Pro SQL Server 2005 Reporting Services

Rodney Landrum
and Walter J. Voytek II

Apress®

Pro SQL Server 2005 Reporting Services

Copyright © 2006 by Rodney Landrum and Walter J. Voytek II

ISBN (pbk): 1-59059-498-3

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

Lead Editor: Tony Davis
Technical Reviewer: Trish Middleton, Chris Rausch, Thomas Rizzo
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett, Julie McNamee
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor and Artist: Kinetic Publishing Services, LLC
Proofreader: April Eddy, Linda Seifert
Indexer: Broccoli Information Management
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, 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, or visit http://www.apress.com.

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

CHAPTER 5

Using Custom .NET Code with Reports

SSRS 2005 offers software developers a variety of options when it comes to customizing reports through code. These options give software developers the ability to write custom functions using .NET code that can interact with report fields, parameters, and filters in much the same way as any of the built-in functions. To give just two examples, you can create a custom function that does the following:

Implements a business rule and returns `true` *or* `false` *based on the logic*: You can use such a function as part of an expression to change the value or style of a field based on the fields or parameters passed to the function.

Reads data from sources not otherwise available to SSRS 2005 directly: You can do this by having your custom code read data directly from the source. In this chapter, you will examine how to read data from an XML file. The sample code for this chapter also includes an example of reading data from a Web service. Although we won't cover it in this chapter, with SSRS 2005 you can also create custom extensions that will allow you to view data in the Report Designer as a data source.

In short, using custom .NET code gives developers the ability to extend the capabilities of SSRS 2005 far beyond those that are available out of the box.
This chapter will cover the following:

Custom code for use within your report using code embedded in the report: This method is the simplest way to add custom code to your report, and it deploys along with your report since it is contained in the RDL. However, it limits what you can do, must be written in VB .NET, and offers limited debugging support.

Custom code for use within your report using a custom assembly called by the report: This method is more involved to implement and more difficult to deploy, but it offers you nearly unlimited flexibility. Your custom code has the full power of the .NET Framework at its disposal and has the added benefit that you can use the custom assembly across multiple reports. You can also use the full debugging capabilities of Visual Studio while developing your custom assembly.

Generally, you will add custom code to your report when you need to perform complex functions and you need the capabilities of a full programming language to accomplish them. However, before you embark on writing custom .NET code, you should first evaluate whether using the built-in expression functionality can meet your needs.

Using Embedded Code in Your Report

Using embedded code is by far the easiest way to implement custom .NET code in your reports, for two main reasons. First, you simply add the code directly to the report using the Report Designer's user interface (UI) in either BIDS or Visual Studio. Second, this code becomes a segment within the report's RDL file, making its deployment simple because it is a part of your report and will be deployed with it.

Although it is easier to use, embedded code does have a few considerations that you should take into account:

Embedded code must be written in VB .NET: If you are a C# programmer or use some other .NET-compatible language as your primary development language, this may force you to use the custom assembly for all but the simplest of functions.

All methods must be instance based: This means the methods will belong to an instantiated instance of the `Code` object and you cannot have static members.

Only basic operations are available: This is because, by default, code access security will prevent your embedded code from calling external assemblies and protected resources. You could change this through SSRS 2005 security policies, but this would require granting `FullTrust` to the report expression host, which would grant full access to the CLR and is definitely not recommended. If you need these capabilities, use custom assemblies so you can implement security policies to grant each assembly only the security it needs. You will look at custom assemblies and how to set security for them in the "Deploying a Custom Assembly" section.

Before you run the included examples, make sure to read the `ReadMe.htm` file included with the sample code for this chapter. It is located in a file in the samples root folder. If you have the code open in Visual Studio, it will be under the `Solution Items` folder. It contains setup and configuration steps that are required before running the examples.

Let's take a look at how this feature of SSRS 2005 works by adding some embedded code to one of the reports you have already created. In this case, start with the sample Employee Service Cost report included with this chapter. It is a slightly modified version of the Employee Service Cost report you created in Chapter 4. We will show you how to use the embedded code feature to add a function that will determine whether you have exceeded a certain number of visits for a patient in a given time period. You will then use that function to determine the color of one of the text fields in the report to help draw attention to those specific patients.

Note In this chapter's example, we'll use a slightly modified version of the report created in Chapter 4 so that the employee report parameter will include an employee with patients who have exceeded the maximum number of visits. We've also set defaults for the report parameters to make sure the results included this patient.

Using the ExceedMaxVisits Function

Listing 5-1 is the full listing of the custom code that you will add to the Employee Service Cost report. It is a simple function, called `ExceedMaxVisits`, which determines whether a patient has exceeded a certain number of visits over some period of time. This allows you to identify cases for review to determine why they have such a high utilization of services.

Listing 5-1. *The* `ExceedMaxVisits` *Function*

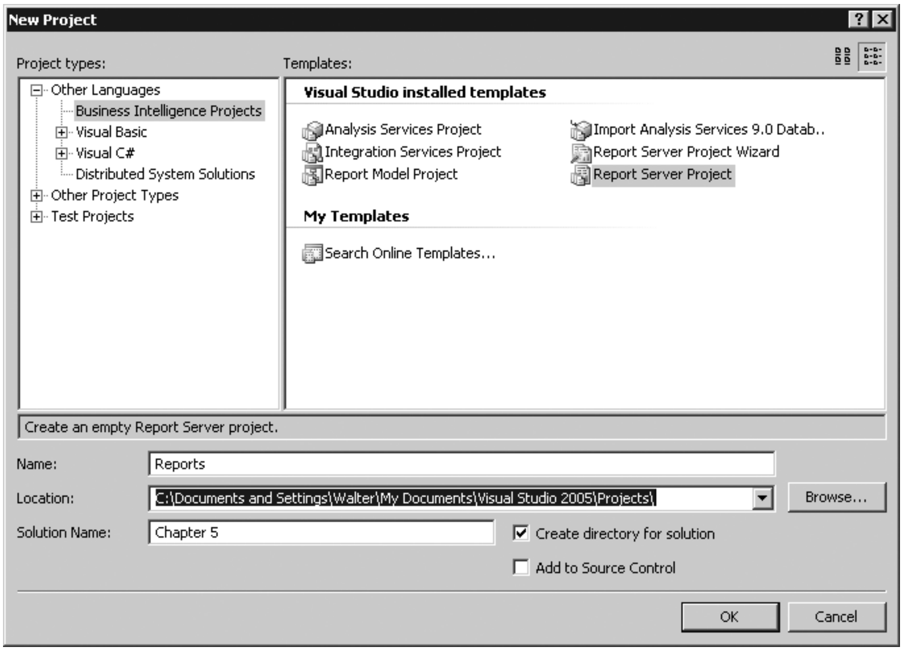
```
Function ExceedMaxVisits(ByVal visitCount As Integer, ➥
    ByVal visitMonth As Integer, ➥
    ByVal visitYear As Integer) As Boolean

    ' Our businesses logic dictates that we need to know whether
    ' we exceed 240 visits per patient per visitYear
    ' or 20 visits per patient per visitMonth
    If (visitMonth = Nothing And visitYear <> Nothing) Then
        If visitCount > 240 Then
            Return True
        End If
    ElseIf (visitMonth <> Nothing) Then
            If visitCount > 20 Then
                Return True
        End If
    End If

    Return False

End Function
```

If you are following along with the code in the book, you will need to create a new Visual Studio 2005 BI project, as shown in Figure 5-1. If you don't have Visual Studio 2005 installed, you can create this first project in BIDS. For this example, call the solution Chapter 5 and the project Reports.


New Project
Project types:
Other Languages
Business Intelligence Projects
Visual Basic
Visual C#
Distributed System Solutions
Other Project Types
Test Projects
Templates:
Visual Studio installed templates
Analysis Services Project
Integration Services Project
Report Model Project
Import Analysis Services 9.0 Datab..
Report Server Project Wizard
Report Server Project
My Templates
Search Online Templates...
Browse...
OK
Cancel

Create an empty Report Server project.

Name: Reports

Location: C:\Documents and Settings\Walter\My Documents\Visual Studio 2005\Projects\

Solution Name: Chapter 5

[✓] Create directory for solution

[ ] Add to Source Control

Figure 5-1. *Creating a Visual Studio BI project*

■ **Note** For those of you who are not familiar with Visual Studio 2005 and/or BIDS, they are essentially the same IDE except Visual Studio 2005 adds full programming language support such as C# and VB .NET as well as other software development tools. Also note that both organize individual projects into solutions so you can keep related projects together.

To add the existing `EmployeeServiceCost-NoCode.rdl` file to your new project, right-click the Reports project, and select Add ➤ Existing Item, as shown in Figure 5-2. Alternatively, with the Reports project highlighted, select Project ➤ Add Existing Item from the menu. Next, browse to the location where you installed the Chapter 5 samples, and select `EmployeeServiceCost-NoCode.rdl`. You will also need to add the shared data source by adding an existing item and picking the `Pro_SSRS.rds` file.

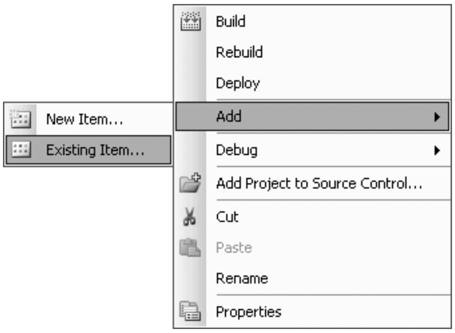
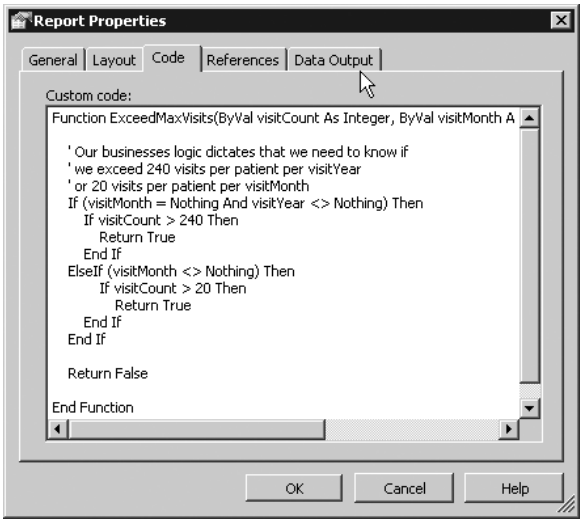


Figure 5-2. *Adding* `EmployeeServiceCost-NoCode.rdl` *to your project*

To add the code from Listing 5-1 to the Employee Service Cost report, first open the report by double-clicking it or by right-clicking it and selecting Open in the Solution Explorer. Next, with the report on the Layout tab, select Report Properties from the Visual Studio Report menu; alternatively, right-click within the report design area, and select Properties. On the Report Properties dialog box's Code tab, add the code from Listing 5-1 to the Custom Code box, as shown in Figure 5-3.



Custom code:

```
Function ExceedMaxVisits(ByVal visitCount As Integer, ByVal visitMonth A

   ' Our businesses logic dictates that we need to know if
   ' we exceed 240 visits per patient per visitYear
   ' or 20 visits per patient per visitMonth
   If (visitMonth = Nothing And visitYear <> Nothing) Then
      If visitCount > 240 Then
         Return True
      End If
   ElseIf (visitMonth <> Nothing) Then
         If visitCount > 20 Then
            Return True
      End If
   End If

   Return False

End Function
```

Figure 5-3. *Entering embedded code in the custom code editor*

Note You must enter the function declaration (the first line) as a single line in the embedded code editor, or you will receive an error when you try to preview the report. In Listing 5-1, it is shown with returns but should be entered into the embedded code editor without them.

Now that you have defined your custom code, you'll want to use it to highlight the patients who have exceeded the maximum visit count. To do this, you need to access the `ExceedMaxVisits` method as part of an expression.

Methods in embedded code are available through a globally defined `Code` member. When a report's RDL file is compiled into a .NET assembly (at publish time), SSRS 2005 creates a global member of the class called `Code` that you can access in any expression by referring to the `Code` member and method name, such as `Code.ExceedMaxVisits`.

Listing 5-2 shows how to use a conditional expression in the `Color` property of a textbox to set the color of the text depending on the return value of the function call.

Listing 5-2. *Using a Conditional Expression*

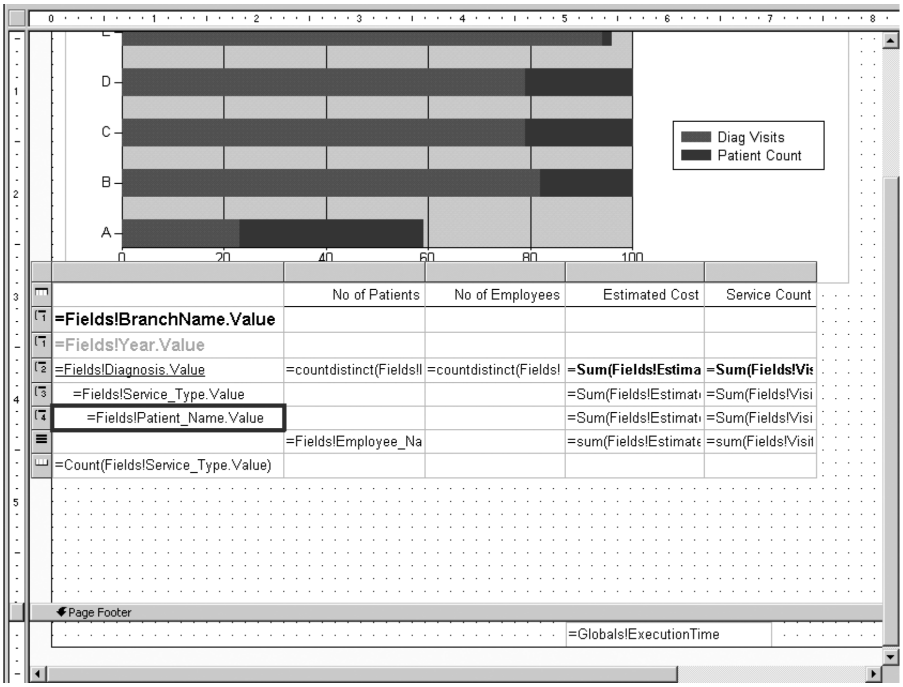
```
=iif(Code.ExceedMaxVisits(Sum(Fields!Visit_Count.Value),
    Parameters!ServiceMonth.Value,Parameters!ServiceYear.Value),
    "Red", "Black")
```

The method `ExceedMaxVisits` determines whether the patient has had more visits in the time span than allowed and returns `true` if so or `false` if not. Using a Boolean return value makes it easy to use the method in formatting expressions, because the return value can be tested directly instead of comparing the returned value to another value.

When a patient exceeds the maximum visits allowed, `ExceedMaxVisits` returns `true`, which sets the value of the textbox `Color` property to `Red`, which in turn will cause the report to display the text in red. If the patient has not exceeded the allowable number of visits, then `ExceedMaxVisits` returns `false`, and the `Color` property is set to `Black`.

Using the ExceedMaxVisits Function in a Report

Now we'll walk you through how to actually add this expression to the report. First, select the field in the report to which you want to apply the expression. In this case, select the patient name textbox, as shown in Figure 5-4.

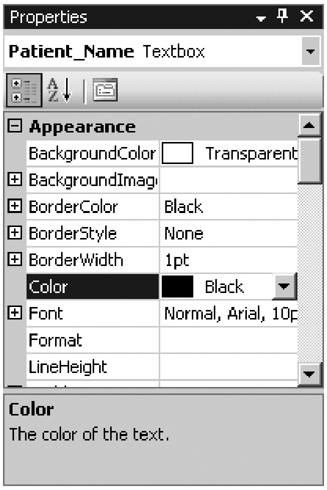


	No of Patients	No of Employees	Estimated Cost	Service Count
=Fields!BranchName.Value				
=Fields!Year.Value				
=Fields!Diagnosis.Value	=countdistinct(Fields!I	=countdistinct(Fields!	=Sum(Fields!Estima	=Sum(Fields!Vis
=Fields!Service_Type.Value			=Sum(Fields!Estimat	=Sum(Fields!Visi
=Fields!Patient_Name.Value			=Sum(Fields!Estimat	=Sum(Fields!Visi
	=Fields!Employee_Na		=sum(Fields!Estimate	=sum(Fields!Visit
=Count(Fields!Service_Type.Value)				

=Globals!ExecutionTime

Figure 5-4. *Adding the expression to the report*

Second, with the textbox selected, go to the Properties window, and select the `Color` property (see Figure 5-5).


Properties

Patient_Name Textbox

Appearance

BackgroundColor	Transparent
BackgroundImag	
BorderColor	Black
BorderStyle	None
BorderWidth	1pt
Color	Black
Font	Normal, Arial, 10p
Format	
LineHeight	

Color
The color of the text.

Figure 5-5. `Color` *property in the Properties window*

Next, click the down arrow, and from the list select Expression (see Figure 5-6).

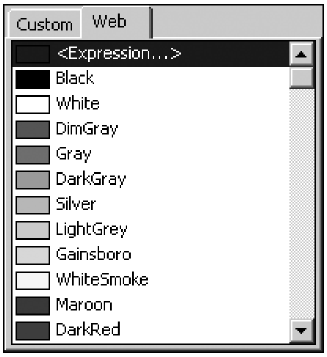


Figure 5-6. *Color selection list*

Now you will see the Edit Expression dialog box, as shown in Figure 5-7. Enter the expression using your custom code here. You can just type the expression in, or you can use the expression editor to insert the parameters that you need into your expression.

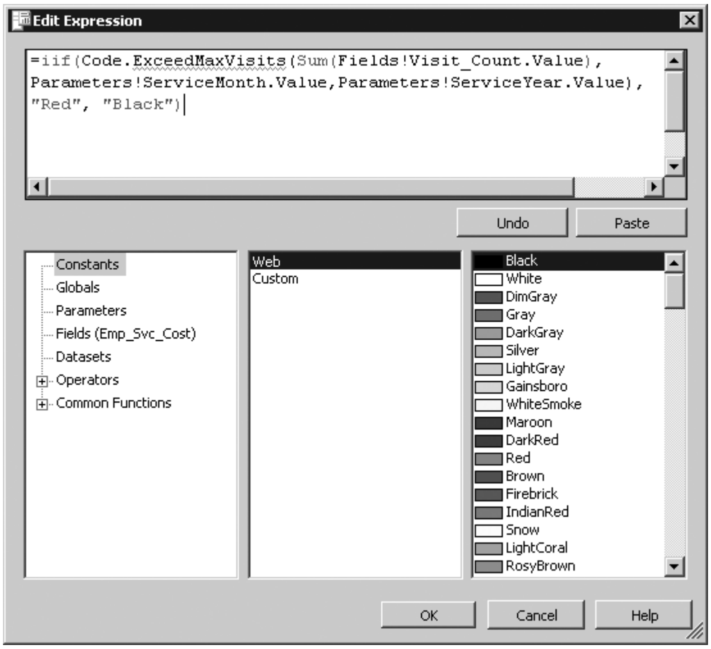
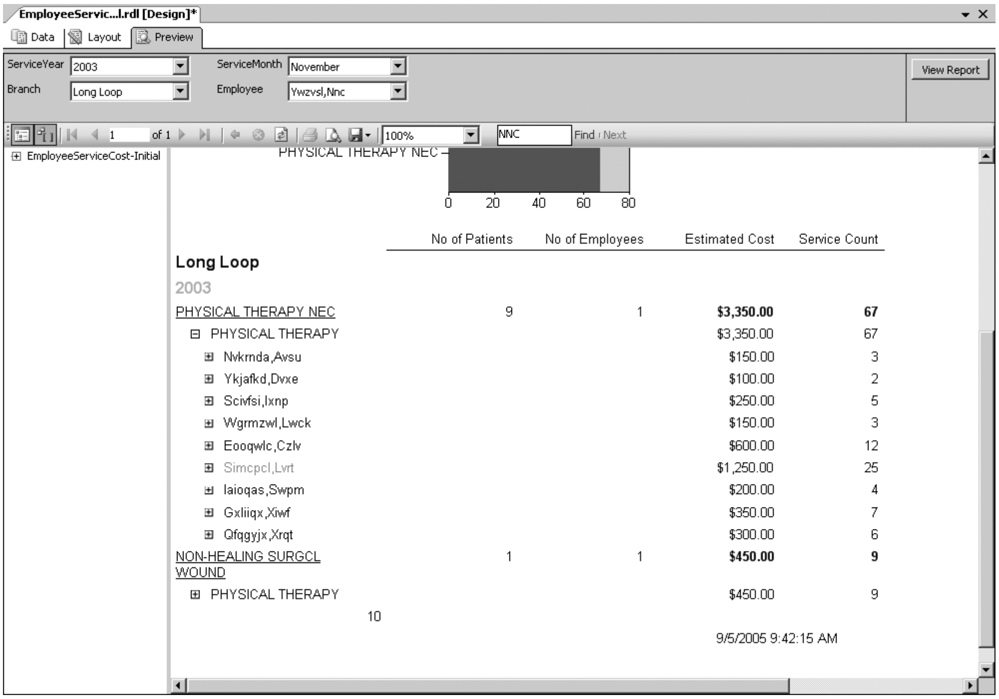

```
=iif(Code.ExceedMaxVisits(Sum(Fields!Visit_Count.Value),
Parameters!ServiceMonth.Value,Parameters!ServiceYear.Value),
"Red", "Black")
```

Figure 5-7. *Entering an expression in the expression editor*

You can now run your report, and the patient name will be displayed in red or black according to the business logic in the `Code` element of the report.

Now that you have modified the report to use the ExceedMaxVisits function, you can preview the report to see it in action. To do this, select the Preview tab, and select ServiceYear 2003, Branch Long Loop, Service Month November, and Employee Ywzvsl, Nnc. You should now see a report that looks similar to Figure 5-8.



ServiceYear 2003 · ServiceMonth November · Branch Long Loop · Employee Ywzvsl,Nnc

1 of 1 · 100% · NNC

PHYSICAL THERAPY NEC

	No of Patients	No of Employees	Estimated Cost	Service Count
Long Loop				
2003				
PHYSICAL THERAPY NEC	9	1	**$3,350.00**	**67**
⊟ PHYSICAL THERAPY			$3,350.00	67
⊞ Nvkrnda,Avsu			$150.00	3
⊞ Ykjafkd,Dvxe			$100.00	2
⊞ Scivfsi,Ixnp			$250.00	5
⊞ Wgrmzwl,Lwck			$150.00	3
⊞ Eooqwlc,Czlv			$600.00	12
⊞ Simcpcl,Lvrt			$1,250.00	25
⊞ Iaioqas,Swpm			$200.00	4
⊞ Gxliiqx,Xiwf			$350.00	7
⊞ Qfqgyjx,Xrqt			$300.00	6
NON-HEALING SURGCL WOUND	1	1	**$450.00**	**9**
⊞ PHYSICAL THERAPY			$450.00	9
	10			

9/5/2005 9:42:15 AM

Figure 5-8. *Report with the embedded code*

Accessing .NET Assemblies from Embedded Code

The Code element of the report was primarily designed for basic use of the .NET Framework and VB .NET language syntax. Access to many of the framework namespaces is not included by default in the Code element. Referencing many of the standard .NET assemblies in your embedded custom code requires that you create a reference to it in the report. To do this, go to the References tab of the Report Properties dialog box, click the ellipsis by the References: Assembly name grid, and then select the appropriate assembly you want to reference. Note that, by default, these referenced assemblies will have only Execution permission.

Although it is possible to use other .NET Framework assemblies and third-party assemblies directly within the Code element of the report, as just described, it is highly recommended that you consider using a custom assembly instead. One of the primary reasons for this is security. By default, the Code element runs with Execution permission only, which means it can run but cannot access protected resources. If you need to perform certain protected operations, such as reading data from a file, you'll have to set the security policy for the code group named Report_Expressions_Default_Permissions to FullTrust. This code group controls permissions for the report expression host assembly, which is an assembly that is created from all the expressions found within a report and is stored as part of the compiled report. To set the security policy, you need to edit the policy configuration files of the report server and the Report Designer. See the "Deploying a Custom Assembly" section later in this chapter for the standard location of these files.

But making this change to the security policy is not recommended. When you change the permissions for the code that runs in the `Code` element, you also change the permissions for all reports that run on that report server. By changing permissions to `FullTrust`, you enable all expressions used in reports to make protected system calls. This will essentially give anyone who can upload a report to your report server complete access to your system.

If you need to use features outside the VB .NET language syntax, need additional security permissions, have complicated logic to implement, need to use more of the .NET Framework, or want to use the same functionality within multiple reports, then you should move your code into a custom assembly. You can then reference that assembly in your report and use the code through methods and properties of your custom class. Not only does a custom assembly allow you a lot more flexibility in the code itself, it also allows you to control security at a much more granular level. With a custom assembly, you can add a permission set and code group for your custom code just to that specific assembly without having to modify the permissions for all code that runs in the `Code` element.

You'll want to use custom assemblies for another reason. With embedded code, you do not have the benefit of developing the `Code` section of your report using the full Visual Studio IDE with features such as IntelliSense and debugging at your disposal. Writing code in the `Code` section of your report is not much different from working in Notepad.

However, you can work around this. If the code you choose to place in the `Code` element is more than just a few simple lines of code, it can be easier to create a separate project within your report solution to write and test your code. A quick VB .NET Windows Forms or console project can provide the ideal way to write the code you intend to embed in your report. You get the full features of the IDE, and once you have the methods working the way you want, you can just paste them into the code window of the report. Remember to use a VB .NET project, since the `Code` element works only with code written in VB .NET.

Using Custom Assemblies with Your Report

Custom assemblies are harder to implement but offer you greater flexibility than embedded code. The process of creating them is a bit more involved because they are not part of the report's RDL and must be created outside the Report Designer. This also makes them more difficult to deploy because, unlike the embedded code, which becomes a part of the report's RDL, the custom assembly is a separate file.

However, your hard work is repaid in many ways:

Code reuse: You can reuse custom code across multiple reports without the need to copy and paste code into each report. This allows you to centralize all the custom logic into a single location, making code maintenance much simpler.

Task separation: Using assemblies allows you to more easily separate the tasks of writing a report from creating the custom code. This is somewhat similar in concept to writing an ASP.NET application using the code-behind feature. This allows ASP.NET developers to separate the page markup, layout, and graphics from the code that will interact with it. If you have several people involved, you can let those who specialize in report writing handle the layout and creation of the report while others who may have more coding skills write the custom code.

Language neutrality: You can use the .NET language of your choice. Choose from C#, VB, J#, or any third-party language that is compatible with the .NET Framework.

Productive development environment: If you use Visual Studio 2005 to develop your custom assemblies, you get the full power of its editing and debugging features.

Security control: You can exercise fine-grained control over what your assembly can do using security policies.

To use a custom assembly from within your report, you need to create a class library to hold the code, add the methods and properties you want to use from the report to your class, and then compile it into an assembly. To use it from within the Report Designer, you can add a class library project to the solution in Visual Studio, allowing you easy access to both the report and the code you will use in it. Note that you must be using Visual Studio 2005 and not just BIDS for this project. Before you run the included examples, make sure to read the `ReadMe.htm` file in the solution's `Solution Items` folder to see whether any steps are required before running the examples for your particular configuration.

Adding a Class Library Project to Your Reporting Solution

To use a custom assembly with your report, you will first need to write your custom code in the form of a .NET class. You can do this by adding a class library project to your existing solution so that you can work on the report and custom code at the same time.

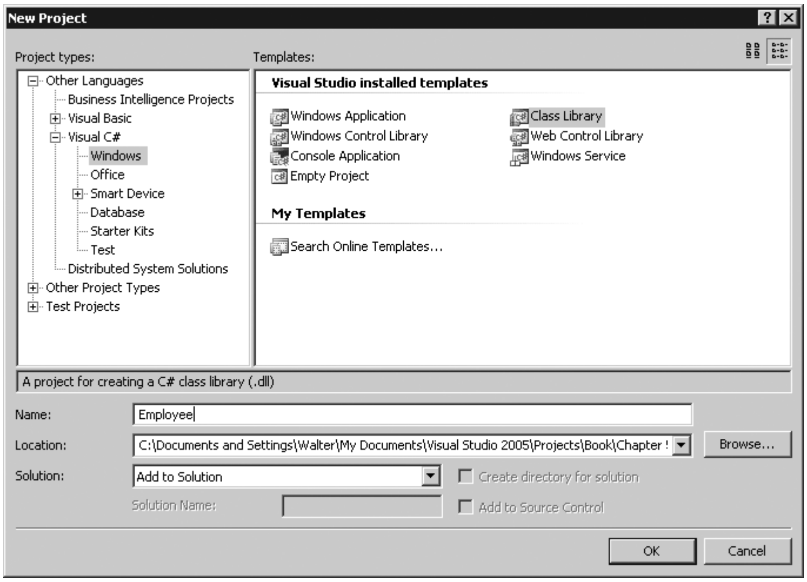
For this example, you want to display the amount an employee is paid for a visit to a patient. The class will get this information from an XML file that is periodically exported from the human resources (HR) system.

■ **Note** If possible, you would want to get this information directly from the HR system, possibly through a Web service. The sample code included with this chapter includes a sample Web service and a method to call it.

Using the XML file `EmployeePay.xml` (supplied as part of the code download for this chapter) in this example allows you not only to write a custom assembly but also to see the steps necessary to access a protected resource such as a local file. To get the information from the XML file and make it available to your report, create a class with a method that takes `EmployeeID` and a date as a parameter and that will read the employee pay per visit rates from the XML file and then return the pay rate. Although we will not cover it step by step in this chapter, the sample code included also contains an example of doing the same thing using a Web service. This allows you to simulate being able to interact with the HR system via a Web service instead of an exported file.

You can then reference the assembly from an expression in the report and use it to calculate the total visiting costs per patient.

To start, select File ➤ Add Project ➤ New Project from the menu. Pick Visual Basic Projects or Visual C# Projects, depending on your preference. Select Class Library, and enter **Employee** for the name of the project. In this example, we will show you how to use a Visual C# class library project, as shown in Figure 5-9.



A project for creating a C# class library (.dll)

Name: Employee

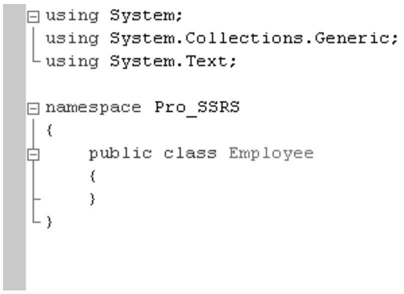
Location: C:\Documents and Settings\Walter\My Documents\Visual Studio 2005\Projects\Book\Chapter !

Solution: Add to Solution

Figure 5-9. *New Project dialog box*

■ **Note** Make sure to select Add to Solution instead of Create New Solution in the Solution drop-down list.

Select `Class1.cs`, and rename it to something a bit more descriptive, such as Employee, because you will use this class to calculate the cost of a visit provided by the employees. Open the `Employee.cs` file in the Visual Studio 2005 IDE, change the namespace from Employee to Pro_SSRS, and you will see the code editor, as shown in Figure 5-10.



```
using System;
using System.Collections.Generic;
using System.Text;

namespace Pro_SSRS
{
    public class Employee
    {
    }
}
```

Figure 5-10. *The Visual Studio 2005 code editor*

For this example, you'll add a few using statements to import types defined in other namespaces. Specifically, you'll add the System.Data and System.Security.Permissions namespaces so you can reference the DataSet and SecurityAction methods without typing in the full namespace in the Employee assembly, as shown in Listing 5-3.

Listing 5-3. *The* Employee *Assembly*

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Security.Permissions;
using System.Data;

namespace Pro_SSRS
{
    public class Employee
    {
        public Employee()
        {
        }
        [PermissionSetAttribute(SecurityAction.Assert, Unrestricted = true)]
        public static decimal CostPerVisitXML(string employeeID, DateTime visitDate)
        {

            DataSet empDS = new DataSet();
            empDS.ReadXmlSchema(@"C:\Temp\EmployeePay.xsd");
            empDS.ReadXml(@"C:\Temp\EmployeePay.xml");
            DataRow[] empRows =
                empDS.Tables["EmployeePay"].Select("EmployeeID = '" +
                employeeID + "'");
            Decimal empAmt;
            if (empRows.Length > 0)
            {
                empAmt = Convert.ToDecimal(empRows[0]["Amount"]);
                return empAmt;
            }
            else
                return 0M;

        }
    }
}
```

Any assemblies used by the custom assembly must be available both on the computer being used to design the report and on the SSRS 2005 server itself. Since you are just using common .NET Framework assemblies, this should not be a problem because the .NET Framework is installed on your local computer as well as on the SSRS 2005 server. If you reference other custom or third-party assemblies in your custom assembly, you need to make sure they are available on the SSRS 2005 server where you will be running your report.

■**Note** Because this book's focus is on SSRS 2005 and not on writing code, we won't explain the code samples line by line. If you are interested in programming, Apress offers many excellent books for the various programming languages that can help you write custom code for SSRS 2005. Refer to `http://www.apress.com`.

To use the `Employee` assembly in your report, you need to deploy it to the appropriate location first. In the next section, you will learn how to deploy custom assemblies and set up the necessary permissions required. Once you have done that, you will return to the report and use the custom assembly you have created and deployed in the Employee Service Cost report.

■**Note** Remember that each time you make a change to your custom assembly, you must redeploy the assembly. Also, if you added code that requires additional permissions, you may have to grant them.

Deploying a Custom Assembly

Custom assemblies are more difficult to deploy than code embedded in your report through the `Code` element. This is because of the following:

- Custom assemblies are not part of the report itself and must be deployed separately.
- Custom assemblies are not deployed to the same folder as the reports.
- The built-in project deployment method in Visual Studio 2005 will not automatically deploy your custom assemblies.
- Custom assemblies are granted only `Execution` permissions by default. `Execution` permission allows code to run but not to use protected resources.

To use your custom assemblies with SSRS 2005, you need to take the following steps to place them in a location where SSRS 2005 can find them and to edit the files that control security policy when necessary. The location of the files depends on whether you want to use them in the Report Designer within BIDS, within Visual Studio, or on the report server.

1. You need to deploy your custom assemblies to the Report Designer and/or SSRS 2005 applications folder.
   - For the Report Designer, the default is `C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\PrivateAssemblies`.
   - For SSRS 2005, the default is `C:\Program Files\Microsoft SQL Server\MSSQL.n\Reporting Services\ReportServer\bin`.

■**Note** You need to have the necessary permissions to access these folders. By default, members of the standard Users group won't have the necessary write/modify permissions on these folders. Logged in as a user with the appropriate security permissions, such as the administrator, you can set the permissions on the folders to allow the necessary access to the folders when you are logged in under a less privileged account. Alternately, you could move the files to the appropriate folders when logged in or running as a user who has the necessary permissions.

2. Next, you need to edit the SSRS 2005 security policy configuration files if your custom assembly requires permissions in addition to the `Execution` permission. (For SSRS 2005, the default location is `C:\Program Files\Microsoft SQL Server\MSSQL.n\Reporting Services\ReportServer\rssrvpolicy.config`.)

■**Note** The Report Designer runs custom assemblies with `FullTrust` security, so you may not encounter security-related issues when you are previewing the reports. However, should changes be required, the default location of the preview security configuration file is `C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\PrivateAssemblies\RSPreviewPolicy.config`.

■**Note** The `MSSQL.n` folder will vary depending on the particular installation options you selected when installing SQL Server 2005. It may be in a folder with a period and number appended to the end of *MSSQL.n* such as `MSSQL.3`. The `MSSQL` folder for the custom assembly and the security policy file will be the same.

For example, if you were writing a custom assembly to calculate an employee's cost per visit, you might need to read the pay rates from a file. To retrieve the rate information, you would need to grant additional security permissions to your custom assembly. To give your custom assembly `FullTrust` permission, you can add the XML text shown in Listing 5-4 to the appropriate `CodeGroup` section of the `rssrvpolicy.config` file.

Listing 5-4. *Granting* `FullTrust` *Permission to the Custom Assembly*

```
<CodeGroup class="UnionCodeGroup"
    version="1"
    PermissionSetName="FullTrust"
    Name="EmployeePayCodeGroup"
    Description="Employee Cost Per Visit">
    <IMembershipCondition
        class="UrlMembershipCondition"
        version="1"
        Url="C:\Program Files\Microsoft SQL Server\MSSQL.n\ ➥
```

```
            Reporting Services\ReportServer\bin\Employee.dll"
        />
</CodeGroup>
```

■Note If you run your report and you see `#Error` text in a textbox instead of the expected result, it is more than likely a permission problem of some kind.

Because it's generally not a good idea to grant your assemblies `FullTrust` unless absolutely necessary, you can use named permission sets to grant your custom assembly just the permissions it needs rather than `FullTrust`.

To grant the custom assembly just enough permission to read the data files called `C:\Temp\EmployeePay.xml` and `C:\Temp\EmployeePay.xsd`, you first need to add a named permission set in the policy configuration file `rssrvpolicy.config` that grants read permission to the files. You can then apply the specific permission sets to the custom assembly, as shown in Listing 5-5.

Listing 5-5. *Named Permission Sets for Reading Files*

```
<PermissionSet
    class="NamedPermissionSet"
    version="1"
    Name="EmployeePayFilePermissionSet"
    Description="Permission set that grants read access to my employee cost file.">
    <IPermission
        class="FileIOPermission"
        version="1"
        Read="C:\Temp\EmployeePay.xml"
    />
    <IPermission
        class="FileIOPermission"
        version="1"
        Read="C:\Temp\EmployeePay.xsd"
    />
    <IPermission
        class="SecurityPermission"
        version="1"
        Flags="Execution, Assertion"
    />
</PermissionSet>
```

Next, as shown in Listing 5-6, you add a code group that grants the assembly the additional permissions to the `CodeGroup` section of the policy configuration file `rssrvpolicy.config`.

Listing 5-6. *Granting File I/O Permission on the* `Employee` *Assembly*

```
<CodeGroup class="UnionCodeGroup"
   version="1"
   PermissionSetName=" EmployeePayFilePermissionSet "
   Name="EmployeePayCodeGroup"
   Description="Employee Cost Per Visit">
   <IMembershipCondition class="UrlMembershipCondition"
      version="1"
      Url="C:\Program Files\Microsoft SQL Server\MSSQL.n\Reporting ➥
Services\ReportServer\bin\Employee.dll"/>
</CodeGroup>
```

■Note The name of the assembly that you add to the configuration file must match the name that is added to the RDL under the `CodeModules` element. This is the name you set for the custom assembly under the Report Properties ➤ References menu, which was introduced in the "Accessing .NET Assemblies from Embedded Code" section; it is discussed in detail in the "Adding an Assembly Reference to a Report" section.

To apply custom permissions, you must also assert the permission within your code. For example, if you want to add read-only access to the XML files `C:\Temp\EmployeePay.xsd` and `C:\Temp\EmployeePay.xml`, you must add code similar to that shown in Listing 5-7 to your method.

Listing 5-7. *Asserting Permission with Code*

```
// C#
FileIOPermission permissionXSD = new
    FileIOPermission(FileIOPermissionAccess.Read,
    @" C:\Temp\EmployeePay.xml");
    permissionXSD.Assert();
    // Load the schema file
    empDS.ReadXmlSchema(@"C:\Temp\EmployeePay.xsd");

FileIOPermission permissionXML = new
    FileIOPermission(FileIOPermissionAccess.Read,
    @" C:\Temp\EmployeePay.xml");
    permissionXML.Assert();
    empDS.ReadXml(@"C:\Temp\EmployeePay.xml");
```

You can also add the assertion as a method attribute, as shown in Listing 5-8. This is the method shown in this chapter's examples.

Listing 5-8. *Asserting Permission with a Method Attribute*

```
[FileIOPermissionAttribute(SecurityAction.Assert,
    Read=@" C:\Temp\EmployeePay.xsd")]
[FileIOPermissionAttribute(SecurityAction.Assert,
    Read=@" C:\Temp\EmployeePay.xml")]
```

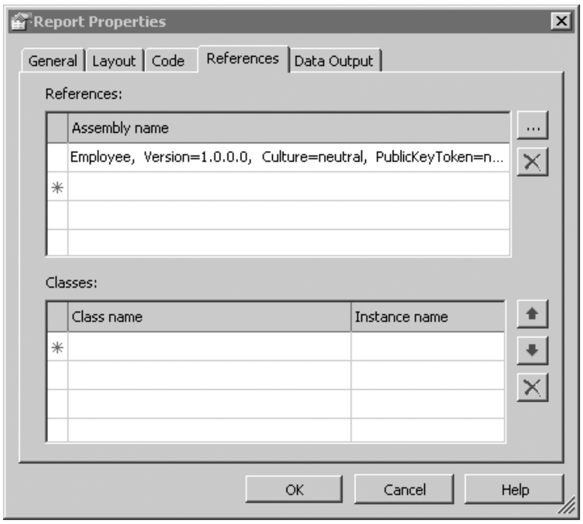
■ **Tip** For more information about code access security and reporting services, see "Understanding Code Access Security in Reporting Services" in the SSRS 2005 Books Online (BOL). For more information about security, see ".NET Framework Security" in the *.NET Framework Developer's Guide*, available on the Microsoft Developer Network (MSDN) Web site at `http://msdn.microsoft.com`. You will also want to read about using the Global Assembly Cache (GAC) to store your custom assembly.

Adding an Assembly Reference to a Report

With the `EmployeeServiceCost-NoCode` report selected and on the Layout tab, select Report ➤ Report Properties; alternatively, right-click within the report design area, and select Properties. Then do the following:

1. In References, click the ellipsis button.

2. Select the Browse tab, and browse to the `Employee.dll` assembly from the Add Reference dialog box. When you are done, the Report Properties dialog box should look like Figure 5-11.



References:

	Assembly name
	Employee, Version=1.0.0.0, Culture=neutral, PublicKeyToken=n...
*	

Classes:

	Class name	Instance name
*		

Figure 5-11. *References tab*

Note The class list on the References tab of the Report Properties dialog box is used only by instance-based members, not static members.

To use the custom code in your assembly in a report expression, you must call a member of a class within the assembly. You can do this in different ways depending on how you declared the method.

If the method is defined as static, it is available globally within the report. You access it in an expression by the namespace, class, and method name. The following example calls the static `CostPerVisit` method in the `Employee` class, which is in the `Pro_SSRS` namespace, passing in an `EmployeeID` value and the visit date. The method will return the cost per visit for the specified employee.

```
=Pro_SSRS.Employee.CostPerVisitXML(empID, visitDate)
```

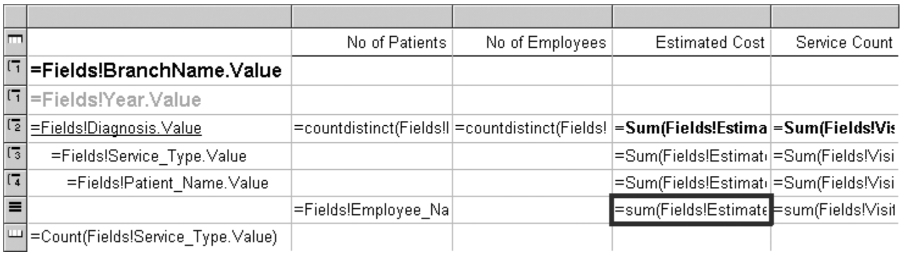
If the custom assembly contains instance methods, you must add the class and instance name information to the report references. You do not need to add this information for static methods.

Instance-based methods are available through the globally defined `Code` member. You access these methods by referring to the `Code` member and then the instance and method name. The following shows how you would call the `CostPerVisitXML` method if it had been declared as an instance method instead of a static method:

```
=Code.Employee.CostPerVisitXML(empID, visitDate)
```

Tip Use static methods whenever possible because they offer higher performance than instance methods. However, be careful if you use static fields and properties, because they expose their data to all instances of the same report, making it possible that the data used by one user running a report is exposed to another user running the same report.

After adding the reference to the `Employee` custom assembly, you will use it by calling the `CostPerVisitXML` method as a part of an expression in the report. Highlight the `Employee_Cost` textbox in the report, as shown in Figure 5-12.



	No of Patients	No of Employees	Estimated Cost	Service Count
=Fields!BranchName.Value				
=Fields!Year.Value				
=Fields!Diagnosis.Value	=countdistinct(Fields!I	=countdistinct(Fields!	**=Sum(Fields!Estima**	**=Sum(Fields!Vis**
=Fields!Service_Type.Value			=Sum(Fields!Estimat	=Sum(Fields!Visi
=Fields!Patient_Name.Value			=Sum(Fields!Estimat	=Sum(Fields!Visi
	=Fields!Employee_Na		=sum(Fields!Estimate	=sum(Fields!Visit
=Count(Fields!Service_Type.Value)				

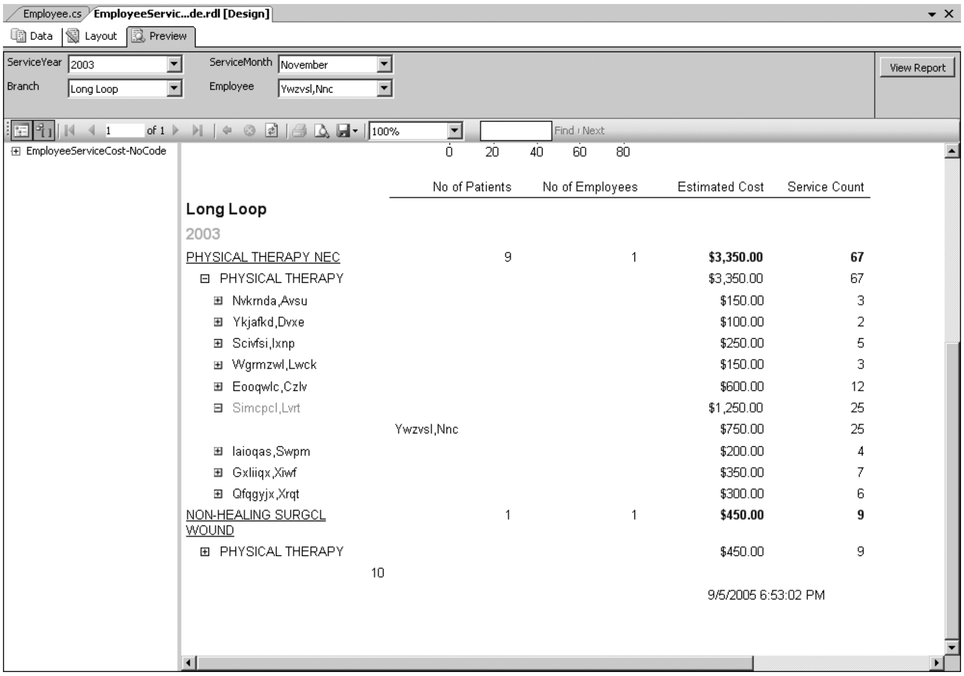
Figure 5-12. `Employee_Cost` *textbox*

Right-click, select Expression, and in the Edit Expression dialog box enter the code shown in Listing 5-9.

Listing 5-9. *Using the* `CostPerVisitXML` *Method in an Expression*

```
=Pro_SSRS.Employee.CostPerVisitXML(Fields!EmployeeID.Value,
"01/01/2004") * sum(Fields!Visit_Count.Value)
```

Now if you preview the report or build and deploy it, you should see a report similar to Figure 5-13.



Employee.cs | EmployeeServic...de.rdl [Design]

Data | Layout | Preview

ServiceYear: 2003 ServiceMonth: November
Branch: Long Loop Employee: Ywzvsl,Nnc

View Report

1 of 1 100% Find | Next

EmployeeServiceCost-NoCode

0 20 40 60 80

	No of Patients	No of Employees	Estimated Cost	Service Count
Long Loop				
2003				
PHYSICAL THERAPY NEC	9	1	**$3,350.00**	**67**
⊟ PHYSICAL THERAPY			$3,350.00	67
⊞ Nvkrnda,Avsu			$150.00	3
⊞ Ykjafkd,Dvxe			$100.00	2
⊞ Scivfsi,Ixnp			$250.00	5
⊞ Wgrmzwl,Lwck			$150.00	3
⊞ Eooqwlc,Czlv			$600.00	12
⊟ Simcpcl,Lvrt			$1,250.00	25
Ywzvsl,Nnc			$750.00	25
⊞ Iaioqas,Swpm			$200.00	4
⊞ Gxliiqx,Xiwf			$350.00	7
⊞ Qfqgyjx,Xrqt			$300.00	6
NON-HEALING SURGCL WOUND	1	1	**$450.00**	**9**
⊞ PHYSICAL THERAPY			$450.00	9
10				

9/5/2005 6:53:02 PM

Figure 5-13. *Final report*

In the sample code included for Chapter 5, we have also included a Web service that can be called from the custom code to access the employee pay information. This simulates accessing information from another system via a Web service and is designed to allow you to replace the exported XML file with a call to a Web service. The `Employee` class included in the sample code already contains a method called `CostPerVisitWS` that uses the Web service rather than the XML file as its source of data. By simply changing the expression from this:

```
=Pro_SSRS.Employee.CostPerVisitXML(Fields!EmployeeID.Value,
"01/01/2004") * sum(Fields!Visit_Count.Value)
```

to the following:

```
=Pro_SSRS.Employee.CostPerVisitWS(Fields!EmployeeID.Value,
"01/01/2004") * sum(Fields!Visit_Count.Value)
```

you can make the report use the Web service instead of the XML file. To try this, just open the sample solution for this chapter, and edit the expression in the `Employee_Cost` field in the `EmployeeServiceCost.rdl` file. To try the Employee Web Service, you also must publish the Web site by selecting the Employee Web Service project, right-clicking, and then selecting Publish Web Site.

We have also included a sample test application that allows you to call the `CostPerVisitXML` and `CostPerVisitWS` methods of the `Employee` class using a Windows Forms application. This allows you to exercise the class and step through the code in the Windows Forms environment, which is easier to test and debug.

Tip Writing a test application is a great way to make sure your custom code is properly performing the expected functions prior to using the code within your SSRS 2005 report. Not only can you create a custom Windows Forms application as we did, but with the proper version of Visual Studio 2005 you can create specialized test code and automated test routines.

Debugging Custom Assemblies

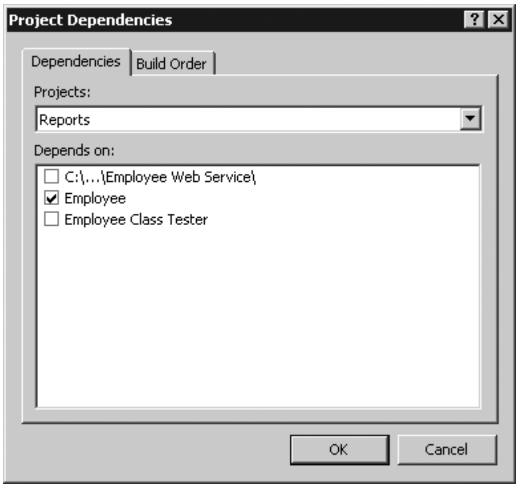
For ease of debugging, the recommended way to design, develop, and test custom assemblies is to create a solution that contains both your test reports and your custom assembly. This will allow you easy access to both the report and the code you will use in it at the same time from within Visual Studio.

Note The Report Preview that is used during debugging does not grant `FullTrust` security to your custom assembly and may require that you set up appropriate permissions to run your custom assembly. In your development environment, you may want to grant it `FullTrust`, which you can do by editing the `RSPreviewPolicy.config` file. The default location of the preview security configuration file is `C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\PrivateAssemblies\RSPreviewPolicy.config`. You can use the code shown in Listing 5-4 to grant `FullTrust` if you just change the URL element to point to the `Employee.dll` in the `PrivateAssemblies` folder.

We'll now show how to set up Visual Studio to allow you to debug the `Employee` assembly you have just written:

1. In the Solution Explorer, right-click the solution, and select Configuration Manager. This will allow you to set the build and deploy options for debugging.
2. Select `DebugLocal` as the Active Solution Configuration option.
3. Make sure the report project in your solution is set to `DebugLocal` and that `Deploy` is unchecked. `DebugLocal` debugs reports on your local system, rendering the report in the Preview pane within Visual Studio. If `Deploy` is checked, it will publish the reports to the report server instead of running them locally, and you will not be able to debug them.

4. Right-click the project containing your reports.
5. Set the `Reports` project as the startup project. This will make sure the report runs first when you start debugging. You will set the specific report that will call the custom assembly in a subsequent step.
6. Right-click again, and select Project Dependencies.
7. In the Project Dependencies dialog box, select the Employee project as the dependent project, as shown in Figure 5-14. This will tell Visual Studio that your report depends on the custom assembly you have written.



Projects: Reports

Depends on:
- [ ] C:\...\Employee Web Service\
- [✓] Employee
- [ ] Employee Class Tester

Figure 5-14. *Project Dependencies dialog box*

8. Click OK to save the changes, and close the Project Dependencies dialog box.
9. Right-click the `Reports` project again, and select Project Properties.
10. Select StartItem, and set it to the report you want to debug. In this case, select `EmployeeServiceCost.rdl`. The StartItem option tells Visual Studio specifically which report to run when you run with debugging.
11. In Solution Explorer, select the Employee custom assembly project.
12. Right-click, and select Properties.
13. Expand Configuration Properties, and click Build.
14. On the Build page, enter the path to the Report Designer folder in the Output Path textbox. (By default, this path is `C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\PrivateAssemblies`.)

■**Note** You will need to have the necessary permissions to this folder in order to use it for the output path. By default, members of the standard Users group won't have the necessary write/modify permissions on this folder. When you're logged in as a user with appropriate security permissions, such as the administrator, make sure to set the permissions on the `PrivateAssemblies` folder to allow the necessary access when you are logged in under a less privileged account.

■**Note** Debugging requires that you set the permissions.

■**Tip** You could leave the default output path, but changing it saves you some work. With the default path, you'd have to build and then manually copy your custom assembly in order for the Report Designer running within Visual Studio to find it and run it. If you update the `Employee` class, you may find that Visual Studio report preview has a copy of the old version in memory that will prevent your solution from being deployed. If this occurs, exit Visual Studio and start it again.

15. Now set breakpoints in your custom assembly code.
16. Make sure to set Report as the startup project, and then press F5 to start the solution in debug mode. When the report uses the custom code in your expression, the debugger will stop at any breakpoints you have set when they are executed. Now you can use all the powerful debugging features of Visual Studio to debug your code.

■**Note** It is also possible to use multiple copies of Visual Studio to debug your custom assembly. See the SSRS 2005 BOL for details.

Troubleshooting Your Project

If you modify a custom assembly and rebuild it, you must redeploy it, because the Report Designer looks for it only in the Report Designer application folder. If you followed our suggestion in the "Debugging Custom Assemblies" section to change the output path, it should be in the correct location each time you rebuild it while debugging. If not, you will need to follow the instructions in the "Deploying a Custom Assembly" section to move it to the Report Designer application folder. Remember, Visual Studio will not deploy your custom assembly to your SSRS 2005 server machine; you must copy it manually.

You may find that you have to exit the Visual Studio IDE in order to replace the files, as they may otherwise be in use.

Finally, you may want to keep the version of any custom assembly the same, at least while you are developing it. Every time you change the version of a custom assembly, the reference to it must change on the References tab of the Report Properties dialog box, as discussed earlier in this chapter. Once your reports are in production where you want to keep track of version information, you can use the GAC, which can hold multiple versions; this means you have to redeploy only those reports that use the new features of the new version. If you want all the reports to use the new version, you can set the binding redirect so that all requests for the old assembly are sent to the new assembly. You would need to modify the report server's `Web.config` file and `ReportService.exe.config` file.

If you are using a custom assembly and the output on your report shows `#Error`, you likely have encountered a permissions issue. See the "Deploying a Custom Assembly" section in this chapter for information on how to properly set up the permissions.

Summary

In this chapter, you learned how to use custom code with your reports, and we discussed some of the other programmatic aspects of dealing with SSRS 2005. Chapters 6, 7, and 8 will build on this as we show you how to write custom applications to render reports, deploy them to the report server, and schedule them to run using subscriptions.