# Chapter 5

# Using Custom .Net Code with Reports

SQL Reporting Services offers the software developer a variety of options when it comes to customizing reports through the use of code. These options give the software developer the ability to write custom functions using .NET code that can interact with report fields, parameters and filters in much the same was as any of the functions that come "built in". This gives the developer the ability to extend the capabilities of SRS far beyond those that are available out of the box.

This chapter will cover the following:

* Using code to customize your report
* Custom code for use within your report using code embedded in the report
* Custom code for use within your report using a custom assembly called by the report

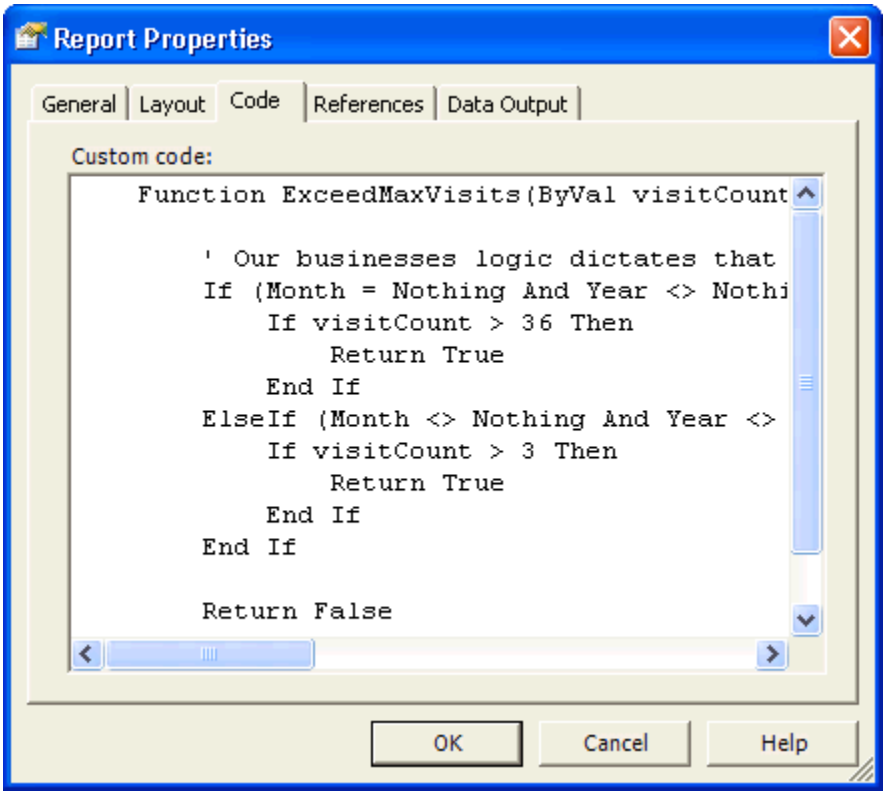
There are two ways to add custom code to your reports, embedded code and code within custom assemblies. This code can then be used in the report through expressions. Generally you will add custom code to your report when you need to perform complex functions and you need the capabilities of a full programming language to accomplish them.

.

## Embedded Code in Your Report

Embedded code is by far the easiest method to implement for two main reasons. First, you simply add the code directly into the report using the report designers UI. Second, this code becomes a segment within the reports rdl file making its deployment simple because it is a part of your report and will be deployed with it.

To add code to our Employee Service Cost report to to the Report menu, click Report Properties or right-click within the report design area. On the Code tab in Custom Code textbox, type the code you want included in your report, as shown in Figure 5-1.




*Insert Figure 0341f0501.tif*

Figure 5-1. Embedded Entering code in the code editor

Listing 5-1 is the full listing of the code that we are going to add to our report. It a simple example of custom business logic that we are going to use to determine if a patient has exceeded a certain number of visits over some period of time that are dictated by our company's business needs.

Listing 5-1. The code for our report

```
Function ExceedMaxVisits(ByVal visitCount As Integer, ByVal Month As Integer, ByVal
Year As Integer) As Boolean


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

```
    Return False
End Function
```

Now that we have our custom code defined, we want to use it to highlight the patients within the report that have exceeded the maximum visit count. Methods in embedded code are available through a globally defined Code member. You access these by referring to the Code member and method name.

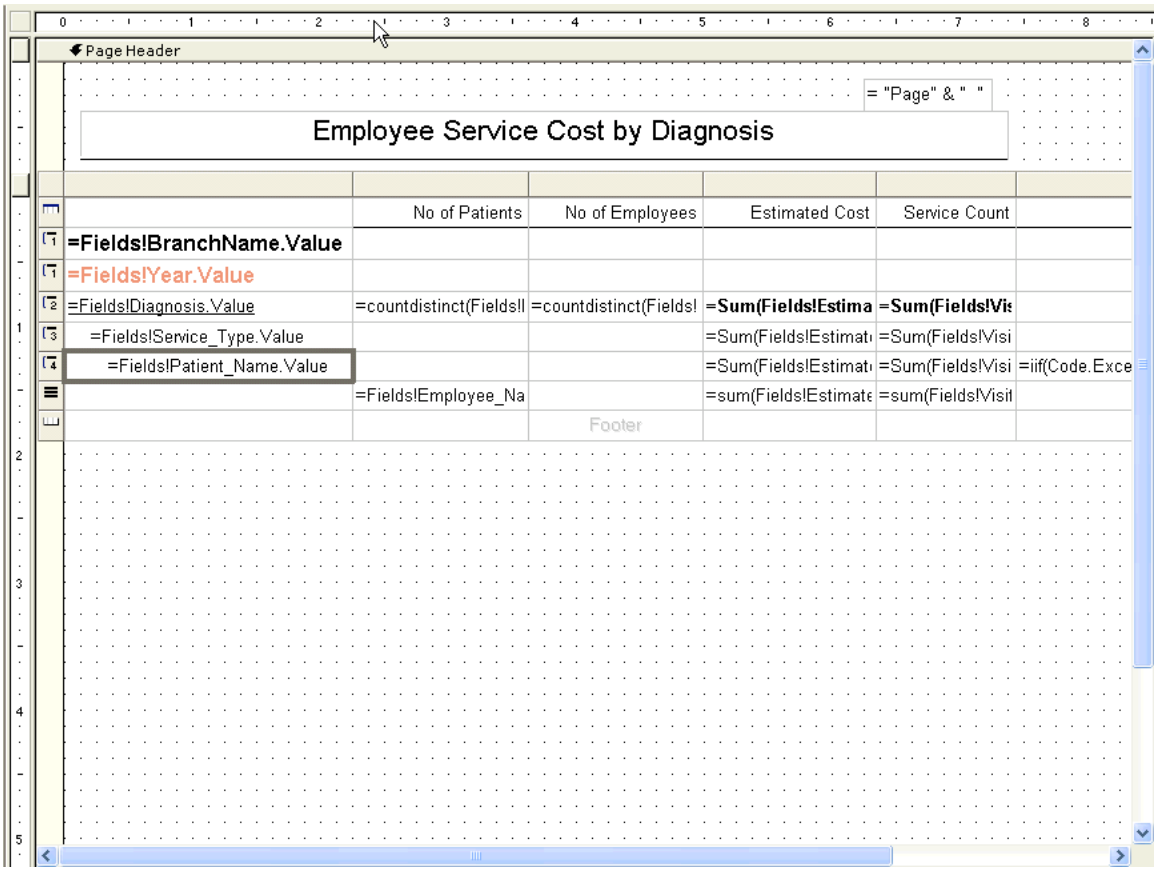
Listing 5-2 shows the use of a conditional expression in the Color property of a textbox to determine the color of the text depending on return value of our method call. The method ExceedMaxVisits determines if the patient has had more visits in the time span than allowed and returns true if so and false if not. Using a Boolean return value makes it very easy to use in formatting expressions, because the return value can be tested directly instead of comparing the returned value to another value.

When we exceed 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.

Listing 5-2. Using a conditional expression

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

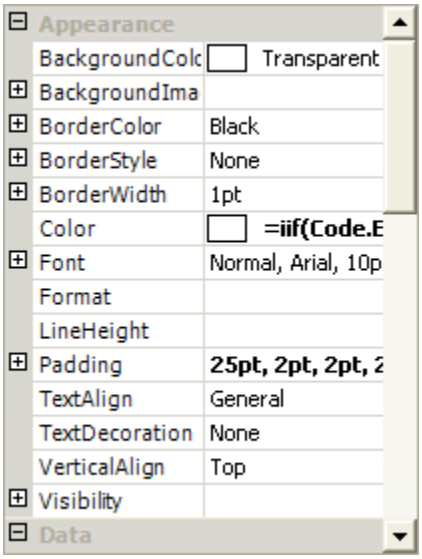
Now let's walk through how to actually add this expression to our report. First select the field in the report you want to apply the expression to. In this case, we are going to select the patient name textbox, as shown in Figure 5-2.




*Insert Figure 0341f0502.tif*

Figure 5-2. Adding the expression to our report

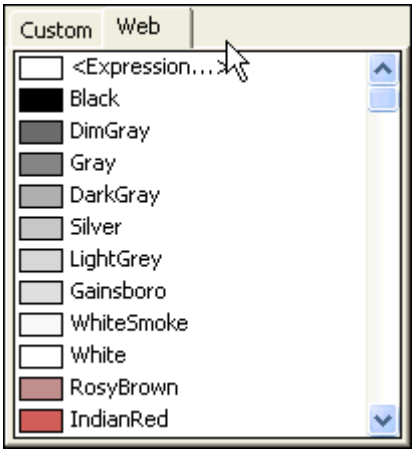
With the textbox selected, go to the Properties dialog box and select the Color property (see Figure 5-3).



*Insert Figure 0341f0503.tif*

Figure 5-3. Field Property Dialog

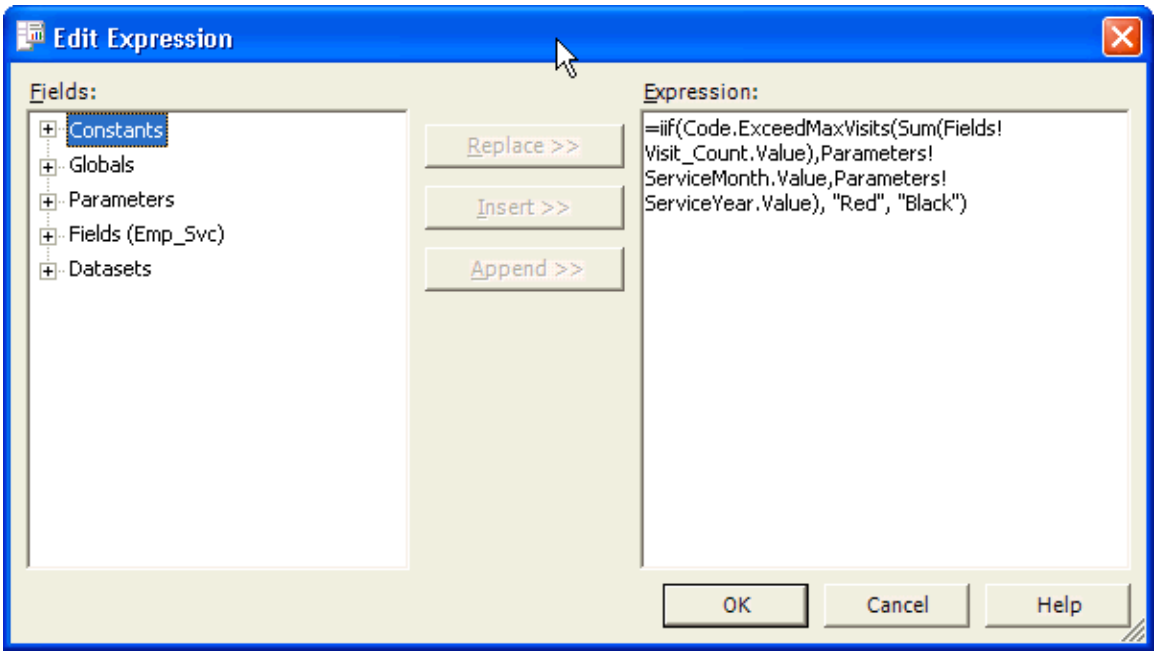
Select Expression from the drop-down list in the color property value (see Figure 5-4).




*Insert Figure 0341f0504.tif*

Figure 5-4. Color Property Dialog

Now you will see the Expression Dialog box, shown in Figure 5-5. Enter the expression using your custom code here. You can just type the expression in or you can use the features of the Expression Editor to help by using it to insert the parameters that you need into your expression.



*Insert Figure 0341f0505.tif*

Figure 5-5. Entering an expression in Expression Editor

You can now run your report and the Patient Name field will be displayed using Red or Black according to our business logic in the Code element of the report.

> Note: The embedded code can contain multiple methods. Embedded code must be written in Visual Basic .NET and must be instance based.
>
> Note: Referencing many of the standard .NET assemblies in your custom code requires that you create a reference in the report. To do this, go to the References Tab of the Report Properties, click on the ellipses by the References: Assembly name grid and then select the appropriate assembly that you want to reference. These will only have Execution permission by default.

The Code section of the report was primarily designed for basic use of the .NET framework and the VB.NET language syntax itself. Access to many framework namespaces is not included by default in the Code element. While it is possible to use other .NET Framework assemblies directly within the Code section of the report, it is highly recommended that you consider using a custom assembly.

One of the primary reasons for this is Security. By default the Code element runs with Execute permission only, so if you perform certain operations such as reading data from a file, you will need to modify the code group for the report expression host assembly. When you change the permissions for the code that runs in the Code element it changes it for all reports that run on that reporting server.

If you need to use features outside of the VB.NET language syntax, have more complicated logic or need to use more of the .NET Framework you should move your code into a custom assembly. You can then reference that assembly in your report and call the assembly through a method in your custom class. This way, you can add the permission set and code group for your custom code without having to modify the permissions for all code that runs in the Code element.

Another reason is that in SRS you do not have the benefit of developing the Code section of your report using the full Visual Studio IDE with all of the features such as IntelliSense and Debugging features at your disposal. Writing code in the Code section of your report is not much different than working in NotePad.

> Tip: If the code you do 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 WinForm or Console project can provide the ideal way to write code you intend to embed in your report. You get the full features of the IDE and once you have the method or 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 only works with code written in VB.NET.

# Using Custom Assemblies

Using custom assemblies is harder to implement but offers you greater flexibility. It is more difficult to use custom assemblies because you must create them outside of the report designer and deploy them to the report server separately from the report. To use it from within the report designer you must make it available to the report designer by adding a reference to the assembly within the report itself.

The main benefit of using assemblies for your custom code is that the custom code can be used across multiple different reports without the need to copy and paste code into each report. This allows you to centralize all of the custom logic into a single location making code maintenance much simpler. This also allows you to more easily separate the tasks of writing a report from the creation of the custom code.

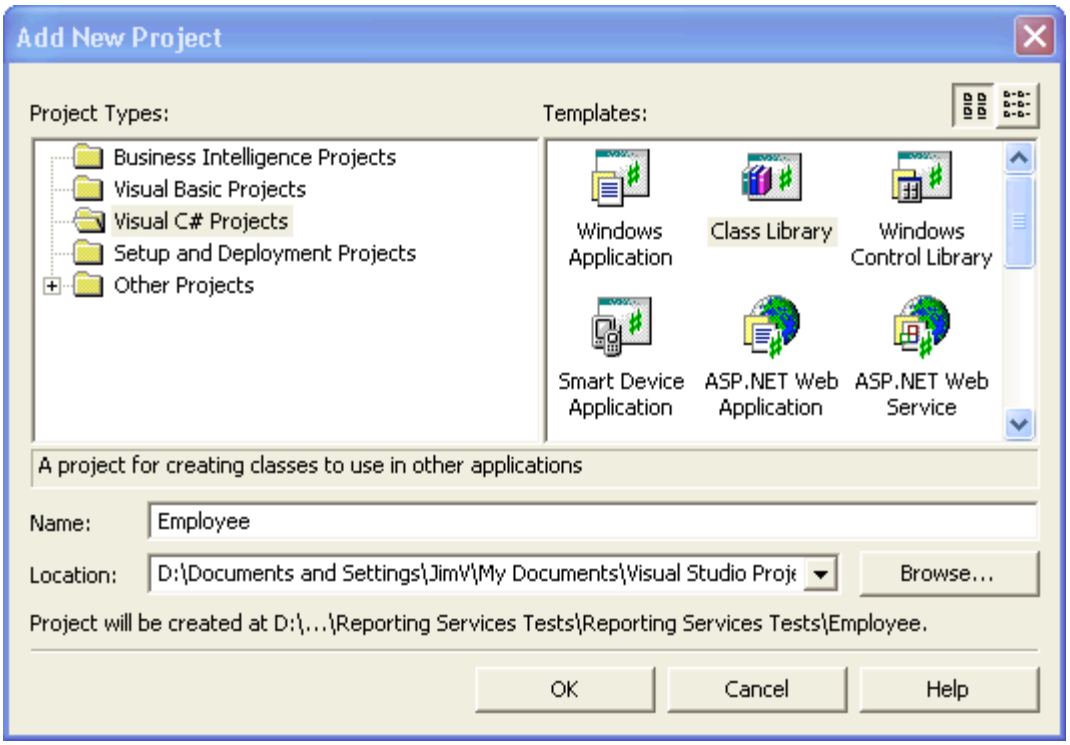
This is somewhat similar in concept to laying out an ASP.NET web page using code behind. If you have several people involved you can let those that specialize in report writing handle the layout and creation of the report while others that may have more coding skills write the custom code.

Another benefit of using a custom assembly is that you can use the .NET language of your choice. Choose from C#, VB.NET, J# or any third part language that is compatible with the .NET Framework and can create compatible assemblies.

## *Adding a Class Library Project to your Reporting Solution*

First you will 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 SRS project solution so that you can work on the report and the custom code at the same time in the same solution.

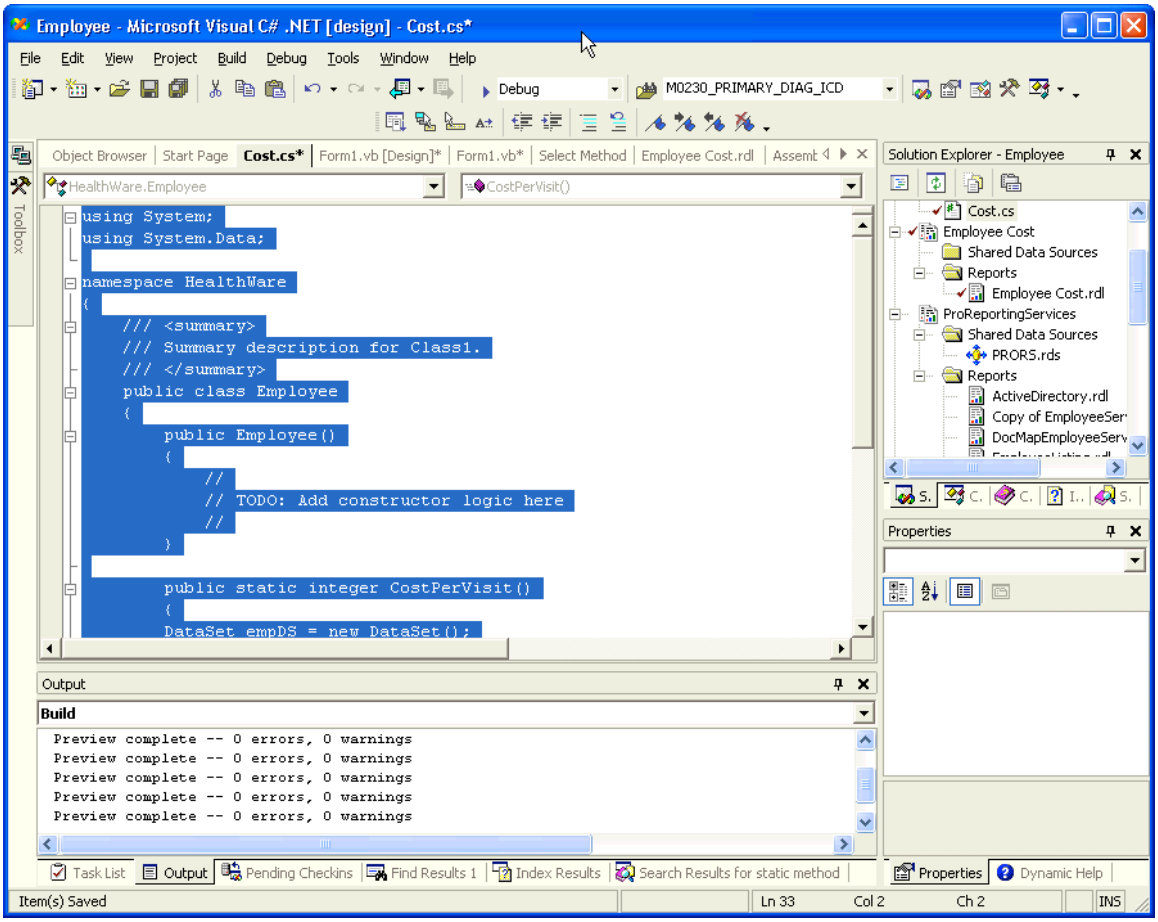
Select File, Add Project, New Project from the menu. Pick Visual Basic Projects or Visual C# Projects, select Class Library, and enter the name of Employee for the name of the Project. In Figure 5-6, we are using a Visual C# Class Library Project.




***Insert Figure 0341f0506.tif***

Figure 5-6. Add New Project Dialog

Select the Class1.cs and let's rename is to something a bit more descriptive such as Cost because we are going to use this class to calculate the cost associated with our employees. Open the Cost.cs file in the Visual Studio.NET IDE and you will see the code editor in Figure 5-7.




*Insert Figure 0341f0507.tif*

Figure 5-7. The Visual Studio.NET 2003 code editor

In our example, we are going to select the System.Data and System.Xml assemblies so we can reference the DataSet and LoadXml methods of the Dataset in our Custom Code. These assemblies must be on both the computer being used to design the report as well as the SQL Reporting Server itself. Since we are just using common .NET Framework assemblies this should not be a problem because the .NET Framework is installed on our local computer as well as on the SRS server. If you reference other custom or third party assemblies in your custom assembly you will need to make sure that they are available on the SRS server where you will be running your report.

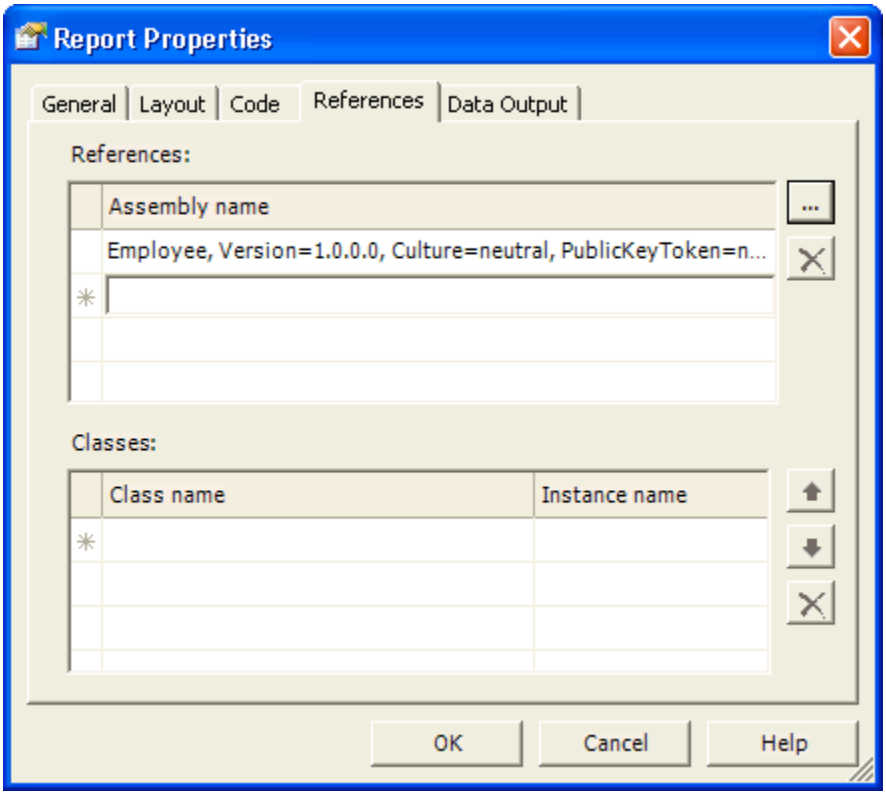
```
using System;
using System.Data;
using System.Xml;

namespace HealthWare
{
   /// <summary>
   /// Summary description for Class1.
   /// </summary>
```

```
public class Employee
{
   public Employee()
   {
   //
   // TODO: Add constructor logic here
   //
   }
   public static integer CostPerVisit()
   {
      DataSet empDS = new DataSet();
      empDS.ReadXml(@"D:\Temp\EmployeeCost.xml");
      DataRow[] empRows = empDS.Tables["Employee"].
         Select("EmployeeID = " & empID);
      Decimal empAmt;
      if (empRows.Length > 0)
      {
         empAmt =  Convert.ToDecimal(empRows(0)("EmployeePayPerVisit"));
         return empAmt;
      }
      else
         return 10;
    }
  }
}
```

## *Adding an Assembly Reference to a Report*

On the Report menu, click Report Properties or click within the report design area. On the References tab, do the following:

1. In References, click the Add (...) button and then select or browse to the assembly from the Add References dialog box.
2. In Classes, type name of the class and for instance based members provide an instance name to use within the report. If you use static members, then you will not need to specify them in the instance list since they are accessed through the globally available Code member as shown in Figure 5-8.




*Insert Figure 0341f0508.tif*

Figure 5-8. References Dialog

To use your custom code in your assembly in a report expression, you must call the member of a class within the assembly. This will be done differently depending on how the method was declared.

If the method is defined as static it is available globally within the report. You access it by namespace, class, and method name. The following example calls the method CostPerVisit that was declared as a static method. This example calculates the cost of a visit for a particular employee based on the employee ID passed to it:

```
=HealthWare.Employee.CostPerVisit(empID)
```

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 example calls the same CostPerVisit method but this time it was not declared static and so it uses the instance based calling convention.

```
=Code.rptHealthWare.Employee.CostPerVisit(empID)
```

Note: Use static methods whenever possible because they offer higher performance than instance methods. However be careful if you use static fields

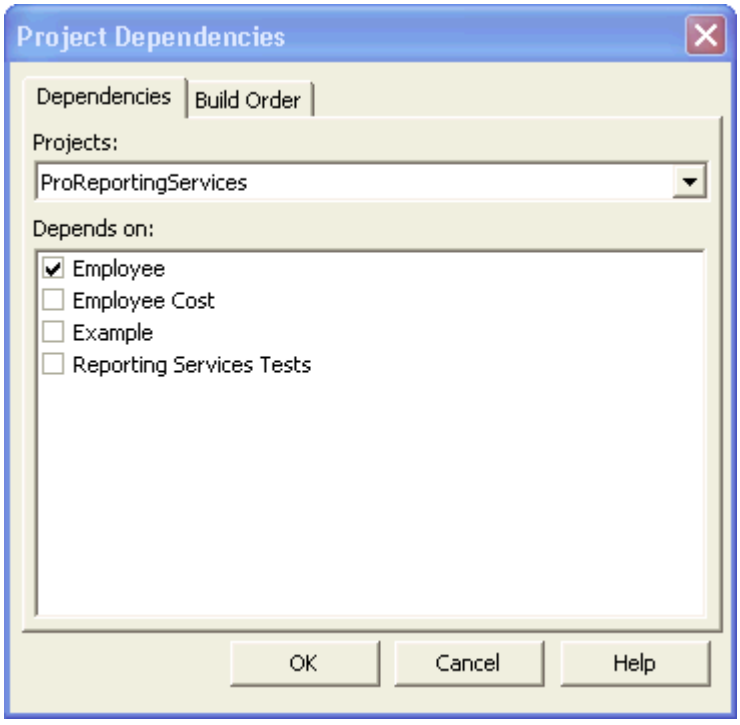
and properties because they expose their data to all instances of the same report making it possible that the data used by one user running a report is exposed to another user running the same report.

Let's take a look at how we are going to use our custom assembly in the report we have created.

## Debugging Custom Assemblies

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 is the way we have done it in our example.

1. In the Solution Explorer Right click on the Solution and Select Configuration Manager.
2. Select DebugLocal as the Active Solution Configuration
3. Make sure that the report project in your solution is set to DebugLocal and that deploy is unchecked.
4. Right click on the project containing your reports
5. Set the Report project as the startup project.
6. Right click again and select Project Dependencies.
7. In the Project Dependencies dialog select the Employee project as the dependent project as shown in Figure 5-8.




*Insert Figure 0341f0509.tif*

8. Click OK to save the changes, and close the Property Pages dialog.

9. Right click again and select Project Properties.
10. Select StartItem and set it to the report you want to debug.
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 (by default, C:\Program Files\Microsoft SQL Server\80\Tools\Report Designer) in the Output Path text box.

    Tip: You could leave the default output path but this saves you the step of building and then manually copying your custom assembly to the folder required in order for Reporting Services Designer to find it and run it.
15. Now set breakpoints in your custom assembly code.
16. Make sure to set the Report as the startup project and then press F5 to start the solution in debug mode. When the report uses the custom code in our expression, the debugger will stop at any breakpoints that you have set when they are executed. Now you can use the all the powerful debugging features of Visual Studio.NET to debug your code.

    Note: It is also possible to use multiple copies of Visual Studio.NET and then from the copy of Visual Studio.NET where you have your custom class library attach to the devenv.exe process running your report in order to debug the code.

    Note: The DebugLocal configuration must be set not to deploy the reports in the project for debugging to work properly

    Note: You will need to specify which project within your solution to run first

## *Deploying a Custom Assembly*

Custom assemblies are much more difficult to deploy than Code embedded in your report through the Code element. There are several reasons for this:

* 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 themselves
* The built in project deployment method in Visual Studio.NET will not automatically deploy your custom assemblies
* Custom assemblies are only granted Execution permissions by default.

In order use your custom assemblies with SQL Reporting Services you will need to take the following steps:

1. You need to deploy your custom assemblies to the application bin folder.

* For the Report Designer the default is: C:\Program Files\Microsoft SQL Server\80\Tools\Report Designer.

* For the Reporting Server the default is: C:\Program Files\Microsoft SQL Server\MSSQL\Reporting Services\ReportServer\bin.

2. Next you need to edit the Reporting Services policy configuration files if your custom assembly requires permissions in addition to Execution permission.

   * For the Report Designer the default location is: C:\Program Files\Microsoft SQL Server\80\Tools\Report Designer.

   * For the Reporting Server the default location is: C:\Program Files\Microsoft SQL Server\MSSQL\Reporting Services\ReportServer.

For example, if you were writing a custom assembly to calculate an employees cost per visit, you might need to read the pay rates from a file. To retrieve the rate information, you would need to add an additional security permission, FileIOPermission, to your permission set for the assembly. To grant this permission we have to make the following two changes to the configuration file.

To add permission to read a file called D:\Temp\EmployeeCost.xml you first need to add a permission set in the policy configuration file that grants read permission to the file that we can then apply to the custom assembly.

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

Next we add a code group that grants our assembly the additional permissions.

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

Note : The name of your assembly that you add to the configuration file must match exactly the name that is added to the RDL under the CodeModules element.

In order to apply custom permissions, you must also assert the permission within your code. For example, if you want to add read-only access to an XML file C:\CurrencyRates.xml, you must add the following code to your method:

```
// C#
FileIOPermission permission = new
    FileIOPermission(FileIOPermissionAccess.Read,
    @" D:\Temp\EmployeeCost.xml");
try
{
   permission.Assert();
   // Load the XML currency rates file
   XmlDocument doc = new XmlDocument();
   doc.Load(@"D:\Temp\EmployeeCost.xml");
...
```

You can also add the assertion as a method attribute:

```
[FileIOPermissionAttribute(SecurityAction.Assert,
    Read=@" D:\Temp\EmployeeCost.xml")]
```

For more information, see ".NET Framework Security" in the .NET Framework Developer's Guide.

Tip: If you change the custom assembly and rebuild it you must redeploy it. Even if you are referencing a custom assemble in the VS.NET IDE from its original location in your project the Report Designer preview only looks for it in the Report Designer application folder and Reporting Services only looks for it in its application folder. You may find that you have to exit the VS.NET IDE in order to replace the files as they may otherwise be in use. You may wish 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 under Report Properties References 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 meaning you only have to redeploy reports that use the new features of the new version. If you want all of 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 servers Web.config file and ReportService.exe.config file.

## Summary

In this chapter we have looked at using custom code within your application and we have discussed some of the other programmatic aspects of dealing with reporting services. Chapters 6, 7, 8 and 9 will build on this as we write applications to render, deploy, manage and secure both the reporting solutions we have developed as well as the SRS server itself.

In subsequent chapters we will look at other aspects of programmatically working with SRS such as how to render reports from within a custom .NET application and how to deploy them to and secure them on the reporting server