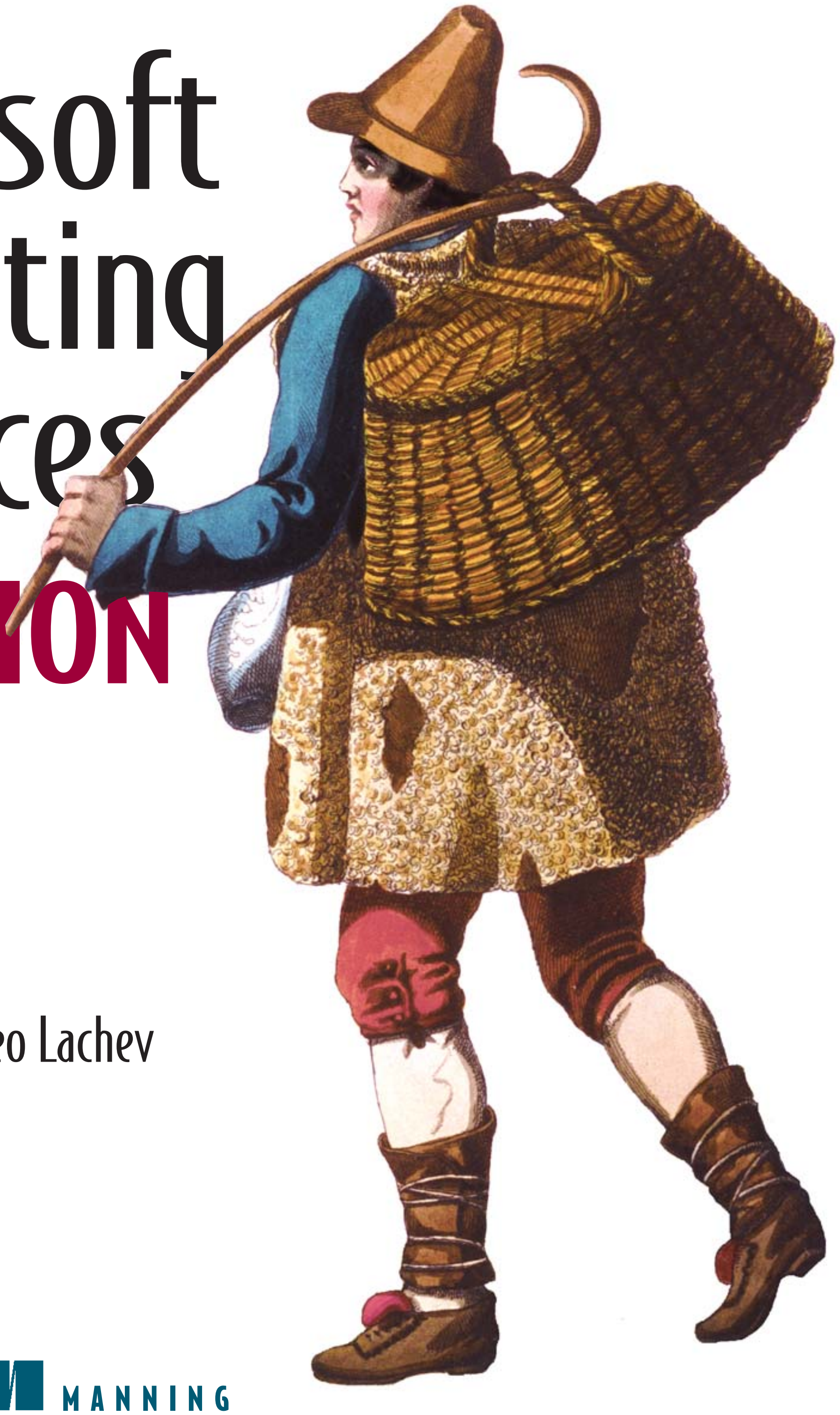
SAMPLE CHAPTER

Microsoft Reporting Services IN ACTION



Teo Lachev

MANNING

Microsoft Reporting Services in Action

by Teo Lachev

Chapter 6

brief contents

CHAPTER 6

Using custom code

Reporting Services doesn't limit your programming options to using inline expressions and functions. In this chapter, we will show you how to supercharge the expression capabilities of your reports by integrating them with custom code. Writing custom code allows us to use advanced programming techniques to meet the most demanding reporting needs.

In this chapter, you will

- See what custom code options RS offers
- Learn how to write embedded code
- Find out how to integrate reports with external .NET assemblies
- Use XSL transformations to produce XML reports

We will put our custom code knowledge into practice by creating an advanced report that will show forecasted sales data.

With the widespread adoption of the XML as an interoperable data exchange format, we will also see how we can export reports to XML and custom-tailor the report output by using XSL transformations.

6.1 UNDERSTANDING CUSTOM CODE

As we mentioned in chapter 1, one of the most prominent features of Reporting Services is its extensible architecture. One way you can extend the RS capabilities is by integrating your reports with custom code that you or somebody else wrote. In general, you have two options for doing so:

- Write embedded (report-specific) code using Visual Basic .NET.
- Use custom code located in an external .NET assembly.

We'll now discuss each custom code option in more detail.

6.1.1 Using embedded code

As its name suggests, *embedded* code gets saved inside the Report Definition Language (RDL) file. Before we jump to a code example, we would like to mention some limitations that embedded code is a subject to:

- You can call embedded code only from within the report that contains the code. Because embedded code is saved in the RDL file, it is always scoped at the report level. For this reason, code embedded in one report cannot be referenced from another report. To create global and reusable functions that could be shared among reports, you have to move them to an external .NET assembly.
- You are restricted to using Visual Basic .NET only as a programming language for writing embedded code.
- As we pointed out in chapter 5, inside custom code you cannot directly reference the report object global collections, such as Fields, ReportItems, and so on. Instead, you have to pass them to your embedded methods as arguments.

To call embedded code in your report, you reference its methods using the globally defined *Code* member. For example, if you have authored an embedded code function called `GetValue`, you can call it from your expressions by using the following syntax:

```
=Code.GetValue()
```

DEFINITION *Shared* (called static in C#) methods can be invoked directly through the class name without first creating an instance of the class. To designate a method as shared, you use the VB.NET Shared modifier. The embedded code option doesn't support shared methods. On the other hand, *instance* methods are accessed through instances of the class and don't require a special modifier.

With the exception of shared methods, your embedded code can include any VB.NET-compliant code. In fact, if you think of the embedded code as a private class inside your project, you won't be far from the truth. You can declare class-level members and constants, private or public methods, and so on.

Maintaining state

One not-so-obvious aspect of working with embedded code is that you can maintain state in it. For example, you can use class-level members to preserve the values of the variables between calls to embedded code methods from the moment the report processing starts until the report is fully processed. We will demonstrate this technique in the forecasting example that we will explore in section 6.2.

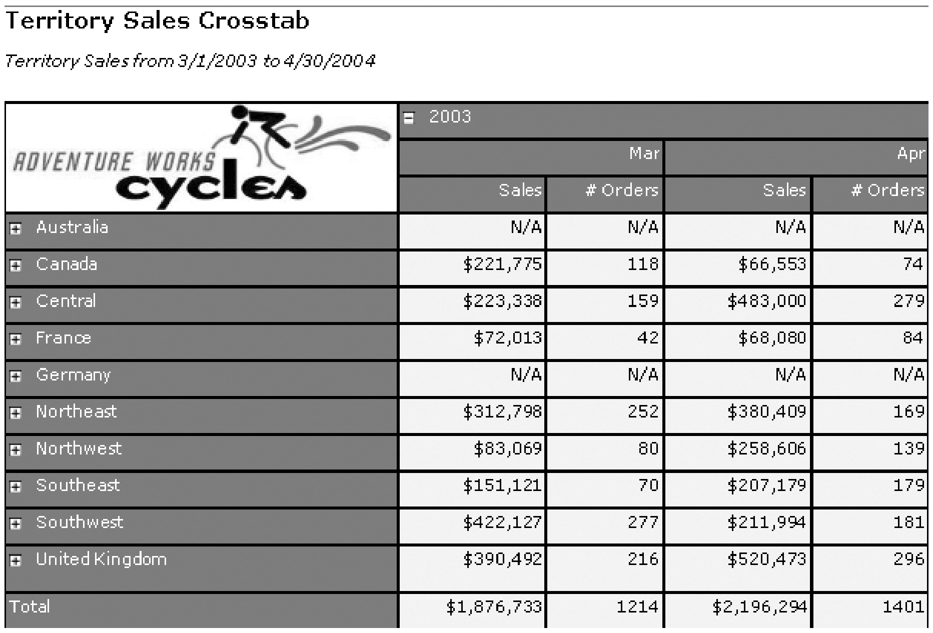
Please note that state can be maintained within the duration of single report request only. As we explained in chapter 2, the RS report-processing model is stateless. For this reason, the report state gets discarded at the end of the report processing. Reporting Services is a web-based application, and just like any other web application, once the request is handled, its runtime state gets released. For this reason, subsequent requests to the same report cannot share state stored in class-level variables.

Let's now look at a practical example where embedded code can be useful.

Writing embedded code

You can write embedded code to create reusable utility functions that can be called from several expressions in your report. Let's examine an example of how we can do just that.

Suppose that the Adventure Works users have requested that we change the Territory Sales Crosstab report to display N/A when data is missing, as shown in figure 6.1.



Territory Sales Crosstab

Territory Sales from 3/1/2003 to 4/30/2004

	2003			
	Mar		Apr	
	Sales	# Orders	Sales	# Orders
Australia	N/A	N/A	N/A	N/A
Canada	$221,775	118	$66,553	74
Central	$223,338	159	$483,000	279
France	$72,013	42	$68,080	84
Germany	N/A	N/A	N/A	N/A
Northeast	$312,798	252	$380,409	169
Northwest	$83,069	80	$258,606	139
Southeast	$151,121	70	$207,179	179
Southwest	$422,127	277	$211,994	181
United Kingdom	$390,492	216	$520,473	296
Total	$1,876,733	1214	$2,196,294	1401

Figure 6.1 You can use embedded code to implement useful utility functions scoped at the report level.

Further, let's assume that we need to differentiate between missing data and NULL values. When the underlying value is NULL, we will translate it to zero. To meet this requirement, we could write a simple embedded function called `GetValue`.

Using the Code Editor

To write custom embedded code, you use the Report Designer Code Editor, which you can invoke from the Report Properties dialog. You can open this dialog in either of two ways:

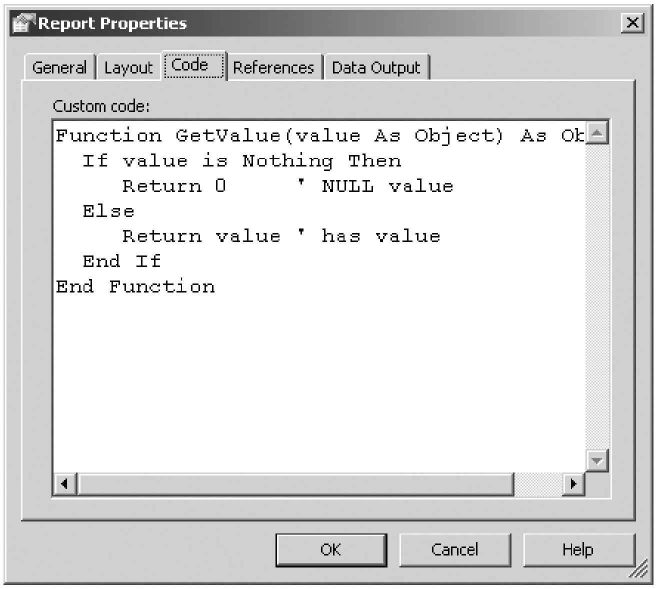
- Select the report by right-clicking the Report Selector and choosing Properties.
- Right-click anywhere on the report outside the body area, and choose Properties.

Then, from the Report Properties dialog, choose the Code tab, as shown in figure 6.2.

Granted, function GetValue can easily be replaced with an Iif-based expression. However, encapsulating the logic in an embedded function has two advantages. First, it centralizes the logic of the expression in one place instead of using Iif functions for every field in the report. Second, it makes the report more maintainable because if you decide to make a logical change to your function, you do not have to track down and change every Iif function in the report.

As you can see, the Code Editor is nothing to brag about. It is implemented as a simple text area control, and its feature set doesn't go beyond copying and pasting text. For this reason, I highly recommend that you use a standard VB Windows Forms or Console application to write your VB.NET code in a civilized manner and then copy and paste it inside the Code Editor.

The Report Designer saves embedded code under the `<Code>` element in the RDL file. When doing so, the Report Designer URL-encodes the text. Be aware of this if you decide to change the `<Code>` element directly for some reason.



```
Function GetValue(value As Object) As Ob
  If value is Nothing Then
     Return 0      ' NULL value
  Else
     Return value ' has value
  End If
End Function
```

Figure 6.2
Use the Code Editor for writing embedded code. The `GetValue` function, shown in the Code Editor, determines whether a value is missing or NULL.

Handling missing data

Once the `GetValue` function is ready, to differentiate between NULL and missing data in our report, we could base the txtSales and txtNoOrders values on the following expressions:

```
=Iif(CountRows()=0, "N/A", Code.GetValue(Sum(Fields!Sales.Value)))
```

and

```
=Iif(CountRows()=0, "N/A", Code.GetValue(Sum(Fields!NoOrders.Value)))
```

respectively.

The `CountRows` function returns the count of rows within a specified scope. If no scope is specified, it defaults to the innermost scope, which in our case resolves to the static group that defines the values in the data cells. Both expressions first check for missing data (no rows) by using `CountRows` and display N/A if no missing data is found. Otherwise, they call the `GetValue` embedded function to translate the NULL values.

We recommend that you use embedded code for writing simple report-specific utility-like functions. When your programming logic gets more involved, you should consider moving your code to external assemblies, as we discuss next.

6.1.2 Using external assemblies

The second way of extending RS programmatically is by using prepackaged logic located in external .NET assemblies that can be written in any .NET-supported language. The ability to integrate reports with custom code in external assemblies increases your programming options dramatically. For example, by using custom code, you can do the following:

- Leverage the rich feature set of the .NET Framework. For example, let's say you need a collection to store crosstab data of a matrix region in order to perform some calculations. You can "borrow" any of the collection classes that come with .NET, such as Array, ArrayList, Hashtable, and so on.
- Integrate your reports with custom .NET assemblies, written by you or third-party vendors. For example, to add forecasting features to the Sales by Product Category report in section 6.2, we leveraged the Open Source OpenForecast package.
- Write code a whole lot easier by leveraging the powerful Visual Studio .NET IDE instead of the primitive Code Editor.

I hope that at some point in future, RS will get better integrated with the Visual Studio .NET IDE and support other .NET languages besides VB.NET. Ideally, RS should allow developers to add custom classes to their business intelligence projects and write code using the Visual Studio .NET editor. If this gets implemented, enhancing RS programmatically will be no different than writing code in traditional .NET development projects.

Based on preliminary feedback that I got from Microsoft, this seems to be the long-term direction that RS will follow.

Referencing external assemblies

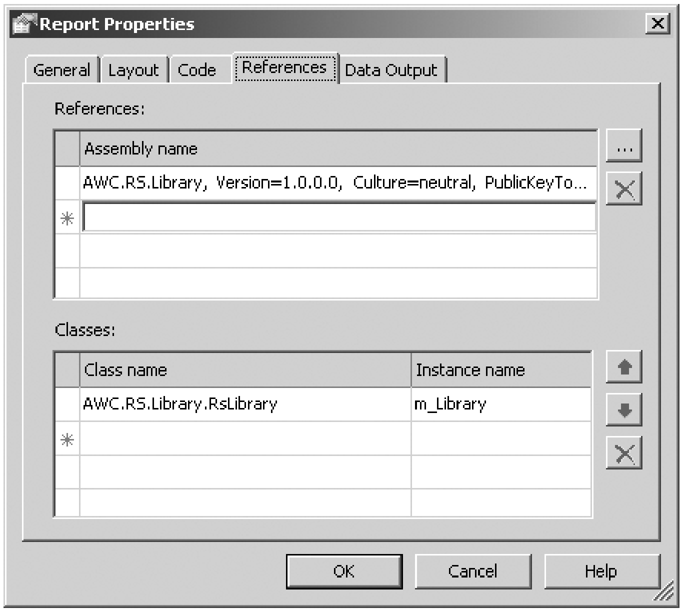
To use types located in an external assembly, you have to first let the Report Designer know about it by using the References tab in the Report Properties dialog, as shown in figure 6.3.

Assuming that our report needs to use the custom AWC.RS.Library assembly (included with this book's source code), we must first reference it using the References tab. While this tab allows you to browse and reference an assembly from an arbitrary folder, note that when the report is executed, the .NET Common Language Runtime (CLR) will try to locate the assembly according to CLR probing rules. In a nutshell, these rules give you two options for deploying the custom assembly:

- Deploy the assembly as a private assembly.
- Deploy the assembly as a shared assembly in the .NET Global Assembly Cache (GAC). As a prerequisite, you have to strong-name your assembly. For more information about how to do this, please refer to the .NET documentation.

If you choose the first option, you will need to deploy the assembly to the Report Designer folder so that the assembly is available during the report-testing process. Assuming that you have accepted the default installation settings, to deploy the assembly to the Report Designer folder, copy the assembly to C:\Program Files\Microsoft SQL Server\80\Tools\Report Designer. Once you have done this, you can build and render the report in preview mode inside VS.NET.

Before the report goes live, you need to deploy the assembly to the Report Server binary folder. Specifically, you need to copy to the assembly to the Report Server binary



References:

	Assembly name
	AWC.RS.Library, Version=1.0.0.0, Culture=neutral, PublicKeyTo...
*	

Classes:

	Class name	Instance name
	AWC.RS.Library.RsLibrary	m_Library
*		

Figure 6.3 Use the Report Properties dialog to reference an external assembly.

folder, which by default is C:\Program Files\Microsoft SQL Server\MSSQL\ Reporting Services\ReportServer\bin.

Please note that deploying the custom assembly to the right location is only half of the deployment story. Depending on what your code does, you may need also to adjust the code access security policy so the assembly code can execute successfully. We will discuss the code access security model in chapter 8. If you need more information about deploying custom assemblies, please refer to the "Using Custom Assemblies with Reports" section in the RS documentation.

Calling shared methods

When using custom code in external assemblies, you can call both instance and shared methods. If you need to call only shared methods (also called static in C#) inside the assembly, you are ready to go because shared methods are available globally within the report.

You can call shared methods by using the fully qualified type name using the following syntax:

```
<Namespace>.<Type>.<Method>(argument1, argument2, …, argumentN)
```

For example, if we need to call the `GetForecastedSet` shared method located in the RsLibrary class (AWC.RS.Library assembly) from an expression or embedded code, we would use the following syntax:

```
=AWC.RS.Library.RsLibrary.GetForecastedSet(forecastedSet, forecastedMonths)
```

where `AWC.RS.Library` is the namespace, `RsLibrary` is the type, `GetForecastedSet` is the method, and `forecastedSet` and `forecastedMonths` are the arguments.

If the custom assembly is your own, how can you decide whether to define your methods as shared or instance? My short answer is to use shared methods if you don't need instance methods. Shared methods are convenient to call. However, instance methods allow you to maintain state within the duration of the report request. For example, you can preserve the class-level variable values between multiple method invocations of the same type. The state considerations for using code in external .NET assemblies are the same as the ones we discussed in the section 6.1.1 for embedded code.

One thing to watch for is using shared class-level fields to maintain state because their values are shared across all instances of the same report. So, depending on how many users are accessing a single report at any one time, the value of a shared field may be changing. In addition, the values of shared fields are not private to a report user, so sensitive user-only data should never be accessed through a shared field or property. Finally, static class-level fields are subject to multithreading locking issues. To avoid these issues, create your classes as stateless classes that don't have class-level shared fields or use instance class-level fields and methods. For more information about shared vs. instance methods, see the Visual Studio .NET documentation.

Sometimes, you simply won't have a choice and your applications requirements will dictate the type of method invocation. For example, if the method needs to be also invoked remotely via .NET Remoting, it has to be an instance method.

Calling instance methods

To invoke an instance method, you have some extra work left to do. First, you have to enumerate all instance classes (types) that you need to instantiate in the Classes grid (see figure 6.3). For each class, you have to assign an instance name. Behind the scenes, RS will create a variable with that name to hold a reference to the instance of the type.

NOTE When you specify the class name in the Classes grid, make sure that you enter the fully qualified type name (namespace included). In our example (shown previously in figure 6.3), the namespace is `AWC.RS.Library` while the class name is RsLibrary. When you are in doubt as to what the fully qualified class name is, use the VS.NET Object Browser or another utility, such as Lutz Roeder's excellent .NET Reflector (see section 6.5 for information on this utility), to browse to the class name and find out its namespace.

For example, assuming that we need to call an instance method in the AWC.RS.Library assembly, we have to declare an instance variable m_Library, as shown in figure 6.3. In our case, this variable will hold a reference to the RsLibrary class.

If you declare more than one variable pointing to the same type, each will reference a separate instance of that type. Behind the scenes, when the report is processed, RS will instantiate as many instances of the referenced type as the number of instance variables.

Once you have finished with the reference settings, you are ready to call the instance methods via the instance type name that you specified. Just as with embedded code, you use the `Code` keyword to call an instance method. The difference between a shared and an instance method is that instead of using the class name, you use the variable name to call the method.

For example, if the `RsLibrary` type had an instance method named `DummyMethod()`, we could invoke it from an expression or embedded code like this:

```
Code.m_Library.DummyMethod().
```

Having seen what options we have as developers for programmatically expanding our report features, let's see how we can apply them in practice. In the next section, we will find out how we can use embedded and external code to add advanced features to our reports.

6.2 CUSTOM CODE IN ACTION: IMPLEMENTING REPORT FORECASTING

In this section, we will show you how to incorporate forecasting capabilities in our reports. These are the design goals of the sample report that we are going to create:

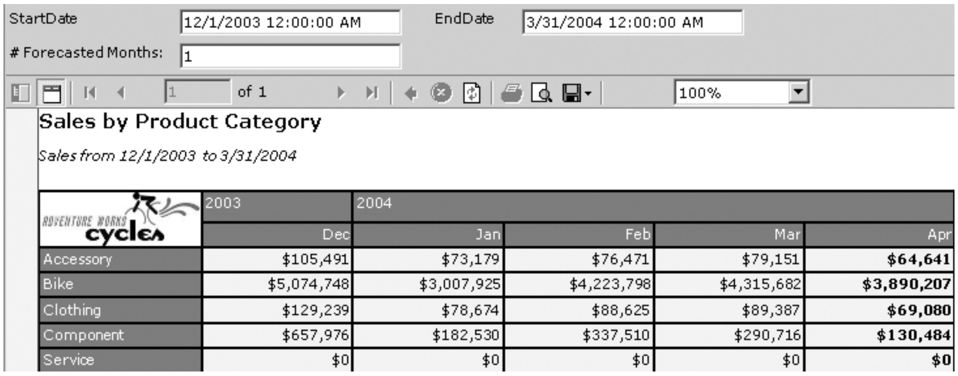
- Allow the user to generate a crosstab report of sales data for an arbitrary period.
- Allow the user to specify the number of forecasted columns.
- Use data extrapolation to forecast the sales data.

Here is our fictitious scenario. Imagine that the AWC management has requested to see forecasted monthly sales data grouped by product category. To make these things more interesting, let's allow the report users to specify a data range to filter the sales data, as well as the number of forecasted months. To accomplish the above requirements, we will author a crosstab report, Sales by Product Category, as shown in figure 6.4.

The user can enter a start date and an end date to filter the sales data. In addition, the user can specify how many months of forecasted data will be shown on the report. The report shows the data in a crosstab fashion, with product categories on rows and time periods on columns. The data portion of the report shows first the actual sales within the requested period, followed by the forecasted sales in bold font.

For example, if the user enters 4/30/2003 as a start date and 3/31/2004 as an end date and requests to see three forecasted months, the report will show the forecasted data for April, May, and June 2004 (to conserve space, figure 6.4 shows only one month of forecasted data).

As you would probably agree, implementing forecasting features on your own is not an easy undertaking. But what if there is already prepackaged code that does this for us? If this code can run on .NET, our report can access it as custom code. Enter OpenForecast.



StartDate 12/1/2003 12:00:00 AM   EndDate 3/31/2004 12:00:00 AM
# Forecasted Months: 1

Sales by Product Category

Sales from 12/1/2003 to 3/31/2004

	2003	2004			
	Dec	Jan	Feb	Mar	Apr
Accessory	$105,491	$73,179	$76,471	$79,151	**$64,641**
Bike	$5,074,748	$3,007,925	$4,223,798	$4,315,682	**$3,890,207**
Clothing	$129,239	$78,674	$88,625	$89,387	**$69,080**
Component	$657,976	$182,530	$337,510	$290,716	**$130,484**
Service	$0	$0	$0	$0	**$0**

Figure 6.4 The Sales by Product Category report uses embedded and external custom code for forecasting.

6.2.1 Forecasting with OpenForecast

Forecasting is a science in itself. Generally speaking, forecasting is concerned with the process used to predict the unknown. Instead of looking at a crystal ball, forecasting practitioners use mathematical models to analyze data, discover trends, and make educated conclusions. In our example, the Sales by Product Category report will predict the future sales data by using the data extrapolating method.

There are number of well-known mathematical models for extrapolating a set of data, such as polynomial regression and simple exponential smoothing. Implementing one of those models, though, is not a simple task. Instead, for the purposes of our sales forecasting example, we will use the excellent Open Source OpenForecast package, written by Steven Gould.

OpenForecast is a general-purpose package that includes Java-based forecasting models that can be applied to any data series. The package require no knowledge of forecasting, which is great for those of us who have decided to focus on solving pure business problems and kissed mathematics goodbye a long time ago.

OpenForecast supports several mathematical forecasting models, including single-variable linear regression, multi-variable linear regression, and so on. The current OpenForecast version as of the time of this writing is 0.3, but version 0.4 is under development and probably will be released by the time you read this book. Please see section 6.5 for a link to the OpenForecast web site.

Let's now see how we can implement our forecasting example and integrate with OpenForecast by writing some embedded and external code.

6.2.2 Implementing report forecasting features

Creating a crosstab report with forecasting capabilities requires several implementation steps. Let's start with a high-level view of our envisioned approach and then drill down into the implementation details.

Choosing an implementation approach

Figure 6.5 shows the logical architecture view of our solution.

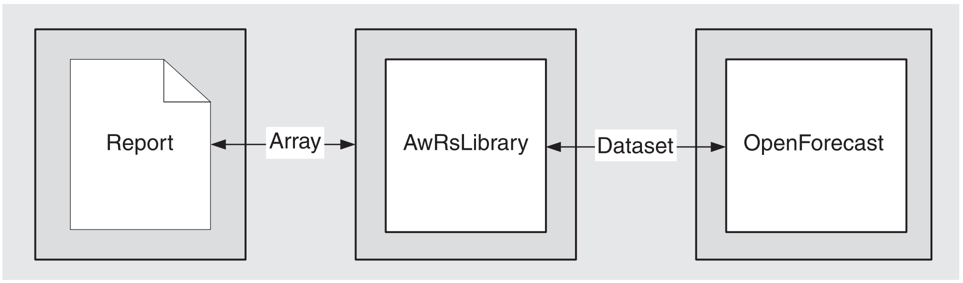


Figure 6.5 The Sales by Product Category report uses embedded code to call the AwRsLibrary assembly, which in turns calls the `J# OpenForecast` package.

Our report will use embedded code to call a shared method in a custom assembly (AwRsLibrary) and get the forecasted data. AwRsLibrary will load the existing sales data into an OpenForecast dataset and obtain a forecasting model from OpenForecast. Then, it will call down to OpenForecast to get the forecasted values for the requested number of months. AwRsLibrary will return the forecasted data to the report, which in turn will display it.

We have at least two implementation options for passing the crosstab sales data to AwRsLibrary:

- Fetch the sales data again from the database. To accomplish this, the report could pass the selected product category and month values on a row-by-row basis. Then, AwRsLibrary could make a database call to retrieve the matching sales data.
- Load the existing sales data in a structure of some kind using embedded code inside the report and pass the structure to AwRsLibrary.

The advantages of the latter approach are as follows:

- *The custom code logic is self-contained.* We don't have to query the database again.
- *It uses the default custom code security policy.* We don't have to elevate the default code access security policy for the AwRsLibrary assembly. If we choose the first option, we won't be able to get away with the default code access security setup, because RS will grant our custom assemblies only Execution rights, which are not sufficient to make a database call. Actually, in the case of OpenForecast, we had to grant both assemblies FullTrust rights because any J# code requires FullTrust to execute successfully. However, we wouldn't have had to do this if we had chosen C# as a programming language.
- *No data synchronization is required.* We don't have to worry about synchronizing the data containers, the matrix region and the AwRsLibrary dataset.

For the above reasons, we will choose the second approach. To get it implemented, we will use an expression to populate the matrix region data values. The expression will call our embedded code to load an array structure on a row-by-row basis. Once a given row is loaded, we will pass the array to AwRsLibrary to get the forecasted data.

Now, let's discuss the implementation details, starting with converting OpenForecast to .NET.

Migrating OpenForecast to .NET

OpenForecast is written in Java, so one of the first hurdles that we had to overcome was to integrate it with .NET. We had two options to do so:

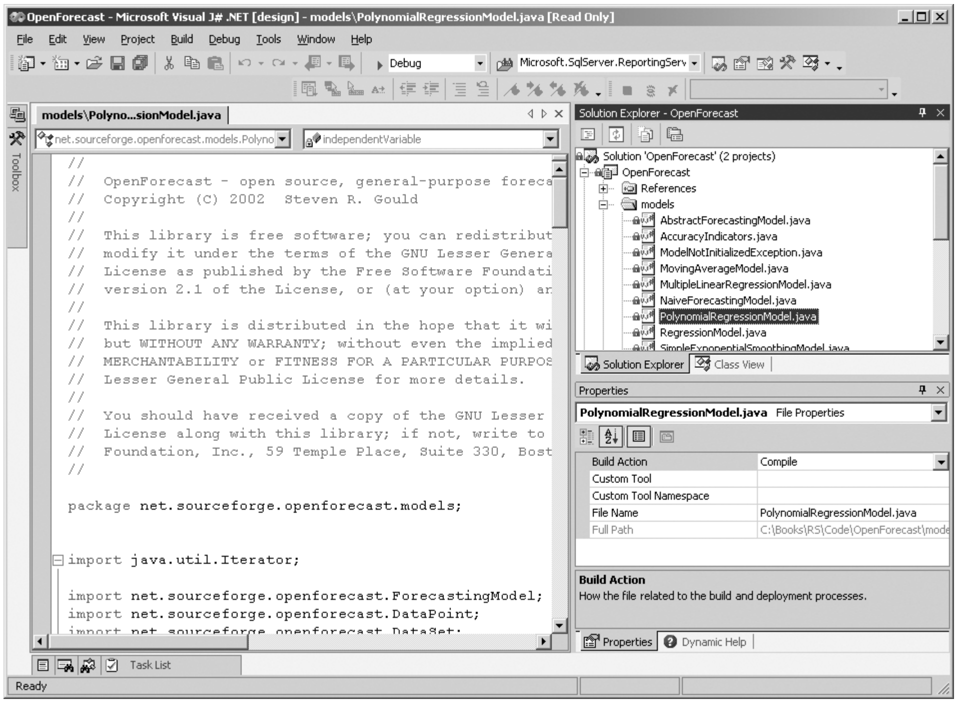
- *Use a third-party Java-to-.NET gateway to integrate both platforms.* Given the complexities of this approach we quickly dismissed it.

- *Port OpenForecast to one of the supported .NET languages.* Microsoft provides two options for this. First, we can use the Microsoft Java Language Conversion Assistant (see section 6.5 for more information) to convert Java-language code to C#. Second, we could convert OpenForecast to J#. The latter option would have preserved the Java syntax although that code will execute under the control of the .NET Common Language Runtime instead of the Java Virtual Machine.

We decided to port OpenForecast to J#. The added benefit to this approach is that the Open Source developers could maintain only one Java-based version of OpenForecast. Porting OpenForecast to J# turned out to be easier than we thought. We created a new J# library project, named it OpenForecast, and loaded all *.java source files inside it. We included the .NET version of OpenForecast in the source code that comes with this book.

Figure 6.6 shows the converted to J# version of OpenForecast open in Visual Studio.NET.

We had take care of only a few compilation errors inside the MultipleLinearRegression, because several Java hashtable methods are not supported in J#, such as `keySet()`, `entries()`, and hashtable cloning. We also included a WinForm application (TestHarness) that you can use to test the converted OpenForecast. We included the OpenForecast DLL so you could still run the report even if you don't have J# installed.


OpenForecast - Microsoft Visual J# .NET [design] - models\PolynomialRegressionModel.java [Read Only]
File Edit View Project Build Debug Tools Window Help
Debug
Microsoft.SqlServer.ReportingServ
models\Polyno...sionModel.java
net.sourceforge.openforecast.models.Polyno
independentVariable
Toolbox
Task List
Ready

```
//
//  OpenForecast - open source, general-purpose foreca
//  Copyright (C) 2002  Steven R. Gould
//
//  This library is free software; you can redistribut
//  modify it under the terms of the GNU Lesser Genera
//  License as published by the Free Software Foundati
//  version 2.1 of the License, or (at your option) ar
//
//  This library is distributed in the hope that it wi
//  but WITHOUT ANY WARRANTY; without even the implied
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOS
//  Lesser General Public License for more details.
//
//  You should have received a copy of the GNU Lesser
//  License along with this library; if not, write to
//  Foundation, Inc., 59 Temple Place, Suite 330, Bost
//

package net.sourceforge.openforecast.models;


import java.util.Iterator;

import net.sourceforge.openforecast.ForecastingModel;
import net.sourceforge.openforecast.DataPoint;
```

Solution Explorer - OpenForecast

- Solution 'OpenForecast' (2 projects)
  - OpenForecast
    - References
    - models
      - AbstractForecastingModel.java
      - AccuracyIndicators.java
      - ModelNotInitializedException.java
      - MovingAverageModel.java
      - MultipleLinearRegressionModel.java
      - NaiveForecastingModel.java
      - PolynomialRegressionModel.java
      - RegressionModel.java

Properties

PolynomialRegressionModel.java File Properties

Build Action	Compile
Custom Tool	
Custom Tool Namespace	
File Name	PolynomialRegressionModel.java
Full Path	C:\Books\RS\Code\OpenForecast\mode

Build Action
How the file related to the build and deployment processes.

Figure 6.6 To convert Java-based OpenForecast to .NET, we migrated its code to J#.

Implementing the AwRsLibrary assembly

The next step was to create the custom .NET assembly, AwRsLibrary, that will bridge the report-embedded code and OpenForecast. We implemented AwRsLibrary as a C# class library project. Inside it we created the class RsLibrary that exposes a static (shared) method, `GetForecastedSet`. The abbreviated code of this method is shown in listing 6.1.

Listing 6.1 The report-embedded code calls the AwRsLibrary `GetForecastedSet` method, which in turns calls OpenForecast.

```
public static void GetForecastedSet(double[] dataSet,
  int numberForecastedPoints ) {
  DataSet observedData = new DataSet();          <-- Define an OpenForecast dataset and
  Observation dp;                                    load it with the matrix row array
  for (int i=0;i<dataSet.Length-numberForecastedPoints;i++) {
    dp = new Observation( dataSet[i]);
    dp.setIndependentValue( "x", i);
    observedData.add( dp );
  }

  ForecastingModel forecaster = new MultipleLinearRegressionModel();  <-- Obtain a forecasting
                                                                          model from
                                                                          OpenForecast
  forecaster.init(observedData);
  DataSet requiredObservations = new DataSet();    <-- Specify placeholders for
for ( int i=dataSet.Length-numberForecastedPoints;     the forecasted data
 i < dataSet.Length; i++ ) {
    dp = new Observation( 0.0 );
    dp.setIndependentValue( "x", i );
    requiredObservations.add( dp );
  }

  forecaster.forecast( requiredObservations );    <-- Perform
                                                      forecasting

  int index =  dataSet.Length - numberForecastedPoints;
  Iterator it = requiredObservations.iterator();
  while ( it.hasNext() ) {    <-- Populate the input array
    dataSet[index] = ((DataPoint)it.next()).getDependentValue();
    index++;
  }
}
```

The `GetForecastedSet` method receives the existing sales data for a given product category in the form of a dataSet array, as well as the number of the requested months for forecasted data. Next, integrating with OpenForecast is a matter of five steps.

Step 1 We create a new OpenForecast dataset and load it with the existing data from the matrix row array.

Step 2 We obtain a given forecasting model. OpenForecast allows developers to get the optimal forecasting mathematical model based on the given data series by calling the `getBestForecast` method. This method will examine the dataset and will try a few forecasting models to select the most optimal. If the returned model is not a good fit, you can request a forecasting model explicitly by instantiating any of the classes found under the model's project folder.

NOTE When testing the report, I noticed that with my sales data `getBestForecast()` returns the PolynomialRegressionModel model, which returns negative values when the sales data varies considerably. For this reason, I explicitly request the MultipleLinearRegressionModel model. I recommend that you try `getBestForecast()` first for your forecasting applications, and only if the returned model doesn't meet your needs should you request a model explicitly.

Step 3 We prepare another dataset to hold the forecasted data and initialize it with as many elements as the number of forecasted months.

Step 4 We call the forecast method to extrapolate the data and return the forecasted results.

Step 5 We load the forecasted data back to the dataSet array so we can pass it back to the report's embedded code.

Once we have finished with both the AwRsLibrary and OpenForecast .NET assemblies, we need to deploy them.

Deploying custom assemblies

As we explained in section 6.1, we need to deploy custom assemblies to both the Report Designer and Report Server binary folders. The custom assembly deployment process consists of the following steps:

Step 1 Copy the assemblies to the Report Designer and Report Server binary folders.

Step 2 Adjust the code-based security if the custom code needs an elevated set of code access security permissions.

To make both assemblies, AwRsLibrary and OpenForecast, available during design time, we have to copy AWC.RS.Library.dll and OpenForecast.dll to the Report Designer folder, which by default is C:\Program Files\Microsoft SQL Server\80\Tools\Report Designer.

Similarly, to successfully render the deployed report under the Report Server, we have to deploy both assemblies to the Report Server binary folder, which by default is C:\Program Files\Microsoft SQL Server\MSSQL\Reporting Services\ReportServer\bin. In fact, the Report Server will not let you deploy a report from within the VS.NET IDE if all referenced custom assemblies are not already deployed.

The default RS code access security policy grants Execution rights to all custom assemblies by default. However, J# assemblies require FullTrust code access rights. Because the .NET Common Language Runtime walks up the call stack to verify that all callers have the required permission set, we need to elevate the code access security policy for both assemblies to full trust. This will require changes to the Report Designer and Report Server security configuration files.

We will provide more details about how code access security works and how it can be configured in chapter 8. If you don't want to wait until then, you can find a copy of our rssrvpolicy.config configuration file enclosed with the AwRsLibrary project. Toward the end of the file, you will see two CodeGroup XML elements that point to the AwRsLibrary and OpenForecast files. You will need to copy these elements to the Report Server security configuration file (rssrvpolicy.config).

In addition, as we discussed in chapter 2, if you want to preview (run) the report in the Preview window from the Report Designer, you will need to propagate the changes to the Report Designer security configuration file (rspreviewpolicy.config) as well.

Once the custom assemlies are deployed, we will need to write some VB.NET embedded code in our report to call the AwRsLibrary assembly, as we will discuss next.

Writing report embedded code

To integrate the report with AwRsLibrary we added an embedded function called `GetValue` to the Sales by Product Category report as shown in listing 6.2.

Listing 6.2 The embedded `GetValue` function calls the AwRsLibrary assembly.

```
Dim forecastedSet() As Double  ' array with sales data
Dim productCategoryID As Integer = -1
Dim bNewSeries As Boolean = False
Public Dim m_ExString = String.Empty

Function GetValue(productCategoryID As Integer, _
  orderDate As DateTime, _
sales As Double, reportParameters as Parameters, _
txtRange as TextBox) As Double

  Dim startDate as DateTime = reportParameters!StartDate.Value
  Dim endDate as DateTime = reportParameters!EndDate.Value
Dim forecastedMonths as Integer = _
  reportParameters!ForecastedMonths.Value

  If (forecastedSet Is Nothing) Then
    ReDim forecastedSet(DateDiff(DateInterval.Month, _
       startDate, endDate) + forecastedMonths)     <-- Redim the array only once
  End If                                               to hold existing sales data
                                                       plus forecasted sales
  If Me.productCategoryID <> productCategoryID Then     <-- The array holds
    Me.productCategoryID = productCategoryID                sales data per
    bNewSeries = True                                       product category
    Array.Clear(forecastedSet, 0, forecastedSet.Length - 1)
  End If

  Dim i = DateDiff(DateInterval.Month, startDate , orderDate)
  'Is this a forecasted value?
   If orderDate <= endDate Then
            ' No, just load the value in the array
            forecastedSet(i) = sales
   Else
      If bNewSeries Then
         Try
           AWC.RS.Library.RsLibrary.GetForecastedSet(_      <-- Call AwRsLibrary
             forecastedSet, _                                   to get the
             forecastedMonths)                                  forecasted set
           bNewSeries = False
         Catch ex As Exception
           m_ExString  = "Exception: " & ex.Message
           System.Diagnostics.Trace.WriteLine(ex.ToString())
           throw ex
         End Try
     End If
   End If ' is it forecasted value
   Return forecastedSet(i)
End Function
```

Because the matrix region data cells use an expression that references the `GetValue` function, this function gets called by each data cell. Table 6.1 lists the input arguments that the `GetValue` function takes.

Table 6.1 Each data cell inside the matrix region will call the `GetValue` embedded function and pass the following input arguments.

Argument	Purpose
productCategoryID	The productCategoryID value from the rowProductCategory row grouping corresponding to the cell
orderDate	The orderDate value from the colMonth column grouping corresponding to the cell
sales	The aggregated sales total for this cell
reportParameters	To calculate the array dimensions, `GetValue` needs the values of the report parameters. Instead of passing the parameters individually using Parameters!ParameterName.Value, we pass a reference to the report Parameters collection.
txtRange	A variable that holds the error message in case an exception occurs when getting the forecasted data

To understand how `GetValue` works, note that each data cell inside the matrix region is fed from the forecastedSet array. If the cell doesn't need forecasting (its corresponding date is within the requested date range), we just load the cell value in the array and pass it back to display it in the matrix region. To get this working, we need to initialize the array to have a rank equal to the number of requested months plus the number of forecasted months. Once the matrix region moves to a new row and calls our function, we are ready to forecast the data by calling the `AwRsLibrary.GetForecastedSet` method.

Implementing the Sales by Product Category crosstab report

The most difficult part of authoring the report itself was setting up its data to ensure that we always have the correct number of columns in the matrix region to show the forecasted columns. By default, the matrix region won't show columns that don't have data. This will interfere with calculating the right offset to feed the cells from the array.

Therefore, we have to ensure that the database returns records for all months within the requested data range. To implement this, we need to preprocess the sales data at the database. This is exactly what the spGetForecastedData stored procedure does. Inside the stored procedure, we prepopulate a custom table with all monthly periods within the requested date range, as shown in listing 6.3.

Listing 6.3 The spGetForecastedData stored procedure ensures that the returned rowset has the correct number of columns.

```
CREATE  PROCEDURE spGetForecastedData (
  @StartDate smalldatetime,
  @EndDate smalldatetime
)
AS

DECLARE @tempDate smalldatetime

DECLARE @dateSet TABLE       <-- Define a custom table to hold all months
  (                              within the requested date range
  ProductCategoryID   tinyint,
  OrderDate    smalldatetime
  )

SET   @tempDate = @EndDate

WHILE (@StartDate <= @tempDate)    <-- Insert the
BEGIN                                  month records
  INSERT INTO @dateSet
  SELECT ProductCategoryID,  @tempDate
  FROM ProductCategory

  SET @tempDate = DATEADD(mm, -1, @tempDate)
END

SELECT    DS.ProductCategoryID, PC.Name as ProductCategory,
          OrderDate AS Date, NULL AS Sales
FROM      @dateSet DS INNER JOIN ProductCategory PC ON
          DS.ProductCategoryID=PC.ProductCategoryID
UNION ALL    <-- Return the actual sales data
                 plus the dummy records
```

```
SELECT    PC.ProductCategoryID, PC.Name AS ProductCategory,
          SOH.OrderDate AS Date,
          SUM(SOD.UnitPrice * SOD.OrderQty) AS Sales
FROM      ProductSubCategory PSC INNER JOIN
          ProductCategory PC ON PSC.ProductCategoryID =
          PC.ProductCategoryID
INNER JOIN
          Product P ON PSC.ProductSubCategoryID =
          P.ProductSubCategoryID
INNER JOIN SalesOrderHeader SOH INNER JOIN
          SalesOrderDetail SOD ON SOH.SalesOrderID =
          SOD.SalesOrderID
ON        P.ProductID = SOD.ProductID
WHERE     (SOH.OrderDate BETWEEN @StartDate AND @EndDate)
GROUP BY SOH.OrderDate, PC.Name, PC.ProductCategoryID
ORDER BY PC.Name, OrderDate
```

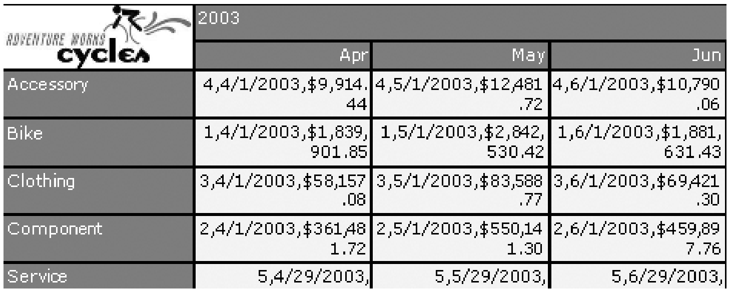
Finally, we union all records from the @dateSet table (its Sales column values are set to NULL) with the actual SQL statement that fetches the sales data.

Once the dataset is set, authoring the rest of the report is easy. We use a matrix region for the crosstab portion of the report. To understand how the matrix region magic works and how it invokes the embedded `GetValue` function, you may want to replace the expression of the txtSales textbox with the following expression:

```
= Fields!ProductCategoryID.Value & "," & Fields!Date.Value _
  & "," &  Format(Fields!Sales.Value, "C")
```

Figure 6.7 shows what the Sales by Product Category crosstab report looks like when this expression is applied.

As you can see, we can easily get to the corresponding row and column group values that the matrix region uses to calculate the aggregate values in the region data cells. Now we have a way to identify each data cell. The matrix region is set up as shown in table 6.2.



	2003		
	Apr	May	Jun
Accessory	4,4/1/2003,$9,914.44	4,5/1/2003,$12,481.72	4,6/1/2003,$10,790.06
Bike	1,4/1/2003,$1,839,901.85	1,5/1/2003,$2,842,530.42	1,6/1/2003,$1,881,631.43
Clothing	3,4/1/2003,$58,157.08	3,5/1/2003,$83,588.77	3,6/1/2003,$69,421.30
Component	2,4/1/2003,$361,481.72	2,5/1/2003,$550,141.30	2,6/1/2003,$459,897.76
Service	5,4/29/2003,	5,5/29/2003,	5,6/29/2003,

Figure 6.7 How the matrix region aggregates data

Table 6.2 The trick to getting the matrix region populated with forecasted values is to base its data cells on an expression.

Matrix Area	Name	Expression
Rows	rowProductGroup	=Fields!ProductCategory.Value
Columns	colYear	=Fields!Date.Value.Year
	colMonth	=Fields!Date.Value.Month
Data	txtSales	=Code.GetValue(Fields!ProductCategoryID.Value, Fields!Date.Value, Sum(Fields!Sales.Value), Parameters, ReportItems!txtRange)

To implement conditional formatting for the forecasted columns (show them in bold), we used the following expression for the font property of the txtSales textbox:

```
=Iif(Code.IsForecasted(Fields!Date.Value, Parameters!EndDate.Value),
"Bold", "Normal")
```

This expression calls the `IsForecasted` function located in the report-embedded code. The function simply compares the sales monthly date with the requested end date and, if the sales date is before the end date, returns false.

The only thing left for us to do is to reference the AwRsLibrary assembly using the Report Properties dialog's References tab, as shown previously in figure 6.3. Please note that for the purposes of this report, we don't need to set up an Instance Name (no need to enter anything in the Classes grid), because we don't call any instance methods.

Debugging custom code

You may find debugging custom code challenging. For this reason, I would like to share with you a few techniques that I have found useful for custom code debugging.

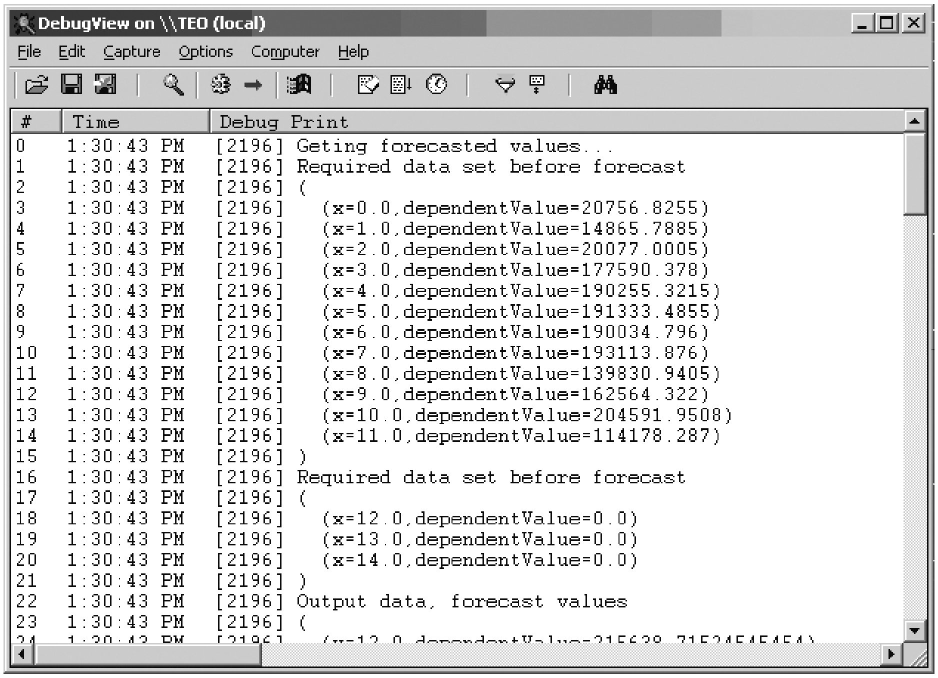
There aren't many options for debugging embedded code. The only one I have found so far is to use the `MsgBox` function to output messages and variable values when the report is rendered inside the Report Designer. Make sure to remove the calls to `MsgBox` before deploying the report to the Report Server. If you don't, all `MsgBox` calls will result in an exception. For some reason, trace messages using System.Diagnostics.Trace (OutputDebugString API) inside embedded code get "swallowed" and don't appear either in the VS.NET Output window or by using an external tracing tool.

When working with external assemblies, you have at least two debugging options:

- Output trace messages.
- Use the VS.NET debugger to step through the custom code.

Tracing

For example, in the `AwRsLibrary.GetForecastedSet` method, we are outputting trace messages using System.Dianogistics.Trace.WriteLine to display the observed and forecasted values. To see these messages when running the report inside VS.NET or Report Server, you can use Mark Russinovich's excellent DebugView tool, shown in figure 6.8.


DebugView on \\TEO (local)
File Edit Capture Options Computer Help

```
#    Time          Debug Print
0    1:30:43 PM   [2196] Geting forecasted values...
1    1:30:43 PM   [2196] Required data set before forecast
2    1:30:43 PM   [2196] (
3    1:30:43 PM   [2196]   (x=0.0,dependentValue=20756.8255)
4    1:30:43 PM   [2196]   (x=1.0,dependentValue=14865.7885)
5    1:30:43 PM   [2196]   (x=2.0,dependentValue=20077.0005)
6    1:30:43 PM   [2196]   (x=3.0,dependentValue=177590.378)
7    1:30:43 PM   [2196]   (x=4.0,dependentValue=190255.3215)
8    1:30:43 PM   [2196]   (x=5.0,dependentValue=191333.4855)
9    1:30:43 PM   [2196]   (x=6.0,dependentValue=190034.796)
10   1:30:43 PM   [2196]   (x=7.0,dependentValue=193113.876)
11   1:30:43 PM   [2196]   (x=8.0,dependentValue=139830.9405)
12   1:30:43 PM   [2196]   (x=9.0,dependentValue=162564.322)
13   1:30:43 PM   [2196]   (x=10.0,dependentValue=204591.9508)
14   1:30:43 PM   [2196]   (x=11.0,dependentValue=114178.287)
15   1:30:43 PM   [2196] )
16   1:30:43 PM   [2196] Required data set before forecast
17   1:30:43 PM   [2196] (
18   1:30:43 PM   [2196]   (x=12.0,dependentValue=0.0)
19   1:30:43 PM   [2196]   (x=13.0,dependentValue=0.0)
20   1:30:43 PM   [2196]   (x=14.0,dependentValue=0.0)
21   1:30:43 PM   [2196] )
22   1:30:43 PM   [2196] Output data, forecast values
23   1:30:43 PM   [2196] (
24   1:30:43 PM   [2196]   (x=12.0,dependentValue=215638.7152454545454)
```

Figure 6.8 Outputting trace messages from external assemblies in DebugView

For more information about DebugView, see section 6.5.

Debugging custom code

You can also step through the custom assembly code using the VS.NET debugger by attaching to the Report Designer process, as follows:

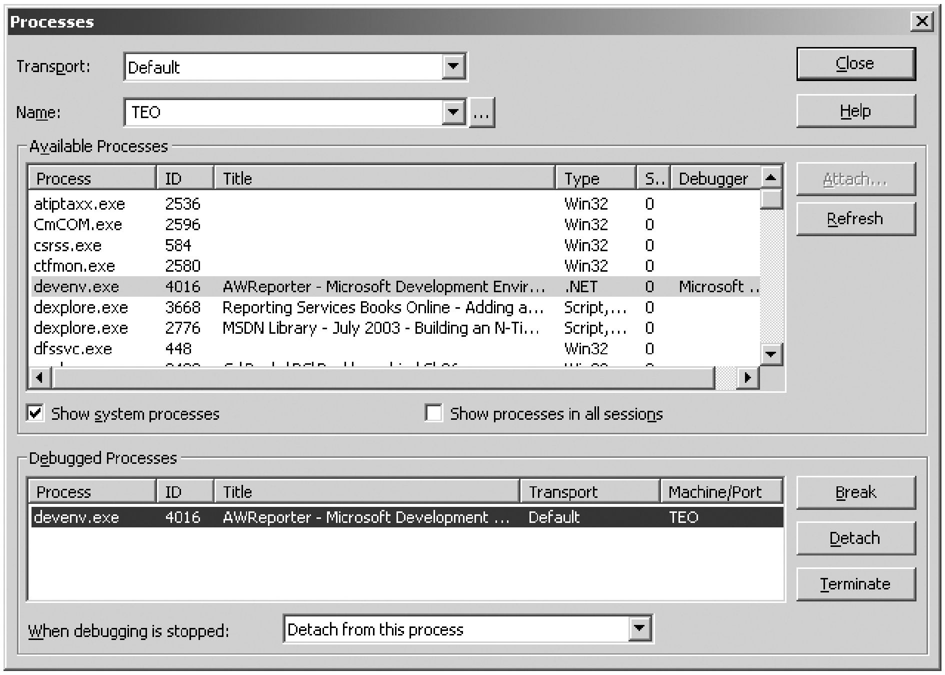
Step 1 Open the custom assembly that you want to debug in a new instance of VS.NET. Set breakpoints in your code as usual.

Step 2 In your custom assembly project properties, expand the Configuration Properties node and select Debugging. Set Debug Mode to Wait to Attach to an External Process.

Step 3 Open your business intelligence project in another instance of VS.NET.

Step 4 Back at the custom assembly project, click on the Debug menu and then choose Processes. Locate the devevn process that hosts that the Business Intelligence project and attach to it. In the Attach To Process dialog, make sure that the Common Language Runtime check box is selected, and click Attach. At this point, your Processes dialog should look like the one shown in figure 6.9.

In this case, we want to debug the code in the AwRsLibrary assembly when it is invoked by the Sales by Product Category report. For this reason, in the AwRsLibrary project we attach to the AWReporter devenv process.



Transport: Default

Name: TEO

Available Processes

Process	ID	Title	Type	S..	Debugger
atiptaxx.exe	2536		Win32	0	
CmCOM.exe	2596		Win32	0	
csrss.exe	584		Win32	0	
ctfmon.exe	2580		Win32	0	
devenv.exe	4016	AWReporter - Microsoft Development Envir...	.NET	0	Microsoft ..
dexplore.exe	3668	Reporting Services Books Online - Adding a...	Script,...	0	
dexplore.exe	2776	MSDN Library - July 2003 - Building an N-Ti...	Script,...	0	
dfssvc.exe	448		Win32	0	

☑ Show system processes ☐ Show processes in all sessions

Debugged Processes

Process	ID	Title	Transport	Machine/Port
devenv.exe	4016	AWReporter - Microsoft Development ...	Default	TEO

When debugging is stopped: Detach from this process

Figure 6.9 To debug custom assemblies, attach to the Visual Studio instance that hosts your BI project.

Step 5 In the Business Intelligence project, preview the report that calls the custom assembly. Or, if you have already been previewing the report, press the Refresh Report button on the Preview Tab toolbar. At this point, your breakpoints should be hit by the VS.NET debugger.

As you will soon find out, if you need to make code changes and recompile the custom assembly, trying to redeploy it to the Report Designer folder results in the following exception:

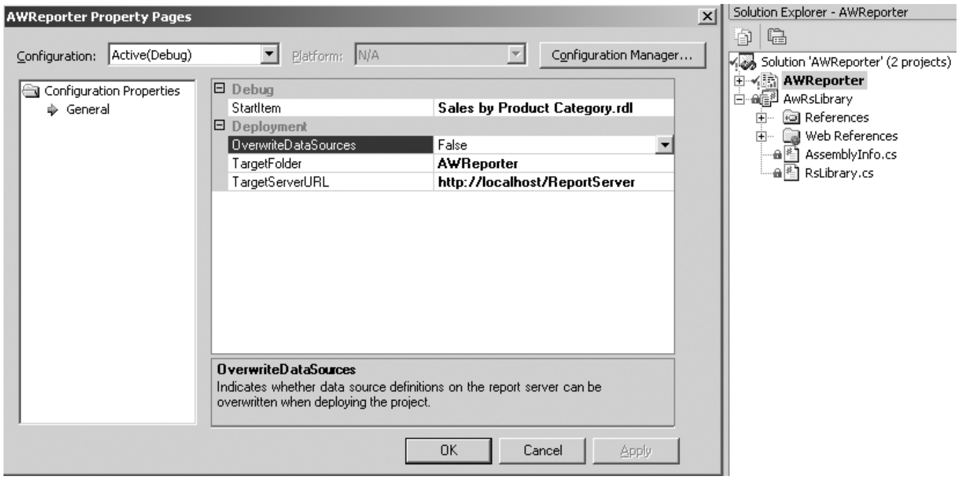
```
Cannot copy <assembly name>: It is being used by another person or program.
```

The problem is that VS.NET IDE holds a reference to the custom assembly. You will need to shut down VS.NET and then redeploy the new assembly. To avoid this situation and make the debugging process even easier, you could debug the custom assembly code by using the Report Host (Preview Window). To do this, follow these steps:

Step 1 Add the custom assembly to the VS.NET solution that includes your BI project.

Step 2 Change the BI project start item to the report that calls the custom code, as shown in figure 6.10.

Step 3 Press F5 to run the report in the Preview window. When the report calls the custom code, your breakpoints will be hit.



Debug	
StartItem	Sales by Product Category.rdl
Deployment	
OverwriteDataSources	False
TargetFolder	AWReporter
TargetServerURL	http://localhost/ReportServer

OverwriteDataSources
Indicates whether data source definitions on the report server can be overwritten when deploying the project.

Figure 6.10 Use the Report Host debug option to avoid locking assemblies.

NOTE As explained in chapter 2, what happens when you press F5 to debug a report depends on your project settings. If both the Build and Deploy options are selected in Configuration Manager, VS.NET will build and deploy all reports in your Business Intelligence project before the report is displayed in the Preview window. To avoid this problem and launch your report faster, clear these options or switch to DebugLocal configuration. This configuration doesn't include the Deploy option by default.

When using the Preview window approach, VS.NET doesn't lock the custom assemblies. This allows you to change the build location of your assembly to the Report Designer folder so that it always includes the most recent copy when you rebuild the assembly. As we explained in chapter 2, running your projects in the Preview window is a result of the code access security policy settings specified in the Report Designer configuration file (rspreviewpolicy.config).

Let's now look at another way of using custom code in reports in the form of XSL transformations.

6.3 USING XML-BASED REPORTS

So far in this chapter, we've seen how we can use custom code to extend report capabilities programmatically. For all its flexibility, custom code has its limitations. For example, besides hiding report items, you cannot control the report output programmatically. However, if you export your reports to XML, you can use custom code in the form of XSL transformations to precisely control the XML presentation of the report, as we will discuss next.

Strictly speaking, from an implementation standpoint, exporting a report to XML is no different than exporting it to any other rendering format, because the actual work is performed by the XML rendering extension (Microsoft.ReportingServices.XmlRendering.dll), which happens to be one of the supported RS extensions. However, I decided to devote a special place for it because, in my opinion, this is an extremely useful and important option.

Given the fact that the IT industry has embraced XML as the de facto standard for data exchange between heterogeneous platforms, exporting a report to XML opens a whole new world of opportunity. For example, in the B2B (business-to-business) scenario, an organization could expose an inventory report to its vendors. A vendor could request the report in XML to find out the current inventory product levels. The XML document could then be sent to a BizTalk server, which could extract the product information and send it to the manufacturing department. We will implement a similar solution in chapter 11.

6.3.1 Understanding XML exporting

The content of the following report elements could be exported to XML: textbox, rectangle, subreport, table region, list region, and matrix region. As a report author, you have full control over the XML presentation of these elements. To customize the XML-rendered output of the report, you use the Data Output tab of the report element's property pages. Which settings can be customized depends on the type of the element. In general, you can specify the following:

- Whether the report element and its content (for regions, groups, and rectangles) will be exported
- The XML element name
- Whether the report element will be rendered as an XML attribute or element

For example, at a report level, you can specify the root node name and XML schema. At the region level, you can specify whether the region and its items will be rendered at all. At the textbox level, you can tell the Report Server whether the textbox content will be rendered as an XML attribute or element.

When the Data Output settings are not enough, you can further fine-tune the XML output by using custom XSL transformations. For example, while skipping report elements is easy, adding additional XML nodes is not. In cases such as this, you can write an XSL transformation that will be applied by the Report Server after the report is rendered to XML.

Let's now look at a practical example that demonstrates how exporting to XML could be useful.

6.3.2 Exposing the report content as an RSS feed

While I was trying to figure out what a good XML report could be, my favorite RSS reader (IntraVNews) popped up a new window to let me know about the current news headlines. For those of you who are not familiar with this great information medium, RSS (which stands for all of the following: RDF Site Summary, Rich Site Summary, or Really Simple Syndication) is an XML-based format that allows information workers to describe and syndicate web content. Many organizations and individuals use RSS for *blogging*.

DEFINITION A *blog*, short for *web log*, is a personal journal that is frequently updated and intended for general public consumption.

This inspired me to see if we could expose a report as an RSS feed. To give our example a touch of reality, let's say that Adventure Works Cycles would like to take advantage of the increasing popularity of blogging with RSS feeds. In particular, the company management has requested these requirements:

- Future promotional campaigns must be exposed as an RSS feed. The AWC customers could subscribe to the feed using their favorite RSS newsreader and be notified about future product promotions.
- Each promotional item must include a hyperlink that will show more details about the campaign, such as discounted products and their sale prices.

Implementation options

How can we implement the above requirements? One approach could be to add the promotional information as static or dynamic web content to the company's web portal. For example, the products page could include a section that lists the current promotions. As far as exporting the promotional data as XML for the purposes of the RSS feed, we could create a Web Service that would query the Adventure Works database, retrieve the promotion details in XML, and write them into an RSS blog file.

Another implementation option could be to author an RS report that would supply both the HTML and XML content. The RSS Web Service could then request the report as XML and append the promotional information to the RSS blog file. The RSS item hyperlink could bring the customer to the HTML version of same report. Of course, the latter option assumes that you are willing to allow web users to access your Report Server directly by URL. This is not as bad as it sounds. If Windows authentication is an issue, you can replace it with a custom security extension to authenticate and authorize your web users, as we will discuss in Chapter 15.

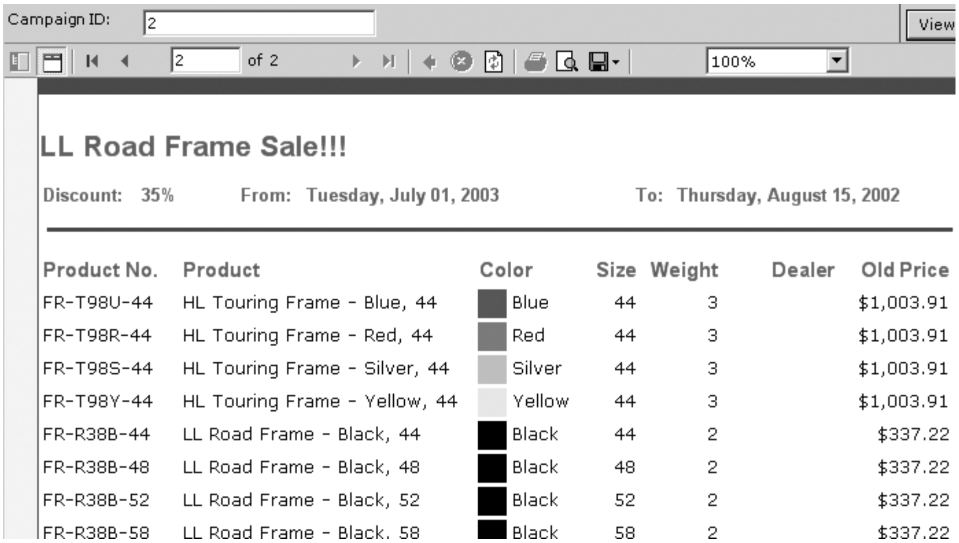
Which approach will work better for you depends on your particular needs and limitations. In our case, we will go for the latter to demonstrate the exporting-to-XML feature. To recap, our design goals for the new report sample will be as follows:

- Export the report to RSS-compliant XML format.
- Append the report XML to an RSS feed (we will postpone the actual implementation to chapter 9).

Implementing the report

Let's start by creating a new report called Sales Promotion. The report gets the promotional data from the SpecialOffer and SpecialOfferProduct tables. In addition, it takes one parameter, Campaign ID, which the user can use to request a specific campaign.

For example, figure 6.11 shows the second page of the Sales Promotion report when the user requests a campaign with an ID of 2.



Campaign ID: 2

2 of 2 — 100%

LL Road Frame Sale!!!

Discount: 35% From: Tuesday, July 01, 2003 To: Thursday, August 15, 2002

Product No.	Product	Color	Size	Weight	Dealer	Old Price
FR-T98U-44	HL Touring Frame - Blue, 44	■ Blue	44	3		$1,003.91
FR-T98R-44	HL Touring Frame - Red, 44	■ Red	44	3		$1,003.91
FR-T98S-44	HL Touring Frame - Silver, 44	■ Silver	44	3		$1,003.91
FR-T98Y-44	HL Touring Frame - Yellow, 44	□ Yellow	44	3		$1,003.91
FR-R38B-44	LL Road Frame - Black, 44	■ Black	44	2		$337.22
FR-R38B-48	LL Road Frame - Black, 48	■ Black	48	2		$337.22
FR-R38B-52	LL Road Frame - Black, 52	■ Black	52	2		$337.22
FR-R38B-58	LL Road Frame - Black, 58	■ Black	58	2		$337.22

Figure 6.11 The Sales Promotion report serves as both the RSS feed source and the HTML campaign details page.

As you can see, this report is very similar to the RS Product Catalog report sample, so we won't spend much time discussing its implementation details. Instead, we will focus on explaining how to export the report's content to XML.

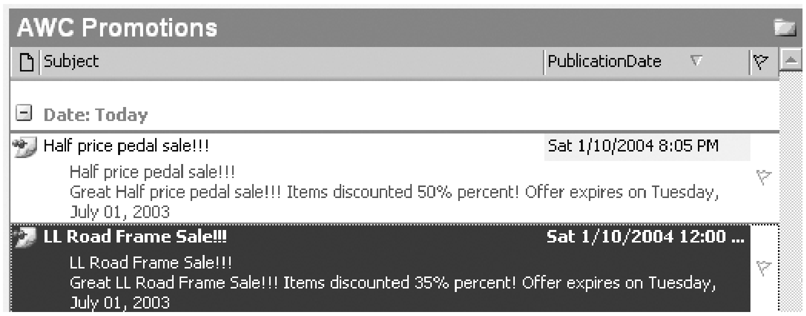
Understanding the RSS schema

What the report's XML output needs to be depends on which version of the RSS specification you have to support. For example, listing 6.4 shows what the sales promotion RSS feed should look like if it conforms to RSS version 2.0.

Listing 6.4 The Sales Promotion RSS feed to which the AWC subscribers will subscribe to be notified about sales promotions

```
<rss version="2.0">
<channel>    <— General feed-related header
  <title>AWC Promotions</title>
  <link>http://www.adventure-works.com/</link>
  <description>Great discounted deals!</description>
  <language>en-us</language>
  <ttl>1440</ttl>
  <item xmlns:n1="http://www.awc.com/sales" xmlns:xs="http://www.w3.org/
2001/XMLSchema">    <— Feed item
    <title>LL Road Frame Sale!!!</title>
    <link>http://localhost/reportserver?/AWReporter/Sales
                Promotion&SpecialOfferID=2&rs:Command=Render&rs:Format=XML
    </link>
    <description>Great LL Road Frame Sale!!!</description>
    <pubDate>Saturday, January 10, 2004</pubDate>
  </item>
  <item xmlns:n1="http://www.awc.com/sales" xmlns:xs="http://www.w3.org/
2001/XMLSchema">
    <!-Another item information here-
  </item>
</channel>
</rss>
```

Given the above feed, figure 6.12 shows how it gets rendered in the IntraVNews RSS Reader, which is integrated with Outlook:


AWC Promotions

Subject	PublicationDate
Date: Today	
Half price pedal sale!!!	Sat 1/10/2004 8:05 PM
Half price pedal sale!!! Great Half price pedal sale!!! Items discounted 50% percent! Offer expires on Tuesday, July 01, 2003	
LL Road Frame Sale!!!	Sat 1/10/2004 12:00 ...
LL Road Frame Sale!!! Great LL Road Frame Sale!!! Items discounted 35% percent! Offer expires on Tuesday, July 01, 2003	

Figure 6.12 The AWC Promotions feed rendered in IntraVNews

Let's now examine what needs to be done to massage the report output in order to make it compliant with the RSS schema.

Defining the report XML output

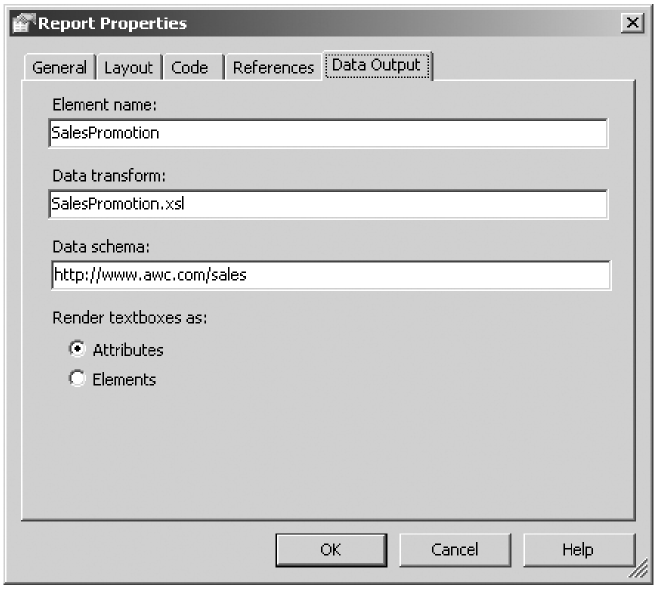
The first step required to export the report to an RSS-compliant format is to fine-tune its XML output. We've made a few changes using the Data Output tab for various elements, so the report renders to the abbreviated XML schema shown in listing 6.5.

Listing 6.5 The Sales Promotion report rendered in XML

```
<SalesPromotion xmlns="http://www.awc.com/sales" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="..." Name="Sales
Promotion" Date="2004-01-10T00:00:00.0000000-05:00">
  <Promotions>
    <Promotion Description="LL Road Frame Sale!!!">   <-- The Promotion element will represent an item in the RSS feed
      <ProductInfo>
        <Products>
          <Product ProductNumber="FR-T98U-44"
             Product="HL Touring Frame - Blue, 44" Color="Blue"
             Size="44" Weight="2.92" ListPrice="1003.9100"/>
          <Product ProductNumber="FR-T98R-44" Product="HL Touring Frame -
Red, 44" Color="Red"
             Size="44" Weight="2.92" ListPrice="1003.9100"/>
        </Products>
      </ProductInfo>
    </Promotion>
  </Promotions>
</SalesPromotion>
```

The most important change that you have to make is to explicitly set the XML Schema setting at the report level, as shown in figure 6.13.

If the Data Schema setting is not specified, the Report Server will autogenerate the XML document global namespace to include the date when the report is processed. This will interfere with referencing the document elements from an XSL transformation, so make sure you explicitly set the schema namespace.



Element name: SalesPromotion

Data transform: SalesPromotion.xsl

Data schema: http://www.awc.com/sales

Render textboxes as: [✓ Attributes] Elements

Figure 6.13
Use the Data Output report settings to define the report XML root element name and namespace.

Writing the XSL tranformation

Once you have finished making adjustments to the XML schema, the next step will be to write an XSL transformation to transform the XML output to an RSS-compliant format. To fit the Sales Promotion output to the RSS schema, we wrote the simple XSL transformation shown in listing 6.6.

Listing 6.6 Use XSL transformations to fine-tune the report's XML output.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:n1="http://www.awc.com/sales"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xsl:template match="/">
   <xsl:for-each select="n1:SalesPromotion/n1:Promotions/
```

```
      n1:Promotion">    <—  Loop through all Promotion elements
     <item>    <—  Generate an RSS item
      <title><xsl:value-of select="./@Description"/></title>
      <link>http://www.adventure-workds.com/promotions</link>
      <description>Great <xsl:value-of select="./@Description"/>
        Items discounted
        <xsl:value-of select="./@DiscountPct"/> percent! Offer
           expires on
         <xsl:value-of select="./@StartDate"/>
       </description>
      <pubDate><xsl:value-of select="./@StartDate"/></pubDate>
     </item>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

The XSL transformation simply loops through all sales promotions and outputs them in XML according to the RSS item specification. Strictly speaking, in our case there is always going to be only one XML sales promotion node, because we use a report parameter to select a single campaign. Finally, we need to add the XSL transformation file to our project. Similarly to working with images, we have to add the XSLT file to the same report project and subsequently upload it to the report catalog when the report is deployed. The Report Server cannot reference external XSLT files.

The last implementation step is to take care of appending the current sales promotion item to the RSS blog file. The easiest way to accomplish this would be to manually update the RSS feed XML file on the web server when there is a new promotional campaign. RSS newsreaders could reference this file directly, for example, by going to http://www.adventure-works.com/promotions.rss. Of course, if the requirements call for it, the process could also be fully automated. We will see how this could be done in chapter 9, where we will implement a table trigger that invokes a custom web service when a new sales promotion record is added to the database.

To subscribe to the RSS feed, AWC customers would configure their favorite RSS readers to point to the blog file. Once they do so, they will be notified each time the blog file is updated.

6.4 SUMMARY

In this chapter we learned how to integrate our reports with custom code that we or someone else wrote.

For simple report-specific programming logic, you can use embedded VB.NET code. When the code complexity increases or you prefer to use programming languages other than VB.NET, you can move your code to external assemblies.

For interoperability with different platforms and languages, you can export your reports to XML. You can control precisely the report output by using the Data Output tab coupled with custom XSL transformations.

By now, you should have enough knowledge to be able to author reports with Reporting Services. We'll now move on to the second phase of the report lifecycle: report management.

6.5 RESOURCES

The OpenForecast web site (http://openforecast.sourceforge.net/)

Microsoft Java Language Conversion Assistant
(http://msdn.microsoft.com/vstudio/downloads/tools/jlca/default.aspx)
Converts Java-language code to C#

Mark Russinovich's DebugView tool
(http://www.sysinternals.com/ntw2k/freeware/debugview.shtml)

What is RSS?
(http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html)
A good introduction to RSS

Lutz Roeder's .NET Reflector
(http://www.aisto.com/roeder/dotnet/)
Similar to the VS.NET Object Browser, Reflector is a class browser for .NET components.

Microsoft Reporting Services IN ACTION

Teo Lachev

Business reporting is a lifeline of business, so a better reporting environment is a big deal. With a sophisticated, modern tool like Microsoft Reporting Services, you can report-enable any type of application, regardless of its targeted platform or development language.

Written for information workers, system administrators, and developers, this book is a detailed and practical guide to the functionality provided by Reporting Services. It systematically shows off many powerful RS features by leading you through a dizzying variety of possible uses. Following a typical report lifecycle, the book shows you how to create, manage, and deliver RS reports.

In the first half, you will master the skills you need to create reports. System administrators will learn the ropes of managing and securing the report environment. The second half of the book teaches developers the techniques they need to integrate RS with their WinForm or web-based applications. It exercises RS through a wide variety of real-world scenarios—one of this book's strengths are its many useful examples.

What's Inside

- Extend RS with custom code
- Expose reports as RSS feeds
- Implement dynamic reports with Office Web Components
- Create reports off ADO.NET datasets
- Deliver reports to Web Services
- Customize RS security
- Evaluate RS performance and capacity
- and much more

A technology consultant with the Enterprise Application Services practice of Hewlett-Packard, Teo Lachev has more than 11 years' experience designing and developing Microsoft-centric solutions. He is a Microsoft Certified Solution Developer and a Microsoft Certified Trainer. Teo lives in Atlanta, GA.

AUTHOR ONLINE
Ask the Author

Ebook edition

www.manning.com/lachev

53995

9 781932 394221

ISBN 1-932394-22-2

MANNING

$39.95 US/$55.95 Canada