

# Pro SQL Server 2005



Thomas Rizzo, Adam Machanic,  
Julian Skinner, Louis Davidson,  
Robin Dewson, Jan Narkiewicz,  
Joseph Sack, Rob Walters

## **Pro SQL Server 2005**

**Copyright © 2006 by Thomas Rizzo, Adam Machanic, Julian Skinner, Louis Davidson, Robin Dewson, Jan Narkiewicz, Joseph Sack, Rob Walters**

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-477-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Tony Davis

Technical Reviewers: Sajal Dam, Cristian Lefter, Alejandro Leguizamo, Alexzander Nepomnjashiy, Andrew Watt, Richard Waymire, Joe Webb, Roger Wolter

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser

Project Manager: Kylie Johnston

Copy Edit Manager: Nicole LeClerc

Copy Editors: Ami Knox, Nicole LeClerc

Assistant Production Director: Kari Brooks-Copony

Production Editor: Kelly Winquist

Compositor: Susan Glinert

Proofreaders: Kim Burton, Linda Marousek, Linda Seifert, Liz Welch

Indexer: Broccoli Information Management

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail [orders-ny@springer-sbm.com](mailto:orders-ny@springer-sbm.com), or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail [info@apress.com](mailto:info@apress.com), or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Source Code section.



# .NET Integration

**T**ruly devoted (or is it insane?) SQL Server programmers might think back wistfully on days spent debugging extended stored procedures, yearning for those joyfully complicated times. The rest of us, however, remember plunging headfirst into a process that always felt a lot more esoteric than it needed to be and never quite lived up to the functionality we hoped it could provide.

SQL Server 7.0 introduced the idea of *extended stored procedures* (XPs), which are DLLs—usually written in C++—that can be used to programmatically extend SQL Server’s functionality. Programming and debugging these is unfortunately quite difficult for most users, and their use gives rise to many issues, such as memory leaks and security concerns, that make them less than desirable. Luckily, extended stored procedures are a thing of the past (or are deprecated, at the very least), and SQL Server 2005 gives programmers much better options with tightly integrated common language runtime (CLR) interoperability. Developers can now use any .NET language they feel comfortable with to create powerful user-defined objects within SQL Server; note, however, that only C#, VB .NET, and Managed C++ are officially supported languages. Although other languages can be used, getting them to work properly may require a bit of additional effort.

In this chapter, programming with CLR objects will be introduced with a step-by-step tour through development of a CLR stored procedure. Also discussed will be the .NET object model provided for SQL Server CLR development, best practices for developing CLR objects, and various deployment issues.

Chapter 6 builds upon this foundation, covering all of the other types of objects that can be created in SQL Server using .NET: CLR user-defined types, CLR user-defined functions, CLR aggregates, and CLR triggers.

Please note that both this chapter and Chapter 6 assume familiarity with .NET programming using the C# language. Those readers who haven’t yet picked up .NET skills should consider starting with Andrew Troelsen’s excellent book, *C# and the .NET Platform, Second Edition* (Apress, 2003).

## Introduction to SQL Server .NET Integration

SQL Server developers have had few choices in the past when it came to doing things in the database that Transact-SQL (T-SQL) wasn’t especially well suited for. This includes such things as complex or heavily mathematical logic, connecting to remote services or data stores, and manipulating files and other non-SQL Server–controlled resources. Although many of these tasks are best suited for operation on the client rather than within SQL Server, sometimes system architecture, project funding, or time constraints leave developers with no choice—business problems must be solved in some way, as quickly and cheaply as possible. Extended stored procedures were one option to help with these situations, but as mentioned previously, these were difficult to write and debug, and were known for decreasing server stability. Another option was to use the `sp_OA` (Object Automation) stored procedures to call COM objects, but this had its own issues, including performance penalties and dealing with COM “DLL hell” if the correct versions weren’t registered on the SQL Server.

CLR integration does away with these issues—and provides a structured, easy-to-use methodology for extending SQL Server in a variety of ways.

## Why Does SQL Server 2005 Host the CLR?

There are some things that T-SQL just isn't meant to do. For instance, it's not known as a language that excels at accessing data from web services. Although there are some ways that this can be done, T-SQL isn't something we developers would think of as the first choice for this operation. Another good example is data structures; in T-SQL, there is only one data structure: tables. This works fine for most of our data needs, but sometimes something else is needed—an array or a linked list, for instance. And although these things can be simulated using T-SQL, it's messy at best.

The common language runtime is a managed environment, designed with safety and stability in mind. Management means that memory and resources are automatically handled by the runtime—it is very difficult (if not impossible) to write code that will cause a memory leak. Management also means that SQL Server can control the runtime if something goes wrong. If SQL Server detects instability, the hosted runtime can be immediately restarted.

This level of control was impossible with the extended stored procedure functionality that was present in earlier versions of SQL Server. Extended stored procedures were often known for decreasing the stability of SQL Server, as there was no access control—an unwitting developer could all too easily write code that could overwrite some of SQL Server's own memory locations, thereby creating a time bomb that would explode when SQL Server needed to access the memory. Thanks to the CLR's "sandboxing" of process space, this is no longer an issue.

The CLR builds virtual process spaces within its environment, called *application domains*. This lets code running within each domain operate as if it has its own dedicated process, and at the same time isolates virtual processes from each other. The net effect in terms of stability is that if code running within one application domain crashes, the other domains won't be affected—only the domain in which the crash occurred will be restarted by the framework and the entire system won't be compromised. This is especially important in database applications; developers certainly don't want to risk crashing an entire instance of SQL Server because of a bug in a CLR routine.

## When to Use CLR Routines

T-SQL is a language that was designed primarily for straightforward data access. Developers are often not comfortable writing complex set-based solutions to problems, and end up using cursors to solve complex logical problems. This is never the best solution in T-SQL. Cursors and row-by-row processing aren't the optimal data access methods. Set-based solutions are preferred.

When non-set-based solutions are absolutely necessary, CLR routines are faster. Looping over a `SqlDataReader` can be much faster than using a cursor. And complex logic will often perform much better in .NET than in T-SQL. In addition, if routines need to access external resources such as web services, using .NET is an obvious choice. T-SQL is simply not adept at handling these kinds of situations.

## When Not to Use CLR Routines

It's important to remember an adage that has become increasingly popular in the fad-ridden world of IT in the past few years: "To a hammer, everything looks like a nail."

Just because you can do something using the CLR, doesn't mean you should. For data access, set-based T-SQL is still the appropriate choice in virtually all cases. Access to external resources from SQL Server, which CLR integration makes much easier, is generally not appropriate from SQL Server's process space. Think carefully about architecture before implementing such solutions. External

resources can be unpredictable or unavailable—two factors that aren't supposed to be present in database solutions!

In the end, it's a question of common sense. If something doesn't seem to belong in SQL Server, it probably shouldn't be implemented there. As CLR integration matures, best practices will become more obvious—but for the meantime, take a minimalist approach. Overuse of the technology will cause more problems in the long run than underuse.

## How SQL Server Hosts .NET: An Architectural Overview

The CLR is completely hosted by SQL Server. Routines running within SQL Server's process space make requests to SQL Server for all resources, including memory and processor time. SQL Server is free to either grant or deny these requests, depending on server conditions. SQL Server is also free to completely restart the hosted CLR if a process is taking up too many resources. SQL Server itself is in complete control, and the CLR is unable to compromise the basic integrity that SQL Server offers.

### Why Managed Objects Perform Well

SQL Server 2005 CLR integration was designed with performance in mind. Compilation of CLR routines for hosting within SQL Server is done using function pointers in order to facilitate high-speed transitions between T-SQL and CLR processes. Type-specific optimizations ensure that once routines are just-in-time compiled (JITted), no further cost is associated with their invocation.

Another optimization is streaming of result sets from CLR *table-valued functions* (which will be covered in detail in the next chapter). Unlike some other rowset-based providers that require the client to accept the entire result set before work can be done, table-valued functions are able to stream data a single row at a time. This enables work to be handled in a piecemeal fashion, thereby reducing both memory and processor overhead.

### Why CLR Integration Is Stable

SQL Server both hosts and completely controls the CLR routines running within the SQL Server process space. Since SQL Server is in control of all resources, routines are unable to bog down the server or access unavailable resources, like XPs were able to.

Another important factor is the `HostProtection` attribute, a new feature in the .NET 2.0 Base Class Library. This attribute allows methods to define their level of cross-process resource interaction, mainly from a threading and locking point of view. For instance, synchronized methods and classes (for example, `System.Collections.ArrayList.Synchronized`) are decorated with the `Synchronization` parameter of the attribute. These methods and classes, as well as those that expose a shared provider state or manage external processes, are disallowed from use within the SQL Server-hosted CLR environment, based on permission sets chosen by the DBA at deployment time. Permission sets are covered in more detail later in this chapter, in the section "Deploying CLR Routines."

Although an in-depth examination of CLR code safety issues is beyond the scope of this book, it's important that DBAs supporting the CLR features in SQL Server 2005 realize that this is no longer the world of XPs. These objects can be rolled out with a great deal of confidence. And as will be discussed later in this chapter, the DBA has the final say over what access the CLR code will have once it is deployed within the server.

## SQL Server .NET Programming Model

ADO.NET, the data access technology used within the .NET Framework, has been enhanced to operate within SQL Server 2005 hosted routines. These enhancements are fairly simple to exploit; for most operations, the only difference between coding on a client layer or within the database will be

modification of a connection string. Thanks to this, .NET developers will find a shallow learning curve when picking up SQL CLR skills. And when necessary, moving code between tiers will be relatively simple.

## Enhancements to ADO.NET for SQL Server Hosting

CLR stored procedures use ADO.NET objects to retrieve data from and write data to the database. These are the same objects you're already familiar with if you use ADO.NET today: `SqlCommand`, `SqlDataReader`, `DataSet`, etc. The only difference is, these can now be run in SQL Server's process space (in-processes) instead of only on a client.

When accessing SQL Server via an ADO.NET client, the `SqlConnection` object is instantiated, and a connection string is set, either in the constructor or using the `ConnectionString` property. This same process happens when instantiating an in-process connection—but the connection string has been rewired for SQL Server. Using the connection string `"Context connection=true"` tells SQL Server to use the same connection that spawned the CLR method as the connection from which to perform data access.

This means, in essence, that only a single change is all that's necessary for migration of the majority of data access code between tiers. To migrate code into SQL Server, classes and methods will still have to be appropriately decorated with attributes describing how they should function (see the section, "Anatomy of a Stored Procedure"), but the only substantial code change will be to the connection string! Virtually all members of the `SqlClient` namespace—with the notable exception of asynchronous operations—will work within the SQL Server process space.

The other major code difference between CLR routines and ADO.NET programming on clients is that inside of CLR routines the developer will generally want to communicate back to the session that invoked the routine. This communication can take any number of forms, from returning scalar values to sending back a result set from a stored procedure or table-valued function. However, the ADO.NET client has until now included no mechanisms for which to do that.

## Overview of the New .NET Namespaces for SQL Server

Two namespaces were added to the .NET Framework for integration with SQL Server 2005. These namespaces contain the methods and attributes necessary to create CLR routines within SQL Server 2005, and perform manipulation of database objects within those routines.

### Microsoft.SqlServer.Server

The `Microsoft.SqlServer.Server` namespace contains attributes for defining managed routines, as well as ADO.NET methods specific to the SQL Server provider.

In order for classes and methods to be defined as hosted CLR routines, they must be decorated with attributes to tell SQL Server what they are. These attributes include, among others, the `SqlProcedureAttribute` for defining CLR stored procedures, and the `SqlFunctionAttribute` for CLR user-defined functions. All of these attributes will be explained in detail in the next chapter.

The namespace also contains ADO.NET methods that allow CLR routines to communicate back to the session that invoked them. What can be communicated back depends on the type of CLR routine. For instance, a stored procedure can return messages, errors, result sets, or an integer return value. A table-valued user-defined function, on the other hand, can only return a single result set.

When programming CLR routines that need to return data, an object called `SqlContext` is available. This object represents a connection back to the session that instantiated the CLR routine. Exposed by this object is another object, `SqlPipe`. This is the means by which data is sent back to the caller. Sending properly formatted messages or result sets "down the pipe" means that the calling session will receive the data.

Note that not all `SqlContext` features are available from all routine types. For instance, a scalar user-defined function cannot send back a result set. Developers must remember to carefully test CLR routines; using a feature that's not available won't result in a compile-time error! Instead, an error will occur at runtime when the system attempts to use the unavailable feature. It's very important to keep this in mind during development in order to avoid problems once routines are rolled to production systems.

## Programming a CLR Stored Procedure

Now that the basic overview of what's available is complete, it's time to get into some code! The example used in this chapter will be a dynamic cross-tabulation of some sales data in the AdventureWorks sample database that's included with SQL Server 2005. Given the data in the `Sales.SalesOrderHeader` and `Sales.SalesOrderDetail` tables, the goal will be to produce a report based on a user-specified date range, in which the columns are sales months and each row aggregates total sales within each month, by territory.

Before starting work on any CLR routine, the developer should ask the following question: "Why should this routine be programmed using the CLR?" Remember that in most cases, T-SQL is still the preferred method of SQL Server programming—so give this question serious thought before continuing.

In this case, the argument in favor of using a CLR routine is fairly obvious. Although this problem can be solved using only T-SQL, it's a messy prospect at best. In order to accomplish this task, the routine first must determine in which months sales occurred within the input date range. Then, using that set of months, a query must be devised to create a column for each month and aggregate the appropriate data by territory.

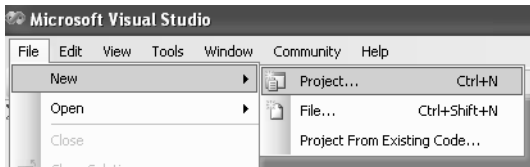
This task is made slightly easier than it was in previous versions of SQL Server, thanks to the inclusion of the `PIVOT` operator (see Chapter 3 for more information on this operator). This operator allows T-SQL developers to more easily write queries that transform rows into columns, a common reporting technique known as either *pivoting* or *cross-tabulating*. However, `PIVOT` doesn't provide dynamic capabilities—the developer still needs to perform fairly complex string concatenation in order to get things working. Concatenating strings is tricky and inefficient in T-SQL; using the .NET Framework's `StringBuilder` class is a much nicer prospect—and avoiding complex T-SQL string manipulation is argument enough to do this job within a CLR routine.

Once the determination to use a CLR routine has been made, the developer next must ask, "What is the appropriate type of routine to use for this job?" Generally speaking, this will be a fairly straightforward question; for instance, a CLR user-defined type and a CLR user-defined trigger obviously serve quite different purposes. However, the specific problem for this situation isn't so straightforward. There are two obvious choices: a CLR table-valued function and a CLR stored procedure.

The requirement for this task is to return a result set to the client containing the cross-tabulated data. Both CLR table-valued functions and CLR stored procedures can return result sets to the client. However, as will be discussed in the next chapter, CLR table-valued functions must have their output columns predefined. In this case, the column list is dynamic; if the user enters a three-month date range, up to four columns will appear in the result set—one for each month in which there were sales, and one for the territory sales are being aggregated for. Likewise, if the user enters a one-year date range, up to 13 columns may be returned. Therefore, it isn't possible to predefine the column list, and the only choice is to use a CLR stored procedure.

## Starting a Visual Studio 2005 SQL Server Project

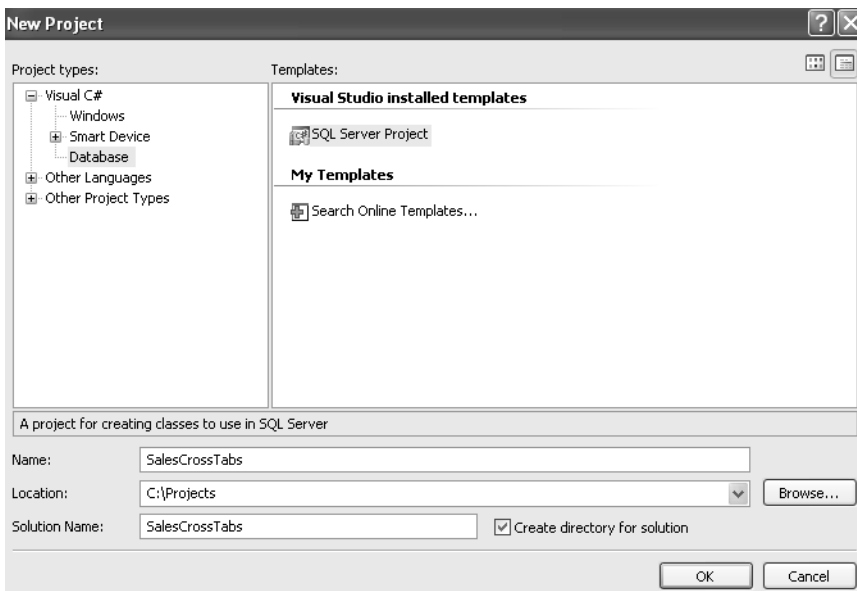
Once you have decided to program a CLR routine, the first step is to start Visual Studio 2005 and create a new project. Figure 5-1 shows the menu option to pick in order to launch the New Project Wizard.



**Figure 5-1.** Open a new project in Visual Studio 2005.

Visual Studio 2005 includes a project template for SQL Server projects, which automatically creates all of the necessary references and can create appropriate empty classes for all of the SQL Server CLR routine types. Although you could use a Class Library template instead and do all of this manually, that's not an especially efficient use of time. So we definitely recommend that you use this template when developing CLR routines.

Figure 5-2 shows the SQL Server Project template being chosen from the available database project templates. On this system, only C# has been installed; on a system with Visual Basic .NET, the same option would appear under that language's option tree.

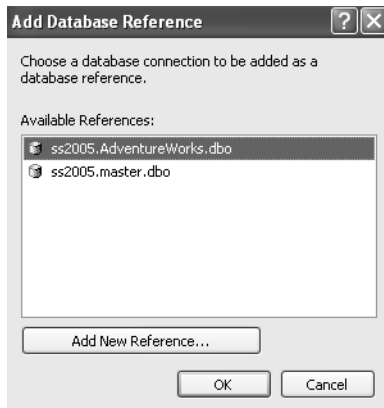


**Figure 5-2.** Select the SQL Server Project template.

This project has been named SalesCrossTabs, since it's going to contain at least one cross-tabulation of sales data—and perhaps more will be added in the future. A single SQL Server project can contain any number of CLR routines. However, it's recommended that any one project logically groups only a small number of routines. If a single change to a single routine is made, you should not have to reapply every assembly referenced by the database.



After clicking the OK button, a dialog box will appear prompting for a database reference, as shown in Figure 5-3. This reference indicates the SQL Server and database to which the project can be automatically deployed by Visual Studio for testing and debugging during the development process.



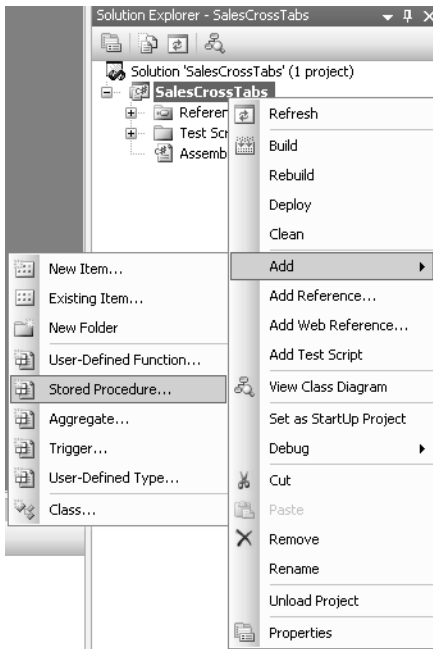
**Figure 5-3.** Add a database reference to allow Visual Studio to automatically deploy the routine.

If the correct reference hasn't already been created, click the Add New Reference button, and the dialog box shown in Figure 5-4 will appear. As development is being done for sales data in the AdventureWorks database, that's what has been selected for this reference. Once the server and database have been selected and the connection has been tested, click the OK button to continue.



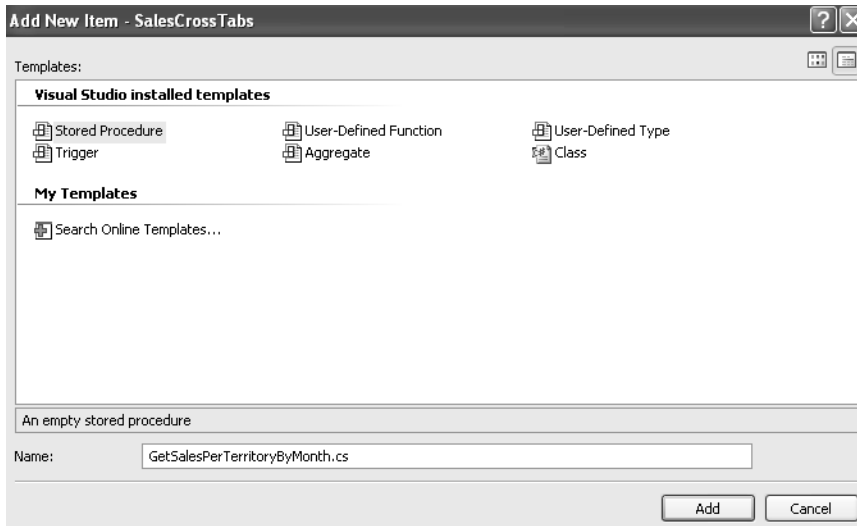
**Figure 5-4.** Create a new database reference if the correct one doesn't already exist.

At this point, a new, blank project has been created and is ready to have some code added. Right-click the project name in the Solution Explorer window, click Add, and then click Stored Procedure, as shown in Figure 5-5.



**Figure 5-5.** Adding a stored procedure to the project

The final step in adding the new stored procedure is to name it. Figure 5-6 shows the window that will appear after clicking Stored Procedure. The Stored Procedure template is selected, and the procedure has been named `GetSalesPerTerritoryByMonth`. Developers should remember that, just as in naming T-SQL stored procedures, descriptive, self-documenting names go a long way towards making development—especially maintenance—easier.



**Figure 5-6.** Naming the stored procedure

## Anatomy of a Stored Procedure

After the new stored procedure has been added to the project, the following code will appear in the editing window:

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

public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void GetSalesPerTerritoryByMonth()
    {
        // Put your code here
    }
};
```

There are a few important features to notice here. First of all, note that the `Microsoft.SqlServer.Server` and `System.Data.SqlTypes` namespaces have been automatically included in this project. Both of these namespaces have very specific purposes within a routine—and will be necessary within most SQL Server CLR projects.

The `Microsoft.SqlServer.Server` namespace is necessary, as previously mentioned, for the attributes that must decorate all routines to be hosted within SQL Server. In this case, the `GetSalesPerTerritoryByMonth` method has been decorated with the `SqlProcedure` attribute. This indicates that the method is a stored procedure. The method has also been defined as `static`—since this method will be called without an object instantiation, it would not be available if not defined as `static`. The namespace is also included in order to provide access to the calling context, for data access and returning data—but more on that shortly.

The `System.Data.SqlTypes` namespace is also included. This namespace provides datatypes that correspond to each of the SQL Server datatypes. For instance, the equivalent of SQL Server's `INTEGER` datatype isn't .NET's `System.Int32` datatype. Instead, it's `SqlTypes.SqlInt32`. Although these types can be cast between each other freely, not all types have direct equivalents—many of the SQL Server types have slightly different implementations than what would seem to be their .NET siblings. For that reason, and to provide some insulation in case of future underlying structural changes, it's important to use these types instead of the native .NET types when dealing with data returned from SQL Server, including both parameters to the routine and data read using a `SqlDataReader` or `DataSet`.

Aside from the included namespaces, note that the return type of the `GetSalesPerTerritoryByMonth` method is `void`. SQL Server stored procedures can return either 32-bit integers or nothing at all. In this case, the stored procedure won't have a return value. That's generally a good idea, because SQL Server will override the return value should an error occur within the stored procedure; so output parameters are considered to be a better option for returning scalar values to a client. However, should a developer want to implement a return value from this stored procedure, the allowed datatypes are `SqlInt32` or `SqlInt16`.

## Adding Parameters

Most stored procedures will have one or more parameters to allow users to pass in arguments that can tell the stored procedure which data to return. In the case of this particular stored procedure, two parameters will be added in order to facilitate getting data using a date range (one of the requirements outlined in the section “Programming a CLR Stored Procedure”). These parameters will be called `StartDate` and `EndDate`, and each will be defined as type `SqlDateTime`.

These two parameters are added to the method definition, just like parameters to any C# method:

```
[Microsoft.SqlServer.Server.SqlProcedure]
public static void GetSalesPerTerritoryByMonth( SqlDateTime StartDate,
                                              SqlDateTime EndDate)
{
    // Put your code here
}
```

In this case, these parameters are required input parameters. Output parameters can be defined by using the C# `ref` (reference) keyword before the datatype. This will then allow developers to use SQL Server's `OUTPUT` keyword in order to get back scalar values from the stored procedure.

Unfortunately, neither optional parameters nor default parameter values are currently supported by CLR stored procedures.

## Defining the Problem

At this point, the stored procedure is syntactically complete and could be deployed as is; but of course, it wouldn't do anything! It's time to code the meat of the procedure. But first, it's good to take a step back and figure out what it should do.

The final goal, as previously mentioned, is to cross-tabulate sales totals per territory, with a column for each month in which sales took place. This goal can be accomplished using the following steps:

1. Select a list of the months and years in which sales took place, from the `Sales.SalesOrderHeader` table.
2. Using the list of months, construct a query using the `PIVOT` operator that returns the desired cross-tabulation.
3. Return the cross-tabulated result set to the client.

The `Sales.SalesOrderHeader` table contains one row for each sale, and includes a column called `OrderDate`—the date the sale was made. For the sake of this stored procedure, a distinct list of the months and years in which sales were made will be considered. The following query returns that data:

```
SELECT DISTINCT
    DATEPART(yyyy, OrderDate) AS YearPart,
    DATEPART(mm, OrderDate) AS MonthPart
FROM Sales.SalesOrderHeader
ORDER BY YearPart, MonthPart
```

Once the stored procedure has that data, it needs to create the actual cross-tab query. This query needs to use the dates from `Sales.SalesOrderHeader` and for each month should calculate the sum of the amounts in the `LineTotal` column of the `Sales.SalesOrderDetail` table. And of course, this data should be aggregated per territory. The `TerritoryId` column of the `Sales.SalesOrderHeader` table will be used for that purpose.

The first step in creating the cross-tab query is to pull the actual data required. The following query returns the territory ID, order date formatted as `YYYY-MM`, and total line item amount for each line item in the `SalesOrderDetail` table:

```
SELECT
    TerritoryId,
    CONVERT(CHAR(7), h.OrderDate, 120) AS theDate,
    d.LineTotal
FROM Sales.SalesOrderHeader h
JOIN Sales.SalesOrderDetail d ON h.SalesOrderID = d.SalesOrderID
```

Figure 5-7 shows a few of the 121,317 rows of data returned by this query.

	TerritoryId	theDate	LineTotal
1	5	2001-07	2024.994000
2	5	2001-07	6074.982000
3	5	2001-07	2024.994000
4	5	2001-07	2039.994000
5	5	2001-07	2039.994000
6	5	2001-07	4079.988000
7	5	2001-07	2039.994000
8	5	2001-07	86.521200
9	5	2001-07	28.840400
10	5	2001-07	34.200000

**Figure 5-7.** *Unaggregated sales data*

Using the `PIVOT` operator, this query can be turned into a cross-tab. For instance, to report on sales from June and July of 2004, the following query could be used:

```

SELECT
    TerritoryId,
    [2004-06],
    [2004-07]
FROM
(
    SELECT
        TerritoryId,
        CONVERT(CHAR(7), h.OrderDate, 120) AS YYYY_MM,
        d.LineTotal
    FROM Sales.SalesOrderHeader h
    JOIN Sales.SalesOrderDetail d ON h.SalesOrderID = d.SalesOrderID
) p
PIVOT
(
    SUM (LineTotal)
    FOR YYYY_MM IN
    (
        [2004-06],
        [2004-07]
    )
) AS pvt
ORDER BY TerritoryId

```

Figure 5-8 shows the results of this query. The data has now been aggregated and cross-tabulated. Note that a couple of the values are NULL—this indicates a territory that did not have sales for that month.

	TerritoryId	2004-06	2004-07
1	1	779625.967724	10165.250000
2	2	240725.227509	NULL
3	3	298101.019968	NULL
4	4	993295.830953	9155.300000
5	5	269604.574888	113.960000
6	6	717837.710783	10853.700000
7	7	316740.799297	3491.950000
8	8	349467.104000	3604.830000
9	9	711086.975552	9234.230000
10	10	688354.968664	4221.410000

**Figure 5-8.** Cross-tabulated sales data

In the actual stored procedure, the tokens representing June and July 2004 will be replaced by tokens for the actual months from the input date range, as determined by the *StartDate* and *EndDate* parameters and the first query. Then the full cross-tab query will be concatenated. All that's left from the three steps defined previously is to return the results to the caller—but you have a couple of choices for how to tackle that challenge.

## Using the SqlPipe

As mentioned in previous sections, *SqlContext* is an object available from within the scope of CLR routines. This object is defined in the *Microsoft.SqlServer.Server* namespace as a sealed class with a private constructor—so you don't create an instance of it; rather, you just use it. The following code, for instance, is invalid:

```

//This code does not work -- the constructor for SqlContext is private
SqlContext context = new SqlContext();

```

Instead, just use the object as is. To use the `SqlPipe`, which is the object we need for this exercise, the following code might be used:

```
//Get a reference to the SqlPipe for this calling context
SqlPipe pipe = SqlContext.Pipe;
```

So what is the `SqlPipe`? This object allows the developer to send data or commands to be executed from a CLR routine back to the caller.

## The Send() Method

The `Send()` method, which as you can guess is used to actually send the data, has three overloads:

- `Send(string message)` sends a string, which will be interpreted as a message. Think `InfoMessage` in ADO.NET or the messages pane in SQL Server Management Studio. Sending strings using `Send()` has the same effect as using the T-SQL `PRINT` statement.
- `Send(SqlDataRecord record)` sends a single record back to the caller. This is used in conjunction with the `SendResultsStart()` and `SendResultsEnd()` methods to manually send a table a row at a time. Getting it working can be quite a hassle, and it's really not recommended for most applications. See the section "Table-Valued User-Defined Functions" in the next chapter for a much nicer approach.
- `Send(SqlDataReader reader)` sends an entire table back to the caller, in one shot. This is much nicer than doing things row-by-row, but also just as difficult to set up for sending back data that isn't already in a `SqlDataReader` object. Luckily, this particular stored procedure doesn't have that problem—it uses a `SqlDataReader`, so this method can be used to directly stream back the data read from the SQL Server.

The `Send()` methods can be called any number of times during the course of a stored procedure. Just like native T-SQL stored procedures, CLR procedures can return multiple result sets and multiple messages or errors. But by far the most common overload used will be the one that accepts `SqlDataReader`. The following code fragment shows a good example of the utility of this method:

```
command.CommandText = "SELECT * FROM Sales.SalesOrderHeader";
SqlDataReader reader = command.ExecuteReader();
SqlContext.Pipe.Send(reader);
```

In this fragment, it's assumed that the connection and command objects have already been instantiated and the connection has been opened. A reader is populated with the SQL, which selects all columns and all rows from the `Sales.SalesOrderHeader` table. The `SqlDataReader` can be passed to the `Send()` method as is—and the caller will receive the data as a result set.

Although this example is quite simplistic, it illustrates the ease with which the `Send()` method can be used to return data back to the caller when the data is already in a `SqlDataReader` object.

## Using the ExecuteAndSend() Method

The problem with sending a `SqlDataReader` back to the caller is that all of the data will be marshaled through the CLR process space on its way back. Since, in this case, the caller generated the data (it came from a table in SQL Server), it would be nice to be able to make the caller return the data on its own—without having to send the data back and forth.

This is where the `ExecuteAndSend()` method comes into play. This method accepts a `SqlCommand` object, which should have both `CommandText` and `Parameters` (if necessary) defined. This tells the calling context to execute the command and process the output itself.

Letting the caller do the work without sending the data back is quite a bit faster. In some cases, performance can improve by up to 50 percent. Sending all of that data between processes is a lot of

work. But this performance improvement comes at a cost; one of the benefits of handling the data within the CLR routine is control. Take the following code fragment, for example:

```
command.CommandText = "SELECT * FROM Sales.ERRORSalesOrderHeader";
try
{
    SqlDataReader reader = command.ExecuteReader();
    SqlContext.Pipe.Send(reader);
}
catch (Exception e)
{
    //Do something smart here
}
```

This fragment is similar to the fragment discussed in the “Using the Send() Method” section. It requests all of the rows and columns from the table, and then sends the data back to the caller using the Send() method. This work is wrapped in a try/catch block—the developer, perhaps, can do something to handle any exceptions that occur. And indeed, in this code block, an exception will occur—the table Sales.ERRORSalesOrderHeader doesn’t exist in the AdventureWorks database.

This exception will occur in the CLR routine—the ExecuteReader() method will fail. At that point, the exception will be caught by the catch block. But what about the following code fragment, which uses the ExecuteAndSend() method:

```
command.CommandText = "SELECT * FROM Sales.ERRORSalesOrderHeader";
try
{
    SqlContext.Pipe.ExecuteAndSend(command)
}
catch (Exception e)
{
    //Do something smart here
}
```

Remember that the ExecuteAndSend() method tells the caller to handle all output from whatever T-SQL is sent down the pipe. This includes exceptions; so in this case the catch block is hit, but by then it’s already too late. The caller has already received the exception, and catching it in the CLR routine isn’t especially useful.

So which method, Send() or ExecuteAndSend(), is appropriate for the sales cross-tabulation stored procedure? Given the simplicity of the example, the ExecuteAndSend() method makes more sense. It has greater performance than Send(), which is always a benefit. And there’s really nothing that can be done if an exception is encountered in the final T-SQL to generate the result set.

## Putting It All Together: Coding the Body of the Stored Procedure

Now that the techniques have been defined, putting together the complete stored procedure is a relatively straightforward process.

Recall that the first step is to get the list of months and years in which sales took place, within the input date range. Given that the pivot query will use date tokens formatted as YYYY-MM, it will be easier to process the unique tokens in the CLR stored procedure if they’re queried from the database in that format—so the query used will be slightly different than the one shown in the “Defining the Problem” section. The following code fragment will be used to get the months and years into a SqlDataReader object:



```
//Get a SqlCommand object
SqlCommand command = new SqlCommand();

//Use the context connection
command.Connection = new SqlConnection("Context connection=true");
command.Connection.Open();

//Define the T-SQL to execute
string sql =
    "SELECT DISTINCT " +
        "CONVERT(CHAR(7), h.OrderDate, 120) AS YYYY_MM " +
    "FROM Sales.SalesOrderHeader h " +
    "WHERE h.OrderDate BETWEEN @StartDate AND @EndDate " +
    "ORDER BY YYYY_MM";
command.CommandText = sql.ToString();

//Assign the StartDate and EndDate parameters
SqlParameter param =
    command.Parameters.Add("@StartDate", SqlDbType.DateTime);
param.Value = StartDate;
param = command.Parameters.Add("@EndDate", SqlDbType.DateTime);
param.Value = EndDate;

//Get the data
SqlDataReader reader = command.ExecuteReader();
```

This code uses the same `SqlCommand` and `SqlDataReader` syntax as it would if this were being used for an ADO.NET client; keep in mind that this code won't work unless the `System.Data.SqlClient` namespace is included with a `using` directive. The only difference between this example and a client application is the connection string, which tells SQL Server that this should connect back to the caller's context instead of a remote server. Everything else is the same—the connection is even opened, as if this were a client instead of running within SQL Server's process space.

As a result of this code, the reader object will contain one row for each month in which sales took place within the input date range (that is, the range between the values of the `StartDate` and `EndDate` parameters). Looking back at the fully formed pivot query, you can see that the tokens for each month need to go into two identical comma-delimited lists: one in the outer `SELECT` list and one in the `FOR` clause. Since these are identical lists, they only need to be built once. The following code handles that:

```
//Get a StringBuilder object
System.Text.StringBuilder yearsMonths = new System.Text.StringBuilder();

//Loop through each row in the reader, adding the value to the StringBuilder
while (reader.Read())
{
    yearsMonths.Append "[" + (string)reader["YYYY_MM"] + ", ";
}

//Close the reader
reader.Close();

//Remove the final comma in the list
yearsMonths.Remove(yearsMonths.Length - 2, 1);
```

A `StringBuilder` is used in this code instead of a `System.string`. This makes building the list a bit more efficient. For each row, the value of the `YYYY_MM` column (the only column in the reader) is enclosed in square brackets, as required by the `PIVOT` operator. Then a comma and a space are appended to the end of the token. The extra comma after the final token is removed after the loop is done. Finally, `SqlDataReader` is closed. When working with `SqlDataReader`, it's a good idea to close it as soon as data retrieval is finished in order to disconnect from the database and save resources.

Now that the comma-delimited list is built, all that's left is to build the cross-tab query and send it back to the caller using the `ExecuteAndSend()` method. The following code shows how that's done:

```
//Define the cross-tab query
sql =
    "SELECT TerritoryId, " +
        yearsMonths.ToString() +
    "FROM " +
    "(" +
        "SELECT " +
            "TerritoryId, " +
            "CONVERT(CHAR(7), h.OrderDate, 120) AS YYYY_MM, " +
            "d.LineTotal " +
        "FROM Sales.SalesOrderHeader h " +
        "JOIN Sales.SalesOrderDetail d " +
        "ON h.SalesOrderID = d.SalesOrderID " +
        "WHERE h.OrderDate BETWEEN @StartDate AND @EndDate " +
    ") p " +
    "PIVOT " +
    "(" +
        "SUM (LineTotal) " +
        "FOR YYYY_MM IN " +
        "(" +
            yearsMonths.ToString() +
        ")" +
    ") AS pvt " +
    "ORDER BY TerritoryId";

//Set the CommandText
command.CommandText = sql.ToString();

//Have the caller execute the cross-tab query
SqlContext.Pipe.ExecuteAndSend(command);

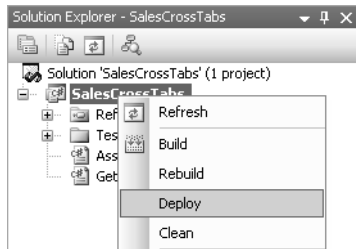
//Close the connection
command.Connection.Close();
```

Note that the same command object is being used as was used for building the comma-delimited list of months in which sales took place. This command object already has the `StartDate` and `EndDate` parameters set; since the cross-tab query uses the same parameters, the parameters collection doesn't need to be repopulated. Just like when programming in an ADO.NET client, the connection should be closed when the process is finished with it.

At this point, the CLR stored procedure is completely functional as per the three design goals; so it's ready for a test drive.

## Testing the Stored Procedure

Visual Studio 2005 makes deploying the stored procedure to the test SQL Server quite easy. Just right-click the project name (in this case, SalesCrossTabs) and click Deploy. Figure 5-9 shows what the option looks like.



**Figure 5-9.** Deploying the assembly to the test server

Once the procedure is deployed, testing it is simple. Log in to SQL Server Management Studio and execute the following batch of T-SQL:

```
USE AdventureWorks
GO

EXEC GetSalesPerTerritoryByMonth
    @StartDate = '20040501',
    @EndDate = '20040701'
GO
```

If everything is properly compiled and deployed, this should output a result set containing cross-tabulated data for the period between May 1, 2004, and July 1, 2004. Figure 5-10 shows the correct output.

TerritoryId	2004-05	2004-06	2004-07
1	837807.162368	779625.967724	513.140000
2	168740.205705	240725.227509	NULL
3	249727.190221	298101.019968	NULL
4	1007034.715534	993295.830953	248.140000
5	264907.861900	269604.574888	NULL
6	533267.247946	717837.710783	472.150000
7	766232.284720	316740.799297	178.940000
8	407253.815030	349467.104000	149.230000
9	551630.604000	711086.975552	221.730000
10	407520.435480	688354.968664	51.460000

**Figure 5-10.** Cross-tabulated sales data for the period between May 1, 2004, and July 1, 2004

Note that running the stored procedure might result in the message, “Execution of user code in the .NET Framework is disabled. Use sp\_configure 'clr enabled' to enable execution of user code in the .NET Framework. If this happens, execute the following batch of T-SQL to turn on CLR integration for the SQL Server:

```
USE AdventureWorks
GO
```

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

Running the `sp_configure` system stored procedure to enable CLR integration is required before CLR routines can be run in any database. Keep in mind that enabling or disabling CLR integration is a server-wide setting.

Once the stored procedure is running properly, it will appear that the stored procedure works as designed! However, perhaps some deeper testing is warranted to ensure that the procedure really is as robust as it should be. Figure 5-11 shows the output from the following batch of T-SQL:

```
USE AdventureWorks
GO
```

```
EXEC GetSalesPerTerritoryByMonth
    @StartDate = '20050501',
    @EndDate = '20050701'
GO
```

```
Msg 50000, Level 16, State 1, Line 1
No data present for the input date range.
Msg 6522, Level 16, State 1, Procedure GetSalesPerTerritoryByMonth, Line 0
A .NET Framework error occurred during execution of user defined routine or aggregate
System.Data.SqlClient.SqlException: No data present for the input date range.
System.Data.SqlClient.SqlException:
    at System.Data.SqlClient.SqlConnection.OnError(SqlException exception, Boolean brei
    at System.Data.SqlClient.SqlInternalConnection.OnError(SqlException exception, Boo
    at System.Data.SqlClient.SqlInternalConnectionSmi.ProcessMessages()
    at System.Data.SqlClient.SqlCommand.RunExecuteNonQuerySmi(Boolean sendToPipe)
    at System.Data.SqlClient.SqlCommand.InternalExecuteNonQuery(DbAsyncResult result, I
    at System.Data.SqlClient.SqlCommand.ExecuteToPipe(SmiContext pipeContext)
    at Microsoft.SqlServer.Server.SqlPipe.ExecuteAndSend(SqlCommand command)
    at StoredProcedures.GetSalesPerTerritoryByMonth(SqlDateTime startDate, SqlDateTime
```

**Figure 5-11.** Attempting to cross-tabulate sales data for the period between May 1, 2005, and July 1, 2005

## Debugging the Procedure

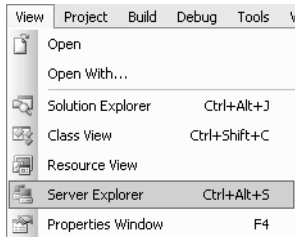
What a difference a year makes! Luckily, since this stored procedure is being coded in Visual Studio, the integrated debugging environment can be used to track down the problem. In the Solution Explorer, expand the Test Scripts tree and double-click `Test.sql`. This will open a template that can contain T-SQL code to invoke CLR routines for debugging. Paste the following T-SQL into the Stored procedure section:

```
EXEC GetSalesPerTerritoryByMonth
    @StartDate = '20050501',
    @EndDate = '20050701'
```

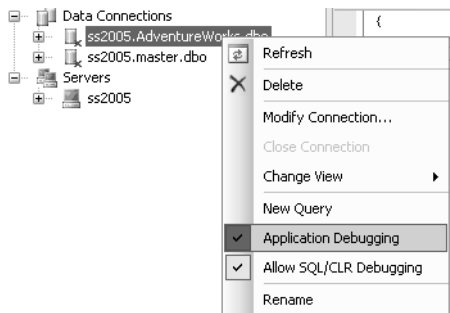
Now return to the code for the managed stored procedure and put the cursor on the first line: `SqlCommand command = new SqlCommand();`. Pressing the F9 key will toggle a breakpoint for that line.

Before starting the debug session, open the Server Explorer by clicking View ► Server Explorer, as shown in Figure 5-12. In the Server Explorer, right-click the database connection being used for this project and make sure that both Application Debugging and Allow SQL/CLR Debugging are

checked, as shown in Figure 5-13. Keep in mind that allowing SQL/CLR debugging should not be done on a production SQL Server. During debugging, all managed code running within the SQL Server process will be halted should any breakpoints be hit. This can wreak havoc on a live system that makes use of CLR routines, so make sure to only debug on development systems.



**Figure 5-12.** *Opening the Server Explorer in Visual Studio*



**Figure 5-13.** *Allowing SQL/CLR Debugging for the project's database connection*

Once debugging is enabled, press the F5 key, and Visual Studio will enter debug mode. If all is working as it should, the breakpoint should be hit—and code execution will stop on the first line of the stored procedure.

Use the F10 key to step through the stored procedure one line at a time, using the Locals pane to check the value of all of the variables. Stop stepping through on the line `yearsMonths.Remove(yearsMonths.Length - 2, 1);`, and look at the value of the `yearsMonths` variable in the Locals pane—it's empty; no characters can be removed from an empty string!

As it turns out, this stored procedure wasn't coded properly to be able to handle date ranges in which there is no data. This is definitely a big problem, since the output requires a column per each month in the input date range that sales occurred. Without any data, there can be no columns in the output. The stored procedure needs to return an error if no data is present.

## Throwing Exceptions in CLR Routines

Any exception that can be thrown from the CLR will bubble up to the SQL Server context if it's not caught within the CLR routine. For instance, the sales cross-tab stored procedure could be made a bit more robust by raising an error if `yearsMonths` is zero characters long, instead of attempting to remove the comma:

```

if (yearsMonths.Length > 0)
{
    //Remove the final comma in the list
    yearsMonths.Remove(yearsMonths.Length - 2, 1);
}
else
{
    throw new ApplicationException("No data present for the input date range.");
}

```

Instead of getting a random error from the routine, a well-defined error is now returned—theoretically. In reality, the error isn't so friendly. As shown in Figure 5-14, these errors can get quite muddled—not only is the error returned as with native T-SQL errors, but the call stack is also included. And although that's useful for debugging, it's overkill for the purpose of a well-defined exception.

```

Msg 6522, Level 16, State 1, Procedure GetSalesPerTerritoryByMonth, Line 0
A .NET Framework error occurred during execution of user defined routine or aggregate 'GetSalesPerTerritoryBy
System.ApplicationException: No data present for the input date range.
System.ApplicationException:
    at StoredProcedures.GetSalesPerTerritoryByMonth(SqlDateTime StartDate, SqlDateTime EndDate)
.

```

**Figure 5-14.** Standard CLR exceptions aren't formatted well for readability.

A better option, obviously, would be to use a native T-SQL error, invoked with the `RAISERROR()` function. A batch can be sent using `SqlPipe.ExecuteAndSend()`, as in the following code fragment:

```

if (yearsMonths.Length > 0)
{
    //Remove the final comma in the list
    yearsMonths.Remove(yearsMonths.Length - 2, 1);
}
else
{
    command.CommandText =
        "RAISERROR('No data present for the input date range.', 16, 1)";
    SqlContext.Pipe.ExecuteAndSend(command);
    return;
}

```

Alas, as shown in Figure 5-15, this produces an even worse output. The T-SQL exception bubbles back into the CLR layer, where a second CLR exception is thrown as a result of the presence of the T-SQL exception.

```

Msg 50000, Level 16, State 1, Line 1
No data present for the input date range.
Msg 6522, Level 16, State 1, Procedure GetSalesPerTerritoryByMonth, Line 0
A .NET Framework error occurred during execution of user defined routine or aggregate 'GetSalesPerTerritoryBy
System.Data.SqlClient.SqlException: No data present for the input date range.
System.Data.SqlClient.SqlException:
    at System.Data.SqlClient.Internal.StandardEventsSink.HandleErrors()
    at System.Data.SqlClient.Internal.RequestExecutor.HandleExecute(EventTranslator eventTranslator, SqlConnection
    at System.Data.SqlClient.Internal.RequestExecutor.ExecuteToPipe(SqlConnection conn, SqlTransaction tran, C
    at System.Data.SqlClient.SqlPipe.Execute(SqlCommand command)
    at StoredProcedures.GetSalesPerTerritoryByMonth(SqlDateTime StartDate, SqlDateTime EndDate)
.

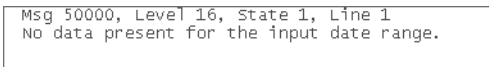
```

**Figure 5-15.** `RAISERROR` alone doesn't improve upon the quality of the exception.

The solution, as strange as it seems, is to raise the error using `RAISERROR` but catch it so that a second error isn't raised when control is returned to the CLR routine. The following code fragment shows how to accomplish this:

```
if (yearsMonths.Length > 0)
{
    //Remove the final comma in the list
    yearsMonths.Remove(yearsMonths.Length - 2, 1);
}
else
{
    command.CommandText =
        "RAISERROR('No data present for the input date range.', 16, 1)";
    try
    {
        SqlContext.Pipe.ExecuteAndSend(command);
    }
    catch
    {
        return;
    }
}
```

After catching the exception, the method returns—if it were to continue, more exceptions would be thrown by the `PIVOT` routine, as no pivot columns could be defined. Figure 5-16 shows the output this produces when run with an invalid date range. It's quite a bit easier to read than the previous exceptions.



```
Msg 50000, Level 16, State 1, Line 1
No data present for the input date range.
```

**Figure 5-16.** *Catching the exception in the CLR routine after firing a `RAISERROR` yields the most readable exception message.*

The complete code for the sales cross-tab stored procedure, including handling for invalid date ranges, follows:

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

public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void GetSalesPerTerritoryByMonth( SqlDateTime StartDate,
                                                    SqlDateTime EndDate)
    {
        //Get a SqlCommand object
        SqlCommand command = new SqlCommand();
```

```

//Use the context connection
command.Connection = new SqlConnection("Context connection=true");
command.Connection.Open();

//Define the T-SQL to execute
string sql =
    "SELECT DISTINCT " +
        "CONVERT(CHAR(7), h.OrderDate, 120) AS YYYY_MM " +
    "FROM Sales.SalesOrderHeader h " +
    "WHERE h.OrderDate BETWEEN @StartDate AND @EndDate " +
    "ORDER BY YYYY_MM";
command.CommandText = sql.ToString();

//Assign the StartDate and EndDate parameters
SqlParameter param =
    command.Parameters.Add("@StartDate", SqlDbType.DateTime);
param.Value = StartDate;
param = command.Parameters.Add("@EndDate", SqlDbType.DateTime);
param.Value = EndDate;

//Get the data
SqlDataReader reader = command.ExecuteReader();

//Get a StringBuilder object
System.Text.StringBuilder yearsMonths = new System.Text.StringBuilder();

//Loop through each row in the reader, adding the value to the StringBuilder
while (reader.Read())
{
    yearsMonths.Append("[ " + (string)reader["YYYY_MM"] + " ], ");
}

//Close the reader
reader.Close();

if (yearsMonths.Length > 0)
{
    //Remove the final comma in the list
    yearsMonths.Remove(yearsMonths.Length - 2, 1);
}
else
{
    command.CommandText =
        "RAISERROR('No data present for the input date range.', 16, 1)";
    try
    {
        {
            SqlContext.Pipe.ExecuteAndSend(command);
        }
        catch
        {
            return;
        }
    }
}
}

```



```

//Define the cross-tab query
sql =
    "SELECT TerritoryId, " +
        yearsMonths.ToString() +
    "FROM " +
    "(" +
        "SELECT " +
            "TerritoryId, " +
            "CONVERT(CHAR(7), h.OrderDate, 120) AS YYYY_MM, " +
            "d.LineTotal " +
        "FROM Sales.SalesOrderHeader h " +
        "JOIN Sales.SalesOrderDetail d " +
            "ON h.SalesOrderID = d.SalesOrderID " +
        "WHERE h.OrderDate BETWEEN @StartDate AND @EndDate " +
    ") p " +
    "PIVOT " +
    "(" +
        "SUM (LineTotal) " +
        "FOR YYYY_MM IN " +
        "(" +
            yearsMonths.ToString() +
        ")" +
    ") AS pvt " +
    "ORDER BY TerritoryId";

//Set the CommandText
command.CommandText = sql.ToString();

//Have the caller execute the cross-tab query
SqlContext.Pipe.ExecuteAndSend(command);

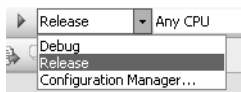
//Close the connection
command.Connection.Close();
}
};

```

## Deploying CLR Routines

Once a routine is written, tested, and—if necessary—debugged, it can finally be rolled to production. The process of doing this is quite simple: the release version of the DLL is copied to the server, and a few T-SQL statements are executed.

In order to produce a release version, change the build option on the Standard toolbar from Debug to Release, as shown in Figure 5-17. Once the configuration is set, click Build from the main toolbar, and then click Build Solution. This will produce a release version of the DLL—a version with no debug symbols—in the [Project Root]\bin\Release folder. So if the root folder for the project is C:\Projects\SalesCrossTabs, the DLL will be in C:\Projects\SalesCrossTabs\bin\Release.



**Figure 5-17.** *Configuring the project for a release build*

The release version of the DLL can be copied from this location onto any production server in order to deploy it. Only the DLL is required in order to deploy the CLR routines compiled within it.

The DLL is registered with SQL Server 2005 using the `CREATE ASSEMBLY` statement. The syntax for this statement is

```
CREATE ASSEMBLY assembly_name
[ AUTHORIZATION owner_name ]
FROM { <client_assembly_specifier> | <assembly_bits> [,...n] }
[ WITH PERMISSION_SET = { SAFE | EXTERNAL_ACCESS | UNSAFE } ]
[ ; ]
```

The `assembly_name` represents a user-defined name for the assembly—generally, it's best to use the name of the project. The `AUTHORIZATION` clause is optional, and allows the DBA to specify a particular owner for the object. The important part of the `FROM` clause is the `client_assembly_specifier`—this is the physical path to the DLL file. The `assembly_bits` option is used for situations in which the DLL has been binary serialized, and won't be covered in this book.

The most important clause of `CREATE ASSEMBLY`, however, is the optional `WITH PERMISSION_SET` clause. The DBA is in complete control when it comes to what CLR routines can do. Routines can be assigned to one of three permission sets—`SAFE`, `EXTERNAL_ACCESS`, or `UNSAFE`. Each permission set is progressively less restrictive. By controlling CLR routine permission, the DBA can keep a close watch on what routines are doing—and make sure that none are violating system policies.

The default `SAFE` permission set restricts routines from accessing any external resources, including files, web services, the registry, or networks. The `EXTERNAL_ACCESS` permission set opens up access to these external resources. This can be useful for situations in which data from the database needs to be merged with data from other sources. Finally, the `UNSAFE` permission set opens access to all CLR libraries. It is recommended that this permission set not be used, as there is potential for destabilization of the SQL Server process space if libraries are misused.

Assuming that the `SalesCrossTabs` DLL was copied to the `C:\Assemblies` folder on the SQL Server, it could be registered using the following T-SQL:

```
CREATE ASSEMBLY SalesCrossTabs
FROM 'C:\Assemblies\SalesCrossTabs.DLL'
```

Since this assembly doesn't use any external resources, the default permission set doesn't need to be overridden. Keep in mind that if the assembly has already been deployed using Visual Studio, this T-SQL would fail; assembly names must be unique within a database. If there is already an assembly called `SalesCrossTabs` from a Visual Studio deployment, it can be dropped using the `DROP ASSEMBLY` statement.

Once `CREATE ASSEMBLY` has successfully registered the assembly, the physical file is no longer accessed—the assembly is now part of the database it's registered in.

The next step is to tell SQL Server how to use the procedures, functions, and types in the assembly. This is done using slightly modified versions of the `CREATE` statements for each of these objects. To register the `GetSalesPerTerritoryByMonth` stored procedure, the following T-SQL would be used:

```
CREATE PROCEDURE GetSalesPerTerritoryByMonth
    @StartDate DATETIME,
    @EndDate DATETIME
AS
EXTERNAL NAME SalesCrossTabs.StoredProcedures.GetSalesPerTerritoryByMonth
```

The parameter list must match the parameters defined on the CLR method. The `EXTERNAL NAME` clause requires three parameters, delimited by periods: the user-defined name of the assembly, the name of the class defined in assembly (in this case, the default `StoredProcedures` class), and finally the name of the method defined as the stored procedure in the assembly. This clause is case sensitive, so be careful. Changing the case from that defined in the routine will result in an error.

Once the stored procedure is defined in this way, it can be called just like any native T-SQL stored procedure.

## Summary

CLR integration allows developers to extend the functionality of SQL Server 2005 using safe, well-performing methods. Coding CLR stored procedures is an easy way to improve upon some of the things that T-SQL doesn't do especially well.

In the next chapter, we'll cover the other types of CLR objects available to developers: functions, aggregates, user-defined types, and triggers. We'll also present a more in-depth look into managing routines from a DBA's perspective.

