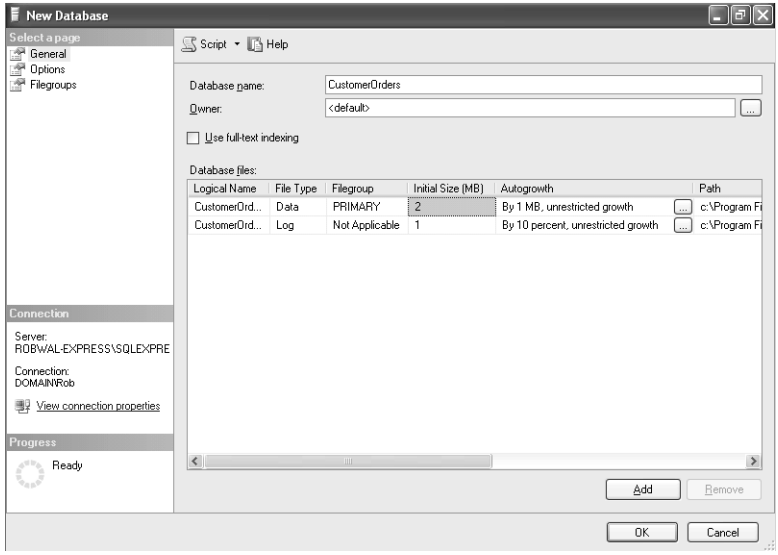

Figure 14-7 The New Database dialog box in Management Studio Express

Using this dialog box, we can specify the most common parameters, such as data and log file location, autogrowth parameters, and other database attributes. Almost all dialog boxes in Management Studio Express (including those in the full version of Management Studio) allow you to script the action instead of actually executing the action against the server. This is useful in a variety of ways—for example, if we want to know exactly what will be executed against our server without actually executing any commands. All you do is click the Script button in the dialog box and choose where to dump the script.

Once we have specified the name of the database—in our case, CustomerOrders—we can click OK, and we will see the new CustomerOrders node in Object Explorer. Now it's time to create a table for our new database. We could open a new query window and type the *CREATE TABLE* syntax with all the interesting columns we want, but there is a much simpler way to use Management Studio Express.

Management Studio Express provides a table designer for creating and modifying tables, as well as a view designer. To launch the table designer, right-click on the Tables node under our new database and choose New Table. This launches the table designer in another document window, as shown in Figure 14-8. This modeless approach allows users to go back and forth between document windows without having to close down each one first. Every dialog box used in Management Studio Express is modeless, so you do not have to launch another instance of Management Studio Express to do other work while another task (such as restoring a database) is taking a long time.

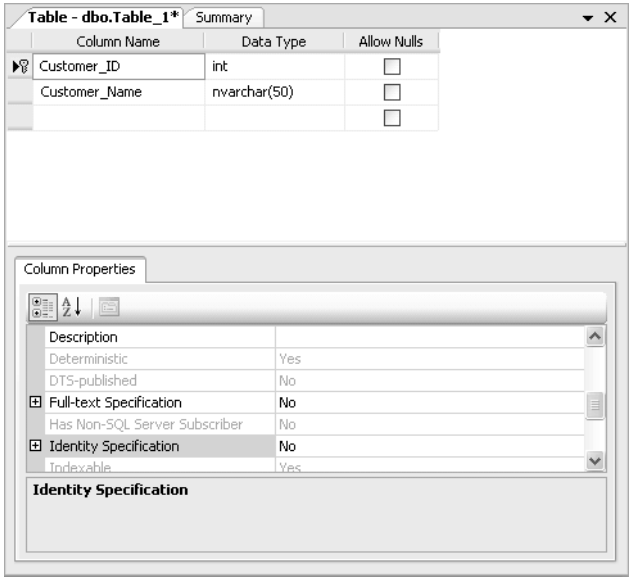

Figure 14-8 The table designer in Management Studio Express

The table designer has two main sections, the column definition grid and the column properties panel at the bottom of the window. The grid allows you to simply type the name of the column, select a data type from the combo box, and specify whether the column will allow null values. You can set the primary key, create an index, and create check constraints—among other things—by right-clicking a row in the grid. The power of the table designer and its ease of use make it a valuable asset in Management Studio Express.

The View designer in Management Studio Express provides a feature-rich environment for creating and modifying views. This designer has four panes: Diagram, Criteria, SQL, and Results. You can enable all panes at once (which results in a busy user interface, but if you have a big enough monitor this might be OK). For a clearer demonstration, Figure 14-9 shows the designer with the Diagram and Criteria panes, and Figure 14-10 shows the SQL and Results panes. These figures show data from the Northwind database because it offers a more complex relationship than the single table we have built so far in this chapter.

The Diagram pane shows an entity relationship among the various tables within the view.

Note To diagram the database, you can use a separate Diagram node in Object Explorer, which produces a user interface that is similar to the designers previously described.

The boxes indicate a table; you can resize and move them by clicking and dragging with the mouse. This view is not read-only; it's interactive—you can modify the contents of the tables and add or remove tables. For example, if you selected the Description check box in the Categories table, this would add Description to the criteria pane and you can add this to your view.

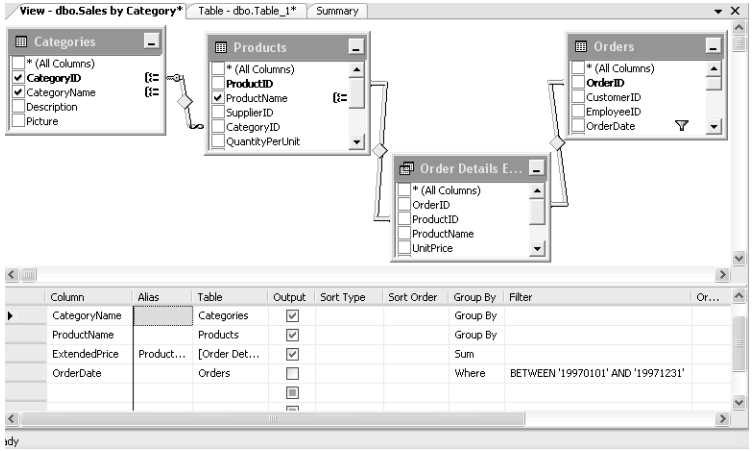

Figure 14-9 The View designer with the Diagram and Criteria panes

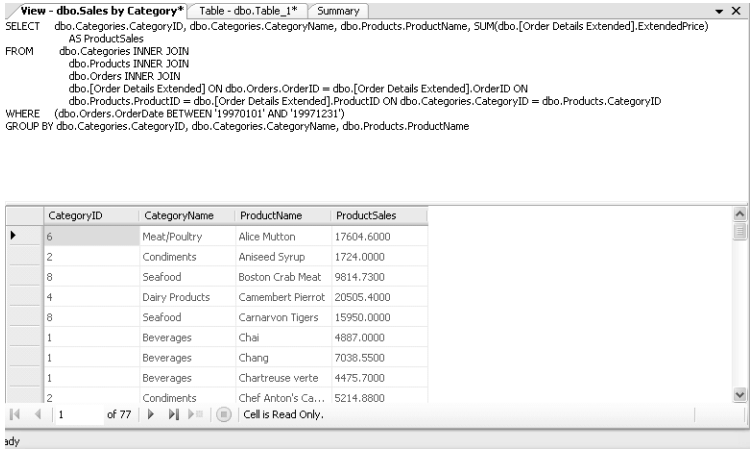

Figure 14-10 The View designer with the SQL and Results panes

If you want to tweak the T-SQL in the view, you can do so through the SQL pane. Any changes made in the SQL pane are carried over to all the other panes, and vice-versa. So if you checked the Description check box in the Categories table, this information is automatically added to the Criteria pane and the T-SQL within the SQL pane.

To conclude our discussion of Management Studio Express, we'll look at the ability to create and manage ad hoc queries. To create a new query, you simply click the New Query button. This creates a new document window and allows you to free-type T-SQL or load or save a script file. The query editor (as it is called) does not have IntelliSense (which offers autocompletion based on the first few characters of an object name that are typed), but it does have some other helpful features, such as line numbering and displaying the execution plan. Figure 14-11 shows the execution plan of the Invoices view in the Northwind database.

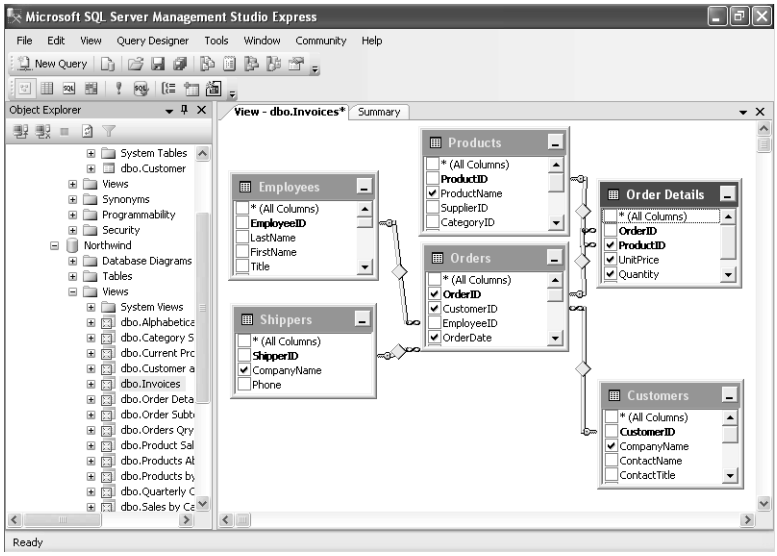

Figure 14-11 Invoices view

Notice the three tabs in the bottom section of the document window: Results, Messages, and Execution Plan. The Results tab normally shows up by default, displaying the results of whatever we submitted to SQL Server. In this example, to get the Execution Plan tab to show up, we had to first choose Include Actual Execution Plan from the Query menu.

Management Studio Express goes far beyond any management tool provided with MSDE—which is, basically, nothing. Most important, its tools are free and can be freely distributed with any custom application as long as the SQL Server Express Edition engine is shipped in the same application.

SQLCMD Command-Line Tool

Traditionally, command-line tools have always been available for executing script or ad hoc queries against SQL Server. These have included iSQL and oSQL, which each use a different technology, such as Open Database Connectivity (ODBC), to connect to the server instance and have different features. In SQL Server 2005, a new command-line tool replaces both oSQL and iSQL. SQLCMD is not just a replacement—it extends the functionality of the current tools in a variety of ways. Some of the new features include the following:

- Scripting variables
- Ability to include multiple in-line scripts
- Ability to dynamically change connections
- Support for Dedicated Administrator Connection (DAC)
- Support for the new SQL Server 2005 data types: *xml* and *nvarchar(max)*

Suppose you need to create a maintenance script that performs a DBCC *CHECKDB*, performs a database backup, and then copies the backup file to another location. Here is an example T-SQL script:

```
DBCC CHECKDB ('Northwind') WITH NO_INFOMSGS
GO
BACKUP DATABASE [Northwind] TO DISK='C:\Backups\Northwind\Northwind.bak'
GO
Exec xp_cmdshell 'copy C:\Backups\Northwind\*.bak D:\TapeBackupDropFolder\Northwind'
GO
```

Life with this single script file goes well, and eventually you are tasked with doing this for another database. Fine—you copy and paste the code and change the Northwind entries to pubs entries, and now you have the script shown in Listing 14-1.

Listing 14-1 T-SQL script

```
DBCC CHECKDB ('Northwind') WITH NO_INFOMSGS
GO
BACKUP DATABASE [Northwind] TO DISK='C:\Backups\Northwind\Northwind.bak'
GO
Exec xp_cmdshell 'copy C:\Backups\Northwind\*.bak D:\TapeBackupDropFolder\Northwind'
GO
DBCC CHECKDB ('Pubs') WITH NO_INFOMSGS
GO
BACKUP DATABASE [Pubs] TO DISK='C:\Backups\Pubs\Pubs.bak'
GO
Exec xp_cmdshell 'copy C:\Backups\Pubs\*.bak D:\TapeBackupDropFolder\Pubs'
GO
```

Time passes, and you are asked to do this for another three databases, and the location of the drop folder has changed. You can imagine that every time you want to add databases to this script, another copy and paste will be required, which introduces the possibility of bugs within the script. Wouldn't it be great if we could use variables within the script instead? SQLCMD in SQL Server 2005 supports the use of variables for this type of scenario. To create a variable, we use *:SETVAR*, as shown in the modified script in Listing 14-2.

Listing 14-2 T-SQL script (modified)

```
:SETVAR DatabaseName Northwind
DBCC CHECKDB ($(DatabaseName)) WITH NO_INFOMSGS
GO
BACKUP DATABASE $(DatabaseName) TO DISK='C:\Backups\$(DatabaseName)\$(DatabaseName).bak'
GO
Exec xp_cmdshell 'copy C:\Backups\$(DatabaseName)\*.bak D:\TapeBackupDropFolder\
$(DatabaseName)'
GO

:SETVAR DatabaseName Pubs
```

```
DBCC CHECKDB ($(DatabaseName)) WITH NO_INFOMSGS
GO
BACKUP DATABASE $(DatabaseName) TO DISK='C:\Backups\$(DatabaseName)\$(DatabaseName).bak'
GO
Exec xp_cmdshell 'copy C:\Backups\$(DatabaseName)\*.bak
D:\TapeBackupDropFolder\$(DatabaseName)'
GO
```

Before we optimize this example, we should mention a few key points. You can create and change variables throughout the script. You can also place the value of the variable by typing the name of the variable preceded by *$(* and followed by *)*. The replacement happens at run time for the script, and as you can see in this example, it dynamically places the backup in the folder whose name is defined in the variable. In addition, a list of environment variables are available to use within your script. They are listed in Table 14-3.

Table 14-3 SQLCMD Environment Variables

Variable	Related Switch	Read/Write
SQLCMDUSER	*U*	R
SQLCMDPASSWORD	*P*	N/A
SQLCMDSERVER	*S*	R
SQLCMDWORKSTATION	*H*	R
SQLCMDDBNAM	*d*	R
SQLCMDLOGINTIMEOUT	*l*	R/W
SQLCMDSTATTIMEOUT	*t*	R/W
SQLCMDHEADERS	*h*	R/W
SQLCMDCOLSEP	*s*	R/W
SQLCMDCOLWIDTH	*w*	R/W
SQLCMDPACKETSIZE	*a*	R
SQLCMDERRORLEVEL	*m*	R/W
SQLCMDMAXVARTYPEWIDTH	*y*	R/W
SQLCMDMAXFIXEDTYPEWIDTH	*Y*	R/W

Using these special variables is the same as using a custom variable. For example, if we want to print out the name of the workstation executing the script, we can do the following:

```
Print "$(SQLCMDWORKSTATION)"
```

Returning to our example, if we simply left the script as is, our managers and colleagues might question our programming skills. To make this script more efficient, let's just write the process once and define the variables on the command line. So our script, which we will call BackupMyDatabase.sql, becomes that shown in Listing 14-3.

Listing 14-3 BackupMyDatabase.sql

```
DBCC CHECKDB ($(DatabaseName)) WITH NO_INFOMSGS
GO
BACKUP DATABASE $(DatabaseName) TO DISK='C:\Backups\$(DatabaseName)\$(DatabaseName).bak'
GO
Exec xp_cmdshell 'copy C:\Backups\$(DatabaseName)\*.bak D:\TapeBackupDropFolder\
$(DatabaseName)'
GO
```

To back up each database, we can use SQLCMD as follows:

```
SQLCMD –E –S.\SQLEXPRESS –i "C:\MaintenanceScripts\BackupMyDatabase.sql" –
v DatabaseName="Northwind"
```

Notice that the parameters are case sensitive. A *-v* tells SQLCMD that the subsequent value is a variable name, not a severity level (as in the case of a capital *V*). In this example, we tell SQLCMD to connect to the *SQLEXPRESS* instance, execute the .sql script file BackupMyDatabase.sql, and define the *DatabaseName* variable to be *Northwind*. Now all we have to do is repeat this for pubs, and we are all set. However, we can leverage yet another feature to make this even more efficient. We can include a script within an existing script and still keep our variable values. Expanding on this example, let's create a new script called LaunchBackup.sql, as defined in Listing 14-4.

Listing 14-4 LaunchBackup.sql

```
:SETVAR DatabaseName Northwind
:R "BackupMyDatabase.sql"
:SETVAR DatabaseName Pubs
:R "BackupMyDatabase.sql"
```

The *:R* tells SQLCMD to load the .sql script in-line at run time, before execution. Now it's quite easy to add another database to our process of backing up the database.

Note To run this script, all we have to do is tell SQLCMD to launch the script file. We don't have to pass variables because they are defined in the script file itself.

```
SQLCMD –E –S.\SQLEXPRESS –i "LaunchBackup.sql"
```

What if we want to perform this operation on multiple servers or multiple instances of SQL Server? SQLCMD has yet another feature that allows the script to make connections from within the script using the *:CONNECT* command. Expanding on our example, say we want to connect to our test server and perform this operation. We can do this with the code in Listing 14-5.

Listing 14-5 LaunchBackup.sql connecting to multiple servers

```
--Connect to my local express machine
:CONNECT .\SQLEXPRESS
:SETVAR DatabaseName Northwind
:R "BackupMyDatabase.sql"
--Now connect to my test server and backup Northwind
:CONNECT TESTSERVER1\SQLEXPRESS
:SETVAR DatabaseName Northwind
:R "BackupMyDatabase.sql"
```

Combining the powerful functionality of connecting to different servers and variables gives us the ability to do a plethora of tasks within our database scripts. One additional key feature of SQLCMD is worth mentioning here. In the SQL Server 2005 engine there is a special thread that basically waits just for a connection from an SQLCMD client with a *-A* parameter. This special connection is called the Dedicated Administrator Connection (DAC), and its purpose is to provide administrators with a way to connect to SQL Server if, for whatever reason, the server becomes unresponsive. The DAC is available only when you connect to SQL Server via SQLCMD with the special *-A* parameter. You cannot make this connection from any other client software (such as SQL Server Management Studio). In addition, only one connection of this type is allowed per instance of SQL Server because this feature is designed to be used only in the event that administrators cannot gain access through a normal network connection.

There is one more important point about the DAC: In Express Edition, the DAC is disabled. To use the DAC, you must start the service running a trace flag of 7806. If you have trouble connecting to the DAC once you restart the service with this trace flag, make sure the SQL Server Browser service is running.

Note To set a trace flag in SQL Server, use *-T#* as a startup parameter, where # is the trace flag number. If you view the SQL Server service properties from the Service control panel, you will see a Start Parameters text box. To experiment with the DAC, you can place *-T7806* (no space between the T and 7) there.

SQLCMD can connect to the Express Edition–only feature called User Instances. However, there is another tool called SSEUTIL (available for download on MSDN) that looks and feels like SQLCMD but is used primarily to manage Express Edition user instances. Before we go into the details of SSEUTIL and how it compares with SQLCMD, we should first provide an overview of User Instances.

User Instances

SQL Server 2005 Express Edition supports a new feature called User Instances that is available only when you use the .NET Framework data provider for SQL Server (SqlClient). This feature is available only with Express Edition. A user instance is basically a separate

instance of the Express Edition database engine that is generated by a parent instance. For example, as a Windows administrator, I can install Express Edition on my client machine. I can then take an application that will connect to Express Edition and create a separate instance of Express Edition for each client user of the application. When connected to this user instance, the client is the sysadmin for that instance but does not have any special privileges on the parent Express Edition instance. This is important because software executing on a user instance with limited permissions cannot make system-wide changes. This is because the instance of Express Edition is running under the non-administrator Windows account of the user, not as a service. Each user instance is isolated from its parent instance and from any other user instances running on the same computer. Databases running on a user instance are opened in single-user mode only, and multiple users cannot connect to databases running on a user instance. Replication, distributed queries, and remote connections are also disabled for user instances.

When you install Express Edition, one of the setup wizard pages asks if you want to enable User Instances. This is one way to enable them; the other is to toggle the setting using *sp_configure*, as follows:

```
-- Enable user instances.
sp_configure 'user instances enabled','1'
-- Disable user instances.
sp_configure 'user instances enabled','0'
```

There are a couple more restrictions on user instances. For one thing, the only way to connect to them is through Named Pipes. In addition, SQL Server logins are not supported on User Instances; connections to the User Instance must be made using Windows Authentication.

Now that we have discussed User Instances and some of the restrictions on them, take a look at the example connection string shown here:

```
Data Source=.\\SQLExpress;Integrated Security=true;
User Instance=true;AttachDBFilename=|DataDirectory|\InstanceDB.mdf;
Initial Catalog=InstanceDB;
```

Note the following:

- The *Data Source* keyword refers to the parent instance of Express Edition that is generating the user instance. The default instance is *.\sqlexpress*.
- *Integrated Security* is set to *true*. To connect to a user instance, you must use Windows Authentication; SQL Server logins are not supported.
- *User Instance* is set to *true*, which invokes a user instance. (The default is *false*.)
- *AttachDBFilename* indicates the location of the InstanceDB database.

- The *DataDirectory* substitution string enclosed in the pipe symbols refers to the data directory of the application opening the connection and provides a relative path indicating the location of the .mdf and .ldf database and log files. If you want to locate these files elsewhere, you must provide the full path to the files.

When the *SqlConnection* is opened, it is redirected from the default Express Edition instance to a run-time-initiated instance running under the caller's account.

Unlike with versions of SQL Server that run as a service, SQL Server Express Edition user instances do not need to be manually started and stopped. Each time a user logs in and connects to a user instance, the user instance is started if it is not already running. User instance databases have the *AutoClose* option set so that the database is automatically shut down after a period of inactivity. The Sqlservr.exe process that is started is kept running for a limited timeout period after the last connection to the instance is closed, so it does not need to be restarted if another connection is opened before the timeout has expired. The user instance automatically shuts down if no new connection opens before that timeout period has expired. A system administrator on the parent instance can set the duration of the timeout period for a user instance by using *sp_configure* to change the user instance timeout option. The default is 60 minutes.

The first time a user instance is generated for each user, the master and msdb system databases are copied from the Template Data folder to a path under the user's local application data repository directory for exclusive use by the user instance. This path is typically C:\Documents and Settings\<UserName>\Local Settings\Application Data\Microsoft\Microsoft SQL Server Data\SQLEXPRESS. When a user instance starts up, the tempdb, log, and trace files are also written to this directory. A name is generated for the instance, which is guaranteed to be unique for each user.

By default, all members of the Windows Builtin\Users group are granted permissions to connect on the local instance as well as read and execute permissions on the SQL Server binaries. Once the credentials of the calling user hosting the user instance have been verified, that user becomes the sysadmin on that instance. Only shared memory is enabled for user instances, which means that only operations on the local machine are possible.

Note that users must be granted both read and write permissions on the .mdf and .ldf files specified in the connection string. These files represent the database and log files, respectively, and are a matched set. The main database file will not open if it is coupled with the wrong log file.

To prevent data corruption, a database in the user instance is opened with exclusive access. If two different user instances share the same database on the same computer, the user on the first instance must close the database before it can be opened in a second instance. User instances are given a globally unique ID (GUID) as a name. This makes them a bit more challenging to manage. The SSEUTIL command-line tool makes managing user instances easier.

SSEUTIL

SSEUTIL is available as a free download from MSDN at *http://www.microsoft.com/downloads/details.aspx?FamilyID=fa87e828-173f-472e-a85c-27ed01cf6b02&DisplayLang=en*. The tool lets you easily interact with SQL Server. Among other things, it allows you to:

- Connect to the main instance or user instance of SQL Server
- Create, attach, detach, and list databases on the server
- Upgrade database files to match the version of the server
- Execute SQL statements via the console (similar to SQLCMD)
- Retrieve the version of SQL Server that is running
- Enable and disable trace flags (for example, to trace SQL statements sent to the server by any client application)
- List the instances of SQL Server on the local machine or on remote machines
- Checkpoint and shrink a database
- Measure the performance of executing specific queries
- Create and play back lists of SQL commands for the server to execute
- Log all input and output

Although SSEUTIL might appear to be very similar to SQLCMD, it uses different command-line switches to connect to SQL Server. Once you connect to SQL Server Express Edition using SSEUTIL, you can either use a command prompt (which resembles the SQLCMD behavior) or you can request a graphical console window. Either way, you must specify either *-c* for a command prompt or *-consolewnd* to use the graphical user interface, as shown here:

```
C:\>sseutil –s .\sqlexpress –c
```

The default in Express Edition is to call the named instance, *SQLEXPRESS*. SSEUTIL does not require you to specify the *-s* (*servername*) parameter; if you omit it, SSEUTIL assumes that you want to connect to the *SQLEXPRESS* instance on your local box.

Another important point about SSEUTIL and defaults is worth noting. From the preceding information, you might think that we are connecting to the default parent instance of Express Edition. In reality, we are connecting to the server instance, *.\sqlexpress*, but our connection is then redirected to the user instance of the user who is logged in. If we specifically want to connect to the Express Edition parent instance, we must pass the *-m* parameter, as shown here:

```
C:\>sseutil –s .\sqlexpress –m
```

If you do not specify *-m,* SSEUTIL uses the *-child* parameter, which tells SSEUTIL to connect to the user instance of the user who is currently logged in to the box. You can append a

different user name to this parameter if you want to connect to another user instance. To obtain a list of active user instances, you can use the *-childlist* parameter, as shown here:

```
C:\>sseutil -childlist
User             Pipe                                      ProcessId   Status
-------------------------------------------------------------------------------
DOMAIN\robwal   \\.\pipe\07EB27D8-877E-4F\tsql\query   2168        alive
```

Now that we have discussed how to connect to both the parent and child instances using SSEUTIL, it's time to do something useful with SSEUTIL. Suppose we want to attach the Northwind database to our user instance. We can enumerate the databases that are attached to our instance by using the *-l* parameter:

```
C:\>sseutil -child DOMAIN\robwal -l
Using instance '\\.\pipe\07EB27D8-877E-4F\tsql\query'.

1. master
2. tempdb
3. model
4. msdb
```

SSEUTIL lets you attach or detach a database from both the parent instance and any user instance on your box. Let's attach Northwind, whose database files we'll assume are located in a directory called C:\databases.

```
C:\>sseutil -child DOMAIN\robwal -a "C:\databases\northwnd.mdf" Northwind
Using instance '\\.\pipe\07EB27D8-877E-4F\tsql\query'.

Command completed successfully.
```

For completeness, we specified the *-child* parameter, but it does not actually need to be there because it's the default. The *-a* attach parameter takes two values. The first is the location of the .mdf file to be attached. The second parameter, which is optional, is the name of the database. Once attached, the database exists on the instance but not on the parent instance, again reinforcing our belief that user instances are truly a separate instance of SQL Server.

Here is a list of databases on the parent Express Edition instance:

```
C:\>sseutil -m -l master
2. tempdb
3. model
4. msdb
5. CustomerOrders
```

Here is a list of databases on the user instance of the logged-in user:

```
C:\>sseutil -child DOMAIN\robwal -l
Using instance '\\.\pipe\07EB27D8-877E-4F\tsql\query'.

1. master
2. tempdb
3. model
4. msdb
5. Northwind
```

If you develop an application that uses user instances, you will find that SSEUTIL is a useful tool for quickly obtaining information and performing tasks against the instance. Alternatively, you can use SQL Server Management Studio Express against a user instance. To do this, you simply obtain the pipename of the user instance. One way to get the pipename is to use the *-childlist* parameter. In the examples in this section, our pipename was *\\.\pipe\07EB27D8-877E-4F\tsql\query*. Open Management Studio Express and, in the Connection dialog box, paste the pipename into the Server Name text box.

Installing SQL Server Express Edition

Express Edition offers three ways to perform an installation or upgrade: via the Setup Wizard, silently through the use of command-line parameters, or via a configuration file. Using the wizard to quickly get your development or test environment set up is useful if you don't already have Express Edition on your box. When bundling Express Edition with your application, you might find that installing Express Edition via a configuration file provides the best user experience.

Note You might already have Express Edition installed if you already have Visual Studio installed on your box. You can check to see whether Express Edition is installed by going to the Services applet in Control Panel and looking for "SQL Server (SQLEXPRESS)." Note that if you installed Express Edition and gave it a different instance name from the default SQLEXPRESS, the name shown in Control Panel might be different.

You can also run the Express Edition setup to install another instance of Express Edition. Express Edition supports up to 50 instances on a single computer.

Existing Beta Versions of SQL Server 2005

If your box contains previously released versions of SQL Server, you might run into problems if you attempt to upgrade to the released version by simply executing the setup program of the released version. To ensure a successful installation, manually remove all pre-release versions of SQL 2005, Visual Studio 2005, and the .NET

Framework 2.0. If you are lucky enough to be in this situation, it will be beneficial to read through the MSDN article "Uninstalling Previous Versions of Visual Studio 2005," which can be found at *http://msdn.microsoft.com/vstudio/express/support/uninstall*.

If you want to download SQL Server 2005 Express Edition from the Microsoft Web site, you can go to *http://go.microsoft.com/fwlink/?LinkId=64064*. On this Web page, you will see the prerequisites for installing Express Edition. You need Windows Installer 3.0 and the .NET Framework 2.0. The Web page includes links for downloading these—you must install them before you can run setup for Express Edition, which is the file on this Web page called SQLEXPR.exe. If you simply run this executable, it self-extracts and runs through the Setup Wizard. To perform a silent installation or an installation using the configuration file, you can download this file and run SQLEXPR.exe /X with the */X* attribute; this extracts the files without running the Setup Wizard. If you do not want to extract all the Express Edition installation files, you can also pass installation parameters to SQLEXPR.exe.

After we have the Express Edition setup bits on our box, we can either run through the wizard to install or upgrade to SQL Server 2005 Express Edition or we can use a configuration file called Template.ini to silently perform an installation. We will first cover installing Express Edition using the Setup Wizard.

Using the Setup Wizard to Manually Install Express Edition

Once you have the .NET Framework 2.0 and at least version 3.0 of the Windows Installer, you are ready to run the Setup Wizard for Express Edition. In this example, we'll use the setup for SQL Server Express Edition with Advanced Services (SQLEXPR_ADV.exe).

Setup first installs the SQL Native Client and some additional setup support files. The SQL Native Client is a standalone data access API that is used for both the Object Linking and Embedding Database (OLE DB) and ODBC. It combines the SQL OLE DB provider and the SQL ODBC driver into one native dynamic link library (DLL). It also provides new functionality beyond that supplied by the Microsoft Data Access Components (MDAC).

Next, the familiar Welcome screen appears. Continuing through the wizard brings us to the System Configuration Check page. This page does some rudimentary checks to see whether your operating system is appropriate for Express Edition and whether you have any pending reboot requests from another application, and it looks at a number of other issues that could prevent setup from working correctly.

The next page in the wizard is for registration (Figure 14-12).

Notice the Hide Advanced Configuration Options check box at the bottom of the page. This is selected by default; if we uncheck it, we can change the name of the instance and database

collation, as well as specify some of the service settings for the SQL Server and SQL Server Browser service.

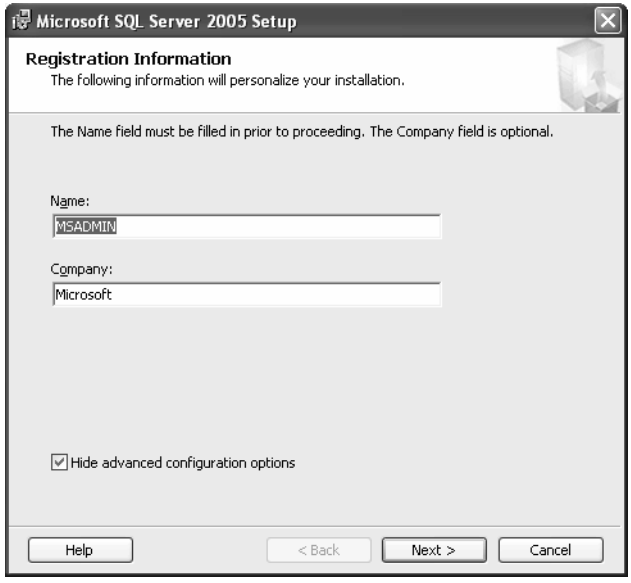

Figure 14-12 Registration Information page of the Setup Wizard

If we uncheck the box and continue with the wizard, we are presented with the Feature Selection page. This page is not an advanced option; users see it whether or not they chose to hide the advanced options. Figure 14-13 shows this page.

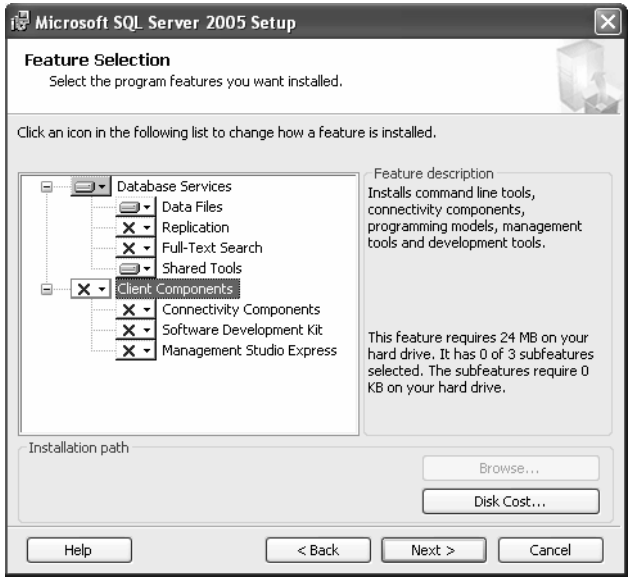

Figure 14-13 Feature Selection page of the Setup Wizard

Note If you are running the Advanced Services setup and have IIS installed, you will see an option for installing Reporting Services. Choosing to install Reporting Services leads to a few other dialog boxes not listed here. If you choose to install Reporting Services, the wizard will ask if you want to configure Report Server now or later. If you select later, you can always configure it through the Report configuration tool, which is one of the tools provided under the Microsoft SQL Server node on the Start Menu.

Note that by default, only the database engine and its support files (such as the master database) are installed. If you want Replication, Full-Text, or Management Tools, you must explicitly add them using the drop-downs in the tree view.

If we had requested to see the advanced options, we would see the Instance Name page next. This page has two option buttons, one for installing Express Edition as the default instance and one for installing it as a named instance. The default is to install Express Edition as a named instance with the name SQLEXPRESS. This is how all Visual Studio installations of Express Edition create their installations of Express Edition, and it is the recommended installation.

The next advanced page is the Service Account page (Figure 14-14).

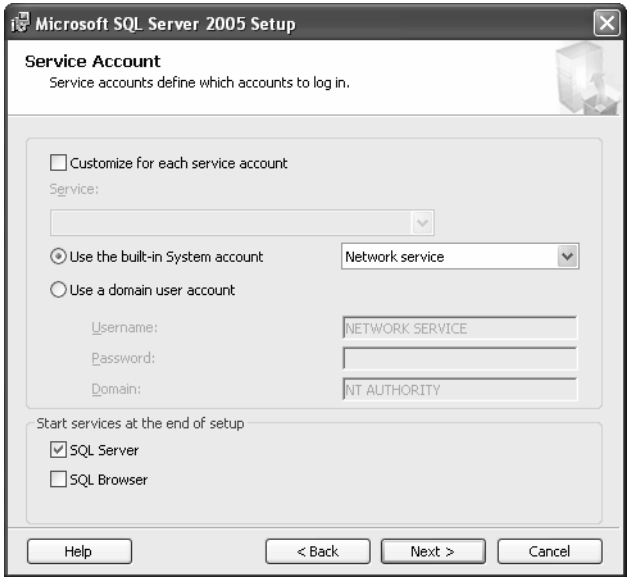

Figure 14-14 Service Account page of the Setup Wizard

The first option you see is the ability to specify credentials for each service account. This is applicable if you are planning to use both the SQL Server service and the SQL Browser service. Most of the time, you can run both services using the Network service account.

Most developers can probably intuit what the "SQL Server" service is. But some of us might not be so sure about the SQL Browser service. In previous versions of SQL Server that supported the idea of "instances," there needed to be a way for client machines to connect to SQL and ask, "What are all your available instances and their corresponding port numbers?" Traditionally, UDP port 1434 was used for this. As you can imagine, having this inside the server process was not only taxing but there was no way to effectively turn it off. In SQL Server 2005, this functionality has been separated out into a new service called the SQL Browser service.

The next page is the Authentication Mode page. Here you can set Express Edition to support only Windows Authentication or both SQL Authentication and Windows Authentication. If you choose Windows Authentication, you will still have an "sa" account, but that account will be disabled and have a random complex password.

Continuing on with the wizard, if you elected to show advanced pages, you will see the Collation Settings page. This page allows you to specify whether databases should be case sensitive or case insensitive, as well as what kind of sort order the database should use.

The next page asks you if you want to enable User Instances. User Instances is an Express Edition–specific feature—it is not available on any other edition of SQL Server. (User Instances were covered in the earlier section titled "Working with SQL Server Express Edition.")

At this point, we are almost done with the wizard. The last page that requires us to make some sort of decision is the Error And Usage Report Settings page. This page asks two questions: "Can SQL automatically send error reports to either a central server in your organization or directly to Microsoft?" and "Can SQL send data on feature usage to Microsoft?" Both of these options enable Microsoft to compile a lot of information about the use of SQL Server and common errors. With the second option, no personally identifiable information is sent, so if you are in the mood to contribute to future products, you might want to select this box.

The next page shows a summary of the actions the wizard will perform. After this page, the wizard proceeds and installs Express Edition, including whatever options you have selected. The final page (Figure 14-15) displays some information about the installation, as well as two hyperlinks.

Figure 14-15 The final page of the Setup Wizard

The Summary Log link loads the installation log into Notepad for your reading or troubleshooting enjoyment. The other link launches the Surface Area Configuration tool (described previously in this chapter). If you do not run this tool now, you can always launch it from the Start menu later.

At this point, you are ready to start working with Express Edition.

Installing via Command-Line Parameters or a Configuration File

There are two ways to programmatically install, modify, and remove SQL Server Express Edition components. First, you can call Setup.exe and pass a series of parameters on the command line. Alternatively, you can configure all the parameters within a single file and pass that file as a command-line parameter to Setup.exe. This file is called Template.ini, and it is located in the root directory of SQL Server Express Edition. An example of launching Setup and passing the configuration file is as follows:

```
start /wait setup.exe /qb /settings c:\template.ini
```

For those not familiar with the Windows command line, *start* is an application that opens a new console window. The */wait* parameter tells the console window to wait until the program finishes execution before terminating itself. A complete list of parameters for the *start* command can be obtained by passing */?* as a parameter. The next parameter in the line of code is *setup.exe*. This is the name of the application to launch; in this case, it's SQL Server Express Edition Setup. The rest of the parameters are arguments for the setup program. The Setup.exe argument */qb* tells Setup to run in quiet mode, which requires no user interaction. This mode does provide some visual indicators about the status of the installation.

Note Alternatively, you can specify */qn*. The only difference between */qb* and */qn* is that when you use */qn*, there is no visual status indicator.

All errors from Setup in SQL Server are recorded in the setup log files. If you call Microsoft for support on a setup-related issue, the Product Support specialist will probably want you to find these files. By default, the setup log files are located at C:\Program Files\Microsoft SQL Server\90\Setup Bootstrap\LOG\Files. If you encounter problems when developing a custom SQL Server Express Edition installation, these files are a good place to start debugging.

The */settings* parameter in the command-line code tells Setup to obtain all installation information from the file that is defined in the next parameter. In the code example, the Template.ini file is stored in the root of the C drive.

Important If you have downloaded SQL Server Express Edition from the Web, you might not see the Setup.exe application if you downloaded a single SQLEXPR.exe file. If this is the case, you need to call Setup.exe from the command line to extract the Express Edition files from this compressed executable. To perform this extraction, run *SQLEXPR /X* from the command line. A dialog box appears, prompting you for a location to extract the files to. The Express Edition files are copied to the location that you specify. These files include Setup.exe and Template.ini, among many other files and folders.

The Template.ini file is a plain-text file that can be opened by using a text editor such as Notepad. When you open this file, you see a long commented introduction citing examples of how to use the file. The file itself is well documented, and a lot of the options are explained in great detail within the file itself. For that reason, this chapter will not cover all the options. Instead, here are just a few parameters of interest.

- ***PIDKEY*** This parameter is required for all SQL Server editions except Express Edition.
- ***ADDLOCAL*** This parameter specifies which components to install. If *ADDLOCAL* is not specified, Setup will fail. You can specify *ADDLOCAL=ALL*, which installs all components. There are some rules around this parameter:

1. Feature names are case sensitive.
2. To use *ADDLOCAL*, you must provide a comma-delimited list of features to install, with no spaces between them.
3. Selecting a parent feature installs only the parent feature, not the child features. Installing a child feature installs the parent and the child. Removing the parent feature removes both the parent and the child.
4. You can also use *ADDLOCAL* to add components in maintenance mode.

Confused yet? We will give a few examples to demonstrate various installation combinations. First we must explain the possible valid feature names for each edition of SQL Server. Tables 14-4, 14-5, and 14-6 list the feature names unique to Express Edition, Express Edition with Advanced Services, and the Express Edition Toolkit, respectively.

Table 14-4 Valid *ADDLOCAL* Parameters for Express Edition

Feature	Parent Feature Name	Child Feature Name
SQL Server Database Services	SQL_Engine	
Data Files		SQL_Data_Files
Replication		SQL_Replication
Client Components	Client_Components	
Connectivity Components		Connectivity
Software Development Kit		SDK

Table 14-5 Valid *ADDLOCAL* Parameters for Express Edition with Advanced Services

Feature	Parent Feature Name	Child Feature Name
SQL Server Database Services	SQL_Engine	
Data Files		SQL_Data_Files
Replication		SQL_Replication
Full-Text Search Engine		SQL_FullText
Reporting Services	RS_Server	
Report Manager		RS_Web_Interface
Client Components	Client_Components	
Connectivity Components		Connectivity
Software Development Kit		SDK
Management Studio Express		SQL_SSMSE

Table 14-6 Valid *ADDLOCAL* Parameters for Express Edition Toolkit

Feature	Parent Feature Name	Child Feature Name
Client Components	Client_Components	
Connectivity Components		Connectivity
Software Development Kit		SDK
BI Development Studio		SQL_WarehouseDevWorkbench
Management Studio Express		SQL_SSMSE

Even though some of these feature names are self-descriptive, Table 14-7 provides a description of each feature, for completeness of our discussion.

Table 14-7 Feature Descriptions

Feature	Description
SQL_Engine	Installs the SQL Server database, including the SQL Server and SQL Browser services.
SQL_Data_Files	Installs core SQL Server databases, including master, the resource database, and tempdb.
SQL_Replication	Installs files necessary for replication support in SQL Server Express Edition.

Table 14-7 Feature Descriptions

Feature	Description
SQL_FullText	Installs files necessary for Full-Text Search support in SQL Server Express Edition.
RS_Server	Installs the Report Server service, which manages, executes, and renders reports.
RS_Web_Interface	Installs a Web-based tool used for managing a report server.
Client_Components	Installs components for communication between clients and servers, including network libraries for ODBC and OLE DB. Also installs applications such as the sqlcmd utility (oSQL replacement), SQL Server Configuration Manager, and the Surface Area Configuration tool.
Connectivity	Installs components for communication between clients and servers, including network libraries for ODBC and OLE DB.
SDK	Installs software development kits containing resources for model designers and programmers. This includes SQL Server Management Objects (SMO) and Replication Management Objects (RMO).
SQL_SSMSE	Installs SQL Server Management Studio Express.
SQL_WarehouseDevWorkbench	Installs BI Development Studio. It also installs Visual Studio Premier Partner Edition if no other edition of Visual Studio is installed on the computer.

With this many options, quite a number of installation combinations are possible. The following are examples of common installation configurations.

- **Install everything**
  `ADDLOCAL=All`
- **Install just the database engine**
  `ADDLOCAL=SQL_Engine,SQL_Data_Files,Connectivity`
- **Install just the management tool**
  `ADDLOCAL=SQL_SSMSE`
- **REMOVE** This parameter is similar to *ADDLOCAL*, but instead of adding components, it either removes a specific component or completely uninstalls SQL Server Express Edition if you use *REMOVE=ALL*. The following example removes the client components of an existing Express Edition installation:

  ```
  REMOVE=Client_Components.
  ```

  You do not have to specify an instance name because the *Client_Components* are not instance-specific. If you were removing *SQL_Replication* support, you would also need to add the following parameter:

  ```
  INSTANCENAME=<<name of the SQL Server Express Instance>>
  ```

  The *INSTANCENAME* parameter is needed whenever the feature is instance-specific.

- **UPGRADE** This parameter is used for upgrading from MSDE to SQL Server 2005 Express Edition. When you use *UPGRADE*, you must also specify the same instance name as the name of the MSDE instance you want to upgrade. This is because it is possible to have up to 16 MSDE instances on a single computer. Here is an example upgrade parameter:

```
UPGRADE=SQL_Engine INSTANCENAME=<<name of the MSDE instance>>
```

When you write custom installation applications, it can be difficult to remember these parameter names for all your projects. To make things easier, you can write a custom wrapper class to encapsulate setting the parameters and to provide a reusable stub for your custom applications. The wrapper class does not expose every option available, but it should give enough direction to suit your own custom installation needs.

Deploying Express Edition Applications Using a Wrapper

As a custom application developer, you have three options for including SQL Server Express Edition within your application:

- Install Express Edition and then install the custom application.
- Install the custom application and then install Express Edition.
- Create a wrapper that combines the two-step process into a single step.

Note An SQL Server Express Edition wrapper cannot be MSI–based because Windows Installer does not support multiple instantiation of the Windows Installer service.

The remainder of this section focuses on creating a wrapper for your custom application. In the example code (Listing 14-6), the wrapper is a simple class that exposes three public methods: *IsExpressInstalled*, *EnumSQLInstances*, and *InstallExpress*. Ideally, you would not need to know if Express Edition or any other instance of SQL Server is already installed on the local computer. This example includes this information in case you want to give the user the flexibility of selecting an existing instance of Express Edition to install your application against instead of always creating a new instance.

The first step is to create a simple class. This class will contain local variables of most of the command-line switches supported by the SQLEXP.exe installation executable. These switches will be exposed as properties of the class object.

Important The following code is only a guideline for installing SQL Server Express Edition with your custom application. It is not complete and does not contain robust error-handling routines.

Listing 14-6 *EmbeddedInstall* class definition

```
  public class EmbeddedInstall
       {
           #region Internal variables

           //Variables for setup.exe command line
           private string instanceName = "SQLEXPRESS";
           private string installSqlDir = "";
           private string installSqlSharedDir = "";
           private string installSqlDataDir = "";
           private string addLocal = "All";
           private bool sqlAutoStart = true;
           private bool sqlBrowserAutoStart = false;
           private string sqlBrowserAccount = "";
           private string sqlBrowserPassword = "";
           private string sqlAccount = "";
           private string sqlPassword = "";
           private bool sqlSecurityMode = false;
           private string saPassword = "";
           private string sqlCollation = "";
           private bool disableNetworkProtocols = true;
           private bool errorReporting = true;
           private string sqlExpressSetupFileLocation =
System.Environment.GetEnvironmentVariable("TEMP") + "\\sqlexpr.exe";

           #endregion
           #region Properties
           public string InstanceName
           {
               get
               {
                   return instanceName;
               }
               set
               {
                   instanceName = value;
               }
           }

           public string SetupFileLocation
           {
               get
               {
                   return sqlExpressSetupFileLocation;
               }
               set
               {
                   sqlExpressSetupFileLocation = value;
               }
           }

           public string SqlInstallSharedDirectory
           {
               get
```

```
        {
            return installSqlSharedDir;
        }
        set
        {
            installSqlSharedDir = value;
        }
    }
    public string SqlDataDirectory
    {
        get
        {
            return installSqlDataDir;
        }
        set
        {
            installSqlDataDir = value;
        }
    }
    public bool AutostartSQLService
    {
        get
        {
            return sqlAutoStart;
        }
        set
        {
            sqlAutoStart = value;
        }
    }
    public bool AutostartSQLBrowserService
    {
        get
        {
            return sqlBrowserAutoStart;
        }
        set
        {
            sqlBrowserAutoStart = value;
        }
    }
    public string SqlBrowserAccountName
    {
        get
        {
            return sqlBrowserAccount;
        }
        set
        {
            sqlBrowserAccount = value;
        }
    }
    public string SqlBrowserPassword
    {
        get
```

```
        {
            return sqlBrowserPassword;
        }
        set
        {
            sqlBrowserPassword = value;
        }
    }
    //Defaults to LocalSystem
    public string SqlServiceAccountName
    {
        get
        {
            return sqlAccount;
        }
        set
        {
            sqlAccount = value;
        }
    }
    public string SqlServicePassword
    {
        get
        {
            return sqlPassword;
        }
        set
        {
            sqlPassword = value;
        }
    }
    public bool UseSQLSecurityMode
    {
        get
        {
            return sqlSecurityMode;
        }
        set
        {
            sqlSecurityMode = value;
        }
    }
    public string SysadminPassword
    {
        set
        {
            saPassword = value;
        }
    }
    public string Collation
    {
        get
        {
            return sqlCollation;
        }
```

```
            set
            {
                sqlCollation = value;
            }
        }
        public bool DisableNetworkProtocols
        {
            get
            {
                return disableNetworkProtocols;
            }
            set
            {
                disableNetworkProtocols = value;
            }
        }
        public bool ReportErrors
        {
            get
            {
                return errorReporting;
            }
            set
            {
                errorReporting = value;
            }
        }
        public string SqlInstallDirectory
        {
            get
            {
                return installSqlDir;
            }
            set
            {
                installSqlDir = value;
            }
        }

        #endregion
```

Now that you have set up the local variables and properties for the class object, you can work on the public methods *IsExpressInstalled*, *EnumSQLInstances*, and *InstallExpress*.

Assuming a local server installation, you can simply look to the local registry to see if Express Edition or any other instance of SQL Server is installed. The method in Listing 14-7 enumerates and checks the *Edition* value for keys under this location, where *X* is an instance of SQL Server:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Microsoft SQL Server\MSSQL.X
```

Listing 14-7 *IsExpressInstalled* method

```
public bool IsExpressInstalled()
            {
                using (RegistryKey Key =
Registry.LocalMachine.OpenSubKey("Software\\Microsoft\\Microsoft SQL
Server\\", false))
                {
                    if (Key == null) return false;
                    string[] strNames;
                    strNames = Key.GetSubKeyNames();

                    //If we cannot find a SQL Server registry key, we
don't have SQL Server Express installed
                    if (strNames.Length == 0) return false;

                    foreach (string s in strNames)
                    {
                        if (s.StartsWith("MSSQL."))
                        {
                            //Check to see if the edition is "Express Edition"
                            using (RegistryKey KeyEdition =
Key.OpenSubKey(s.ToString() + "\\Setup\\", false))
                            {
                                if ((string)KeyEdition.GetValue("Edition") ==
"Express Edition")
                                {
                                    //If there is at least one instance of
SQL Server Express installed, return true
                                    return true;
                                }
                            }
                        }
                    }
                }
                return false;
            }
```

By using the local registry, you can determine more information about all the SQL Server instances, regardless of edition, that are installed on the local server. Having this information is useful if you want to provide a better installation experience. The method in Listing 14-8 takes a reference and populates a string array for instances, editions, and versions. It returns the number of instances of SQL Server that are installed on the local computer.

Listing 14-8 *EnumSQLInstances* method

```
public int EnumSQLInstances(ref string[] strInstanceArray, ref string[] strEditionArray,
 ref string[] strVersionArray)
            {
                using (RegistryKey Key = Registry.LocalMachine.OpenSubKey("Software\\
Microsoft\\Microsoft SQL Server\\", false))
                {
```

```
                    if (Key == null) return 0;
                    string[] strNames;
                    strNames = Key.GetSubKeyNames();

                    //If we can not find a SQL Server registry key, we return 0 for none
                    if (strNames.Length == 0) return 0;

                    //How many instances do we have?
                    int iNumberOfInstances = 0;

                    foreach (string s in strNames)
                    {
                        if (s.StartsWith("MSSQL."))
                            iNumberOfInstances++;
                    }

                    //Reallocate the string arrays to the new number of instances
                    strInstanceArray = new string[iNumberOfInstances];
                    strVersionArray = new string[iNumberOfInstances];
                    strEditionArray = new string[iNumberOfInstances];
                    int iCounter = 0;

                    foreach (string s in strNames)
                    {
                        if (s.StartsWith("MSSQL."))
                        {
                            //Get Instance name
                            using (RegistryKey KeyInstanceName =
Key.OpenSubKey(s.ToString(), false))
                            {
                                strInstanceArray[iCounter] =
(string)KeyInstanceName.GetValue("");
                            }

                            //Get Edition
                            using (RegistryKey KeySetup =
Key.OpenSubKey(s.ToString() + "\\Setup\\", false))
                            {
                                strEditionArray[iCounter] =
(string)KeySetup.GetValue("Edition");
                                strVersionArray[iCounter] =
(string)KeySetup.GetValue("Version");
                            }

                            iCounter++;
                        }
                    }
                    return iCounter;
                }
            }
```

Now you can install SQL Server Express Edition. First convert the properties of the class into a command-line argument that can be passed to the SQLEXPR.exe installation application. The *BuildCommandLine* method performs this task, as shown in Listing 14-9.

Listing 14-9 *BuildCommandLine* method

```
private string BuildCommandLine()
            {
                StringBuilder strCommandLine = new StringBuilder();

                if (!string.IsNullOrEmpty(installSqlDir))
                {
                    strCommandLine.Append("INSTALLSQLDIR=\"").Append(installSqlDir)
                      .Append("\"");
                }

                if (!string.IsNullOrEmpty(installSqlSharedDir))
                {
                    strCommandLine.Append("INSTALLSQLSHAREDDIR=\"")
                      .Append(installSqlSharedDir).Append("\"");
                }

                if (!string.IsNullOrEmpty(installSqlDataDir))
                {
                    strCommandLine.Append("INSTALLSQLDATADIR=\"")
                      .Append(installSqlDataDir).Append("\"");
                }

                if (!string.IsNullOrEmpty(addLocal))
                {
                    strCommandLine.Append(" ADDLOCAL=\"").Append(addLocal).Append("\"");
                }

                if (sqlAutoStart)
                {
                    strCommandLine.Append(" SQLAUTOSTART=1");
                }
                else
                {
                    strCommandLine.Append(" SQLAUTOSTART=0");
                }

                if (sqlBrowserAutoStart)
                {
                    strCommandLine.Append(" SQLBROWSERAUTOSTART=1");
                }
                else
                {
                    strCommandLine.Append(" SQLBROWSERAUTOSTART=0");
                }

                if (!string.IsNullOrEmpty(sqlBrowserAccount))
                {
                    strCommandLine.Append("SQLBROWSERACCOUNT=\"")
                      .Append(sqlBrowserAccount).Append("\"");
                }

                if (!string.IsNullOrEmpty(sqlBrowserPassword))
```

```
            {
                strCommandLine.Append("SQLBROWSERPASSWORD=\"")
                  .Append(sqlBrowserPassword).Append("\"");
            }

            if (!string.IsNullOrEmpty(sqlAccount))
            {
                strCommandLine.Append(" SQLACCOUNT=\"").Append(sqlAccount)
                  .Append("\"");
            }

            if (!string.IsNullOrEmpty(sqlPassword))
            {
                strCommandLine.Append(" SQLPASSWORD=\"").Append(sqlPassword)
                  .Append("\"");
            }

            if (sqlSecurityMode == true)
            {
                strCommandLine.Append(" SECURITYMODE=SQL");
            }

            if (!string.IsNullOrEmpty(saPassword))
            {
                strCommandLine.Append(" SAPWD=\"").Append(saPassword).Append("\"");
            }

            if (!string.IsNullOrEmpty(sqlCollation))
            {
                strCommandLine.Append(" SQLCOLLATION=\"").Append(sqlCollation)
                  .Append("\"");
            }

            if (disableNetworkProtocols == true)
            {
                strCommandLine.Append(" DISABLENETWORKPROTOCOLS=1");
            }
            else
            {
                strCommandLine.Append(" DISABLENETWORKPROTOCOLS=0");
            }

            if (errorReporting == true)
            {
                strCommandLine.Append(" ERRORREPORTING=1");
            }
            else
            {
                strCommandLine.Append(" ERRORREPORTING=0");
            }

            return strCommandLine.ToString();
        }
```

Now you can create the *InstallExpress* method, as shown in Listing 14-10.

Listing 14-10 *InstallExpress* method

```
       public bool InstallExpress()
            {

                //In both cases, we run Setup because we have the file.
                Process myProcess = new Process();
                myProcess.StartInfo.FileName = sqlExpressSetupFileLocation;
                myProcess.StartInfo.Arguments = "/qb " + BuildCommandLine();
                /*      /qn -- Specifies that setup run with no user interface.
                        /qb -- Specifies that setup show only the basic
user interface. Only dialog boxes displaying progress information are
displayed. Other dialog boxes, such as the dialog box that asks users if
they want to restart at the end of the setup process, are not displayed.
                */
                myProcess.StartInfo.UseShellExecute = false;

                return myProcess.Start();

            }
```

Finally, we can create the sample application that calls the wrapper class and installs Express Edition, as shown in Listing 14-11.

Listing 14-11 Main application

```
class Program
    {
        static void Main(string[] args)
        {
            EmbeddedInstall EI = new EmbeddedInstall();

            if (args.Length > 0)
            {
                int i = 0;
                while (i < args.Length)
                {
                    if ((string)args[i].ToUpper() == "-V")
                    {
                        string[] strInstanceArray = new string[0];
                        string[] strVersionArray = new string[0];
                        string[] strEditionArray = new string[0];

                        int iInstances = EI.EnumSQLInstances(ref strInstanceArray,
                          ref strEditionArray, ref strVersionArray);
                        if (iInstances > 0)
                        {
                            for (int j = 0; j <= iInstances - 1; j++)
                            {
                                Console.WriteLine("SQL Server Instance:
                                  \"" + strInstanceArray[j].ToString() + "\" -- " +
```

```
strEditionArray[j].ToString() + "   (" + strVersionArray[j].ToString() +
")");
                            }

                        }
                        else
                        {
                            Console.WriteLine("No instance of SQL Server
                              Express found on local server.\n\n");
                        }

                        return;
                    }

                    if ((string)args[i].ToUpper() == "-I")
                    {
                        if (EI.IsExpressInstalled())
                        {
                            Console.WriteLine("An instance of SQL Server
                              Express is installed.\n\n");
                        }
                        else
                        {
                            Console.WriteLine("There are no SQL Server
                              Express instances installed.\n\n");
                        }

                        return;
                    }

                    i++;
                }
            }

            Console.WriteLine("\nInstalling SQL Server 2005 Express Edition\n");

            EI.AutostartSQLBrowserService = false;
            EI.AutostartSQLService = true;
            EI.Collation = "SQL_Latin1_General_Cp1_CS_AS";
            EI.DisableNetworkProtocols = false;
            EI.InstanceName = "SQLEXPRESS";
            EI.ReportErrors = true;
            EI.SetupFileLocation = "C:\\Downloads\\sqlexpr.exe";
              //Provide location for the Express setup file
            EI.SqlBrowserAccountName = ""; //Blank means LocalSystem
            EI.SqlBrowserPassword = ""; // N/A
            EI.SqlDataDirectory = "C:\\Program Files\\Microsoft SQL Server\\";
            EI.SqlInstallDirectory = "C:\\Program Files\\";
            EI.SqlInstallSharedDirectory = "C:\\Program Files\\";
            EI.SqlServiceAccountName = ""; //Blank means Localsystem
            EI.SqlServicePassword = ""; // N/A
            EI.SysadminPassword = "ThIsIsALoNgPaSsWoRd1234!!"; //<<Supply
              a secure sysadmin password>>
            EI.UseSQLSecurityMode = true;
```

```
            EI.InstallExpress();

            Console.WriteLine("\nInstalling custom application\n");


            //
If you need to run another MSI install, remove the following comment lines
            //and fill in information about your MSI

            /*Process myProcess = new Process();
            myProcess.StartInfo.FileName = "";//
<<Insert the path to your MSI file here>>
            myProcess.StartInfo.Arguments = ""; //
<<Insert any command line parameters here>>
            myProcess.StartInfo.UseShellExecute = false;
            myProcess.Start();*/

        }
```

Deploying Express Edition Applications Using ClickOnce

ClickOnce is a new feature that is part of the .NET Framework 2.0. ClickOnce lets you deploy Windows-based client applications to a computer by placing the application files on a Web or file server that is accessible to the client, and then providing the user with a link. This lets users download and run applications from centrally managed servers without requiring administrator privileges on the client machine.

In this section, we will illustrate the ClickOnce/SQL Server Express Edition experience by developing a simple Windows Forms application. This application uses the AdventureWorks sample database, which can be downloaded from *http://go.microsoft.com/fwlink/?linkid=65209*.

This example demonstrates how to create a single WinForm for viewing the departments in the HumanResources.Department table in the AdventureWorks database.

To create a Windows Form that displays the Department table:

1. Launch Visual Studio.
2. Create a new Windows Application project.
3. When the Form1 Designer opens, add a reference to the AdventureWorks database.
4. Right-click the Project node in the Solution Explorer pane, and then select both Add and Existing Item. Navigate to the AdventureWorks database and click OK.
5. In the Data Source Configuration Wizard, under the Tables node, select the Department table, and then continue with the wizard.
6. When the wizard finishes, you will notice the AdventureWorks.mdf database icon in the Solution Explorer pane and a new AdventureWorks connection in Database Explorer.

Database Explorer lets you perform database operations such as creating new tables, querying and modifying existing data, and other database development functions.

7. Add the *DataGridView* control to the WinForm. This grid control is located in the toolbox. When you drag the grid control onto the design surface, you have the option of selecting the AdventureWorks dataset that you created when you ran the Data Source Configuration Wizard. This dialog box is shown in Figure 14-16.

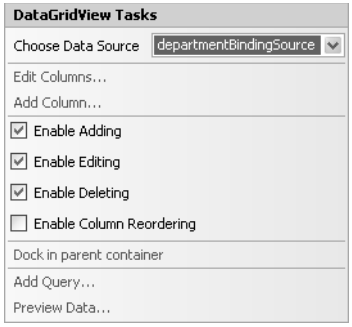

Figure 14-16 *DataGridView* task properties

When a data source is configured, you should be able to run the application and have the grid control display the values for the Department table, as shown in Figure 14-17.

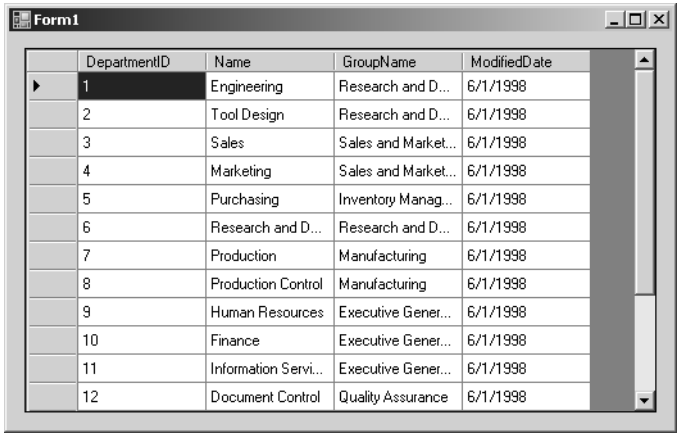

Figure 14-17 Department table enumerated using the *DataGridView* control

You can now deploy this application using ClickOnce.

To deploy the application by using ClickOnce:

1. To publish the application, select Publish from the Build menu. The Publish Wizard opens, as shown in Figure 14-18.

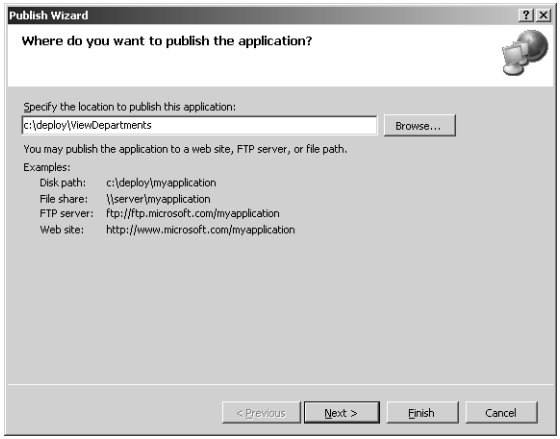

Figure 14-18 The Publish Wizard's Where Do You Want To Publish The Application? page

2. On the first page of the wizard, you specify where the compiled bits should physically be placed. In the Specify The Location To Publish This Application box, enter **C:\deploy\ViewDepartments**. Click Next.

3. On the next page of the wizard (Figure 14-19), you specify the location from which users will install the application. Select From A CD-ROM Or DVD-ROM. Click Next.

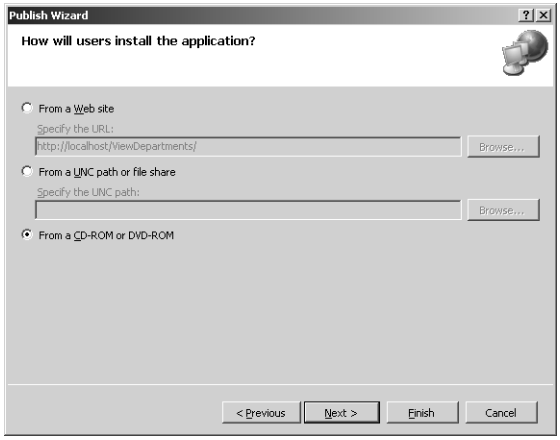

Figure 14-19 The Publish Wizard's How Will Users Install The Application? page

4. On the next page of the wizard (Figure 14-20), you specify whether the application will check for updates.

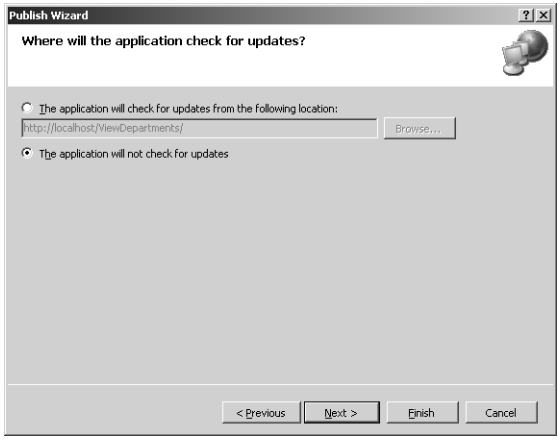

Figure 14-20 The Publish Wizard's Where Will The Application Check For Updates? page

5. ClickOnce gives applications the ability to look for updates at certain times, such as when the application starts or whenever the application developer chooses to call the appropriate update APIs. (There are some issues when you use this feature with a database, which we will discuss later in this chapter.) For this example, select The Application Will Not Check For Updates. Click Next.

6. The last page of the wizard (Figure 14-21) displays summary information and notifies you that, because you are writing to a CD or DVD-ROM, Setup will install a shortcut and entry in Add Or Remove Programs for your application. Click Finish.

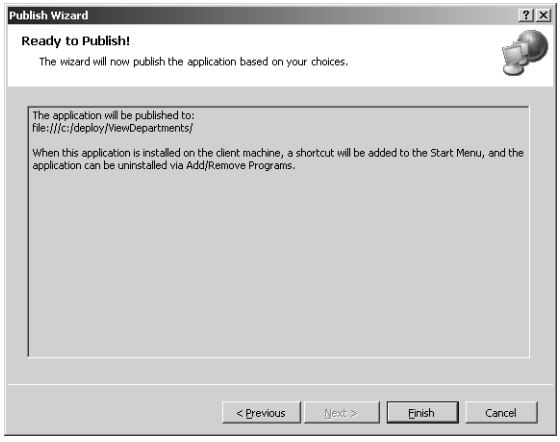

Figure 14-21 The last page of the Publish Wizard

You can write an application that will live on the application server only and will never be installed on the client machine. Regardless, ClickOnce will prompt the user to install any missing prerequisites, such as the .NET Framework 2.0 or SQL Server Express Edition, as shown in Figure 14-22.

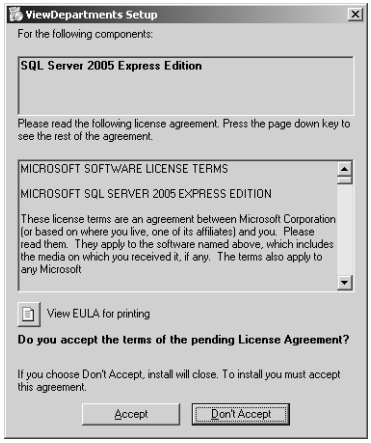

Figure 14-22 Prerequisites not installed when user launches application

Note Whether the application itself is designed to be run on demand from an application server or to be installed locally, SQL Server Express Edition is always installed on the local machine if the custom application requires it.

When the Publish Wizard finishes, new files are placed in the deployment directory. These files include the compressed data files and the setup installer application. You might want to copy these files to a CD and distribute them to your users, to provide them with the necessary information about applications that use SQL Server Express Edition.

Important A user who is not an administrator on the local machine cannot install the .NET Framework or SQL Server Express Edition. In this case, system administrators should deploy these components first. They can do this manually or by using a distributed software management system such as Microsoft Systems Management Server.

Updating ClickOnce Deployments That Use Express Edition

Let's say the user has successfully installed your application. The user had all necessary prerequisites installed, and the application is running successfully.

The user has entered data into version 1.0 of the database and now the developers have come out with version 2.0. This new version has an additional column named Location in the Departments table. This new column stores the geographical location of the department. When the developer deploys version 2.0, the new version of the database is pushed down to the client, and the previous version is automatically moved to a separate folder named Pre. The developer must now write a database migration script to move all the data from the 1.0 version in the Pre folder to the new database. Because Visual Studio does not have any tools to support this migration, it is completely up to the developer to perform the migration.

Otherwise, none of the data that was entered in version 1.0 will be accessible to the application. Additionally, if the developer publishes an interim version (for example, 2.1) to reconcile this migration problem, or if the developer accesses the .mdf file by simply viewing the structure in Server Explorer, ClickOnce will see that the date and time stamp has changed and deploy version 2.1 of the database. This moves version 2.0 of the database to the Pre folder and deletes version 1.0 of the database. This results in complete data loss and a poor customer experience.

To avoid this, Visual Studio should not include the database files when the application is deployed. Instead, it should provide installation scripts to create the database. Also, when you perform a ClickOnce update, you must write and call a separate update script. The ViewDepartments example from the previous section is used in this section to help clarify the workaround solution.

ViewDepartments is a single WinForm application that connects to the AdventureWorks database and enumerates the Departments table. When you developed this application, you pointed Visual Studio to the AdventureWorks .mdf file, which created a new data source. As the application functions now, if you were to use ClickOnce to deploy the application, the application would always include the AdventureWorks .mdf file and cause the overwrite problems mentioned previously.

To avoid unwanted data loss in your application:

1. In the Solution Explorer pane, click the AdventureWorks database icon, as shown in Figure 14-23. In the Properties pane, select Do Not Copy for the Copy To Output Directory property.

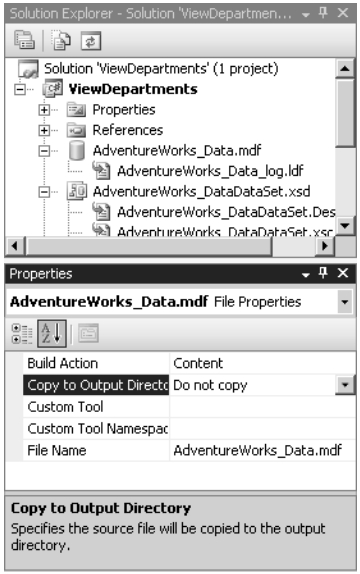

Figure 14-23 Copy To Output Directory property for the AdventureWorks database

2. Choose Properties from the Project menu to open the Project Properties panel. On the Publish tab, click Application Files. This launches a dialog box that contains a list of all the files in the solution. Change the Publish Status to Exclude for the .MDF and .LDF files of the AdventureWorks database, as shown in Figure 14-24.

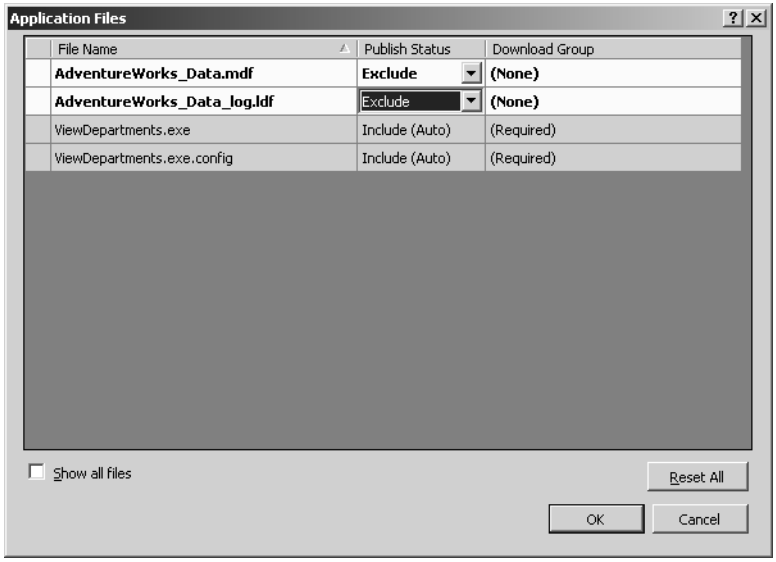

Figure 14-24 Excluding database files

Next you script the creation of the AdventureWorks database. You can script a database in many ways. In SQL Server Management Studio, you can right-click the database in Object Explorer and create the entire script there. Or you can use the Generate SQL Server Scripts Wizard for more scripting options. If you do not have a license for this tool or any other scripting tool, you can easily create a small program that uses the SQL Server Management Objects (SMO) object model to create a script by using the *Scripter* class.

Note If you installed SQL Server Express Edition with the developer components, the SMO DLLs are located by default in C:\Program Files\Microsoft SQL Server\90\SDK\Assemblies.

Listing 14-12 shows a modified AdventureWorks creation script that creates and populates the Departments table.

Listing 14-12 AdventureWorks creation script

```
USE [master]
GO
CREATE DATABASE [AdventureWorks] ON  PRIMARY
( NAME = N'AdventureWorks_Data', FILENAME = N'C:\Program Files\Microsoft
 SQL Server\MSSQL.1\MSSQL\Data\AdventureWorks_Data.mdf' , SIZE = 167936KB
, MAXSIZE = UNLIMITED, FILEGROWTH = 16384KB )
```

```
 LOG ON
( NAME = N'AdventureWorks_Log', FILENAME = N'C:\Program Files\Microsoft
SQL Server\MSSQL.1\MSSQL\Data\AdventureWorks_Log.ldf' , SIZE = 2048KB ,
MAXSIZE = 2048GB , FILEGROWTH = 16384KB )
 COLLATE SQL_Latin1_General_CP1_CI_AS
GO
EXEC dbo.sp_dbcmptlevel @dbname=N'AdventureWorks', @new_cmptlevel=90
GO
USE [AdventureWorks]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TYPE [dbo].[Name] FROM [nvarchar](50) NULL
GO
EXEC sys.sp_executesql N'CREATE SCHEMA [HumanResources] AUTHORIZATION [dbo]'
GO
CREATE TABLE [HumanResources].[Department](
   [DepartmentID] [smallint] IDENTITY(1,1) NOT NULL,
   [Name] [dbo].[Name] NOT NULL,
   [GroupName] [dbo].[Name] NOT NULL,
   [ModifiedDate] [datetime] NOT NULL CONSTRAINT
[DF_Department_ModifiedDate]  DEFAULT (getdate()),
 CONSTRAINT [PK_Department_DepartmentID] PRIMARY KEY CLUSTERED
(
   [DepartmentID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]
GO
insert into [HumanResources].[Department](Name,Groupname)
values('Engineering','Research and Development')
GO
insert into [HumanResources].[Department](Name,Groupname) values('Tool
Design','Research and Development')
GO
insert into [HumanResources].[Department](Name,Groupname)
values('Sales','Sales and Marketing')
GO
insert into [HumanResources].[Department](Name,Groupname)
values('Marketing','Sales and Marketing')
GO
insert into [HumanResources].[Department](Name,Groupname)
values('Purchasing','Inventory Management')
GO
insert into [HumanResources].[Department](Name,Groupname) values('Research
and Development','Research and Development')
GO
insert into [HumanResources].[Department](Name,Groupname)
values('Production','Manufacturing')
GO
insert into [HumanResources].[Department](Name,Groupname)
values('Production Control','Manufacturing')
```

```
GO
insert into [HumanResources].[Department](Name,Groupname) values('Human
Resources','Executive General and Administration')
GO
insert into [HumanResources].[Department](Name,Groupname)
values('Finance','Executive General and Administration')
GO
insert into [HumanResources].[Department](Name,Groupname)
values('Information Services','Executive General and Administration')
GO
insert into [HumanResources].[Department](Name,Groupname) values('Document
 Control','Quality Assurance')
GO
insert into [HumanResources].[Department](Name,Groupname) values('Quality
Assurance','Quality Assurance')
GO
insert into [HumanResources].[Department](Name,Groupname)
values('Facilities and Maintenance','Executive General and
Administration')
GO
insert into [HumanResources].[Department](Name,Groupname) values('Shipping
and Receiving','Inventory Management')
GO
insert into [HumanResources].[Department](Name,Groupname)
values('Executive','Executive General and Administration')
GO
--This next table is used to identify the version of the database
CREATE TABLE AdventureWorks..AppInfo
(Property nvarchar(255) NOT NULL,
Value nvarchar(255))
GO
INSERT INTO AdventureWorks..AppInfo Values('Version','1.0.0.0')
GO
```

Because the actual .mdf file is not included in this solution, you must define and synchronize versions of the database that the application is connected to. An easy workaround is to add the AppInfo table to the AdventureWorks database. When you start the application, it should first check to see if the versions match. If they don't, the application should either run an upgrade script or fail. This is explained in more detail next.

To implement a version check, you add a resource file to your project and then store the script as an embedded resource within the application:

1. Right-click the project in the Solution Explorer pane and select Add, and then select New Item. Select Resource File, and then click Add. This launches the Resource File document window shown in Figure 14-25.
2. You can add the SQL scripts as separate strings or as text files; for simplicity, we'll store the Create and Update scripts as separate files within this resource. From the Add Resource drop-down menu, select Add Existing File. Locate the creation script we produced earlier and add this file.

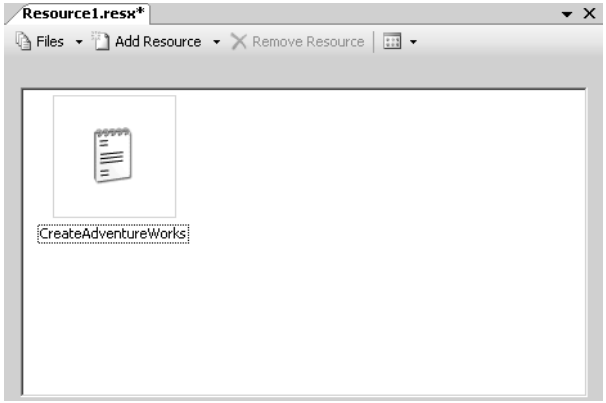

Figure 14-25 Resource document window showing our creation script

3. Next we'll create an upgrade script for the AdventureWorks database. Although you might not have to upgrade your application right away, you should also include the upgrade script to upgrade your database to version 1.0.0.3.

```
USE [AdventureWorks]
GO
ALTER TABLE [HumanResources].[Department]
ADD Location char(2)
GO
UPDATE AdventureWorks..AppInfo set Value='1.0.0.3' where
Property='Version'
GO
```

4. Save this script as UpgradeAdventureWorks.sql. Add it to the resource file, as described earlier.
5. Modify the application to check versions and run any necessary scripts.

Note In the previous example, the *Form_Load* method contains code that was autogenerated when we assigned the AdventureWorks dataset via the UI: *this.departmentTableAdapter.Fill(this.adventureWorks_DataDataSet.Department);* You should remove or comment out this code because you want to perform the database version check first.

You should also set the *DataSource* property (which was pre-populated in the grid control when you used the UI to bind the grid to the data source) to *None* (Figure 14-26).

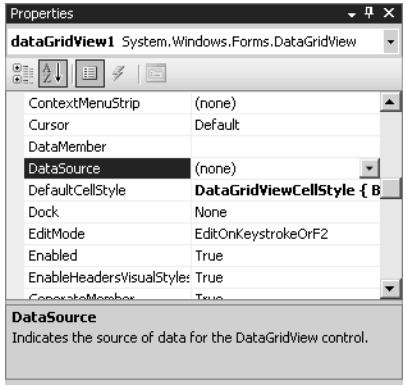

Figure 14-26 *DataSource* property autogenerated by Visual Studio

Listing 14-13 shows the complete code for the *Form1* class:

Listing 14-13 Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Data.SqlClient;
using System.Text.RegularExpressions;


namespace ViewDepartments
{

    public partial class Form1 : Form
    {

        enum VersionCheck { Failed = 0, Equal, DatabaseIsMoreNew,
          DatabaseIsOlder, DatabaseNotFound };

        private SqlConnection sqlCon = new SqlConnection();
        private SqlCommand sqlCmd = new SqlCommand();

        public Form1()
        {
            InitializeComponent();

            if (SetupDatabase() == false)
            {
                return;
            }

            PopulateGrid();
        }
```

```
        public bool SetupDatabase()
        {
            bool bContinue = false;

            //Create a connection to SQL Server
            try
            {
                sqlCon.ConnectionString = "Server=
                  .\\sqlexpress;Integrated Security=true";
                sqlCon.Open();
            }
            catch (SqlException sql_ex)
            {
                MessageBox.Show("Fail to connect to SQL Server Express\n"
+ sql_ex.Number.ToString() + " " + sql_ex.Message.ToString());
                return bContinue;
            }

            //Now that you are connected to Express, check the database versions

            switch (CheckVersion())
            {
                case (int)VersionCheck.Equal:
                    {
                        bContinue = true;
                        break;
                    }
                case (int)VersionCheck.Failed:
                    {
                        bContinue = false;
                        break;
                    }
                case (int)VersionCheck.DatabaseIsOlder:
                    {
                        //Run the upgrade script
                        bContinue = RunScript(Resource1.UpdateAdventureWorks.ToString());
                        break;
                    }
                case (int)VersionCheck.DatabaseIsMoreNew:
                    {
                        bContinue = false;
                        break;
                    }
                case (int)VersionCheck.DatabaseNotFound:
                    {
                        //Run the creation script
                        bContinue = RunScript(Resource1.CreateAdventureWorks.ToString());
                        break;
                    }
                default:
                    {
                        bContinue = false;
                        break;
```

```
                }

            }


            return bContinue;


        }
        public bool RunScript(string strFile)
        {
            string[] strCommands;
            strCommands = ParseScriptToCommands(strFile);
            try
            {
                if (sqlCon.State != ConnectionState.Open) sqlCon.Open();

                sqlCmd.Connection = sqlCon;

                foreach (string strCmd in strCommands)
                {
                    if (strCmd.Length > 0)
                    {
                        sqlCmd.CommandText = strCmd;
                        sqlCmd.ExecuteNonQuery();
                    }
                }
            }
            catch (SqlException sql_ex)
            {
                MessageBox.Show(sql_ex.Number.ToString() + " " +
                  sql_ex.Message.ToString());
                return false;
            }

            return true;
        }
        public int CheckVersion()
        {
            //Get Version information from application
            Version v=new Version(Application.ProductVersion.ToString());

            try
            {

                string strResult;

                //Verify that the AdventureWorks Database exists
                sqlCmd = new SqlCommand("select count(*) from
                  master..sysdatabases where name='AdventureWorks'",sqlCon);
                strResult = sqlCmd.ExecuteScalar().ToString();
                if (strResult == "0")
                {
                    sqlCon.Close();
                    return (int)VersionCheck.DatabaseNotFound;
```

```
                }

                sqlCmd = new SqlCommand("SELECT value from
  AdventureWorks..AppInfo where property='version'", sqlCon);
                strResult=(string)sqlCmd.ExecuteScalar();

                Version vDb = new Version(strResult);

                sqlCon.Close();

                if (vDb == v)
                    return (int)VersionCheck.Equal;

                if (vDb > v)
                    return (int)VersionCheck.DatabaseIsMoreNew;

                if (vDb < v)
                    return (int)VersionCheck.DatabaseIsOlder;

            }
            catch (SqlException sql_ex)
            {
                MessageBox.Show(sql_ex.Number.ToString() + " " +
                  sql_ex.Message.ToString());
                return (int)VersionCheck.Failed;
            }
            catch (Exception system_ex)
            {
                MessageBox.Show(system_ex.Message.ToString());
                return (int)VersionCheck.Failed;
            }

            return (int)VersionCheck.Failed;

        }

        public string[] ParseScriptToCommands(string strScript)
        {
            string[] commands;
            commands = Regex.Split(strScript, "GO\r\n", RegexOptions.IgnoreCase);
            return commands;
        }

        public void PopulateGrid()
        {

            String strCmd = "Select * from [AdventureWorks].[HumanResources].[Department
]";
            SqlDataAdapter da;

            da = new SqlDataAdapter(strCmd, sqlCon);
            DataSet ds = new DataSet();
            da.Fill(ds, "Departments");
            dataGridView1.DataSource = ds;
            dataGridView1.DataMember = "Departments";
```

```
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // TODO: This line of code loads data into the
'adventureWorks_DataDataSet.Department' table. You can move or remove this
line as necessary.
            //this.departmentTableAdapter.Fill(this.adventureWorks_DataDataSet.
Department);
        }
    }
}
```

In the code for Form1.cs, a call is made to *SetDatabase()*. This function first attempts to make a connection to SQL Server Express Edition. When that call succeeds, it calls into the *CheckVersion()* method, which checks to see if the AdventureWorks database exists. If it does, the method obtains the version number from the AppInfo table. If the AdventureWorks database does not exist, the creation script located in the resource file is executed. If the database version is earlier than the application version, the upgrade script is run.

Note The version that is compared against the database comes from the *File Version* property of the project. This property can be set within the Assembly Information dialog box (which is accessible from the Application tab in Project Properties).

When you first execute this application against a blank SQL Server Express Edition database, it creates the AdventureWorks database, and you see the four columns of the Departments table. The next time you execute this application, it will be upgraded to include another column in the table named Location.

Summary

SQL Server Express Edition uses the same database engine as all the other editions of SQL Server, but with memory, disk, and some feature restrictions. Even with these restrictions, it is possible to develop and deploy a variety of Express Edition applications.

SQL Server Express Edition with Advanced Services, which was released simultaneously with Service Pack 1, allows users to develop reports and issue full-text queries against Express Edition databases. This upgrade comes with a stripped-down version of SQL Server Management Studio. The Advanced Services Toolkit also includes a graphical report designer used to create reports for Report Server. With all these features and functionality, SQL Server Express Edition provides users with a powerful relational database engine—for free.