Chapter 8
.NET Business Objects

The VFP community has known about the importance of business objects for several years now. They continue to be extremely important in all types of .NET applications including Web Forms, Window Forms, and Web Services. This chapter explains what business objects are, why you should use them, and how to implement them in .NET.

In most .NET books, documentation, white papers, and Internet resources available these days, you hear little mention of business objects. This isn't because business objects are unimportant—just the opposite; they are extremely important in building flexible applications that are scalable and easy to maintain. Rather, I think this omission is because business objects are perceived as being too advanced when first learning a new technology. While there may be some truth to that, hopefully this chapter will help you grasp the concept of business objects and take your software development to a new level.

In this chapter you'll learn the basic mechanics of creating and using business objects. In subsequent chapters, you'll see how they can be used in Windows Forms, Web Forms, and XML Web Services.

What is a business object?

A business object is an object that represents a real-world entity such as a person, place, or business process (**Figure 1**).

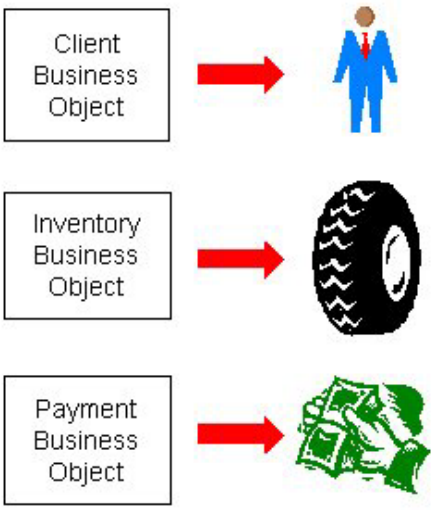

***Figure 1**. Business objects can represent real world entities such as a person, place, or business process.*

For example, in the real world, you have clients. You can create a business object that represents clients. In the real world, you have inventory and payments. You can create business objects that represent inventory and payments.

This concept isn't completely foreign to Visual FoxPro developers. When you design a data model, you often create data that represents the characteristics of real-world entities. For example, you create a client table that represents the characteristics of a client in the real world—a name, address, phone number, e-mail address, and so on. You can create an inventory table that represents the characteristics of different inventory items, a payment table that represents the characteristics of payments, and so on (**Figure 2**).

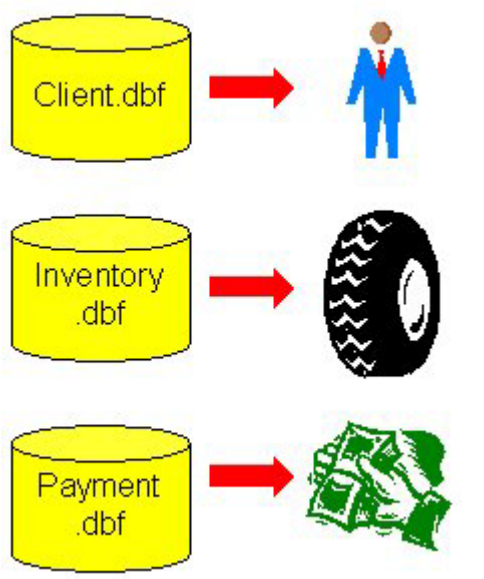

***Figure 2**. Data can represent the characteristics of real world entities, but this is only half the picture.*

Modeling the characteristics of a real-world entity by means of data is only half the story. The other half is modeling its behavior—and you do that by means of code. This includes data manipulation code as well as any business logic (or application logic) code associated with the entity. For example, you may have code that calculates the tax on an invoice, totals the payments made by a client, or calculates a client's credit limit. Business objects bring together both the characteristics and behavior of a particular entity (**Figure 3**).

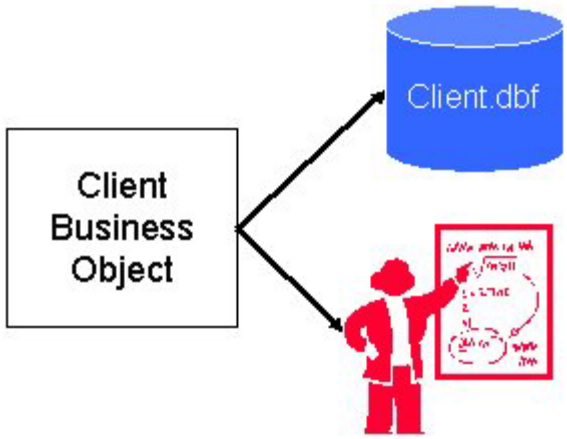

***Figure 3**. Business objects bring together both the characteristics and behavior of a single real-world entity.*

In an application that uses business objects, all the code for a particular entity is stored in the corresponding business object. For example, all the code that has something to do with clients is stored in the client business object; all the code that has something to do with inventory is in the inventory business object; and yes, all the code that has something to do with payments is stored in the payments business object.

The simplest aspect of modeling an entity's behavior is data manipulation. Just about every business object needs methods that allow you to add, edit, retrieve, save, and delete records—these translate to actions that occur in the real world. For example, when you acquire a new client, this corresponds to the ability of a business object to add a new record. When you lose a client, this may correspond to the ability of a business object to delete a record (or possibly mark it as "inactive"). When some attribute of a client changes (address, phone number, e-mail address), this corresponds to the ability of a business object to retrieve, edit, and save a client record.

Examining business objects in popular software applications

The concept of using business objects in software applications is very common among popular off-the-shelf software packages. Examining the business objects in these applications can help you better understand how to implement business objects in the software you create.

A good place to start is by exploring the business object model of Microsoft Word. The primary business object in this model is the Document object, which represents a real-world Microsoft Word document. You can view Word's object model by using Visual Studio .NET's Object Browser. To launch the Object Browser, select View | Other Windows | Object Browser from the menu. To open up the Microsoft Word object model, click the Customize button at the top of the Object Browser, which launches the Selected Components dialog. Click the Add button, which launches the Component Selector dialog (**Figure 4**). Click the COM tab and select Microsoft Word 9.0 Object Library in the component list (the version number differs depending on which version of Microsoft Word is on your computer). Click the Select button to add Microsoft Word to the Selected Components list, and then click OK to close the

Component Selector dialog. Click the OK button in the Selected Components dialog to add it to the Object Browser. This adds a Word node to the Object Browser tree view.

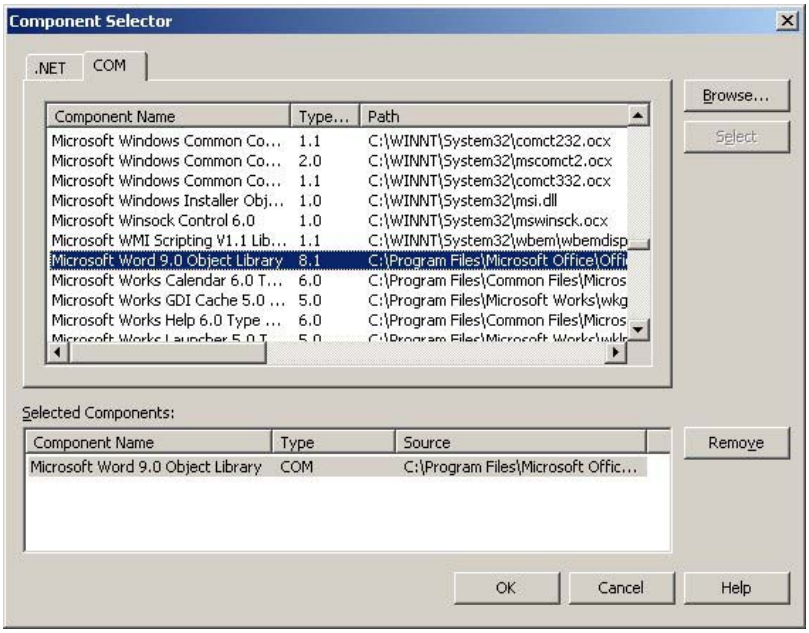

***Figure 4**. The Component Selector allows you to add components to the Object Browser for viewing.*

If you expand the Word node, you see quite a few items listed. If you select the Document business object in the left pane, it displays the members of the Document object in the right pane (**Figure 5**).

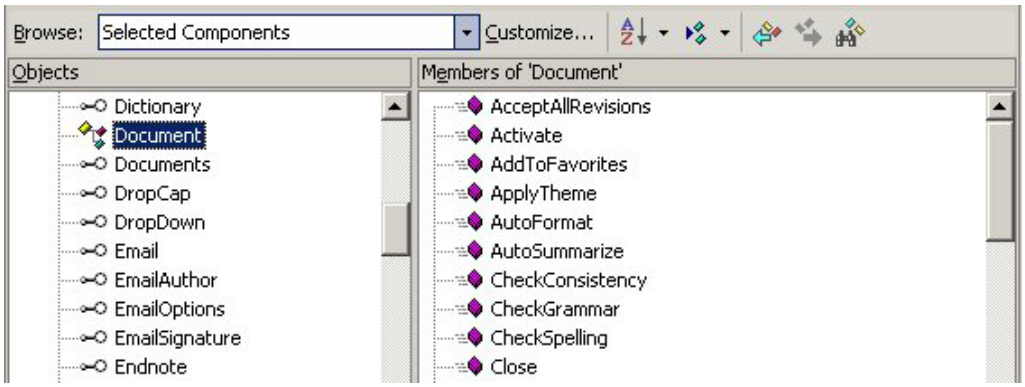

***Figure 5**. You display the members of a business object by selecting the object in the left pane of the Object Browser.*

You can determine which items are business objects in the Object Browser because they have the same icon as shown to the left of the Document business object shown in Figure 5.

You can get quite an education by examining the methods of the Document business object. For example, in the real world you can perform a wide variety of actions against a document such as:

- Check grammar
- Check spelling
- Print it
- Undo changes
- Close It

In the Document business object the methods represent these different actions:

- CheckGrammar
- CheckSpelling
- PrintOut
- Undo
- Close

For another example, open up the Microsoft Internet Controls object library in the VS .NET Object Browser (**Figure 6**).

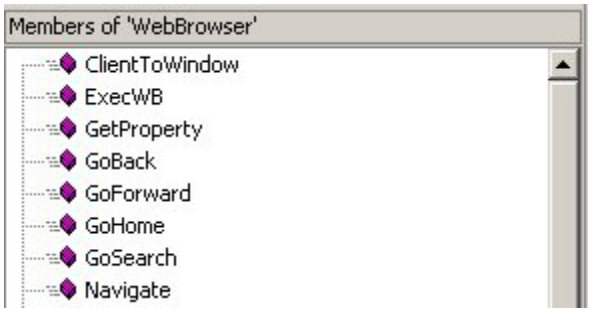

***Figure 6**. The Internet Explorer WebBrowser business object represents a real-world web browser.*

This library contains a WebBrowser business object that represents a real-world browser. It contains methods such as GoBack, GoForward, GoHome, Navigate, Refresh, and so on. Each one represents real-world actions you can perform with a web browser.

One of the main points to realize from looking at these object models is when Microsoft created these tools they did not put the application logic in the user interface of Microsoft Word and Internet Explorer. Instead, they created business objects possessing events and methods that contain the business logic. The methods are intuitive and perform a discrete action that accomplishes a well-defined objective. If they change the interface of the tool, they don't have to move or rewrite all of the internal code that handles operations like SpellCheck and GoHome.

Monolithic vs. three-tier applications

A *monolithic* application is a software application where the user interface, business logic, and data are inextricably bound to each other. Typically this type of application does not use business objects. Despite the benefits that business objects can provide, most developers (Visual FoxPro and otherwise) continue to build monolithic applications. This should come as no surprise, because most of the software development tools on the market actually encourage you to build applications this way.

For example, think about the Data Environment builder in Visual FoxPro. The Form Designer and Report Designer allow you to use the Data Environment builder (**Figure 7**) to specify the data to be loaded by a form or report.

Although these tools can help you build applications rapidly, they don't provide the most scalable solution. For example, if you load Visual FoxPro tables into the data environment of a form, what happens when you want to move to SQL Server or Oracle? You have to spend weeks or months tearing apart your application and putting it back together again.

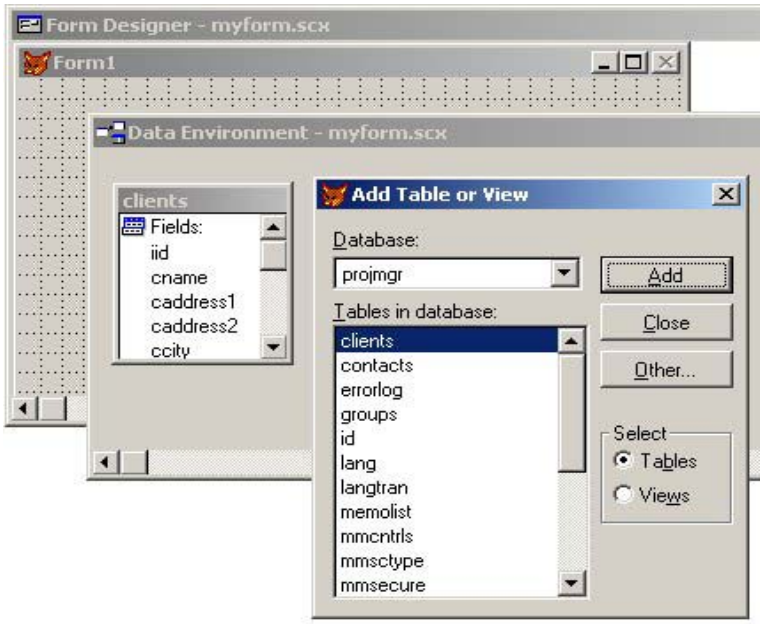

***Figure 7**. The Data Environment builder in Visual FoxPro 7 encourages you to create monolithic applications by binding your user interface directly to your data.*

As mentioned at the beginning of this chapter, most .NET documentation, books, and periodicals also demonstrate creating monolithic applications. Not using business objects means the data access code is placed directly in the user interface, making for a very monolithic application.

In contrast, **Figure 8** shows a three-tier system architecture that includes business objects. This is a far more flexible architecture where any tier can be swapped out (for more information on three-tier architecture, see Chapter 7, "Data Access with ADO.NET").

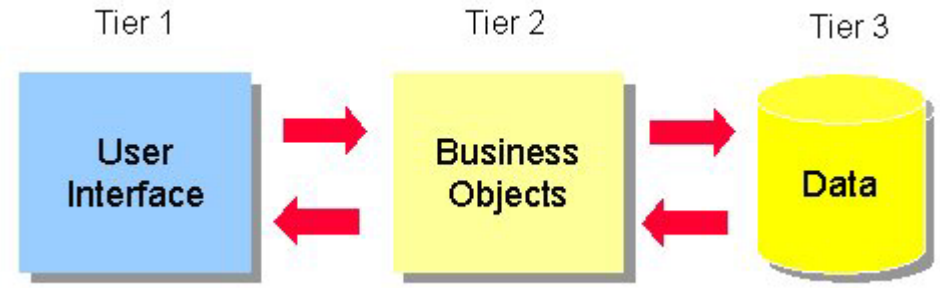

***Figure 8**. Business objects allow you to create a three-tier architecture that is far more scalable than a monolithic architecture.*

For example, you can create a smart client Windows desktop application for tier 1, and then later you can swap it out with a thin client Web browser interface, without affecting the rest of the application (**Figure 9**). You can also change the data tier from Visual FoxPro to SQL Server without changing your application logic.

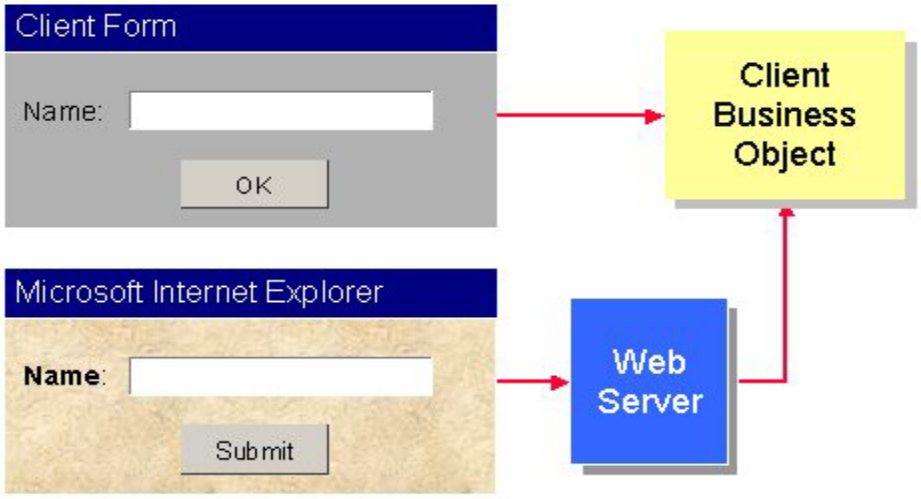

***Figure 9**. Three tier architectures allow you to swap between a fat client Windows desktop front end and a thin client Web browser front end without changing the application logic.*

Additional business object benefits

In addition to the scalability gained with business objects, there are other benefits that come from using them.

Normalizing application logic

One benefit of using business objects is normalizing your application logic. As a Visual FoxPro developer, you know all about normalizing data—eliminating redundancies in the data structure. But how conscious are you of eliminating redundancies in your application logic? When you work with a team of developers, the chances of creating duplicate application logic increases dramatically—especially when creating a monolithic application. You can have five different developers create the same routine five different times when the code is stored in the user interface. Each developer working on a different form has no idea that another developer has already created the code they need.

In contrast, when you use business objects, even when developers work on different forms, each form uses the same set of business objects. Each business object acts as a common repository for code that relates to a particular real-world entity. The chances of adding two or more methods to a business object that perform the same function are pretty slim—especially if you give your methods meaningful names!

Normalizing your application logic means you write, debug, and maintain less code. When a change request comes through there's only one place in your application that needs to change.

Solving the "where's the code" syndrome

Have you ever played "where's the code?" with your software applications? When you create a monolithic application, the code can be located just about anywhere—and Murphy's law predicts that the code you want is probably tucked inside the Click method of a button located on page 3 of a "sub" page frame contained within another page frame.

Finding application code is much easier when you use business objects. For example, if you search for code that has something to do with invoicing, chances are *very* high it can be found in the Invoice business object. If your application has a bug in the logic that performs calculations on inventory, you can bet that the code is probably in the Inventory object. Surfacing your application logic (raising it from the depths of the user interface) and exposing it in high level business objects (**Figure 10**) makes your application far easier to debug and maintain.

***Figure 10**. Surfacing your application logic into business objects (represented by the colored cubes) makes it easier to find the code you're looking for, helps normalize your application logic, and lets you conceive and create complex software systems.*

Ease of conceiving and creating complex software systems

When your code is stuck "in the weeds" of your application's user interface, it can be very difficult to step back and see the big picture of a complex software system.

However, when you place your application logic in business objects, it's far easier to see the big picture and think "big thoughts". Rather than poring through a morass of methods within your user interface code, you can conceptualize complex processes as high-level business objects representing real-world entities and interacting with each other.

I have the same experience over and over again when I visit software development companies to help them solve some of their more thorny issues. As soon as they lay out business objects on a diagram and begin conceptualizing at a higher level, problems that were previously impossible to wrap their minds around can be grasped and solved.

Once you learn how to use business objects, you'll never go back again!

A simple example

So, how do you transform a monster Click event into a business object model? Although this isn't a book about analysis and design, here is a simple example of how this works.

Consider the example of a point-of-sale invoicing application. If you're creating a monolithic application, you might have an Invoice form that has a Save button. Within the Click event of the Save button, you might have code that does the following:

- Scans through each invoice item calculating the tax (if any) on each item.
- Adds the tax and item cost to the invoice header total.
- Subtracts the invoice item quantity from the "quantity on hand" in inventory.
- Saves the invoice items.
- Saves the invoice header

How would you handle this using business objects? Typically, you should create a different business object for each table in your back end database (this is a guideline, not a "set in stone" rule). In this example, you might create Invoice, InvoiceItem, Tax, and Inventory business objects.

Figure 11 shows a UML sequence diagram demonstrating how you might implement business objects to handle all of the processes involved in saving an invoice.

For more information on creating and reading UML sequence diagrams, see my online article "UML Sequence Diagrams" at the following url: http://www.devx.com/codemag/articles/2002/March/umlsequence/umlsequence-1.asp

The stick figure at the top left of diagram represents the Sales Rep who interacts with the application to save an invoice. To the immediate right of the SalesRep a box labeled "UI" is a generic representation of the application's user interface. The Save() message line pointing from the SalesRep to the UI represents the SalesRep pressing the Save button.

The rest of the boxes to the right of the UI box represent the application's business objects. Notice the arrow labeled "Save()" between the UI object and the Invoice object. This indicates the Invoice object has a Save() method being called by the user interface. The Invoice object in turn sends a Save() message to the InvoiceItems object. The InvoiceItems object calls a method on itself named SaveItem(). The asterisk preceding the SaveItem() method indicates this method is called multiple times—in this case, once for each invoice item. From within the SaveItem() method, a call is made to the Tax object (CalcTax) to calculate tax on the item and to the Inventory object (RemoveFromStock) to remove each item from stock.

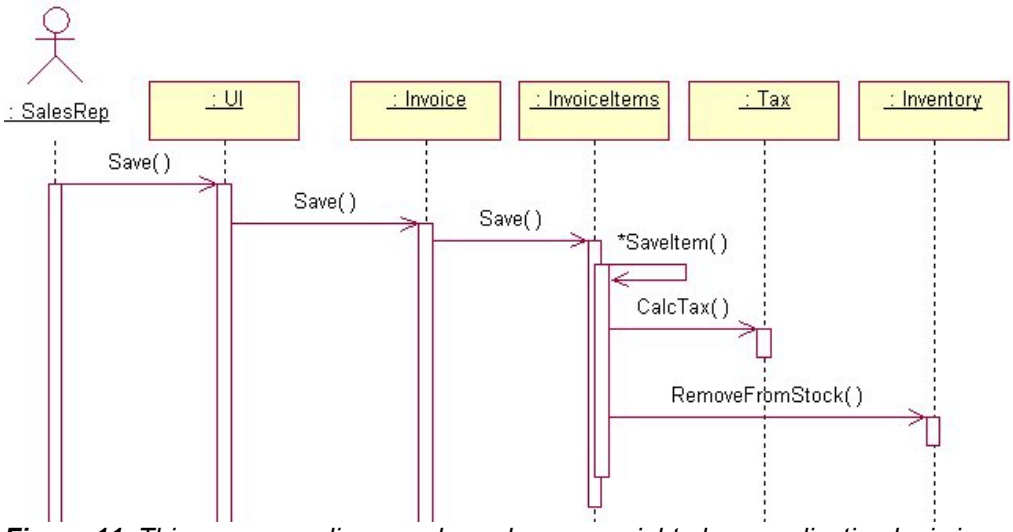

***Figure 11**. This sequence diagram shows how you might place application logic in business objects to save an invoice.*

Note that I haven't included any parameters in this diagram. I did this to make it easier to read. However, in reality most methods in this diagram would receive parameters. For example, the RemoveFromStock() method might receive an inventory item primary key and a quantity, so it can determine which inventory item to adjust and by how much.

The main purpose of this example is to show you the big picture of how to use business objects in applications. You will see more detailed examples of using business objects in Chapter 9, "Building .NET Windows Forms Applications", Chapter 10, "Building Web Applications with ASP.NET", and Chapter 12, "XML Web Services".

Making .NET data access easy

If nothing I've mentioned so far strikes you as a compelling reason to use business objects, the fact that business objects make it much easier to use ADO.NET may be the reason you are looking for.

As Chapter 7, "Data Access with ADO.NET" explained, ADO.NET is very flexible, very scalable, and very object-oriented, but it can be difficult to learn and use. Business objects change all that by creating a high-level interface to ADO.NET that doesn't require all developers on your team to be familiar with creating connections or manipulating and coordinating data objects.

You write your data access logic once, store the code in a family of data access classes used by your business objects, and never worry about the specifics of ADO.NET again—until Microsoft changes the ADO.NET object model.

This actually brings up another compelling reason to create a layer of abstraction between your application and ADO.NET. Microsoft is notorious for changing its data access model every few years. If you follow the pattern set by many .NET code samples found in books, magazines, and online articles, you'll end up sprinkling lots of data access code throughout

your user interface. This becomes a problem if Microsoft makes changes to ADO.NET. It will force you to update all of this data access code accordingly. However, if you use business objects, you have only one place to change your data access code—within the data access classes of the business object.

Enforcing business rules

One of the primary jobs of a business object is to enforce business rules. Business rules fall into two broad categories:

1. Data integrity rules – This encompasses rules that enforce things such as required fields, field lengths, ranges, and so on.
2. Domain rules – This refers to high-level business rules such as "You can't create an invoice for a client who is over their credit limit".

Typically, an application checks business rules at two different points in time. The first is when trying to save a record. After a user clicks the Save button (Windows Forms application) or the Submit button (Web Forms application), the system needs to check if any rules pertaining to the record being saved are broken, and if so, display a message with showing the broken rules.

The second place rules are often checked is when the user leaves a data entry control. For example, when the user leaves an e-mail text box, you may want to immediately check if the e-mail is valid. You can call a business rule method to verify this. For details, check out the section, "The BusinessRules class", below.

.NET business object architecture

To help you grasp the concept of business objects, the sample code that comes with this book provides a simple business object class you can use to access either FoxPro or client-server data. **Figure 12** shows a UML class diagram documenting the basic architecture of this business object class.

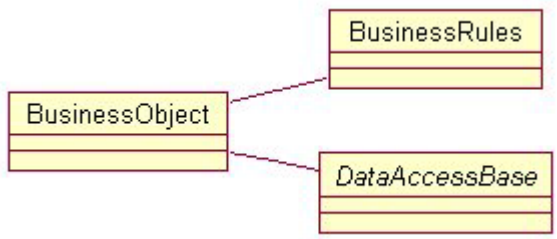

***Figure 12**. Good business object architecture gives you a lot of flexibility—especially in the area of data access.*

The architecture of the business object classes in this book is by no means the one "right" way. This architecture is simple enough to show you the basic mechanics of business objects while still having enough advanced features to demonstrate the flexibility and scalability business objects provide. You'll definitely want to enhance this architecture for more robust production systems.

Each of these objects is covered in detail in the following sections.

The BusinessObject class

The BusinessObject class, shown in Figure 12, is the primary class in the business object architecture that the user directly interfaces with.

The sample code in this chapter is set to use the SQL Server Northwind database by default.

Creating subclasses of the BusinessObject class

You create subclasses of the BusinessObject class that represent entities in your application domain. For example, in this book's sample code, four business objects have been subclassed from BusinessObject:

- Employee
- Customer
- Orders
- OrderDetail

When you create a subclass of BusinessObject, two important properties should be set right away—the TableName and FieldList. The TableName property specifies the primary table in the database with whichwhere the business object retrieves and manipulates data. The FieldList property specifies the default list of fields included in the DataSet when you retrieve data from the back end. This property is set to "*" by default, which specifies that all fields in the table are returned.

You can easily change the value of these properties in your custom business object's constructor method. For example, the following code defines a Customer class derived from BusinessObject and sets the value of TableName and FieldList.

In C#:

```
public class Customer : BusinessObject
{
  /// <summary>
  /// Customer constructor
  /// </summary>
  public Customer()
```

```
  {
        this.TableName = "Customer";
        this.FieldList = "CustomerID, CompanyName, " +
            "Address, City, PostalCode, Country, Phone";
        this.BusinessRuleObject = new CustomerRules();
  }
}
```

And in Visual Basic .NET:

```
Public Class Customer
   Inherits BusinessObject

   '/ <summary>
   '/ Customer constructor
   '/ </summary>

   Public Sub New()

      Me.TableName = "Customer"
      Me.FieldList = "CustomerID, CompanyName, " & _
        Address, City, PostalCode, Country, Phone"
      Me.BusinessRuleObject = New CustomerRules()

   End Sub 'New

End Class 'Customer
```

Retrieving data with the GetDataSet method

The BusinessObject class has a GetDataSet method that executes a SQL SELECT statement and returns a DataSet containing the result set. This method has two overloads.

In C#:

```
protected DataSet GetDataSet()
{
  string Command = "SELECT " + this.FieldList + " FROM " + this.TableName;
  return this.GetDataSet(Command);
}

protected DataSet GetDataSet(string command)
{
  return DataAccessObject.GetDataSet(command, this.TableName);
}
```

And in Visual Basic .NET:

```
Protected Function GetDataSet() As DataSet

   Dim Command As String = "SELECT " & Me.FieldList & " FROM " & Me.TableName
   Return Me.GetDataSet(Command)

End Function 'GetDataSet
```

```
Protected Function GetDataSet(command As String) As DataSet

  Return DataAccessObject.GetDataSet(command, Me.TableName)

End Function 'GetDataSet
```

The first method signature accepts zero parameters. It simply uses the FieldList and TableName properties to automatically build a SELECT command that it passes to the second overload of the GetDataSet method. The second overload accepts a single "command" parameter that it passes to the data access object (discussed below) for execution. It also passes the TableName property, used to specify the name of the main DataTable within the DataSet.

The GetDataSet methods are marked as protected, because you typically don't want to open your back end database to this sort of carte blanche querying capability. For example, if the second GetDataSet method was public, there's nothing stopping someone from issuing a SELECT * that returns all fields and records in a table with millions of records.

To retrieve data from a custom business object, you typically create methods that build a SELECT string and pass it to the second overload of GetDataSet. For example, the following methods of the Customer business object retrieve a customer by ID and phone number.

In C#:

```
public DataSet GetCustomerByID(string customerID)
{
  return this.GetDataSet("SELECT " + this.FieldList + " FROM " + this.TableName
+
         " WHERE customerID='" + customerID + "'");
}

public DataSet GetCustomerByPhone(string phone)
{
  return this.GetDataSet("SELECT " + this.FieldList + " FROM " + this.TableName
+
        " WHERE Phone = '" + phone + "'");
}
```

In Visual Basic .NET:

```
Public Function GetCustomerByID(ByVal customerID As String) As DataSet

  Return Me.GetDataSet(("SELECT " & Me.FieldList & " FROM " & Me.TableName & _
        " WHERE customerID='" & customerID + "'"))

End Function 'GetCustomerByID

Public Function GetCustomerByPhone(ByVal phone As String) As DataSet

  Return Me.GetDataSet(("SELECT " & Me.FieldList & " FROM " & Me.TableName & _
        " WHERE Phone = '" & phone & "'"))

End Function 'GetCustomerByPhone
```

Here is an example of how you call the GetCustomerByPhone method from client code. In C#:

```
Customer CustomerObj = new Customer();
DataSet dsCustomers = CustomerObj.GetCustomerByPhone("555-3425");
```

And in Visual Basic .NET:

```
Dim CustomerObj As New Customer()
Dim dsCustomers As DataSet = CustomerObj.GetCustomerByPhone("555-3425")
```

Saving data with the SaveDataSet method

The SaveDataSet method accepts a single DataSet parameter and updates the back end database with any changes (updates, inserts, deletes) found in the DataSet.

Here is an example of how you call the SaveDataSet method to update a DataSet. In C#:

```
Employee EmployeeBizObj = new Employee();

// Retrieve an Employee record
DataSet dsEmployee = EmployeeBizObj.GetEmployeeByID(1);
DataRow drEmployee = dsEmployee.Tables[0].Rows[0];

// Change a value
drEmployee["Title"] = "Vice president";

// Save the change
int RowsUpdated = EmployeeBizObj.SaveDataSet(dsEmployee);
```

And in Visual Basic .NET:

```
Dim EmployeeBizObj As New Employee()

' Retrieve an Employee record
Dim dsEmployee As DataSet = EmployeeBizObj.GetEmployeeByID(1)
Dim drEmployee As DataRow = dsEmployee.Tables(0).Rows(0)

' Change a value
drEmployee("Title") = "Vice president"

' Save the change
Dim RowsUpdated As Integer = EmployeeBizObj.SaveDataSet(dsEmployee)
```

Before the actual update occurs, this method checks to see if there is a business rule object attached and, if so, calls that object's CheckRules method. If this method is successful, it returns the number of records containing changes that were persisted to the back end (zero or more). If any business rules are broken, this method returns a –1. For details on how to handle broken business rules, see the next section.

The BusinessRules class

The BusinessRules class enforces business rules and keeps track of any rules that are broken. Typically, you should create a subclass of the BusinessRules class for each of your business objects. For example, in the samples for this book, the Customer business object has a CustomerRules object, the Employee object has an EmployeeRules object, and so on.

To associate a business rule class with a business object, you need to instantiate it in the constructor of the business object. For example, you add the following code to the constructor of the Customer business object to instantiate and associate the CustomerRule object with it.

In C#:

```
public Customer()
{
  this.BusinessRuleObject = new CustomerRules();
}
```

And in Visual Basic .NET:

```
Public Sub New()
  Me.BusinessRuleObject = New CustomerRules()
End Sub 'New
```

Typically, you create a separate method in the rules object for each different business rule. For example, the CustomerRules object has IsCompanyNameValid, IsPostalCodeValid, and IsPhoneValid methods. Breaking these methods out allows you to call each method individually (for example, from the event of a user interface control).

In most cases, you also need to check all business rules when you try to save a record. How can this be done if you have each rule in a separate method? The answer is to add a call to each business rule method in the CheckRulesHook method of the BusinessRules class. For example, the CustomerRules object contains the following code in its CheckRulesHook method.

In C#:

```
  public override void CheckRulesHook(DataSet ds, string tableName)
  {
        DataRow dr = ds.Tables[tableName].Rows[0];   // Get the first DataRow

        this.IsCompanyNameValid(dr["CompanyName"].ToString());
        this.IsPostalCodeValid(dr["PostalCode"].ToString());
        this.IsPhoneValid(dr["Phone"].ToString());
  }
```

And in Visual Basic .NET:

```
Public Overrides Sub CheckRulesHook(ds As DataSet, tableName As String)

   Dim dr As DataRow = ds.Tables(tableName).Rows(0) ' Get the first DataRow

   Me.IsCompanyNameValid(dr("CompanyName").ToString())
   Me.IsPostalCodeValid(dr("PostalCode").ToString())
   Me.IsPhoneValid(dr("Phone").ToString())

End Sub 'CheckRulesHook
```

The BusinessObject class automatically calls the CheckRulesHook method before it tries to save a record. It passes the DataSet to be saved as well as the name of the Table within the DataSet. If any rules are broken, the business object does not save the data in the DataSet (the next section shows how you can retrieve and display broken rules).

In this particular implementation, the BusinessRules class only checks the first record in the DataSet. You can enhance this method to check multiple records in a DataTable.

Checking for broken business rules

If the BusinessObject's SetDataSet method returns a –1, you call the GetBrokenRules method of the BusinessObject class to determine the business rules that were broken. This list of broken rules can then be displayed to the user. For example, here's code that checks for broken rules when saving an order.

In C#:

```
Orders OrderObj = new Orders();
DataSet dsOrder = OrderObj.GetOrderByOrderID(10248);
DataRow drOrders = dsOrder.Tables[0].Rows[0];

drOrders["EmployeeID"] = 0;

int RowCount = OrderObj.SaveDataSet(dsOrder);
if (RowCount == -1)
{
  string BrokenRuleList = "";
  foreach (string BrokenRule in OrderObj.BusinessRuleObject.BrokenRules)
  {
        BrokenRuleList += BrokenRule + "\n";
  }
  MessageBox.Show("Broken Rules: \n\n" + BrokenRuleList, "Business Rules");
}
else
{
  MessageBox.Show("Order successfully saved","Business Rules");
}
```

And in Visual Basic .NET:

```
Dim OrderObj As New Orders()
Dim dsOrder As DataSet = OrderObj.GetOrderByOrderID(10248)
Dim drOrders As DataRow = dsOrder.Tables(0).Rows(0)
```

```
drOrders("EmployeeID") = 0

Dim RowCount As Integer = OrderObj.SaveDataSet(dsOrder)
If RowCount = - 1 Then

   Dim BrokenRuleList As String = ""
   Dim BrokenRule As String

   For Each BrokenRule In  OrderObj.BusinessRuleObject.BrokenRules
      BrokenRuleList += BrokenRule + ControlChars.Lf
   Next BrokenRule

   MessageBox.Show("Broken Rules: " + ControlChars.Lf + _
       ControlChars.Lf + BrokenRuleList, "Business Rules")
Else
   MessageBox.Show("Order successfully saved", "Business Rules")
End If
```

This code instantiates the Orders business object, retrieves an order, and saves the DataSet with a broken rule (the Employee ID is empty). It then checks the return value of the SaveDataSet method to see if any rules are broken (indicated by a return value of –1). If any rules are broken, it retrieves all broken rules from the BusinessRule object. A reference to the BusinessRule object is stored in the business object's BusinessRuleObject property. BrokenRules is a string collection contained within the BusinessRule object. The "for each" loop iterates through the string collection building a string containing all broken rules which it then displays in a message box.

Some developers like to display all the broken rules in a single dialog. You can do this by simply concatenating the broken rules together and displaying them. Another option is to have your business object return broken rules as an XML string. You can then display the broken rules in a list box, DataGrid, etc.

Data access classes

Figure 12 showed the BusinessObject class has an associated data access class it uses to retrieve and manipulate data. As shown in **Figure 13**, there is actually a family of data access classes used to access different types of data.

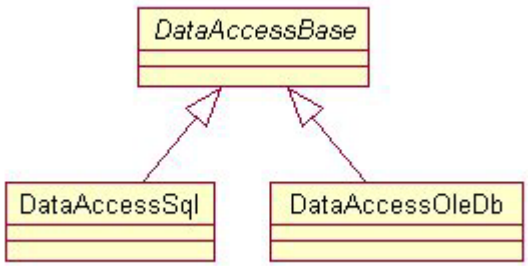

***Figure 13**. Providing a family of data access classes for your business objects allows you to access different data sources (Visual FoxPro, SQL Server, Oracle) without changing your business object.*

The abstract DataAccessBase class defines the interface for the family of data access classes. The DataAccessSql class allows you to access SQL Server 7.0 and later, acting as a wrapper around the .NET SQL Server data provider. The DataAccessOleDb class allows you to access any data with an OleDB Data Provider, acting as a wrapper around the .NET OleDB data provider.

By default, the BusinessObject class uses the DataAccessSql class to access the Northwind SQL Server database. To change the BusinessObject class (and all subclasses) to use the DataAccessOleDb class instead, you change its DataAccessClass property from "DataAccessSql" to "DataAccessOleDb".

Conclusion

You don't have to use business objects in your .NET applications, but doing so makes your applications far more flexible, extensible, and maintainable. In this chapter you've seen how to design and implement business objects. For examples showing how business objects can be used in different types of applications, see Chapter 9, "Building .NET Windows Forms Applications", Chapter 10, "Building Web Applications with ASP.NET", and Chapter 12, "XML Web Services".