Chapter 6 Business Objects

The business object is the worker bee of your application. This is the class that models each entity in your application and encapsulates the rules and processes defined by or for that entity. Most of your processing and rules code will be placed into a business object class.

In the beginning of this book, you learned about Codebook 3.0 and how it introduced the business object to the Visual FoxPro programmer. Business objects were not a new concept in the object-oriented programming world, but they were new to FoxPro developers. After all, version 3 of VFP was the first FoxPro with object-oriented extensions, and even people with five or more years of FoxPro experience were suddenly wet-behind-the-ears OOP beginners.

The business object is the object that is modeled to match the real-world entity. For example, if you wanted to create a business object named “cat,” you would start with some of a cat’s attributes. First you would add whiskers, fur and color attributes. These attributes would become properties of the business object. Next you would model some of the responsibilities or functions of a cat. (Those of us who think a cat is the most useless animal in the world will have to set that thought aside for a moment.) A cat knows how to stretch, yawn, nap, eat and scratch. All of these would become methods of your cat business object.

Up until now, the classes you have looked at in the data layer are really generic and have nothing to do with a specific entity. Their entire purpose is to provide a means to persist or store the properties of a business object. The business object is the first class that becomes specific to your business domain. Your domain might be banking, finance or perhaps school registration. It is in the business object that you implement all the “responsibilities or functions” of the real-world entity.

Tutorial

So far, we have created the table that will physically store the Instructors information. Then, we created a cursor class to provide an object-based method to access the data. Finally, we created a data environment to hold the cursor. Now it is time to create a business object to encapsulate the attributes and actions of the real-world entity.

1. Open the Tutorial application in the Application Builder.
2. Switch to the Classes page and highlight the Business Objects node in the parent page. Press the New button. You will be presented with the wizard selection dialog.
3. As with the cursor wizard, the “Multiple...” wizard will create a bizobj for each data environment that you select without giving you many other options. Select the Single Object wizard and press OK. The Single Business Object wizard will open.

4. Step 1 asks you to select the data environment class for the bizobj. You're provided with a list of all data environment classes in the project. We only have one, so there is not much to do here except click the Next button.
5. Step 2 asks you to set three properties that exist in the business object. These can actually be changed programmatically as well. See the section on the Allow… methods later in this chapter for more information. For now, keep these all checked and press the Next button.
6. Step 3 wants to know the security names. See Chapter 17, "VFE Security," for more information on this. For now, just press the Next button.
7. Step 4 wants to know what to name the class and what library to put it into. The name is based on the DE class you chose. The word "Environment" is removed from it, and "BizObj" is added to that. You may change this if you like. As mentioned earlier, for this tutorial we are going to save all classes for a single entity in a single library. So, select the Instructor class lib from the pull-down menu.
8. At this point, you're done. Press the Finish button.

That was easy, wasn't it? And you thought creating business objects would be difficult. What the wizard did was subclass the framework business object class, iBizObj (which is the I-layer subclass of cBizObj). Because much of your time will be spent dealing with this class, we'll take a detailed look at it.

Business object class

The framework's business object class is designed to contain the common items that all business objects need. Technically, an Instructor can't save itself. In a perfect object-oriented world, a business object class would automatically persist (save) itself to the physical data layer without the programmer needing any knowledge of how that is done. Also, the properties of the object would include the properties of the entity being modeled, like Age, Name, SSN and so forth. But VFE delegates this information to the cursor class.

There are also times when a real-world entity will contain properties that require a separate bizobj in VFE. For example, each instructor may have an addresses property and a phones property. Each of these properties also contains attributes. There is also the possibility that there could be multiple addresses or phones. In a different environment, these types of properties would be defined as a collection.

However, there is no native collection data type in Visual FoxPro. In order to simulate this in VFE, you would create an additional business object for the addresses collection and one for the phones collection. Then, you would use a technique called "relating" the business objects.

Related business objects

Relating business objects in VFE is similar to the parent-child relationship you might find in a database design. The instructor business object would be the parent business object, and the addresses and phones business objects would each be a child business object. As in the database scheme, there would be a one-to-many relationship between these business objects.

While this is a bit removed from a purist approach to a business object, it is the implementation that we must use. In a purist approach, the instructor business object would have an addresses collection and a phones collection that belonged to the instructor. You wouldn't be able to access the addresses or phones outside the parent object. In reality, it takes three business objects to model the single "Instructor" real-world entity. These are what we call inter-object relationships. When representing this entity with multiple business objects, be sure to relate the business objects and files properly. In addition to the business objects, you would want to create referential integrity rules to create these inter-object relationships in the physical layer to mirror the relationships of the business objects that must be created.

This chapter specifically deals with the operation of the business object; for complete information on creating related entities and representative user interfaces, see Chapter 14, "Related Data and Forms," which covers related data.

Parent business object

The parent business object maintains a collection of child business objects. The parent business object cascades commands to all of its child business objects based on several property values. (Refer to the properties and methods sections of this chapter for information on those properties and methods.) The parent object also keeps its children synchronized with it by sending a Requery command to the child business objects when the parent record pointer is changed.

Child business object

In the VFE framework, the same base class is used whether your bizobj is a parent or a child. To indicate to a business object that it is a child, you can populate the cParentBizObj in the child business object. The child will find a reference to its parent and then notify the parent of its existence. VFE has other features that tell the child business object how to "register" with its parent when it is instantiated, but these aren't important right now.

The fact that object references can pass VFP data session boundaries allows you to create related forms that maintain private data sessions. This would not be possible if you wanted to relate two files in separate data sessions without the business objects.

For more information on some of these techniques, see Chapter 14, "Related Data and Forms."

Data Environment Loader

If you open the business object class in the class editor, you will see that it contains an object called the DELoader. This is the Data Environment Loader. You will find several loaders in VFE. The DELoader provides for two things in the case of the business object.

First, the DE that the business object uses can be modified to suit your business domain's purpose. Perhaps you have a different environment for the same business object in different circumstances.

The second reason to use the Data Environment Loader would be to instantiate a DE that resides in a separate physical layer from the business object. This allows the DE to exist in a COM or DCOM component.

The DELoader is a simple class that loads an instance of the DE class specified. The DELoader has a property named "cDataEnvironment" that holds the name of the DE to load. If loading from a COM object, there are other properties to specify the name of the object and the machine to load it from. The cDataEnvironment property is populated when you create the business object using the wizard.

Business rules collection

In addition to the DELoader object in the business object class, you will also find a business rules collection. This collection holds a reference to all the business rules objects that have been added to the business rules. When the business object is told to save a record, if there are any business rules objects in the business rules collection, they will be called one at a time.

Business rules class

The business rules class provides a method to implement rules that will be common across several business objects. This eliminates the duplication of source code. Instead of writing some code in a business class, you would create the code in the business rules class. When a business object is sent the Save() method, it will run the Execute() method of all the business rules objects in its business rules collection.

The Execute() method of the business rule can implement any domain rule or process required. If a True is returned, the business object will continue the save; if a False is returned, the business object will not continue the save.

The business rule provides a GetErrorMessage() method. This method will be called from the business object if a False is returned to display to the user. By default, GetErrorMessage() returns the value of the cErrorMessage property, or the cErrorMessageKey value is used to look up the error message in the MsgSvc table. It is up to the developer to populate the cErrorMessage property prior to returning False to the business object.

cBizObj of cBusiness.VCX

To implement your business objects, the VFE framework provides the cBizObj class. The cBizObj class is shipped in the cBizness library. The cBizObj class is a compound class that contains an oDELoader and an oBizRulesCollection object.

The following sections detail the properties and methods you will work with most often. As always, for the fullest understanding of the VFE framework you should read the code for the classes.

cForeignKeyExpression

Populate this property if the business object is a child business object. This property will hold the name of the field in the parent cursor that relates to the child cursor. This is a somewhat misnamed property. Think of it as the field you want to populate the child foreign key with.

Here’s an example: In the typical overused example of Invoice and InvoiceDetail, the key file of the Invoice table, “InvoiceNo,” is used as the foreign key in the InvoiceDetail file named “Invoice.” In this case, you would populate the cForeignKeyExpress with the value “InvoiceNo.” See the next section, “cKeyField,” for the other half of this example.

The child business object will use the information in this field to populate the foreign key in the child cursor with the parent’s key automatically. When using tables, the business object will use this information to set a relation between the two files.

cKeyField

Populate this property if the business object is a child business object. This property will hold the name of the foreign key field in the child cursor. This property is somewhat misnamed. The name implies that you enter the name of the key field of the child; don't make this mistake.

To continue with the aforementioned example of Invoice and InvoiceDetail, again the key file of the Invoice table, "InvoiceNo," is used as the foreign key in the InvoiceDetail file named "Invoice." In this case, you would populate the cKeyField with the value "Invoice." See the preceding section, "cForeignKeyExpression," for the first half of this example.

The child business object will use the information in this field to populate the foreign key in the child cursor with the parent's key automatically. When using tables, the business object will use this information to set a relation between the two files.

cParentBizObjName

This is the property that informs the business object that it is a child business object. Populate the property with the object instance name of the parent business object. The business object will look for the parent business object, and when it locates it, the parent business object is sent a message to register its new child. The parent business object will add its new child to its child business object collection.

cRelationTag

This property is used in a child business object. This property is only used when using VFP tables and is not needed when using views. This property holds the name of the tag for the child cursor that is used to relate the parent and child cursor.

lAllowDelete, lAllowEdit, lAllowNew

These properties provide a simple interface to disallow any of the specific actions. Recall that these properties were settable in the Business Object wizard. In the case that you want to allow or disallow one of these actions depending on certain values and criteria, leave this property set to True and use the similarly named methods to write code to make such a determination.

lConfirmOnDelete

This property is fairly self-explanatory, but you should be aware that there is no code in the business object class that looks at this property. The code that looks at the property is in the presentation object. So, if you are using a business object via COM and have set the property to True, it is up to your front end to request a confirmation from the user based on this property.

lNewOnParentNew

This property will only be used when the business object is a child business object. If this property is set to new, the parent business object will send a New() message to this, the child business object, when a new is performed at the parent level. (This is one of the big improvements over the Codebook business object. In Codebook, instead of this property there's an lNewChildOnNew, which is much more limited.)

lRequeryChildrenOnSave

This property indicates to the business object that after a save has been performed, a Requery() message will be sent to all of its children business objects. The default for this property is True.

lSaveAllRows

This property will force the business object to save all rows of a record buffered table when the business object is instructed to save. Be aware that this property is set to True by default.

lWriteForeignKeyOnNew, lWriteForeignKeyOnSave

These properties indicate to the business object when to write the foreign key value from the parent record to the child record. The lWriteForeignKeyOnNew property will be checked in the New_Perform() method, and the lWriteForeignKeyOnSave property will be checked in the Save_Perform() method. Be sure that you have coordinated these values with the lWritePrimaryKeyOnNew and lWritePrimaryKeyOnSave in the cursor object.

In other words, be sure that there is a value in the parent cursor's primary key field prior to trying to write it to the foreign key field in the child cursor.

oCursor

This property holds a reference to the primary cursor. The DE class sets this property when the cursors are instantiated. This property is the property the developer will use to access the data that is controlled by the business object.

For example, assume you were writing code in the instructor business object and you needed to check the value of the 'LastName' field. You would write code similar to this:

```
cLastName = This.oCursor.Fields.Item['LastName'].Value
```

oDataEnvironment

This property holds a reference to the DE object that was loaded for the business object. This allows you to use the services of the DE object, such as FindCursor().

oHost

This property holds a reference to the interface object that is hosting the business object. Generally this will be a reference to a presentation object. This reference allows you to send messages to the presentation object, perhaps to run a method in the presentation object that prompts the user for data or a yes/no response.

You will not want to write code in the business object that exposes a user interface in cases where your business object may be running as a COM object. This will violate your layered and n-tier design and make it more difficult to move to that environment. Therefore, use this property with caution and always check to make sure it holds an object reference before addressing it.

oParentBizObj

This property contains a reference to the parent business object. This property will be invaluable when creating related forms where each has a private data session.

oBizObjs

This property contains a reference to the child business object collection. The collection contains a reference to all of the child business objects. This property is added at run time when a child business object is registered with its parent. oBizObjs is actually a member object that gets added to a business object when a child is registered with it, but for all practical purposes it can be thought of as a property.

RecNo, RecordCount, VisibleRecordCount

These properties contain the value indicated by the name. Remember that the cursor object also contains these properties. Each of these properties in the business object has an access method associated with it. The access method actually requests the information from the cursor object and returns it to the business object property.

AllowNew(), AllowDelete(), AllowEdit(), AllowSave()

These methods determine whether the defined behavior is allowed. Here's the syntax:

```
Object.AllowXXXX()
```

Returns:	Logical

Remarks

By default, each of these methods will check the value of its corresponding property, such as lAllowNew or lAllowDelete, and if those are True they will check security and the read-only property to determine whether to allow the function.

To maintain the default functionality and add your own, we recommend that you AND your result with the result of DoDefault(). For example, let's assume you only want to allow a user to delete a record if the termdate is not empty. An example of the code you'd put into the AllowDelete() method would be:

```
local lAllowIt
lAllowIt = .t.
if empty( This.oCursor.Fields.Item['termdate'].Value )
  lAllowIt = .f.
EndIf

Return lAllowIt AND DoDefault()
```

Calling DoDefault() will ensure not only that the record meets your business rules, but also that the user has security to delete a record in this business object, for example. If you want to totally override security, you would omit that call to DoDefault(), of course.

For example, suppose you want to allow anyone to delete a record that has a termdate even if they don't normally have security for this action. You would override AllowDelete() without calling the default behavior. This allows for extremely flexible business rules.

Cancel(), Cancel_Pre(), Cancel_Perform(), Cancel_Post()

The Cancel() method will "undo" any changes made to a record. It will also remove an added record that has not yet been saved. Here's the syntax:

```
Object.CancelXXXX( tlAllRows )
```

Returns:	Constants: • FILE_OK • FILE_CANCEL
Argument:	*tlAllRows*—Pass a True to indicate that all rows of a table buffered cursor should be canceled.

Remarks

Cancel() is the shell method that calls the _Pre, _Perform and _Post methods. The Cancel_Perform() actually calls the cursor object's Cancel() method to do the actual cancel.

The Cancel_Pre() and Cancel_Post() methods are hooks that allow you to perform some task or check prior to or after a cancel. Returning a FILE_CANCEL from Cancel_Pre() will actually cancel the cancel, as strange as that sounds. The Cancel_Post() method is only called after a successful cancel is performed. This allows you to add code that you want run after a successful cancel.

In the case of Cancel_Pre() and Cancel_Post(), if you want to return the default constant we recommend that you return DoDefault(). For example:

```
Local cSomeVariable
* You do some code here prior to a cancel
Return DoDefault()
```

By returning DoDefault(), you don't have to remember the valid return value for a successful completion of your function. If you look at the default methods of these hook properties, you will see that all they do is return FILE_OK.

Delete(), Delete_Pre(), Delete_Perform(), Delete_Post()

These methods delete the current record in the cursor. Here's the syntax:

```
Object.DeleteXXXX()
```

Returns:	Constant FILE_OK or record number

Remarks

Delete() is the shell method that calls the _Pre, _Perform and _Post methods. The Delete_Perform() method calls the Delete() method of the cursor class in order to delete the record.

The Delete_Pre() and Delete_Post() methods are hooks that allow you to implement business rules, which are called before (_Pre) or after (_Post) a delete takes place. The delete will be canceled if you return FILE_CANCEL from the _Pre method. The _Post method will not be called unless the delete is successfully processed.

As indicated previously, be sure to return DoDefault() from your hook methods to assure you are returning the correct data type.

FindCursor()

This method returns a reference to the cursor based on the name you provided. Here's the syntax:

```
Object.FindCursor( tcCursor )
```

Returns:	Object
Argument:	*tcCursor*—Specifies the name of the cursor to find. If no cursor name is passed, the oCursor reference of the business object is returned.

Remarks

FindCursor() will search in the business object's DE object for the specified cursor. If the cursor is not located, the child business object's DE will be searched. If the cursor is not found, a null will be returned. If the cursor is not found and the business object has children, the child business objects will also be searched for a matching cursor.

FindField()

This method returns a reference to a field based on the name you provided. Here's the syntax:

```
Object.FindField( tcField )
```

Returns:	Object
Argument:	*tcField*—Specifies the name of the field to find.

Remarks

FindField() will search the primary cursor first, and then it will search the other DE cursors. If successful, it will return an object reference to the field object. If the business object is a child business object and tcField is not found in the business object's DE, the parent business object is also asked to find the field. If the field can't be found, a null value is returned.

FindViewParameter()

This method returns a reference to the specified view parameter object (see the "FindCursor()" section for more information).

GetValues(), SetValues()

GetValues() will return an object with the values of the current record in the object. SetValues() will take an object created by GetValues() and populate the current cursor with the values. This is an objectified SCATTER/GATHER and can be used to populate a cursor with values from a lookup table, for example.

GetRecordSet()

This method gets an ADO record set of all the data in the primary cursor of the business object. This will come in handy when creating Web pages or using non-VFP front-end user interfaces. Currently the framework does not provide a corresponding method to save a changed RecordSet, so unless you add this functionality yourself, this method is only useful for retrieving data. The designers will probably be adding more ADO support in future versions.

IsAdding(), IsChanged(), IsDeleted(), IsCursorEmpty()

These methods are self-explanatory. Each of these methods requests the information from the cursor object. A logical True or False will be returned as a result.

New(), New_Pre(), New_Perform(), New_Post()

These methods add a record to the primary cursor of the business object. Here's the syntax:

```
Object.NewXXXX( tcCursor )
```

Returns:	FILE_OK or FILE_CANCEL

Remarks

The New() method is the shell method that calls New_Pre(), New_Perform() and New_Post(). New_Pre() is a hook where you can put code that will be run prior to adding a new record. You can also stop a new record from being added here by returning a FILE_CANCEL.

Be aware that New_Pre() is called from New() without checking AllowNew(). AllowNew() is called from New_Perform(). So, if you are using AllowNew() code and also putting code in New_Pre(), it is possible that you have run code and the record will not be added. We suggest that you check the AllowNew() method manually in your New_Pre() code to avoid potential problems.

After New_Perform() gets a True from AllowNew(), it will call on the cursor's AddNew() method to perform the physical add into the cursor. If the cursor returns a FILE_OK, the New_Perform() method will call WriteForeignKey() of the cursor if the business object is a child and if lWriteForeignKeyOnNew is set to True. Then, if the business object has children, it will call the Requery() of its child business objects. (Remember, the business object code supports the bizobj being either a parent or a child, and actually it can be both at the same time.)

Finally, New_Perform() will loop through its child bizobj collection calling the child's New() if its lNewOnParentNew property is True. If all this occurs without a problem, FILE_OK is returned to the New() method, which finally calls your New_Post() method. This is where you should do any additional operations, since you are assured that a record has been added. If a record was not added, the New_Post() will not be called.

OkToMove()

This method determines whether it is okay to move the record pointer. Basically, it is not okay to move if the current record has been edited and the cursor is record buffered, or if the business object is a parent business object and there are pending changes in any of its children.

Requery(), Requery_Pre(), Requery_Perform(), Requery_Post()

These methods pass a Requery() command to the cursor object to retrieve the data from the underlying tables of the cursor object. Here's the syntax:

```
Object.RequeryXXXX()
```

Returns:	REQUERY_SUCCESS or REQUERY_ERROR

Remarks

Requery() is the shell method that calls Requery_Pre(), Requery_Perform() and Requery_Post(). As in the previous methods with hooks, if you return a REQUERY_ERROR from Requery_Pre(), the requery will not be performed.

Requery_Perform() will call the Requery() method of the primary cursor object to retrieve the data from the underlying table. If this is a success, Requery_Perform() will call the Requery() method of its child business objects.

If the Requery_Perform() is successful, the Requery() method will call the Requery_Post() method. The Requery_Post() is the ideal place to add code to Requery() other cursors in the DE to keep them in sync with the primary cursor. There is no need here to refresh controls; you will see in the next chapter, "Presentation Objects," that the Requery() method there will take care of refreshing the display.

Save(), Save_Pre(), Save_Perform(), Save_Post(), OnSaveNew()

These methods save the changes that have been made to the buffered cursors. Here's the syntax:

```
Object.SaveXXXX( tlAllRows, tlForce )
```

Returns:	FILE_OK, FILE_ERRORHANDLED or FILE_CANCEL
Arguments:	*tlAllRows*—Specifies that all edited rows will be saved in a table buffered cursor.
	tlForce—Specifies that the save should be forced even if VFP has sensed an update conflict.

Remarks

Save() is the shell method that calls Save_Pre(), Save_Perform() and Save_Post(). As in the previous methods with hooks, if you return a FILE_CANCEL from Save_Pre(), the save will not be performed.

It's a bit confusing why there is no code to call AllowSave(). It looks like the AllowSave() is only used to determine whether to display the Save button or not, which does make sense because if you allow an edit or a new, you must by default want to allow a save.

Save() will begin a transaction after calling Save_Pre() but prior to calling Save_Perform(). So, if you have to perform processing like Requery() or other things you can do under a transaction, do them in the Save_Pre(). However, if this is a child business object, be aware that the transaction will have been started in the parent business object.

Save_Perform() performs several steps:

1. If the lSaveChildrenFirst property is True, the SaveChildren() method is called. The SaveChildren() method loops through the oBizObjs collection calling each child business object's Save() method.
2. If this is a new record, and lWriteForeignKeyOnSave is True and this object is a child business object, then the WriteForeignKey() method is called.
3. The Validate() method of the oBizRules collection is called. As we discussed earlier, this method will loop through all the business rules objects that have been added to this business object, calling the Execute() method of each. If any of those fail, the save process will be done here and a FILE_CANCEL will be returned.
4. The Update() method of the primary cursor object is called to save the data to the underlying table. If this is a table buffered cursor and we are saving all rows, the UpdateBatch() method of the cursor is called.
5. At this point, if lSaveChildrenFirst is False, so we don't do this twice, the business object will call its SaveChildren() method.

If all of these steps are successful, the Save_Perform() will return a FILE_OK to the Save() method. The Save() method will then commit the transaction by calling the EndTransaction() method. If for some reason the Save_Perform() returned FILE_CANCEL, the Save() method will perform a rollback. At this point, if the lRequeryChildrenOnSave is True, the Save() method will call all the children business objects' Requery() methods.

At this point, Save_Post() is finally called. *Save_Post() is called after the transaction has been committed*. Save_Post() is not called, of course, if the save fails and the transaction is rolled back.

The Save() method has one more hook method called OnSaveNew(). This method is called after a successful save if the record saved was a new record—in other words, if we were saving a newly added record, OnSaveNew() is called after the successful save.

Once again, in the business layer you will see that there are no commands to refresh the user interface at all. That is all done in the presentation object.

SetField(), SetParameter()

These methods can be used to set the value of a field in the cursor object. Here's the syntax:

```
Object.SetField( tcField, tuValue )
Object.SetParameter( tcParameter, tuValue )
```

Returns:	Logical
Arguments:	*tcField*—The name of the field to populate with the specified value.
	tcParameter—The name of the view parameter to populate with the specified value.
	tuValue—The value with which to populate the specified field or view parameter.

Remarks

These methods can be used to replace the direct method of drilling down to the value of the field or view parameter object you want to populate, but in the end, they do the same thing.

However, using “setters” and “getters” is a more common programmatic interface for COM objects, and your code may be more readable if you use these methods.

Creating your business rules

Well, now that you are familiar with all of the default attributes and behaviors of the base business object class, you might ask, “Where do I put the code?” Remember that the business object is just an abstract class that gives you the mundane and common functionality.

It is up to you to add the properties and methods that are required for the entity you are modeling. In the example of the “Cat” entity, you would add the Scratch(), Yawn(), Sleep() and other methods to the business object. Also, keep in mind that if you have a rule or behavior that will be common to several objects, you should create it as a business rules class.

One example of a business rules class is an audit trail. Create a business rules class that writes all the changes of the cursor to an audit file of some kind. This file can have many different structures, depending on what you want to accomplish, from as simple as a time and user ID to as complex as saving the old and new value of every changed field.

Summary

The business object class is the worker bee of your application. This is the base class you will use to implement the entities of your business domain. The bulk of your code will be contained in the business objects. The framework has provided all of the infrastructures you need to persist or store the objects to the physical data layer.

You can also see the Chain of Responsibility pattern emerging here. Each layer is building upon the previous layer and using its services. The business object calls upon the cursor objects to do the physical work of adding records and saving edits. The cursor handles all the details and tells the business object it has completed successfully. The business object then performs value-added services specific to the business object. The business object coordinates with its parent and child business objects to be sure they are synchronized.

Finally, don’t forget the little business rules class. Abstracting specific rules into their own class will allow you to share those rules with other business objects to maximize your code reuse. Each business rules class has the same interface, so you can add and remove the rules objects to the business objects and adapt to future needs of your business domain.

At this point you may want to go back and re-read the VFE documentation sections about the business object and business rules objects. If you can, read through the code of the Save(), New() and other methods as the book walked you through them.

You should now be starting to see the power of this framework. If you are designing a Web application with VFE, this is the point where you would stop. From here we’ll go on to the user interface layer, which you may want to build as a Web front end or—say it isn’t true—a VB front end. If you have created the data layer and the business layer, you have all the functionality in your application. Your Web page or your UI code is nothing more than a control panel that contains switches, dials and knobs so the user can control your application.

So, in the next chapter we will start creating a VFP/VFE-based user interface. The starting point is the presentation object. See you in Chapter 7.