CHAPTER 35

COM Reporting Components

In this chapter

Understanding the Report Designer Component

Business Objects has long viewed the Component Object Model (COM) development platform as one of the key areas it needed to embrace to become successful. Although there were other popular developer platforms in the market, the trend for development projects concerning information delivery was to use Visual Basic. This was because of its good mix of power and simplicity. Now part of the Business Objects product line, Crystal Reports XI mirrors these attributes and delivers a powerful yet productive reporting solution. This chapter covers Business Objects reporting solutions for the COM platform, specifically, the Crystal Report Designer Component.

Although the chapters covering the Java and .NET components focused primarily on Web-based applications, this chapter concentrates on desktop applications because that is the focus of the Business Objects COM Components. Desktop applications, although still popular today, were what started it all. These are standalone applications that run on a single tier and are installed locally on a user's machine. These applications are most commonly built using Visual Basic, but are also sometimes built using Visual C++ or Delphi.

NOTE

All sample code in this chapter uses Visual Basic 6 syntax, but can easily be adapted to other languages that support COM. For sample code in other languages, visit the Business Objects support site at http://support.businessobjects.com.

Many development environments support Microsoft's COM technology. *COM (Component Object Model)* is a standard technology used for exposing Software Development Kits (SDKs) in the Windows world. It implies a set of objects with properties and methods. Much of Microsoft's own SDKs are based on COM. It follows that the recommended Crystal Reports SDK for desktop applications would also be based on COM. Its name is the Report Designer Component, and it consists of the following pieces:

- A report designer integrated into the Visual Basic environment
- An object model built around the report engine used for manipulation of the report
- A report viewer control used for displaying reports inside an application

The following sections describe each of these components in more detail.

Building Reports with the Visual Basic Report Designer

The Visual Basic report designer enables developers to create and edit reports from within the comfort of the Visual Basic environment. Figure 35.1 shows the report designer active inside Visual Basic.

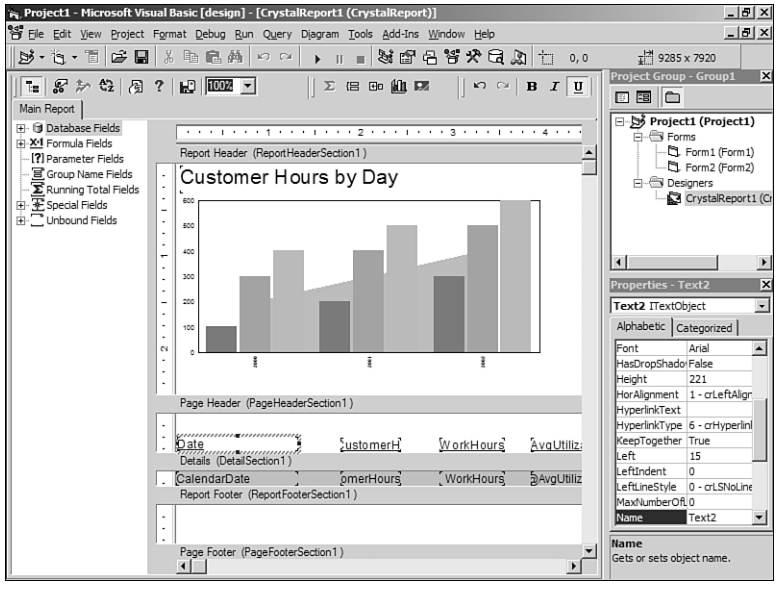

Figure 35.1
Here a report is shown being editing in the Visual Basic report designer.

To add a new report to a project, select Add Crystal Reports XI from the Project menu inside Visual Basic.

NOTE

> If Add Crystal Reports XI is not showing on the Project menu, go to the Project, Components menu, and on the Dialog tab, make sure Crystal Reports XI has a check beside it. If you turn this on, it permanently appears on the Project menu.

From the dialog that opens, select Using the Report Wizard or As a Blank Report to create a new report from scratch. The From an Existing Report option provides you with the capability to import any existing Crystal Report file (`.rpt`) and use the Visual Basic report designer to make further modifications, a great way to leverage any existing work an organization has put into Crystal Reports. A report that is added to a Visual Basic project is saved as a `.dsr` file, which is a container for the actual `.rpt` file along with some other information. At any point, you can click the Save to Crystal Report File button on the designer's toolbar and save the report out to a standard RPT file, so in effect reports can easily go both ways: in and out of Visual Basic. Because the Visual Basic report designer is based primarily on the same code-base as the standalone Crystal Reports designer, the RPT file format is the same. You can also import existing reports from past versions into the Visual Basic report designer.

The Visual Basic report designer supports almost all the features of the Crystal Reports designer and can be used to create everything from simple tabular reports to highly formatted, professional reports. However, even though the capabilities of these two editions are

similar, there are some differences in the way the designer works. This is not meant to be inconsistent, but rather to adapt some of the Crystal Reports tasks to tasks with which Visual Basic developers are familiar. Ideally, the experience of designing a report with the Visual Basic report designer should be like designing a Visual Basic form. The following sections cover these differences.

Understanding the User Interface Conventions

Several user interface components work differently in the Visual Basic report designer. One of the first things you might notice is that the section names are shown above each section on a section band as opposed to being on the left side of the window. However, the same options are available when right-clicking on the section band. This tends to be more convenient anyway.

The Field Explorer resides to the left of the report page. Although it cannot be docked, it can be shown or hidden by clicking the Toggle Field View button on the designer toolbar. Other Explorer windows found in the standalone designer such as the Report Explorer and Repository Explorer are not available in the Visual Basic report designer.

NOTE

Reports that contain objects linked to the Crystal Repository are fully supported; however, no new repository objects can be added to the report without using the standalone designer.

The menus that you would normally find in the standalone Crystal Reports designer can be found by right-clicking on an empty spot on the designer surface. The pop-up menu provides the same functionality.

Modifying the Report Using the Property Browser

To change the formatting and settings for report objects in the standalone designer, users are familiar with right-clicking on a report object and selecting Format Field from the pop-up menu. This opens the Format Editor, which gives access to changing the font, color, style, and other formatting options. In the Visual Basic report designer this scenario is still available; however, there is an additional way to apply most of these formatting options: the Property Browser.

The Property Browser is a window that lives inside of the Visual Basic development environment. It should be very familiar to Visual Basic developers as a way to change the appearance and behavior of a selected object on a form or design surface. In the context of the report designer, the Property Browser is another way to change the settings (properties) for report objects. In general, any setting that is available in the Format Editor dialog is available from the property browser when that object is selected. To see which properties are available for a given object, click on it, and check out the Property Browser window shown in Figure 35.2.

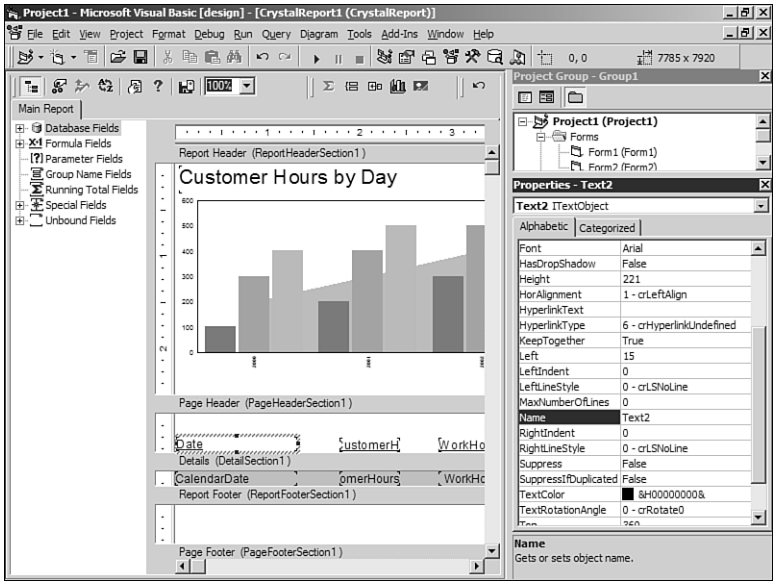

Figure 35.2
Changing a report object's settings via the Property Browser is shown here.

The property names are listed on the left and the current values are listed on the right. To choose a value, simply click on the current value and either type or select from the drop-down list.

One property to pay attention to is the `Name` property. This becomes relevant in the next section when you learn how to use the Report Engine Object Model to manipulate the report on the fly at runtime. This is the way to reference that object in code. Also of note is that the properties shown in the Property Browser map to the same properties that are available programmatically via the object model. If you see a property there, this means it is also available to be changed dynamically at runtime.

UNBOUND FIELDS

The Field Explorer in the Visual Basic report designer has an extra type field not found in the standalone report designer. These are called *unbound fields*. There is one type of unbound field for each data type. These fields are used to build dynamic reports. Because they do not have a predefined database field mapped to them, they provide a way to change the locations of fields on the report by using some application logic. The reason they each have their own data type is so that type-specific formatting can be applied such as the year format for a date object, or the thousands separator for a numeric object. Unbound fields are revisited later in this chapter.

NOTE

When you create an unbound field, it also shows up as a formula in the Formula Fields list. This is because a formula is used behind the scenes of an unbound field. The best practice is not to edit this as a formula field.

PROGRAMMING WITH THE REPORT ENGINE OBJECT MODEL

The object model is the main entry point to the Crystal Reports engine for desktop applications. As mentioned earlier, it is based on COM and can be used from any COM-compliant development environment. Although the main library's filename is `craxdrt.dll`, the more important thing to know is that it shows up in the Project References dialog as Crystal Reports ActiveX Designer Runtime Library 10.0, as shown in Figure 35.3. After a reference is added to this library, a new set of objects will be available to you. These objects are contained in a library called `CRAXDRT`. To avoid name collisions, it's probably a good idea to fully qualify all object declarations with the `CRAXDRT` name, for example, `Dim Param As CRAXDRT.ParameterField`.

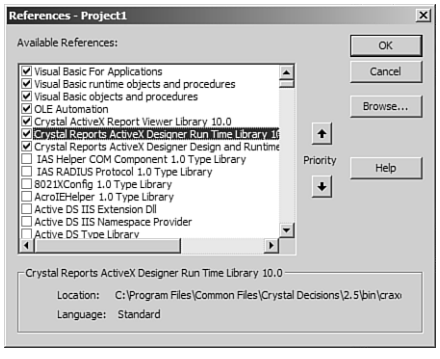

Figure 35.3
Reference the Report Designer Component's Object Model in the References dialog.

NOTE

When adding a Crystal Report to your project, a reference is automatically added to this library for you. You can use the `CRAXDRT` library right away.

In addition to many other features, the object model provides the capability to open, create, modify, save, print, and export reports. This section covers some of the more common scenarios a developer might encounter.

The main entry point to the object model is the `Report` object. This object is the programmatic representation of the report template and provides access to all the functions of the SDK. There are three ways to obtain a `Report` object:

- Load an existing RPT file from disk
- Create a new report from scratch
- Load an existing strongly typed report that is part of the Visual Basic project

The first two methods involve the `Application` object. Its two key methods are `NewReport` and `OpenReport`. As the name implies, the `NewReport` method is used to create a blank report

and the `OpenReport` method is used to open an existing report. Creating a new report is useful if the application needs to make a lot of dynamic changes to the report's layout on the fly. This way all report objects can be added dynamically.

NOTE

> Although many purist developers are tempted to not use any predefined report templates (RPT files) and create all reports on the fly, this tends to be overkill for most projects. There is a lot of work in having to programmatically make every single addition to a report. It's usually a better plan to have some RPT files as part of the application and then make some small modifications at runtime.

The other option is to use the `OpenReport` method, which takes a filename to an RPT file as a parameter. This opens an existing report. When using this method, the RPT files must be distributed along with the application. The advantage of having these externals files is that you can update the reports without updating the application.

The last method is to use a strongly typed report. A *strongly typed report* is one that is added to the Visual Basic project and turned into a DSR file. Whichever name you give that report, a corresponding programmatic object exists with the same name. For example, if you save a report as BudgetReport.DSR, you can create that report programmatically with the following code:

```
Dim Report As New BudgetReport
```

There are several advantages to using strongly typed reports. First, all report files are bound into the project's resulting executable so no external files are available for users to modify and mess up. Second, not only is the name of the report strongly typed (BudgetReport in the previous example), but also the section and report objects. For example, if you have a text object acting as a column header and you want to modify this at runtime, it's very easy to access the object by name like this:

```
Report.ColumnHeader1.SetText "Some text"
```

The long-handed way of doing this would look something like this:

```
Dim field as CRAXDRT.FieldObject
Set field = Report.Sections("PH").ReportObjects(3)
Field.SetText "Some text"
```

Not only is this last method longer, you'd also have to refer to the report object by index instead of name, which can become problematic. The following sections discuss some of the common tasks that are performed after a Report object is obtained.

EXPORTING REPORTS TO OTHER FILE FORMATS

A very common requirement for application developers is to be able to export a report through their application. Not only do developers want a variety of formats, they want the export to happen in a variety of ways, for example, having the user select where to save the report, saving to a temp file and then opening it, e-mailing it to somebody else, and so on.

By being creative with exporting, you can create some very powerful applications. The Report Designer Component object model provides a very flexible API to meet these broad needs. This section covers the basics of exporting.

There are two components to exports: the format and the destination. The developer specifies both of these through the `ExportOptions` property of the `Report` object.

Setting the format options involves two steps. The first step is to choose which format you want to export to. Sometimes an application provides the user a list of export formats and lets him choose, other times the export type will be hardcoded. In any case, simply setting the `FormatType` property of the `ExportOptions` object specifies this. This property accepts a number. For example, to export to PDF, pass in 31. Remembering which number represents which format is tough so there are some enumerations with descriptive names that make this easier.

NOTE

For a full list of enumerations, consult the Crystal Reports Developers Help file and look at the `CRExportFormatType` enumeration in the Visual Basic object browser.

To help you get started, here are some of the more popular export format enumeration values:

- **PDF**—`crEFTPortableDocFormat`
- **Word**—`crEFTWordForWindows`
- **Excel**—`crEFTExcel97`
- **HTML**—`crEFTHTML40`
- **XML**—`crEFTXML`

Generally, all you need to do to set the format options is set the `FormatType` property. However, many of the format types have some additional options. For example, when exporting to Excel there is an option to indicate whether you want the grid lines shown. To handle these extra settings, there are some other properties off the `ExportOptions` object whose names begin with the format type. In the Excel grid lines example, the property is called `ExcelShowGridLines`. For PDF, there are `PDFFirstPageNumber` and `PDFLastPageNumber` properties that indicate which pages of the report you want exported to PDF. You can determine what options are available by checking out the Crystal Reports Developer Help file and looking at the `ExportOptions` object.

After the format is set up, you need to tell Crystal Reports where you want this report to be exported. This is called the *export destination*. The most common destination is simply a file on disk but there are destinations such as e-mail or Microsoft Exchange folders where reports can be automatically sent. The export destination is set via the `DestinationType` property of the `ExportOptions` object. Some example values are listed here:

- **File**—`crEDTDiskFile`
- **E-mail**—`crEDTMailMAPI`
- **Exchange**—`crEDTMicrosoftExchange`

Check out the `CRExportDestinationType` enumeration to see the other available options. Like the format, the destination has a set of additional options. The most obvious one is when setting the destination to a file (`crEDTDiskFile`), you would need to specify where you want this file and what its name should be. This is accomplished by setting the `DiskFileName` property. Other properties on the `ExportOptions` object are available such as the `MailToList` property, which is used to indicate who the report should be mailed to if the e-mail option is selected as the destination.

The final step in exporting is to call the `Report` object's `Export` method. It takes a single parameter: `promptUser`. If this is set to true, any options previously set on the `ExportOptions` object are ignored and a dialog appears asking the user to select the format and destination. This can be useful if you want the user to have the capability to use any export format and any destination. If you would like a more controlled environment, you can set `promptUser` to false. When this is done the previously selected values from the `ExportOptions` object are respected and the export is done without any user interaction besides a progress dialog popping up while the export is happening. This progress dialog can also be suppressed by setting the `Report` object's `DisplayProgressDialog` property to false. Listing 35.1 provides an example of a report being exported to a PDF file without any user interaction.

LISTING 35.1 EXPORTING TO PDF

```
Dim Report As New CrystalReport1

' Set export format
Report.ExportOptions.FormatType = crEFTPortableDocFormat

' Set any applicable options for that format
' In this case, set to only export pages 1-2
Report.ExportOptions.PDFFirstPageNumber = 1
Report.ExportOptions.PDFLastPageNumber = 2

' Set export destination
Report.ExportOptions.DestinationType = crEDTDiskFile

' Set any applicable options for the destination
' In this case, the filename to be exported to
Report.ExportOptions.DiskFileName = "C:\MyReport.pdf"

' Turn all user interface dialogs off and perform the export
Report.DisplayProgressDialog = False
Report.Export False
```

Printing Reports to a Printer Device

Although it's helpful to view reports onscreen and save some paper, many times reports still need to be printed. To accomplish this, there is a collection of methods for printing reports available from the `Report` object. The simplest way to print a report is to call the `PrintOut` method passing in true for the `promptUser` parameter as shown here:

```
Report.PrintOut True
```

This opens the standard Print dialog that enables the user to select the page range and then click OK to confirm the print. The limitation to this is that the pop-up dialog does not enable the user to change the destination printer. Because this is a common scenario, this method isn't used very often. Instead, the `PrinterSetup` method is called. This method pops up a standard printer selection dialog that enables the user to change the paper orientation or printer.

Keep in mind that calling the `PrinterSetup` method does not actually initiate the print; it only collects the settings to be used for the print later on. Luckily it does indicate via a return value whether the user clicked the OK or Cancel button. Listing 35.2 shows an example of how to use the `PrinterSetup` method to set printer options.

Listing 35.2 Printing a Report Interactively

```
' Call PrinterSetup to set printer, paper orientation, and so on
If Report.PrinterSetupEx(Me.hWnd) = 0 Then
   ' If the return value is 0, the user did not click Cancel
   ' so go ahead with the print
   Report.PrintOut False
End If
```

To print a report without any user interaction, call the `PrintOut` method passing in false for the `promptUser` parameter. Options such as pages and collation can be set with the additional argument to the `PrintOut` method. To change the printer, call the `SelectPrinter` method. This accepts the printer driver, name, and port as parameters and performs the printer change without any user interaction. Listing 35.3 illustrates a silent print.

Listing 35.3 Printing a Report Silently

```
' Call PrinterSetup to set printer, paper orientation, and so on

' Set paper orientation
Report.PaperOrientation = crLandscape

' Set printer to print to
'   pDriver -- for example: winspool
'   pName -- for example: \\PRINTSERVER\PRINTER4
'   pPort -- for example: Ne00:
Report.SelectPrinter pDriver, pName, pPort

' Initiate the print
Report.PrintOut False
```

SETTING REPORT PARAMETERS

Often reports delivered through an application need to be dynamically generated based on a parameter value. If a report with parameters is viewed, exported, or printed, a Crystal parameter prompting dialog pops up and asks the user to enter the parameter values before the report is processed. This parameter prompting dialog requires no code. The use of the object model comes into play when a developer wants to set parameters without user interaction. This is done via the `ParameterFieldDefinitions` collection accessed via the `Report` object's `ParameterFields` property. If all parameter values are provided before the report is processed, the parameter dialog is suppressed.

Parameters can be referenced by name or by number. To reference by name, call the `ParameterFields` object's `GetItemByName` method passing in the name of the parameter you want to access. This returns a `ParameterField` object. Alternatively, use the indexer on the `ParameterFields` object; for example, `ParameterFields(1)`. When referencing by index, the parameters will be stored in the same order they appear in the Field Explorer window in the report designer. After a `ParameterField` object is obtained, simply call the `AddCurrentValue` method to set the parameter's value as shown in Listing 35.4.

LISTING 35.4 SETTING PARAMETERS

```
Dim Application As New CRAXDRT.Application
Dim Report As CRAXDRT.Report

' Open the report from a file
Set Report = Application.OpenReport("C:\MyReport.rpt")

Dim p1 as ParameterField
Set p1 = Report.ParameterFields.GetItemByName("Geography")
p1.AddCurrentValue("Europe")


Dim p2 as ParameterField
Set p2 = Report.ParameterFields(2)
p2.AddCurrentValue(1234)
```

If the parameter accepts multiple values, simply call the `AddCurrentValue` method multiple times. For range parameters where there is a start and an end value, use the `AddRangeValue` method.

Sometimes a developer wants to prompt the user to enter some or all of the parameters but they want to control the user interface. Much information about the parameter can be obtained by reading its properties:

- **`ParameterFieldName`**—Name of the parameter
- **`ValueType`**—The data type of the parameter (string, number, and so on)
- **`Prompt`**—The text to use to prompt for this parameter

Also, by using the `NumberOfDefaultValues` property and `GetNthDefaultValue` method, a developer can construct her own pick-list of default parameter values that is stored in the report.

NOTE

> For more information on the other properties and methods available on the `ParameterField` object, consult the Crystal Reports Developer Help file and look for the `ParameterFieldDefinition` object.

Setting Data Source Credentials

Although the sample reports that come with Crystal Reports XI use an unsecured Microsoft Access database as their data source, most real-world reports are based on a data source that require credentials (username, password) to be passed. Also, it's very common to want to change data source information such as the server name or database instance name via code. This section covers these scenarios.

Unlike parameters, there is no default-prompting dialog for data source credentials. They must be passed via code. The server name, location, database name, and username are all stored in the report. However, the password is never saved. A report will fail to run if a password is not provided.

Most reports only have a single data source but because it is possible for reports to have multiple data sources that in turn would require multiple sets of credentials, setting credentials isn't something that's done on a global level. Credentials are set for each table in the report. Tables are represented by an object called a `DatabaseTable` inside the object model. The following code snippet illustrates the hierarchy required to get at the `DatabaseTable` object.

```
Report
   Database
      DatabaseTables
         DatabaseTable
```

Tables are accessed by their index, not their name. The indexes in the object model are all 1-based and are in the order you see them in the Field Explorer in the report designer. To access the first table in the report, you could do this:

```
Dim tbl as DatabaseTable
Set tbl = Report.Database.Tables(1)
```

After the correct `DatabaseTable` object is obtained, use the `ConnectionInfo` property bag to fill in valid credentials. If you do only have one data source in the report, but multiple tables from that data source, you need not set credentials for each one. The information is propagated across all tables. Listing 35.5 illustrates setting the server name, database name, username, and password for a report based off an OLEDB data source.

LISTING 35.5 SETTING DATA SOURCE CREDENTIALS

```
' Provide database logon credentials (in this case
' for an OLEDB connection to a SQL Server database)
Dim tbl as CRAXDRT.DatabaseTable
Set tbl = Report.Database.Tables(1)
tbl.ConnectionInfo("Data Source") = "MyServer"
tbl.ConnectionInfo("Initial Catalog") = "MyDB"
tbl.ConnectionInfo("User ID") = "User1"
tbl.ConnectionInfo("Password") = "abc"
```

Each type of data source has its own set of properties. OLEDB has a Data Source, which is the server name whereas the Microsoft Access driver has a Database Name, which is a file-name to the MDB file. The `ConnectionInfo` property bag is introspective so you can loop through and determine what properties are available.

MAPPING UNBOUND FIELDS AT RUNTIME

Earlier in this chapter you saw that a new type of field called an unbound field can be added to the report with the Visual Basic report designer. Using the object model, these unbound fields can be mapped to database fields in the report at runtime. This is done two different ways: manually or automatically.

The manual method is to use the `SetUnboundFieldSource` method of the `FieldObject`. This method takes a single parameter, which is the name of the database field to be mapped in the Crystal field syntax, such as {`Table.Field`}. If a strongly typed report is being used, that is, a report added to the Visual Basic project, the `UnboundField` objects can be referenced as properties of the `Report` object. For example, an unbound field object given the default name of `UnboundString1` can be referenced like this:

```
Report.UnboundString1.SetUnboundFieldSource "{Customer.Customer Name}"
```

If a report is loaded at runtime, there are no strongly typed properties so the `FieldObject` needs to be found under the Section and ReportObjects hierarchy. The following example gets a reference to the first unbound field in the details section:

```
Dim fld As FieldObject
Set fld = Report.Sections("D").ReportObjects(1)
fld.SetUnboundFieldSource "{Customer.Customer ID}"
```

The automatic method is to simply call the `Report` object's `AutoSetUnboundFieldSource` method. This assumes that any unbound fields to be mapped are named to match a database field. Initially this might seem strange because the whole point of an unbound field is that the developer doesn't know which database field it will be mapped to at design time. However, this automatic method is valuable when the database table doesn't exist at design time, and instead is added at runtime based on some dynamic data.

USING THE CODE-BEHIND EVENTS

One of the reasons that the report is saved as a DSR file instead of just an RPT file is that the DSR file contains some code that is attached to the report file. This code, often called

code-behind, is event-handing code for several events that the report engine fires. The following list describes events that are fired and their corresponding uses:

- **`Initialize (Report)`**—Fired when the report object is first created. This event can be useful for performing initialization-related tasks.
- **`BeforeFormatPage/AfterFormatPage (Report)`**—Fired before and after a page is processed; can be useful for indicating progress.
- **`NoData (Report)`**—Fired when a report is processed but no records were returned from the data source. Sometimes a report with no records is meaningless and thus should be skipped or the user should be warned; this event is a great way to handle that.
- **`FieldMapping (Report)`**—Fired when the database is verified and there has been a schema change; this event enables you to remap fields without user interaction.
- **`Format (Section)`**—Fired for the rendering of each section. This is useful for handling the detail section's event and performing conditional logic.

DELIVERING REPORTS USING THE REPORT VIEWER

In the previous section, only printing and exporting were mentioned as options for delivering reports. You might have been wondering how to view reports onscreen. This section will cover using the report viewer to view reports. This report viewer control is usually referred to as the ActiveX viewer, or the Crystal Reports Viewer Control. It is an ActiveX control, which means that in addition to being able to be dropped on to any Visual Basic form—like the other components of the Report Designer Component—it can be used in any COM-compliant development environment. Its filename is CRViewer.dll. Figure 35.4 depicts the ActiveX viewer displaying a report from a Visual Basic application.

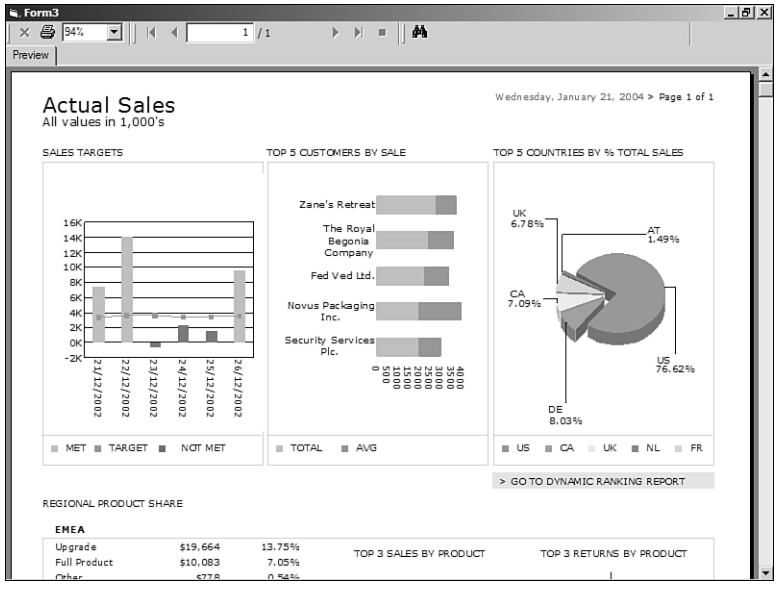

Figure 35.4
A Crystal Report is shown here being displayed in the ActiveX viewer.

The ActiveX viewer works in conjunction with the object model and report engine to render the report to the screen. The object model talks to the report engine to process the report, and then the ActiveX viewer asks the object model for the data for an individual page. After this data is received by the viewer, it displays the report page onscreen. The following code snippet illustrates how to view a report with the report viewer control:

```
Dim Application As New CRAXDRT.Application
Dim Report As CRAXDRT.Report

Set Report = Application.OpenReport("C:\MyReport.rpt")
CRViewer.ReportSource = Report
CRViewer.ViewReport
```

The ActiveX control has many properties and methods that enable you to customize its look and feel. To turn off the toolbar at the top of the viewer control, simply set the `DisplayToolbar` property to false. To turn off the group tree, set the `DisplayGroupTree` property to false. This can result in a very minimalist viewer. In addition, the control has a full event model that notifies you when certain actions are performed, such as a drill-down or page navigation. For more information on the ActiveX viewer control, consult the Crystal Reports XI developer help file.

USING THE OBJECT MODEL TO BUILD BATCH REPORTING APPLICATIONS

So far this chapter has focused on on-demand reporting, meaning that reports are processed as they are requested and they generally go away when the viewing or printing is completed. One of the biggest uses of the Report Designer Component today is for batch reporting; that is, running a large number of reports at once. This section covers some features and best practices relevant to batch reporting.

WORKING WITH REPORTS WITH SAVED DATA

When using the standalone report designer, you might have noticed an option on the File menu called Save Data with Report. This enables a report to be saved with the last returned dataset so that it can be viewed again without connecting to the database. Reports with saved data are in effect an offline report.

Applications using the Report Designer Component can both create and view reports with saved data. This enables you to run a batch of reports and then be able to view them at any point later. This can be useful for reports based on queries that take a long time to run, or also for achieving an archiving process for reports.

Creating a report with saved data is very simple. You just export to the Crystal Reports format by using the `crEFTCrystalReport` identifier. All exported reports have saved data. You can control where this report is saved and archive it for later.

Viewing a report with saved data doesn't actually require any code at all. The logic of the report engine is: If the report has saved data, use it and only hit the database again if the

user clicks the Refresh button or the developer forces a refresh by calling the `DiscardSavedData` method off the `Report` object. You can always tell which copy of the data is being used from examining the `DataDate` property of the `Report` object.

Hopefully you can imagine how applying this principle to batch reporting would be powerful. A set of reports could be run overnight, producing another set of reports with saved data that can be viewed offline.

LOOPING THROUGH REPORTS

Another scenario that is relevant to batch reporting is looping through a set of reports. A common example is running either one report many times with different parameters (such as a bank statement) or running a large collection of reports all at once (such as financial statements).

These scenarios can be accomplished by using external report files and writing a loop that opens a report, prints or exports it, and then closes it. The best way to close a report is to set the `Report` object to `Nothing`:

```
Set Report = Nothing
```

This releases the COM object and releases the report job from memory.

Also, the `CRAXDRT` library is thread safe, which means that multiple threads can be calling into it at the same time. If a large number of reports need to be processing in a very small amount of time, you can spawn as many as five simultaneous threads that are all running reports at the same time.

TROUBLESHOOTING

ADD CRYSTAL REPORTS XI

"Add Crystal Reports XI" is not showing on my Project menu.

If you don't see "Add Crystal Reports XI" on the Project menu, go to the Project, Components menu, and on the Dialog tab make sure Crystal Reports XI has a check beside it. If you turn this on, it permanently appears on the Project menu.