VB Object Framework

Version 1.0a

An Framework for Enhanced Object-Oriented Application Development
for users of Microsoft Visual Basic™ version 4.0 and above.

Table of Contents

Chapter 1: Introduction to the VB Object Framework

The VB Object Framework ("VBOF") is intended to be used by Visual Basic programmers seeking to further exploit the object-oriented capabilities beyond that provided in the Microsoft Visual Basic 4.0 ("VB4") base product. VBOF provides a Framework to facilitate the integration of the VB4 Class Module with VB's powerful GUI and Database facilities. VBOF provides a full complement of object-related services to greatly increase VB4's object-oriented capabilities, including the following:

- automatic implementation of the object containment hierarchy
- implementation of an object-to-database mapping scheme
- automatic object instantiation from database data
- automatic storing of data in the database
- automatic maintenance of object containment links
- automatic implementation of object persistence through the VB or ODBC-connected database
- complete encapsulation of all SQL functions
- automatic avoidance of redundant object instantiation
- object event management for event communication across independent processes and for linking the non-visual BOM to the GUI
    - object-oriented approaches to managing several VB controls

In this Chapter, the section "Overview" describes the general nature of VBOF; section "VBOF Services List" describes the services of the VBOF in a general nature; and section "VBOF Components Overview" outlines the specific VBOF Components and relates them to their respective VBOF services. Also take note of the sections "Intended Use" and "What's not Included" in this Chapter.

Overview

The VB Object Framework is a packaging of several integrated Visual Basic Class Modules in source code form. VB Object Framework includes an Object Manager ("VBOFObjectManager"), an Event Manager ("VBOFEventManager"), a database-aware Collection object ("VBOFCollection"), several Control Wrappers ("VBOF…Wrapper") and this User's Guide. This User's Guide contains descriptions of the required and recommended contents of application Class Modules, Forms and Databases in order to receive the benefits of VB Object Framework.

The VBOF supports the object-oriented design approach of developing and completing the non-visual Business Object Model ("BOM"), then applying the database and user interface aspects of the application, as appropriate. See "Object-Oriented Development Strategies" for additional information.

If desired, the VBOF supports conventional Visual Basic programming techniques, such as GUI-centric and Database-centric approaches which are focused on Visual Basic's GUI DataControl and database facilities. However, its strength is in its support of highly object-oriented application design and development approaches where Visual Basic data-centric components, including the DataControl, are not required. Through VB Object Framework's wrappers, VB's GUI components are fully supported in an object-oriented manner. For example, the Visual Basic ListBox, ComboBox and DBGrid can be processed as visual containers for the presentation of a collection of objects. All interaction with these GUI controls can be conducted through an object-oriented layer. This includes setting or retrieving the contained objects of these GUI controls, and setting or retrieving the selected object(s) or other control properties.

The VBOF is an enhancement beyond the previously released "DataAwareCollection" Shareware product. Regrettably, and unfortunately to those many faithful users of the DataAwareCollection, there is no means of providing the features of the VBOF and also provide 100% backward compatibility with

DataAwareCollection. Thus, "Appendix A" describes in detail the coding enhancements required for compatibility with the VBOF.

VBOF Services List

The following is a summary of the services of the VB Object Framework:

Object Instantiation and Termination Supervision

The VBOFObjectManager supervises the instantiation and termination of all application objects including VBOFCollections , objects instantiated from data sources and contained within VBOFCollections, singular contained objects and VBOF Wrappers.

Object Single-Instance Ensuring

As the VBOFObjectManager monitors the instantiation of all objects, it ensures the uniqueness of each object. When an attempt is made to instantiate an object that has already been instantiated, the VBOFObjectManager cancels the process and directs the user to the instantiated object. This negates the need for the application to develop code to deal with redundant instances of a given object.

Automatic Implementation of the BOM Object Containment Hierarchy

The VBOFCollection automatically implements the object containment hierarchy of the BOM.

For example, if a "Person" Class Module contains collections of "Address" and "Phone" objects, VBOFCollection automatically retrieves the applicable rows of the respective data sources for the Address and Phone objects, instantiates the applicable objects, populates them with data from the respective data source rows and returns them to the application. See below for additional information regarding these functions.

Automatic Database Access

The VBOFCollection automatically generates and executes all of the necessary SQL code for implementation of the BOM object containment hierarchy. Except for an occasional "Order By" or "Where" clause, the application never needs to code any SQL. The rows returned from the data source equate to the objects contained within the appropriate container object.

The VBOFCollection automatically implements object persistence by managing all necessary SQL activity for adding, updating or deleting contained objects from their container object. In addition, the VBOFCollection automatically maintains an independent data source for storing object containment information.

Automatic Conversion of Data Source Rows to Objects

The VBOFCollection cooperates with the Class Module to map the fields of its associated data source with its properties. For each applicable row of the data source, the VBOFCollection instantiates a new object and passes the row information to it to allow the Class Module to map its properties to the data source.

To receive these services, the application need only identify the name of the appropriate data source for the Class Module, provide a "Sample Object" of the contained Class Module type, provide a method to map the fields of a data source row to the properties of the Class Module, and provide a method to map the properties of the Class Module to the fields of a data source row.

Object-Oriented Wrappers for VB GUI and Database Objects

The VBOF provides wrapper-style Class Modules which support object-oriented management of the application's GUI and Database objects. These present an opportunity for increased object-oriented exploitation by the application. As GUI and Database layers are applied to the BOM,

there is no compromise of object-oriented achievements due to having to develop Database-oriented or GUI-oriented code.

For example, the VBOFCollection provides methods which support managing the VB RecordSet like a Collection object. Likewise, the VB Object Framework provides wrapper Class Modules for object-oriented management of the VB ListBox, ComboBox and DBGrid. Additional wrappers will continue to be developed as dictated by customer demands.

When the database and GUI are applied over the BOM, use of these wrappers should be a natural extension of the principles of the BOM.

Object Event Management

The VBOFObjectManager, in conjunction with the VBOFEventManager, provide object-level event management functions. Object events are required for the successful implementation of robust object-oriented applications. For example, events can be used to mark certain state transitions throughout the application, such as objects being instantiated, updated, deleted, etc.

Within an application, the independence between application components achieved through event processing yields increased system life span and reduced system maintenance. This is because there is no need for the triggering component to be sensitive to the requirements of any of the recipients of those events, and all components are free to evolve independently over time. In addition, there can be any number of registered recipients for any given object event, allowing for growth and enhancement of the application, independent of the triggering object.

Object events can also be utilized when applying a GUI over the BOM without compromising the purity of the BOM. To implement this aspect of object event management, the GUI Forms register as interested recipients of events generated by the appropriate BOM objects and respond accordingly. For example, a given Form might be an editor for a given Class Module. When executing over a given instance of the Class Module, the object is known as the Form's Domain Object and certain events triggered by the Domain Object would be of interest to the Form. Thus, the Form would register to receive notification of certain events triggered by the object.

Interested Forms and Class Modules can register with VBOFObjectManager to receive notification of certain events as they are triggered by the appropriate Class Module or particular Domain Object.

Object events can also provide independence between application components, since events can be used as catalysts to cause certain functions to occur without there being any direct link or association between the triggering object and the notified objects. This allows background processing to occur without the triggering object necessarily having any knowledge or direct involvement with those background functions. For example, a product cross-selling application could be running in the background. Its input consists solely of events being triggered by the foreground application. In the event that the cross-selling application finds a need to recommend a product for the Sales Rep to present to the customer, it can display such a message in a special window. Through this implementation, the foreground application never has to know that the background application even exists, and it certainly does not need to integrate the needs of the background application into its own code.

VBOF Components, Overview

VB Object Framework consists of the following Visual Basic Classes Modules[1]:

[1] It is possible to develop additional "Wrapper" Class Modules for object-oriented implementation of certain VB controls that are not already provided in this increment of the VB Object Framework. Please contact the author for support.

File Name	Class Name
VBOFOMgr.cls	VBOFObjectManager
VBOFColl.cls	VBOFCollection (formerly "DataAwareCollection")
VBOFEMgr.cls	VBOFEventManager
VBOFEvnt.cls	VBOFEventObject
VBOFData.cls	VBOFDataWrapper
VBOFDBGr.cls	VBOFDBGridWrapper
VBOFLBox.cls	VBOFListBoxWrapper
VBOFRSet.cls	VBOFRecordSetWrapper

The following section describes these VBOF Class Modules is greater detail.

VBOF Components, Details

The following is a general description of these Class Modules. For specific information regarding the methods and parameters thereof, refer to the appropriate Appendix.

VBOFObjectManager

VBOFObjectManager coordinates all object-level activity, ensuring that no given object is instantiated more than once. It coordinates the object clean-up process at the end of the application, manages the instantiations of all VBOFCollections and manages communications between them. It also coordinates with VBOFEventManager to handle triggered events and deliver the necessary notifications to registered objects.

The application program instantiates the VBOFObjectManager. Refer to Chapter 2, section "Starting VBOF Services".

Upon its instantiation, VBOFObjectManager automatically creates an instance of the VBOFEventManager class. For processing without the assistance of VBOFEventManager, refer to section "NoEventMgr" in the Chapter "Conditional Compilation Options."

As the application needs VBOFCollection instances, it requests them from its instance of VBOFObjectManager (see method `NewVBOFCollection`). This way, the VBOFObjectManager can appropriately control and populate the VBOFCollection and guarantee that any object appearing in any VBOFCollection is unique across the environment.

As the application needs singly contained objects, it requests them from its instance of VBOFObjectManager (see method `NewObject`). This way, VBOFObjectManager can appropriately instantiate the desired object from the database, or return a reference to the object, if it is found to already exist.

As the application needs to introduce an object that it created into the realm of VBOFObjectManager support, it requests that VBOFObjectManager properly initialize the object for (see method `InitializeObject`). This way, VBOFObjectManager can appropriately manage the object as it does any other object.

For additional information regarding the methods and their parameters for the VBOFObjectManager Class Module, refer to Appendix B.1.

VBOFCollection (a version was formerly released as "DataAwareCollection")

VBOFCollection coordinates with VBOFObjectManager to present a database-aware replacement to the Visual Basic Collection object. Since VBOFCollection contains the basic methods of the Collection object, `Add`, `Item` and `Remove`, it can functionally replace the Collection object without changing the base application. It also adds database awareness and other object-oriented properties to the role of the collection object.

VBOFCollection can automatically implement the object containment hierarchy of the application's Business Object Model ("BOM") by tracking object containment. It encapsulates all of the SQL necessary to retrieve the appropriate table rows, then automatically converts those rows into instantiated and populated objects of the appropriate contained Class type, and returns them in a collection form.

As objects within the VBOFCollection are added, changed or deleted, VBOF automatically synchronizes the changes with the underlying database.

VBOFCollection also provides the basis for encapsulation of several Visual Basic GUI controls through "wrappers" to provide an object-oriented manner of manipulating those controls. For example, VBOFCollection provides an object-oriented interface to the Visual Basic DBGrid, ListBox and ComboBox controls, such that the application can deal with the objects and VBOFCollection and not with the GUI controls. These features can be used to create non-visual BOMs long before the GUI is ever designed or developed. The BOM can feature object-level and collection-level manipulations, as necessary, then, as the GUI is developed, these GUI controls can controlled by the BOM without change.

For additional information regarding the methods and parameters thereof for the VBOFCollection Class Module, refer to Appendix C.1.

VBOFEventManager

VBOFEventManager coordinates with VBOFObjectManager to present an application-oriented event management system. Since VBOFEventManager is fully encapsulated by VBOFObjectManager, the application program communicates with VBOFEventManager only through methods in VBOFObjectManager.

Any application component can trigger any event at any time (see method "TriggerObjectEvent" in Class Module "VBOFObjectManager"). Any application Class Module or Form can register as an event notification recipient (see method `RegisterForObjectEvent` in Class Module "VBOFObjectManager"). When an event is triggered, VBOFEventManager and VBOFObjectManager jointly process the trigger and send notifications to all registered objects through each object's `ObjectEventCallback` method.

For processing without triggers, refer to section "NoEventMgr" of chapter "Conditional Compilation Options"

VBOFEventObject

VBOFEventObject is a container Class Module for tracking the reference to each registered event recipient. Since VBOFEventObject is fully encapsulated by VBOFObjectManager and VBOFEventManager, and is applicable only to their internal processing of events, there is no direct communication between VBOFEventObject and the application.

For processing without triggers, refer to section "NoEventMgr" of chapter "Conditional Compilation Options"

VBOFComboBoxWrapper

VBOFDataWrapper

The VBOFDataWrapper is a wrapper Class Module for managing the VB DataControl in an object-oriented manner in conjunction with a VBOFCollection and its underlying RecordSet object. Through the VBOFObjectManager, the application receives a new instance of a VBOFDataWrapper (see method `NewVBOFDataWrapper` of VBOFObjectManager) which is essentially bound to a given VBOFCollection and a DataControl. The application then communicates simultaneously with the DataControl, the VBOFCollection and its underlying RecordSet through the VBOFDataWrapper in an object-oriented manner while the VBOFDataWrapper maintains consistency between the contents of the DataControl and the VBOFCollection. In addition, the VBOFDataWrapper provides a single interface for two separate sets of methods supported by the DataControl and VBOFCollection, respectively.

VB applications typically manage the DataControl as a set of records from a data source in a conventional, row-oriented manner. The VBOFDataWrapper allows the DataControl to be managed as a collection of objects in an object-oriented manner. For example, the VBOFDataWrapper polymorphically implements the RecordSet's "Find" and "Move" methods (e.g., `FindFirst`, `FindNext`, . . . , `MoveLast`, etc.) but instead of simply positioning the RecordSet to the appropriate row, the VBOFDataWrapper continues with actually returning the object which equates to the row in the RecordSet.

Any other Controls on the Form which are bound to the DataControl (e.g., Text fields, Labels, CheckBoxes, etc.) also benefit from the features of the VBOFDataWrapper, since Visual Basic's automatic binding features are not inhibited in any way.

VBOFDBGridWrapper

The VBOFDBGridWrapper is a wrapper Class Module for managing the VB DBGrid in an object-oriented manner. Through the VBOFObjectManager, the application receives an instance of VBOFDBGridWrapper (see method `NewVBOFDBGridWrapper`) which is essentially bound to the associated VBOFCollection, its collection of contained objects and the DBGrid control. The application then communicates with the VBOFCollection and DBGrid through the VBOFListBoxWrapper in an object-oriented manner.

For example, VB applications typically manage the DBGrid in its Bound mode with an accompanying DataControl. However, this does not represent good object-oriented technique because of the procedural manner in which these are managed. Through the VBOFDBGridWrapper the application can manage the DBGrid in an objected-oriented manner. In addition, the application typically benefits from increased performance because VBOF eliminates unnecessary reliance on the DataControl, and the VBOFCollection and its contained objects typically can be referenced more quickly than through a DataControl.

The VBOFDBGridWrapper allows the application to treat the DBGrid as a display and update mechanism for the objects in the associated VBOFCollection. The application simply places small code segments at the event procedures of the DBGrid object within the Form module and VBOF assumes control over a great portion of the necessary management duties.

VBOFListBoxWrapper

The VBOFListBoxWrapper is a wrapper Class Module for managing the VB ListBox in an object-oriented manner. Through the VBOFObjectManager, the application receives an instance of VBOFListBoxWrapper (see method `NewVBOFListBoxWrapper`) which is essentially bound to the associated VBOFCollection, its collection of contained objects and the ListBox control. The application then communicates with the VBOFCollection and ListBox through the VBOFListBoxWrapper in an object-oriented manner.

For example, VB applications typically manage the ListBox as a display mechanism for a collection of strings which are representative of the underlying data. Even though the application utilizes the ListBox properties such as its ListIndex, ListCount and Selected to a great extent, the

application is actually dealing with strings and numbers which represent and are loosely affiliated with the underlying data -- *the application is not dealing with the data itself*. The application typically implements a translation scheme to map the underlying data to the representative strings and numbers.

The VBOFListBoxWrapper allows the application to treat the ListBox as a display mechanism for the objects in the associated VBOFCollection. *The application deals directly with the objects without the need for a translation scheme*. All of the ListBox methods and properties are managed through the VBOFListBoxWrapper. For example, to populate the ListBox, the application executes the AddItems method of the VBOFListBoxWrapper, and to manage selected objects, the application executes the SelectedObject or SelectedObjects Get and Set Property methods. Other methods are provided to support robust object-oriented management of the ListBox.

What's Not Included

The current release of VB Object Framework *does not* contain the following features:

- Heterogeneous collections (i.e., collections containing objects of varying Class Modules);
- Support for automatic database synchronization, object persistence or automatic implementation of the object containment hierarchy for those objects which are recipients of the VBOFObjectManager "InitializeObject" method;
- Synchronous Commits of deferred database updates;
- Dynamic, variable, sorting of the order of objects contained in the VBOFCollection;
    - Anything not explicitly stated in this document or found to be available in a Public Method of any of the VBOF Class Modules.

Availability of these features and others depends on customer demand and continued customer acceptance and support of VBOF in the future.

Intended Use

VB Object Framework is Shareware that can be distributed freely and without charge as long as it is distributed as a complete package in its entirety. It is intended to be widely distributed throughout the Visual Basic programming community to achieve and enjoy a higher level of object-oriented programming than that inherent in the OEM edition.

VBOF is intended to enhance the level of object-oriented programming capable under the Microsoft Visual Basic Programming System. By the very nature of the intention of this product, which is to manipulate, data through objects, including adding, updating and deleting data, the following disclaimer must be made to protect the author from use or misuse of the VBOF product:

> Even though it has been thoroughly tested, is believed to be defect-free and earnest in its intent, design and implementation , this product is provided to the user on an as-is, use-at-your-own-risk basis. By using this product in any way, the user indemnifies the authors of any responsibilities or liabilities associated with the product, its use or the results of the user's implementation thereof. The author cannot, and will not, assume any responsibility for the misuse of, the manipulation of nor the loss of data as a result of the use of this product.

Under no circumstances should any application program invoke any method of any VBOF Class Module whose method name begins with the letters "pvt". These methods are considered private, and are typically given the Private attribute. However, in many cases where communication is required between VBOF Class Modules, those methods have had to have been given the Public attribute. Nonetheless, there is no support for these methods being invoked by any non-VBOF Class Module.

If this product brings some benefit to the VB application programmer or corporation, it is expected that the registration fee will be paid, as described in the accompanying file, "ORDER.TXT." Registered users will receive for free all intermediate upgrades of VB Object Framework until its next major release.

Chapter 2: VBOF Services

This section outlines the object-oriented services offered by VBOF.

Starting VBOF Services

Applications must declare a global-level instance of the VBOFObjectManager Class, using the following technique:

```
Public MyObjectManager as New VBOFObjectManager
```

Applications must instantiate the VBOFObjectManager object at the beginning of the application, using the following technique, to include specification of the Database and Workspace:

```
Public Sub CreateObjectManager()
' instantiate the VBOFObjectManager

    Set MyObjectManager = _
        New VBOFObjectManager

    Set MyObjectManager.Database = _
        MyApplicationDatabase

    Set MyObjectManager.Workspace = _
        Workspaces(0)
End Sub
```

There are numerous VBOFObjectManager properties and method parameters which are available, some of which are ideal to set at this time, such as `AutoDeleteOrphans`, `ANSISQL`, `ODBCPassThrough`, etc. Refer to Appendix B.1 for details.

Implementing Contained Objects

VBOF fully supports the containment of singly-occurring objects within other objects (versus being part of a collection of objects which are contained within another object), such as the Manager object contained within each Employee object, or the State object contained within each Address object.

VBOF provides the `NewObject` method which returns the contained object in a fully instantiated form. This method is typically to be used in the `ObjectInitializeFromRecordSet` method of those Class Modules which contain such singly-occurring objects. For examples, refer to Class Modules "Person" (note the assignment of the Mother and Father properties) and "Address" (note the assignment of the state object) of the VBOF Demonstration Package.

Use of this method requires that the application provide a sample object of the desired Class Module within the containing Class Module (refer to those same examples). VBOF uses this information to retrieve the correct row from the specified data source (as indicated within the object referenced by the `Sample:=` parameter). The sample object is not significant to VBOF after the `NewObject` method has completed.

The recommended database design in support of this type of object containment is to place the `ObjectID` property of any contained objects in the Data Source of the containing object. For example, the data source for the Person Class Module would have columns named "MotherObjectID" and "FatherObjectID". The containing Person Class Module can then utilize these columns in its `ObjectInitializeFromRecordSet` method to instantiate those contained objects using the VBOF `NewObject` method

Because of this service, the application can closely align its Class Modules with their respective data sources. This relieves the application of the need to provide database queries which include JOINs across data sources. Also, it should be noted that providing a JOINed query, as such, can jeopardize the "Updatable" status of the data source and can distort the contents of the RecordSet with respect to the Class Module. For example, a query which JOINs an Address data source to a StateCode data source in order to retrieve the correct StateCode property for the Address would actually distort the contents of the Address Class Module since it would then be containing only the StateCode instead of the full State object. It would be more object-oriented to have the VBOF contain the entire State object within the Address object through the NewObject method.

If such singularly-occurring contained objects are present, the Class Module's Public Function ObjectInitializeFromRecordSet might be the ideal location to instantiate those objects.

The following code segment example implements the containment of a Manager object within the Employee object, where the Manager's ObjectID is defined as column "ManagerObjectID" in the Employee data source:

```
Public Function ObjectInitializeFromRecordSet(Optional RecordSet
As Variant) As Person

    Dim NewEmployee as New Employee
    . . .
' copy values from the RecordSet (not important for this example)
    . . .
' pick-up the contained Manager object
    If Not IsNull(RecordSet("ManagerObjectID")) Then
        Set pvtManager = _
            ObjectManager.NewObject( _
                Sample:=NewEmployee, _
                ObjectID:=CStr(RecordSet("ManagerObjectID")))
    End If
```

Populating Collections

Within every object which contains VBOFCollections, the application must declare each of those VBOFCollection objects similar to the following example:

```
Private pvtAddresses As VBOFCollection
```

Note that the attribute lacks the "New" keyword for the VBOFCollection. This is necessary because the VBOF needs to detect and respond appropriately the first time there is any activity for each VBOFCollection.

VBOFCollection initialization within Forms should occur within the Form_Load event procedure (a recommendation.) The VBOFCollection objects should be registered and initialized by the VBOFObjectManager, similar to the following example:

```
Set pvtAddresses = _
    ObjectManager.NewVBOFCollection
```

> Note that Class Modules have a different and completely automatic mechanism for instantiating and initializing their VBOFCollection objects. Refer to the following section for details.

Managing Collections within Class Modules

Within every Class Module which contains VBOFCollections, the application must provide a wrapper method for either retrieving the VBOFCollection or an Item within it. VBOFObjectManager completely assumes responsibility for the detailed implementation, including instantiating and initializing the VBOFCollection. Nevertheless, the application must provide a basic code segment to invoke the necessary VBOFObjectManager method.

Note: In order for the success of the scheme of automatically instantiating and initializing the VBOFCollection, the application program must declare the VBOFCollection without the `New` keyword, as in the following example:

```
Private pvtAddresses As VBOFCollection
```

In the following code segment example, which would be found in a Person Class Module, the contained Addresses collection or an object within it is returned:

```
Public Function Addresses(Optional ObjectID As Variant) As
Variant
' Returns a VBOFCollection of Address objects which are
'   contained by this Person object,
' or
' Returns an Address object whose ObjectID matches the
'   ObjectID parameter.

    Dim tempNewAddress As New Address

    Set Addresses = _
        ObjectManager. _
            ManageCollection( _
                ObjectID:=ObjectID, _
                Collection:=pvtAddressesCollection, _
                Parent:=Me, _
                Sample:=tempNewAddress)
End Function
```

As demonstrated in the example above, the Class Module must pass the `ObjectID` parameter, the VBOFCollection to be searched (`pvtAddressesCollection`), the reference to the object itself (`Me`) as the Parent and a Sample object (`tempNewAddress`) for use by The VBOFObjectManager. The VBOFObjectManager can then fully manage the referenced collection on behalf of the application.

There are numerous properties and method parameters which are available through the `ManageCollection` method, some of which are ideal to set at this time, such as `WhereClause`, `OrderByClause`, `ANSISQL`, `ODBCPassThrough`, etc. Refer to Appendix C.1 for details.

Initializing Objects

Some applications need to instantiate their own single-occurring objects, yet still have those objects benefit from the services of VBOF. In order to support these objects, VBOFObjectManager must be informed of these objects by the application, as in the following example code segment:

```
Dim MyObject As MyClassModule
 . . .
ObjectManager.InitializeObject _
    Object:=MyObject
```

All objects which are initialized in this manner participate in the VBOFObjectManager's system-wide searches designed to prevent duplicate object instantiation. However, since the VBOFObjectManager did not participate in the retrieval of data from any database for this object, there is no mechanism for automatically posting to a database any changes made to the object or its properties.

Thus, objects having been initialized in this manner are not supported by the full complement of VBOF features, such as automatic database synchronization, automatic implementation of object persistence and the automatic implementation of the object containment hierarchy (through the level occupied by such objects).

VB Control Wrappers

In order to enable the application to achieve a high level of object-oriented programming, the VBOF provides several Wrapper Class Modules for enabling object-oriented management of certain VB controls[2].

All instances of VBOF Wrappers must be declared, but should not use the `New` keyword, as in the following example:

```
Dim MyListBoxWrapper As VBOFListBoxWrapper
```

All instances of VBOF Wrappers must be initialized and registered through the application's instance of VBOFObjectManager, as in the following example:

```
Set MyListBoxWrapper = _
    ObjectManager.NewVBOFListBoxWrapper _
        Collection:=pvtPersons, _
        . . .
```

Each VBOF Wrapper Class Module has a different set of named parameters which can be specified for the appropriate `NewVBOF. . .Wrapper` method[3]. In general, there is a named parameter using the name of the VB control (e.g., `ListBox:=` for the `NewVBOFListBoxWrapper` method, `DBGrid:=` for the `NewVBOFDBGridWrapper` method, etc.).

If necessary, the `Collection:=` named parameter can be set to `Nothing` (for example, if the exact VBOFCollection or its contents are not known at the time that the Wrapper instance is initialized), as in the following code segment example:

```
' the exact VBOFCollection isn't known yet
Set MyListBoxWrapper = _
    ObjectManager.NewVBOFListBoxWrapper _
        Collection:=Nothing, _
        . . .
```

provided the Wrapper is rebound using its `Rebind` method and specifying a valid VBOFCollection object prior to its general use, as in the following example:

```
' the VBOFCollection is now known, so Rebind
MyListBoxWrapper.Rebind _
    Collection:=pvtPersons
```

[2] Refer to the ReadMe.txt for a list of the currently available Wrapper Class Modules. If a given VB control does not have a Wrapper, please contact the VBOF author or the VB control vendor to make arrangements.

[3] Refer to the appropriate Appendix to determine the appropriate named parameters for any given VBOF Wrapper Class Module.

```
' the Wrapper can now be used in full
Set MyListBoxWrapper.TopObject = MyObject
```

Wrapping the Data Object

The VB Object Framework provides object-oriented support to the Visual Basic Data control through the VBOFDataWrapper Class Module. Through the VBOFDataWrapper, the application can manage the Data control as a collection of objects in an object-oriented manner, thus relieving the application of the conventional means of Record-oriented data management.

Note: Some of the other the VBOF Wrappers, such as the VBOFDBGridWrapper and VBOFListBoxWrapper, offer higher levels of object-oriented programming than that of the VBOFDataWrapper. Those Wrappers are supported in Unbound mode, and therefore do not need an associated Data control. It is only through the Unbound mode can a higher level of object-oriented programming be achieved. It is highly recommended that these techniques be examined prior to implementing a design based on the VBOFDataWrapper.

Form modules which contain Data controls which are to receive object-oriented services from the VBOFDataWrapper need to perform the following:

- must have at least one Data control drawn on the Form;
- must have an associated instance of VBOFCollection whose underlying RecordSet object is to be used to initialize the Data control's RecordSet property;
- must have an instance of VBOFDataWrapper for each Data control;
- must request that VBOFObjectManager bind each instance of VBOFDataWrapper with the associated VBOFCollection;
    - if desired, can have defined `<DataControlName>_Reposition` event procedures.

For a complete list of the public methods supported by the VBOFDataWrapper and each method's parameters, refer to Appendix D.

Form Methods

This section outlines the methods and procedures which must be coded within the Form in order to receive the services of the VBOFDataWrapper.

Form-Level Declarations

The following is an example of the Form Declarations which would be necessary to receive the services of the VBOFDataWrapper:

```
Public ObjectManager as VBOFObjectManager
Private pvtPersons as VBOFCollection
Private pvtPersonsDataWrapper as VBOFDataWrapper
Private pvtAddresses as VBOFCollection
Private pvtAddressesDataWrapper as VBOFDataWrapper
```

Note: `ObjectManager` is always present, whether or not any VBOFDataWrappers are present. The above example prepares an environment where the respective underlying RecordSets of the `pvtPersons and pvtAddresses` objects can be managed in an object-oriented manner. For more information, refer to the VBOF Demonstration package.

Form_Load Event Procedure, Preparing for Data Control Processing

In the `Form_Load` event procedure the application should request a new VBOFDataWrapper for the Data controls on the Form and bind them to VBOFCollections. If the exact VBOFCollection is not known for any given Data control, the application can specify `Collection:=Nothing`. However, before any significant service is requested of the VBOFDataWrapper the VBOFDataWrapper must be rebound through the `Rebind` method with the `Collection:=` parameter referencing a valid VBOFCollection. The following code segment meets this objective:

```
Private Sub Form_Load()
    Set pvtPersonsDataWrapper = _
        ObjectManager. _
            NewVBOFDataWrapper( _
                Collection:=publicCompany.Persons, _
                DataControl:=Data1)

' only initialize the pvtAddressesDataWrapper at this time
'    since the exact Address collection can't be determined
'    because the exact Person isn't known yet
   Set pvtAddressesDataWrapper = _
        ObjectManager. _
            NewVBOFDataWrapper( _
                Collection:=Nothing, _
                DataControl:=Data2)
```

Data_Reposition Event Procedure, Tracking the Selected Object

In the `Data_Reposition` event procedure the application should reset its internal variables and properties to follow the user's selections through the Data control. The following code segment is an example that meets all of these objectives:

```
Private Sub Data1_Reposition()

' set variables according to the object in the VBOFCollection
'   which equates to the new position of the DataControl
    Set pvtCurrentPerson = _
        pvtPersonsDataWrapper. _
            AbsolutePositionObject

    If pvtCurrentPerson Is Nothing Then
        Data2.Enabled = False
        Exit Sub
    End If

    RefreshPersonFields

    Data2.Enabled = True

' rebind the Wrapper for the contained Addresses collection
    pvtAddressesDataWrapper.Rebind _
        Collection:=pvtCurrentPerson.Addresses

' display details of the first Address
    pvtAddressesDataWrapper.MoveFirst
```

Wrapping DBGrids

The VB Object Framework provides object-oriented support to the Visual Basic DBGrid control through the VBOFDBGridWrapper Class Module. There are two separate steps to be taken for this support:

1. The Class Module of the objects to be presented through the DBGrid must have coded at least the method `ObjectDBGridUnboundReadData`. This method is executed by the VBOFDBGridWrapper when the DBGrid needs to populate itself. In addition, the Class Module can have coded the method `ObjectDBGridUnboundAddData`. This method is used when the user has created a new row in the DBGrid or when the user has changed the contents of one of the displayed rows. Note that VBOF reuses the capabilities of the `ObjectDBGridUnboundAddData` method to implement the equivalent of what would have been yet a third method, `ObjectDBGridUnboundWriteData`.

2. The Form modules which contain DBGrid controls which are to receive object-oriented services from the VBOFDBGridWrapper need to perform the following:

   - must have at least one DBGrid drawn on the Form, and it must be set to "Unbound" mode;
   - must have an associated instance of VBOFCollection whose contained objects are to be displayed on the DBGrid;
   - must have an instance of VBOFDBGridWrapper for each DBGrid;
   - must request that VBOFObjectManager populate the VBOFDBGridWrapper and bind it to the associated VBOFCollection;
   - must have at least the `<DBGridName>_UnboundReadData` method defined;
     - if desired, can also have defined the `<DBGridName>_UnboundAddData`, `<DBGridName>_UnboundDeleteData`, and `<DBGridName>_UnboundWriteData` event procedures. In general, these event procedures are considered optional, but in order to support the full range of capabilities they must be coded.

Note: Throughout this section, the example code illustrates the use of two DBGrids, "DBGrid1" and "DBGrid2", which are intended to display data for collections of "Person" objects and "Address" objects, respectively. Each Person object has an independent collection of Addresses. The application's exact DBGrids, VBOFDBGridWrappers, VBOFCollections and data fields will likely differ from those presented herein.

For a complete list of the public methods supported by the VBOFDBGridWrapper and each method's parameters, refer to Appendix D.

Class Module Methods

In order to support the VBOFDBGridWrapper, the Class Modules of those objects which are to be displayed through the DBGrid must have coded the `ObjectDBGridUnboundReadData` method, and can also have coded the optional method `ObjectDBGridUnboundAddData`.

Note: There is no need to code a separate method such as `ObjectDBGridUnboundWriteData` because the method `ObjectDBGridUnboundAddData` is reused by the VBOFDBGridWrapper to satisfy the requirements of that function.

ObjectDBGridUnboundAddData Method, Adding and Updating Objects

The objectives of the `ObjectDBGridUnboundAddData` method is receive the data values provided by the user and copy them into the object's properties. The method is invoked by the VBOFDBGridWrapper when the user modifies the contents of any given row or adds a new row to the DBGrid. In the event that the user has added a new row, the VBOFDBGridWrapper would have already instantiated a new object (which is, in fact, the recipient of this message).

Note: according to DBGrid documentation obtained directly from Apex Software, the programmer should be cautioned that only selected values are presented to this method. Therefore, the programmer should first check the RowBuf field for Null before attempting to use it. The following is an example of this method:

```
Public Function ObjectDBGridUnboundAddData(Optional DBGrid As
Variant, Optional RowBuf As Variant, Optional NewRowBookmark As
Variant) As Boolean
' Populate the object variables with the values
'   provided by the user in the new row of the
'   DBGrid
'   (in support of VBOFCollection)
'
' Parameter Description:
'   DBGrid:= the DBGrid which is being
'       populated
'   RowBuf:= the current DBGrid RowBuf object
'   NewRowBookmark:= the row number being processed

    Dim I As Long

    For I = 0 To RowBuf.ColumnCount - 1
        If Not IsNull(RowBuf.Value(0, I)) Then
            Select Case RowBuf.ColumnName(I)
                Case "CustomerNumber"
                    CustomerNumber = RowBuf.Value(0, I)
                Case "FirstName"
                    FirstName = RowBuf.Value(0, I)
                Case "LastName"
                    LastName = RowBuf.Value(0, I)
                Case "SSN"
                    SSN = RowBuf.Value(0, I)
                Case "Sex"
                    Sex = RowBuf.Value(0, I)
                Case "DateOfBirth"
                    DateOfBirth = RowBuf.Value(0, I)
                Case "MaritalStatus"
                    MaritalStatus = RowBuf.Value(0, I)

' Note:  Do not initialize the ObjectID.

            End Select
        End If
    Next I

' return "OK" status
    ObjectDBGridUnboundAddData = True
End Function
```

Note that the ObjectID property must not be programmatically altered -- VBOF reserves this function for its own purposes.

ObjectDBGridUnboundReadData Method, Populating the DBGrid

The objective of the `ObjectDBGridUnboundReadData` method is to populate the DBGrid RowBuf object with the contents of the object's properties. Note how the code `RowBuf.ColumnName(I)` is referenced to determine the Column Name coded on the DBGrid and is then used to determine the appropriate property value to provide , as shown in the following code segment:

```
Public Function ObjectDBGridUnboundReadData(Optional DBGrid As
Variant, Optional RowBuf As Variant, Optional RowNumber As
Variant) As Boolean
' Populate the DBGrid RowBuf with values from
'   variables within this object
'   (in support of VBOFCollection)
' Parameter Description:
'   DBGrid:= the DBGrid which is being
'       populated
'   RowBuf:= the current DBGrid RowBuf object
'   RowNumber:= the row number being processed

    Dim I As Long

    For I = 0 To RowBuf.ColumnCount - 1
        Select Case RowBuf.ColumnName(I)
            Case "CustomerNumber"
                RowBuf.Value(RowNumber, I) = CustomerNumber
            Case "FirstName"
                RowBuf.Value(RowNumber, I) = FirstName
            Case "LastName"
                RowBuf.Value(RowNumber, I) = LastName
            Case "SSN"
                RowBuf.Value(RowNumber, I) = SSN
            Case "Sex"
                RowBuf.Value(RowNumber, I) = Sex
            Case "DateOfBirth"
                RowBuf.Value(RowNumber, I) = DateOfBirth
            Case "MaritalStatus"
                RowBuf.Value(RowNumber, I) = MaritalStatus
            Case "ObjectID"
                RowBuf.Value(RowNumber, I) = ObjectID
        End Select
    Next I
End Function
```

This example copies its property values into the current row of the RowBuf object for each of its columns.

Note: under certain conditions, this method might be invoked repeatedly with the identical information being provided at each iteration. According to documentation retrieved directly from Apex Software, this is considered normal and the programmer should not to be concerned.

Form Methods

This section outlines the methods and procedures which must be coded within the Form in order to receive the services of the VBOFDBGridWrapper.

Form-Level Declarations

The following is an example of the Form Declarations which would be necessary to receive the services of the VBOFDBGridWrapper:

```
Public ObjectManager as VBOFObjectManager
Private pvtPersons as VBOFCollection
Private pvtPersonsDBGridWrapper as VBOFDBGridWrapper
Private pvtAddresses as VBOFCollection
Private pvtAddressesDBGridWrapper as VBOFDBGridWrapper
```

Note: `ObjectManager` is always present, whether or not any VBOFDBGridWrappers are present. The above example prepares an environment where Persons can be displayed in one DBGrid, while the Addresses of the selected Person can be displayed in another. For more information, refer to the VBOF Demonstration package.

Form_Load Event Procedure, Preparing for DBGrid Processing

In the `Form_Load` event procedure the application should request a new VBOFDBGridWrapper for each of the DBGrids on the Form and bind them to VBOFCollections. If the exact VBOFCollection is not known for any given DBGrid, the application can specify `Collection:=Nothing`. However, before any significant service is requested of the VBOFDBGridWrapper the VBOFDBGridWrapper must be rebound through the `Rebind` method with the `Collection:=` parameter referencing a valid VBOFCollection. The following code segment meets all of these objectives:

```
Private Sub Form_Load()
' set pvtPersons to the public collection of Persons
    Set pvtPersons = _
        pubPersons

' initialize pvtAddresses as a new VBOFCollection
    Set pvtAddresses = _
        ObjectManager.NewVBOFCollection

' bind the Collection, DBGrid and DBGridWrapper
    Set pvtPersonsDBGridWrapper = _
        ObjectManager.NewVBOFDBGridWrapper( _
            Collection:=pvtPersons, _
            DBGrid:=DBGrid1)

' bind the Collection, DBGrid and DBGridWrapper
    Set pvtAddressesDBGridWrapper = _
        ObjectManager.NewVBOFDBGridWrapper( _
            Collection:=pvtAddresses, _
            DBGrid:=DBGrid2)
```

In the above code VBOFObjectManager instantiates instances of VBOFDBGridWrapper and returns them to the application. The VBOFDBGridWrapper is automatically bound to the VBOFCollection identified in the `Collection:=` parameter and the DBGrid identified in the `DBGrid:=` parameter.

DBGrid1_RowColChange Event Procedure, Following the User

In order to respond to the users moving from row to row within the DBGrid, it is recommended that the DBGrid's `RowColChange` event procedure be coded. The VBOFDBGridWrapper provides an object-oriented means of implementing this event, as follows:

```
Private Sub DBGrid1_RowColChange(LastRow As Variant, ByVal
LastCol As Integer)

    Dim tempBookmark As Variant

 ' get the Bookmark of the current row
    tempBookmark = _
        pvtPersonsDBGridWrapper.Bookmark

' display the current Person
    If Not IsNull(tempBookmark) Then

        Set pvtCurrentPerson = _
            pvtPersonsDBGridWrapper. _
                BookmarkObject

' set the Addresses collection according to the Person
'   (must rebind the wrapper)
        Set pvtAddresses = _
            pvtCurrentPerson.Addresses
        pvtAddressesDBGridWrapper.Rebind _
            Collection:=pvtAddresses
    End If
End Sub
```

This routine uses the VBOFDBGridWrapper to retrieve the current Bookmark of the DBGrid, then translates that into the corresponding object and stores it in the variable `pvtCurrentPerson`. The example continues with resetting the contents of its VBOFCollection `pvtAddresses`, since that collection represents contained objects relative to the `pvtCurrentPerson` object. The `pvtAddressesDBGridWrapper` is then rebound so the associated VBOFDBGridWrapper can perform any necessary adjustments.

Code such as this should appear any time such drastic changes (such as reassignment) occur to either the bound VBOFCollection or the DBGrid.

DBGrid1_UnboundAddData Event Procedure, Adding Objects

In order to support the `AllowAdd` property of the DBGrid, the following code must appear in the `UnboundAddData` event procedure for the DBGrid within the Form:

```
Private Sub DBGrid1_UnboundAddData(ByVal RowBuf As RowBuffer,
NewRowBookmark As Variant)

    Dim tempPerson As New Person

    pvtPersonsDBGridWrapper. _
        UnboundAddData _
            RowBuf:=RowBuf, _
            NewRowBookmark:=NewRowBookmark, _
            Sample:=tempPerson
End Sub
```

The code is very simple, in that it simply passes its parameters to the VBOFDBGridWrapper. VBOF completely coordinates and synchronizes all updates with the underlying VBOFCollection and objects.

After the user has entered values into the "new row" line at the bottom of the DBGrid, Visual Basic executes the DBGrid's `UnboundAddData` event procedure. The application simply transfers control to the `UnboundAddData` method of the associated VBOFDBGridWrapper. The VBOFDBGridWrapper completely manages the process of:

- instantiating a new object
- populating the object with the data from the new row
- adding the object to the underlying VBOFCollection
- logging the object containment information from the appropriate container object to the new object
- writing the object data to the appropriate Data Source
    - refreshing the display of the DBGrid

Note: In order to support adding new rows to the DBGrid, the `AllowAdd` property of the DBGrid must be set to True.

In order to provide a more intuitive interface, it might be desirable to position to the most recently added object. To do so, the following code segment can be integrated into the above, immediately after the execution of the `UnboundAddData` method:

```
Set pvtCurrentPerson = _
    pvtPersons.MostRecentlyAddedObject

Set pvtPersonsDBGridWrapper.BookmarkObject = _
    pvtCurrentPerson
```

DBGrid1_UnboundDeleteRow Event Procedure, Deleting Objects

In order to support the `AllowDelete` property of the DBGrid, the following code must appear in the `UnboundDeleteRow` event procedure for the DBGrid within the Form:

```
Private Sub DBGrid1_UnboundDeleteRow(Bookmark As Variant)

    pvtPersonsDBGridWrapper. _
        UnboundDeleteRow _
            Bookmark:=Bookmark
End Sub
```

The code is very simple, in that it simply passes its parameters to the VBOFDBGridWrapper. VBOF completely coordinates and synchronizes all updates with the underlying VBOFCollection and objects.

In order to provide a more intuitive interface, it might be desirable to position the display to the first object of the VBOFCollection after having deleted the previous object. To do so, the following code segment can be integrated into the above, immediately after the execution of the `UnboundDeleteRow` method:

```
If pvtPersons.Count > 0 Then
    Set pvtCurrentPerson = _
        pvtPersons.Item(1)

    Set pvtPersonsDBGridWrapper.BookmarkObject = _
        pvtCurrentPerson
```

DBGrid1_UnboundWriteData Event Procedure, Updating Objects

In order to support the AllowUpdate property of the DBGrid, the following code must appear in the `UnboundWriteData` event procedure for the DBGrid within the Form:

```
Private Sub DBGrid1_UnboundWriteData(ByVal RowBuf As RowBuffer,
WriteLocation As Variant)

    pvtPersonsDBGridWrapper. _
        UnboundWriteData _
            RowBuf:=RowBuf, _
            WriteLocation:=WriteLocation
```

The code is very simple, in that it simply passes its parameters to the VBOFDBGridWrapper. VBOF completely coordinates and synchronizes all updates with the underlying VBOFCollection and objects.

DBGrid1_UnboundReadData Event Procedure, Providing Property Values

In order to populate the DBGrid with data from the objects appearing in the VBOFCollection, the following code must appear in the `UnboundReadData` event procedure for the DBGrid within the Form:

```
Private Sub DBGrid1_UnboundReadData(ByVal RowBuf As RowBuffer,
StartLocation As Variant, ByVal ReadPriorRows As Boolean)

    pvtPersonsDBGridWrapper. _
        UnboundReadData _
            RowBuf:=RowBuf, _
            StartLocation:=StartLocation, _
            ReadPriorRows:=ReadPriorRows
End Sub
```

The code is very simple, in that simply passes its parameters to the VBOFDBGridWrapper which assumes all details for populating values into the DBGrid.

Wrapping List Boxes and Combo Boxes

The VB Object Framework provides object-oriented support to the Visual Basic ListBox and ComboBox controls through the VBOFListBoxWrapper Class Module. For example, the VBOFListBoxWrapper supports populating the ListBox from the objects contained within a given VBOFCollection. As the user selects one or more objects, the VBOFListBoxWrapper returns those selected objects to the application. Through the VBOFListBoxWrapper, the application can manage the ListBox as a collection of objects in an object-oriented manner, thus relieving the application of the conventional means of ListBox management through indirect mechanisms such as indexes and representative text strings.

Throughout the "Wrapping List Boxes and Combo Boxes" section of text, support is extended equally to the ListBox and ComboBox. Therefore textual passages referring to the ListBox also apply to the ComboBox unless otherwise specifically stated.

There are two separate steps to be taken for this support:

1. The Class Module of the objects to be presented through a ListBox must have coded at least the method `ObjectListBoxValue`. This method is executed by the VBOFListBoxWrapper when the ListBox needs to populated.
2. The Form modules which contain ListBox controls which are to receive object-oriented services from the VBOFListBoxWrapper need to perform the following:

- must have at least one ListBox drawn on the Form, and it must be set to `Unbound` mode;
- must have an associated instance of VBOFCollection whose contained objects are to be displayed in the ListBox;
- must have an instance of VBOFListBoxWrapper for each ListBox;
- must request that VBOFObjectManager populate the VBOFListBoxWrapper and bind it to the associated VBOFCollection;
    - if desired, can have defined `<ListBoxName>_Clicked` event procedures.

Note: Throughout this section, the example code illustrates the use of a ListBox, "ListBox1", which is intended to display data for the collection of "Person" objects. Your exact ListBox, VBOFListBoxWrappers, VBOFCollections and data fields will likely differ from those presented herein.

For a complete list of the public methods supported by the VBOFDBGridWrapper and each method's parameters, refer to Appendix E.

Class Module Methods

In order to support the VBOFListBoxWrapper, the Class Modules of those objects which are to be displayed through the ListBox must have coded the `ObjectListBoxValue` method.

ObjectListBoxValue, Providing a Representative String for the ListBox

The objective of the `ObjectListBoxValue` method is to provide a string value to represent the object in a ListBox.

In the following example, the object uses its `FormattedName` method to formulate a representative string value for itself:

```
Public Function ObjectListBoxValue() As String
' Return a String that will represent this object
'   in a ListBox

    ObjectListBoxValue = _
        Me.FormattedName
End Function
```

The Class Module is responsible for determining the appropriate textual string value to represent the object in the ListBox.

Form Methods

This section outlines the methods and procedures which must be coded within the Form in order to receive the services of the VBOFListBoxWrapper.

Form-Level Declarations

The following is an example of the Form Declarations which would be necessary to receive the services of the VBOFListBoxWrapper:

```
Public ObjectManager as VBOFObjectManager
Private pvtPersons as VBOFCollection
Private pvtPersonsListBoxWrapper as VBOFListBoxWrapper
Private pvtAddresses as VBOFCollection
Private pvtAddressesListBoxWrapper as VBOFListBoxWrapper
```

Note: `ObjectManager` is always present, whether or not any VBOFListBoxWrappers are present. The above example prepares an environment where "Persons" can be displayed in the ListBox. For more information, refer to the VBOF Demonstration package.

Form_Load Event Procedure, Preparing for ListBox Processing

In the `Form_Load` event procedure, the application should request a new VBOFListBoxWrapper for each of the ListBoxes on the Form and bind them to VBOFCollections. If the exact VBOFCollection is not known for any given ListBox, the application can specify `Collection:=Nothing`. However, before any significant service is requested of the VBOFListBoxWrapper the VBOFListBoxWrapper must be rebound through the `Rebind` method with the `Collection:=` parameter referencing a valid VBOFCollection. The following code segment meets all of these objectives:

```
Private Sub Form_Load()
' initialize pvtPersons as a new VBOFCollection
    Set pvtPersons = _
        ObjectManager.NewVBOFCollection

' bind the Collection, ListBox and ListBoxWrapper
    Set pvtPersonsListBoxWrapper = _
        ObjectManager.NewVBOFListBoxWrapper( _
            Collection:=pvtPersons, _
            ListBox:=ListBox1)

' only initialize the pvtAddressesListBoxWrapper at this time
'    since the exact Person isn't known yet
    Set pvtAddressesListBoxWrapper = _
        ObjectManager.NewVBOFListBoxWrapper( _
            Collection:=Nothing, _
            ListBox:=ListBox2)
```

ListBox_Click Event Procedure, Tracking the Selected Object

In the `ListBox_Click` event procedure the application should reset its internal variables and properties to follow the user's selections through the ListBox. The following code segment is an example that meets all of these objectives:

```
Private Sub ListBox_Click()
    Dim tempObjectID As Long

    On Local Error Resume Next

    Set pvtCurrentPerson = _
        pvtPersonsListBoxWrapper.ListIndexObject

' display the person's detail information
    efCustomerNumber = _
        pvtCurrentPerson.CustomerNumber
    efFirstName = _
        pvtCurrentPerson.FirstName
    efLastName = _
        pvtCurrentPerson.LastName
    efSSN = _
        pvtCurrentPerson.SSN
    efDateOfBirth = _
        Format$(pvtCurrentPerson.DateOfBirth, "mm/dd/yyyy")
```

```
        lbxMaritalStatusCodes = _
            pvtCurrentPerson.MaritalStatus
        lbxGenderCodes = _
            pvtCurrentPerson.Sex
        efFormattedName = _
            pvtCurrentPerson.FormattedName
        efAge = _
            pvtCurrentPerson.Age
    End Sub
```

Wrapping the RecordSet Object

The VB Object Framework provides object-oriented support to the Visual Basic RecordSet control through the VBOFRecordSetWrapper Class Module. Through the VBOFRecordSetWrapper, the application can manage the RecordSet control as a collection of objects in an object-oriented manner, thus relieving the application of the conventional means of Record-oriented data management.

Note: Some of the other the VBOF Wrappers, such as the VBOFDBGridWrapper and VBOFListBoxWrapper, offer higher levels of object-oriented programming than that of the VBOFRecordSetWrapper. Those Wrappers are supported in Unbound mode, and therefore do not need an associated RecordSet or Data control. It is only through the Unbound mode can a higher level of object-oriented programming be achieved. It is highly recommended that these techniques be examined prior to implementing a design based on the VBOFRecordSetWrapper.

Class Modules and Forms which are to receive object-oriented services from the VBOFRecordSetWrapper need to perform the following:

- must have an associated instance of the VBOFCollection Class Module whose underlying RecordSet object is to be bound to the VBOFRecordSetWrapper;
- must request that VBOFObjectManager bind each instance of VBOFRecordSetWrapper with the associated VBOFCollection.

For a complete list of the public methods supported by the VBOFRecordSetWrapper and each method's parameters, refer to Appendix G.

Form Methods

This section outlines the methods and procedures which must be coded within the Form in order to receive the services of the VBOFDataWrapper.

Form-Level Declarations

The following is an example of the Form Declarations which would be necessary to receive the services of the VBOFDataWrapper:

```
Public ObjectManager as VBOFObjectManager
Private pvtPersons as VBOFCollection
Private pvtPersonsRecordSetWrapper as VBOFRecordSetWrapper
Private pvtAddresses as VBOFCollection
Private pvtAddressesRecordSetWrapper as VBOFRecordSetWrapper
```

Note: `ObjectManager` is always present, whether or not any VBOFRecordSetWrappers are present. The above example prepares an environment where the underlying RecordSet of the `pvtPersons` object can be managed in an object-oriented manner. For more information, refer to the VBOF Demonstration package.

Form_Load Event Procedure, Preparing for RecordSet Processing

In the `Form_Load` event procedure the application should request new VBOFRecordSetWrappers. The following code segment meets this objective:

```
Private Sub Form_Load()
    Set pvtPersonsRecordSetWrapper = _
        ObjectManager. _
            NewVBOFRecordSetWrapper( _
                Collection:=publicCompany.Persons)

' only initialize the pvtAddressesRecordSetWrapper at this time
'    since the exact Address collection can't be determined
'    because the exact Person isn't known yet
   Set pvtAddressesRecordSetWrapper = _
        ObjectManager. _
            NewVBOFRecordSetWrapper( _
                Collection:=Nothing)
```

Using VB Object Framework in Conjunction with the Visual Basic Object Browser

Once the VBOF Class Modules have been added to the Visual Basic Project, the VB Object Browser can be used to paste templates of the VBOF methods and properties into the application code. Displayed within the Object Browser is a brief description of each method.

> Do not paste any VBOF method whose name begins with "pvt". These methods are intended only for internal VBOF use. These methods are not published, and are therefore, not guaranteed to be retained or migrated in future releases of VBOF.

Refer to Chapter 7, section "Pasting Code Fragments" in the Microsoft Visual Basic Programmer's Guide for additional information

Chapter 3: Application Requirements and Recommendations

For support by the VB Object Framework, there are several required and optional features for the VB Project, Class Modules and Forms. This section outlines these. Not included in this section are the required and optional features pertaining to the various VBOF Wrapper Class Modules.

Visual Basic Project

This section outlines the Project-level application components which are required or recommended for support by VB Object Framework.

1. (Required) The following VB Object Framework Class Modules must be added to the VB Project:

VBOFOMgr.cls	VBOFObjectManager
VBOFColl.cls	VBOFCollection

2. (Optional) The following Class Modules can be included, if VB Object Framework Event Management is desired:

VBOFEMgr.cls	VBOFEventManager
VBOFEvnt.cls	VBOFEventObject

3. (Optional) Any of the following Class Modules can be included, if GUI wrappers are desired:

VBOFData.cls	VBOFDataWrapper
VBOFDBGr.cls	VBOFDBGridWrapper
VBOFLBox.cls	VBOFListBoxWrapper
VBOFRSet.cls	VBOFRecordSetWrapper

4. (Required) Applications must declare a single, global-level instance of the VBOFObjectManager Class, using the following technique:

```
Public MyObjectManager as New VBOFObjectManager
```

Applications must instantiate the VBOFObjectManager object at the beginning of the application, using the following technique:

```
Public Sub CreateObjectManager()
' instantiate the VBOFObjectManager

    Set MyObjectManager = _
        New VBOFObjectManager

    Set MyObjectManager.Database = _
        MyApplicationDatabase

    Set MyObjectManager.Workspace = _
        Workspaces(0)
End Sub
```

This is also an ideal location to specify some of the VBOFObjectManager's run-time properties, such as `Verbose`, `DebugMode`, `ANSISQL`, `AutoDeleteOrphans`, etc., as shown in the following code segment:

```
With MyObjectManager
    .Verbose = True
    .DebugMode = True
    .ANSISQL = True
End With
```

Refer to Appendix B.1 for a complete reference to the VBOFObjectManager public methods.

Class Modules

This section outlines the application components which are required or recommended for support by VB Object Framework that pertain to the Class Module.

1. (Required) All Class Modules must have the following General Declarations[4]:

```
Public ObjectID As Long
Public ObjectChanged As Long
Public ObjectAdded As Long
Public ObjectDeleted As Long
Public ObjectParentCount As Long
Public ObjectManager As VBOFObjectManager
```

   Note that the VB Object Framework assumes the responsibility of controlling each of the above properties, including propagating the single instance of VBOFObjectManager across all objects in the application, into each object's respective `ObjectManager` property. Therefore, there is no need for the application to perform any maintenance upon these variables. If needed, the application can specifically request the initialization of an object it has instantiated through the VBOFObjectManager's `InitializeObject` method.

2. (Recommended) All Class Modules should code a `Class_Terminate` method to release any links it may hold to other objects, as follows:

```
Private Sub Class_Terminate()
' Terminate Me
    ObjectManager.TerminateObject _
        Me
```

   Note that each instance of any VBOF Wrapper must be identified to the `TerminateObject` method.

3. (Required) Within every object which contains VBOFCollections, the application must provide a wrapper method for retrieving either the VBOFCollection or an Item within it. While VBOFObjectManager completely assumes responsibility for the detailed implementation, the application must provide the following code segment to invoke the necessary VBOFObjectManager method. In the following example, which would be found in a Person Class Module, the Addresses collection which is contained within the Person object is returned or an object within it is returned:

```
Public Function Addresses(Optional ObjectID As Variant) As
Variant
' Returns a VBOFCollection of Address objects which are
'   contained by this Person object,
```

[4] If available, these requirements can be eased through the use of a VB Add-In inheritance tool to simplify the requirements upon the Class Module. For example, Sheridan Software, Inc. offers ClassAssist product which supports inheritance.

```
' or
' Returns an Address object whose ObjectID matches the
'   ObjectID parameter.

    Dim tempNewAddress As New Address

    Set Addresses = _
        ObjectManager. _
            ManageCollection( _
                ObjectID:=ObjectID, _
                Collection:=pvtAddressesCollection, _
                Parent:=Me, _
                Sample:=tempNewAddress)
End Function
```

As demonstrated in the example above, the Class Module must pass-along the `ObjectID` parameter, while adding a reference to the VBOFCollection to be searched, the reference to itself as the `Parent`, and a `Sample` object for use by VBOFObjectManager.

4. (Required) Within every object which contains VBOFCollections, the application must provide a wrapper method for adding Items to the Collection. In the following example, which would be found in a Person Class Module, an Address object is added to the contained collection of Addresses:

```
Public Function AddAddress(Optional Item As Variant,
Optional Parent As Variant) As VBOFCollection
' Add an Address to my pvtAddressesCollection

    Set AddAddress = Me.Addresses.Add( _
        Item:=Item, _
        Parent:=Me)

' (Optional) trigger the "Changed" event
    ObjectHasChanged
End Function
```

5. (Required) Each Class Module must define a `Public` method to instantiate a new copy of itself, as follows:

```
Public Function ObjectNewInstanceOfMyClass() As Person
' Return a new instance of this class
    Set ObjectNewInstanceOfMyClass = New Person
End Function
```

6. (Required) Each Class Module must define a `Public` method to return a string which names the Data Source to be used to retrieve the appropriate data rows for populating instances of the Class Module, as follows:

```
Public Function ObjectDataSource() As String
' Return the Data Source with which this Class is
associated
    ObjectDataSource = "Persons"
End Function
```

7. (Required) Each Class Module must define a `Public` method to copy the contents of a row of a RecordSet object to the object's Properties, as in the following example:

```
Public Function ObjectInitializeFromRecordSet(Optional
RecordSet As Variant) As Person
' Populate my variables from the RecordSet

    On Local Error Resume Next

    CustomerNumber = RecordSet("CustomerNumber")
    FirstName = RecordSet("FirstName")
    LastName = RecordSet("LastName")
    SSN = RecordSet("SSN")
    Sex = RecordSet("Sex")
    DateOfBirth = RecordSet("DateOfBirth")
    MaritalStatus = RecordSet("MaritalStatus")
    ObjectID = RecordSet("ObjectID")
    Set ObjectInitializeFromRecordSet = Me
End Function
```

If singularly-occurring contained objects are present (such as the `pvtManager` property of an Employee object where `pvtManager` is declared as a Person object), this might be the ideal location to instantiate those objects, as in the following code segment:

```
    Dim NewPerson as New Person

' pick-up the Manager object
    If Not IsNull(RecordSet("ManagerObjectID")) Then
        Set pvtManager = _
            ObjectManager.NewObject( _
               Sample:=NewPerson, _

ObjectID:=CStr(RecordSet("ManagerObjectID")))
    End If
```

8. (Required) Each Class Module must define a `Public` method to copy its Properties to the current row of a RecordSet object (the opposite of the above requirement), as in the following example:

```
Public Function ObjectInitializeRecordSet(Optional
RecordSet As Variant) As Long
' Populate the RecordSet with my variables.
'   Return any error code encountered.
' Note:  Do not initialize the ObjectID column.

    On Local Error GoTo InitializeRecordSet_SetError
    Err = 0

    RecordSet("CustomerNumber") = CustomerNumber
    RecordSet("FirstName") = FirstName
    RecordSet("LastName") = LastName
    RecordSet("SSN") = SSN
    RecordSet("Sex") = Sex
    RecordSet("DateOfBirth") = DateOfBirth
    RecordSet("MaritalStatus") = MaritalStatus
```

```
    GoTo InitializeRecordSet_SetError

InitializeRecordSet_SetError:
    ObjectInitializeRecordSet = Err
    Exit Function
End Function
```

If singularly-occurring contained objects are present (such as the `pvtManager` property of an Employee object where `pvtManager` is declared as a Person object), this might be the ideal location to set any appropriate foreign key values, as in the following code segment:

```
' set the Manager object
    If Not pvtManager Is Nothing Then
        RecordSet("ManagerObjectID") = _
            pvtManager.ObjectID
    Else
        RecordSet("ManagerObjectID") = Null
    End If
```

Class Modules and Forms

This section outlines the application components which are required or recommended for support by VBOF pertaining to the Class Module or the Form.

1. (Optional) Each Class Module or Form can specify the `ObjectEventCallBack` method to respond to events which have occurred elsewhere within the application. This method is invoked by the VBOFEventManager to notify the object of the triggering of an event for which the object is a registered recipient. The following example is from a Form module:

```
Public Function ObjectEventCallBack(Optional Event As
Variant, Optional Object As Variant) As Long
' Receive the Trigger notification and process accordingly

    Dim tempObjectType As String

    On Local Error Resume Next

' determine the type of object triggering the event
    tempObjectType = TypeName(Object)

' process events triggered by the Person object
    If tempObjectType = "Person" Then
        If UCase$(Event) = "REMOVEDITEM" _
        Or UCase$(Event) = "CHANGED" _
        Then
            RefreshCustomerList

' if it was my Domain Object, refresh my displays
            If Object.ObjectID = pvtCurrentPerson.ObjectID
_
            Then
                RefreshPersonFields
            End If
        End If
    End If
End Function
```

Note that the conditional compilation parameter `#If NoEventMgr = False` must be set for the VBOFEventManager to be in effect.

2. (Optional) Each Class Module or Form can define a `Private` method to trigger the "Changed" event. The triggering of events is a commonly used technique in object-oriented implementations. It allows a given object to inform any number of other anonymous other objects and Forms (those which are so registered as being interested) in its changed state. Triggering any event causes the VBOFObjectManager to propagate the event throughout the remainder of the application to all such registered recipient objects. Those registered objects are notified through their respective `ObjectEventCallBack` methods (see above).

   The following example from a Class Module is designed to be invoked from elsewhere within the Class Module as its properties are changed. This causes the VBOFObjectManager to trigger the "Changed" event:

```
Private Sub ObjectHasChanged()
' Mark this object as "Changed" and trigger the "Changed"
'   event

    ObjectChanged = True

#If NoEventMgr = False Then
' Trigger the "Changed" event
    If Not ObjectManager Is Nothing Then
        ObjectManager.TriggerObjectEvent _
            Event:="Changed", _
            Object:=Me
    End If
#End If
```

   Note that the conditional compilation parameters `#If` and `#End If` are present. This is in support of enabling or disabling the services of the VBOFEventManager through conditional compilation.

Forms

This section outlines the requirements and recommendations for Forms.

1. (Recommended) Each Form should have a `Form_QueryUnload` method which invokes the `TerminateForm` or the `Form_QueryUnload` method[5] (both are equivalent) of the application's instance of the VBOFObjectManager, as in the following code segment:

```
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode
As Integer)

' Terminate Me and all my Wrappers
    ObjectManager.TerminateForm _
        Me, _
        pvtPersonsListBoxWrapper, _
        pvtAddressesListBoxWrapper, _
```

[5] These two equivalent methods are provided by the VBOFObjectManager because the `TerminateForm` method is roughly polymorphic with the `TerminateObject` method used to terminate object instances of Class Modules, while the `Form_QueryUnload` method is provided because it is easy to remember, given that it belongs in the `Form_QueryUnload` method of the Form.

```
. . .
```

2. (Optional) Each Form can register as a recipient of events, particularly those triggered by the objects being displayed on the form and their supporting VBOFCollections, as in the following code segment:

```
Private Sub RegisterForEvents()

#If NoEventMgr = False Then
    ObjectManager.RegisterForObjectEvent _
        TriggerObjectType:="Person", _
        RegisterObject:=Me

    ObjectManager.RegisterForCollectionEvent _
        Collection:=pvtPersons, _
        RegisterObject:=Me, _
        TriggerEvent:="AddedItem"

    ObjectManager.RegisterForCollectionEvent _
        Collection:=pvtPersons, _
        RegisterObject:=Me, _
        TriggerEvent:="RemovedItem"
#End If
```

This code segment registers the Form as a recipient of notification upon the triggering of any event by any of the Person objects, and the triggering of either the "AddedItem" or "RemovedItem" events triggered by the `pvtPersons` VBOFCollection object.

In order to receive notification of event triggers, the form must provide an appropriate `ObjectEventCallBack` method.

By registering for events, as such, the Form and the Class Module can remain fully independent for an effective separation of the BOM and GUI, yet can work together through events, when appropriate.

Data Sources

This section outlines the application components which are required or recommended for support by VB Object Framework that pertain to the Database and Tables which serve as Data Sources for Class Modules.

1. (Required) Each Class Module's referenced Data Source must be updatable.

2. (Required) Each Data Source must be either a Visual Basic (a.k.a. MS Access) Database or must be ODBC-accessible.

3. (Required) Each Data Source must have a Column with the following attributes:

```
Name:       ObjectID
Type:       Numeric (Long)
Attributes: Counter
```

Since, by default, VBOF can access all data rows via their ObjectID column, it can be argued that the ObjectID column should be the Primary Key column. However, if the Data Source will be processed by facilities other than VBOF, this may not necessarily deliver optimum performance. In this case, the ObjectID should be placed in an Index with the Unique attribute on the Data Source.

4. (Required) The Database must have a Table with the following attributes:

```
TableName:  VBObjectFrameworkObjectLinks

Column:     FromObjectType, Character(64)
Column:     FromObjectID, Number (Long)
Column:     ToObjectType, Character(64)
Column:     ToObjectID, Number (Long)

PrimaryKey: (FromObjectType, FromObjectID,
            ToObjectType, ToObjectID)
```

Chapter 4: Object-Oriented Development Strategies

This section outlines some object-oriented techniques and how they apply to using the VBOF.

Developing the Non-Visual BOM First

One of the most interesting opportunities presented by object-oriented technology is that it allows for the development of a non-visual BOM ("Business Object Model") before applying any GUI or database aspects to the application. Similar to the construction of homes and buildings where the foundation is laid and verified before the walls go up, object-oriented design methodologies typically dictate that a non-visual BOM should be developed, tested and fully working before the user interface and databases are finalized.

The BOM-first approach allows frequent, even drastic, changes to be modeled without having to perform simultaneous maintenance to corresponding GUI and database components. In fact, the longer the design and finalization of the user interface and databases can be deferred, the better, since this extends this early period of high-flexibility. This technique supports wide-open creativity because it encourages the pursuit of numerous, varying and unconventional design approaches which can be modeled and evaluated cheaply and quickly. This would not be considered an option under conventional, GUI-centric design techniques because of the time and costs associated with taking such ventures.

Rather than following the conventional "waterfall" methodologies to application design and development, this approach supports cyclical development, such that each subsequent iteration of the BOM typically offers more than the previous and is closer to being "correct". Nearly all first-time OO development projects go through several "false-start" scenarios, but the very nature of OO prevents any of these from actually being considered a loss or a failure. Rather, each subsequent iteration is typically an improvement upon the previous, and each contributes something to the ultimate solution.

Another advantage of the non-visual BOM is that it can be deployed in many forms, not just under a GUI. For example, Class Modules which have no user interface are ideal for deployment as OLE components to support reports, spreadsheets and other OLE Automation purposes. By comparison, if the class modules contain user interface code or if the business rules are interlaced throughout the GUI, as is the case with conventional VB development approaches, it might be very difficult to provide such a multifaceted degree of support.

The VBOF integrates very well with the BOM-first approach, since its capabilities would include effortlessly bringing object and RecordSet management capabilities to the role of the Collection. This allows the BOM to be designed and developed as usual before any database is available or even defined.

Then, when the database becomes available, the features of the VBOF can be exploited without any changes to the BOM. This allows the BOM to be developed with focus being placed on meeting the business requirements, rather than having to be concerned about making allowances for future object and RecordSet processing. As the GUI layer is added, the VBOF Wrapper classes can be exploited.

Using Events in Conjunction with OO Programming

Event Management is an important part of object-oriented programming because it enhances class independence and encapsulation. In this capacity, this means that each class needs to be concerned primarily with itself -- it has the option to become intimate with selected other classes, but only where appropriate. This environment of high levels of independence between classes allows each class to evolve independently of any other towards an overall higher quality BOM.

For example, suppose a customer-oriented application is successfully deployed. Previously unknown business requirements are stated which demand an integrated product cross-selling mechanism to be integrated with the customer-oriented functions and suggest qualifying products to the sales representative during the course of the conversation with the customer. At this point there is usually a temptation by the

development team to integrate the product cross-selling functions directly into the customer-oriented application, but that effectively makes the customer-oriented application very sensitive to changes to either set of functions. Unfortunately, it is this very type of application evolution which erodes the flexibility and life span of applications over time – trying to do too much about too many requirements and functions, and having to be too sensitive to too many sources of change. The better option is to develop the product cross-selling application as a separate application which responds to events triggered by the customer-oriented application. As changes are posted within the customer-oriented application it would trigger events. As those events are received by the product cross-selling application, the underlying objects can be evaluated for the need to display any necessary suggestions to the sales representative. This allows both application to exist and evolve separately without eroding the flexibility of either application. This should also increase the life span of the applications.

There are certain restrictions which should be observed, such as a BOM object should never communicate directly with a GUI object, since this forces the reliance of the BOM upon that GUI object. Thus, in order to keep the BOM independent of the GUI, the GUI should register as an interested party for any events triggered by the BOM.

If desired, the BOM can also register as an interested party of events triggered by the GUI in a scheme known as collaboration. Collaboration, when used very carefully, can be an excellent communication mechanism. For example, the BOM and GUI must be cooperatively designed for collaboration, since the GUI would need to execute special BOM methods to let the BOM know about the GUI. Likewise, the GUI needs to be respectful that there might be many such parties registered as interested in a given BOM object, and that it's not necessarily the BOM's responsibility to track all of them.

As the BOM and GUI are developed, as issues are raised regarding responsibility, collaboration, independence and communication, the following general guidelines should be followed:

- At no point should any class within the BOM find itself doing something that would be better performed by some other class.

- Each BOM class should communicate with a minimum number of other classes.

It should be noted that the VBOF communicates across its objects through events. For example, as VBOFCollections change their state (e.g., by adding, changing or deleting objects, populating from the database, etc.) they trigger events which other VBOFCollections receive and process. This is how the VBOF Demonstration package is able to immediately reflect changes to a given object across multiple windows – even though the object was changed on only one of those windows.

Refer to Appendix B.2 and C.2 for a listing of events triggered by VBOF objects.

Using VB Object Framework in Collection-Emulation mode

Most object-oriented leaders recommended that neither a Database nor a GUI should be defined or even be a concern to the object designers until a significant portion of the non-visual Business Object Model ("BOM") has been completed. However, in order to support the implementation of the object containment hierarchy, the Collection object must be available. When designing for VBOF services, the VBOFCollection should be used in lieu of the VB Collection (note that the services of the VBOFObjectManager are required for support of the VBOFCollection.) When the database and GUI layers are introduced into the design, the advanced features of the VBOFCollection can be easily invoked without change to the BOM.

In order to implement VBOF's collection-emulation mode, the application Class Modules simply return a null string value from their `ObjectDataSource()` method or remove their `ObjectDataSource()` methods altogether.

Introducing a Database into the Non-Visual BOM

Prior to the implementation of a database, the VBOFCollections can be used in "Collection-Emulation" mode. Later, when the database is available, the VBOFCollections can begin using the database without any changes to the application. During any interim period, the application's Class Modules can independently migrate to database mode.

To begin operating in full database support mode, the application Class Modules provide legitimate values from their respective `ObjectDataSource()` methods.

Introducing a GUI over a Non-Visual BOM

When the time has come to begin developing a GUI over the application's BOM, it is important to avoid making compromises to the higher level of OO programming achieved through the VBOF. Since the GUI becomes the focus of the development effort at this point, it may be tempting for long-time VB programmers to immediately begin applying conventional VB GUI-centric programming tactics. The VBOF can help retain the achieved level of OO programming through its Wrapper classes and Event Management facilities.

The VBOF Wrapper classes provide an object-oriented interface for managing several of VB's GUI controls, such as the ListBox, ComboBox, DBGrid, etc. For more information, refer to Chapter 2, section "VB Control Wrappers".

The VBOF Event Management facilities can also be used to avoid any dependence of the BOM upon the GUI. Event Management offers the ability to communicate between tasks without necessarily either party knowing exactly the party. There is risk to the quality of the BOM if the BOM is "aware" of the GUI, since the GUI could change unexpectedly which might disrupt the BOM if it has an intimate relationship with the GUI. Also, if the BOM is, in fact, intimate with a GUI, it likely has a reduced potential to be deployed in any other capacity, such as an OLE Server.

If the GUI and BOM are designed to communicate only through events then the BOM can be shielded from these risks. To achieve this, the GUI registers as an interested party for any events triggered by the respective BOM Domain Objects. For example, a GUI which manages information about a specific Person object would register with the VBOFObjectManager as an interested party in the events triggered by that Person object. If the GUI manages a collection of Person objects then it would register as an interested party in the events triggered by the VBOFCollection which contains those Person objects. In addition, there might be situations where a GUI would need to register as an interested party in the events triggered by any Person object, not necessarily any given object or collection of objects.

In order to receive notification of a triggered event, the Form must provide an `ObjectEventCallBack` method, as described in Chapter 3, section "Class Modules and Forms".

Performance Discussions

Few, if any, promotional information sources list "improved application performance" as a feature of object-oriented. This is because the trade-off of the benefits of object-oriented programming typically include reduced application performance. While this largely holds true for applications developed under the VBOF, there are some welcome exceptions to this as outlined in the following discussions.

Maximizing Performance by Eliminating the DataControl

One of the best techniques for significantly improving application performance is to eliminate the Data control and the RecordSet. In their places, the application should be designed to use VBOFListBoxWrappers and VBOFDBGridWrappers and their underlying VBOFCollections.

This action essentially allows all processing to be based on the objects which had been instantiated by either the VBOFObjectManager or an instance of VBOFCollection. After being instantiated these objects

remain in memory and are quickly located and returned to the application even though the underlying RecordSets may since have been redirected to retrieve other rows. By comparison, conventional programming techniques would require that the database be referenced each time a given row is needed, even though some of those rows had been previously retrieved.

For an example of these performance benefits, execute the VBOF Demonstration Package. Open and navigate through the example which is based on the Data control and notice the achieved performance levels. In particular, notice the performance of the application as different Person objects are clicked, while the application must gather the contained Addresses and Phone numbers for the clicked Person object. Then, try exactly the same operations using either the DBGrid- or ListBox-based examples and notice how quickly the Addresses and Phones collections are displayed. Even if the order of this experiment is reversed (run the object-oriented examples first and the Data control example last) the result is the same.

Performance Impacts of Conventional Programming Techniques Compared to VBOF

The performance gain, as discussed in the previous section, is not necessarily because of any performance issues with the RecordSet or Data control. It has more to do with the nature of conventional programming techniques versus having the ability to take advantage of the nature of object-oriented programming techniques.

For example, under conventional VB programming techniques, the application's RecordSet objects frequently follow a given cycle where the RecordSet is created, processed, then closed, or altered entirely to contain a different set of rows. In the event that a given row is retrieved more than once across several iterations of such RecordSet reassignments, it is repeatedly searched and retrieved each time from the database, optionally transmitted to the client, assembled into a RecordSet then presented to the application.

While this is actually true even under the VBOF, it is only true the first time any given row is retrieved. It is also true that additional processing occurs as VBOF instantiates objects from the returned rows and populates them the first time the RecordSet is processed. However, during subsequent operations, the VBOFObjectManager immediately locates the already existing object in memory and incurs no delays attributable to database searching, transmitting or assembling into RecordSets.

Chapter 5: Conditional Compilation Options

The following conditional compilation options are available for VB Object Framework code compilation (which can be set in the "Conditional Compilation" field of the VB Project's Options Sheet.)

Using "NoEventMgr" to Suppress Event Management

Event management is an advanced object-oriented concept that does not necessarily bring immediate benefit to all VB application development projects, particularly those which are quite small. In addition, it is typical for most first-time object-oriented programmers to fail to grasp the significance of event management. For these reasons, the VBOF Event Management support services have been designed to be easily removed or avoided, as the case may be.

The VBOFEventManager and VBOFEventObject can be safely excluded from the application project after specifying "NoEventMgr = True" in the "Conditional Compilation" field of the VB Project's Options Sheet. The VB Project can then remove the files "VBOFEMgr.cls" and "VBOFEvnt.cls".

Using "NoDebugMode" to Suppress Generation of Debugging Code

Interlaced throughout the VBOF Class Modules are instances of debug-oriented code. This can be of significant benefit while developing and testing application Class Modules, since the state of the VBOF Class Modules can be easily detailed.

While testing , the debug tracing can be enabled or disabled by setting the `DebugMode` property of the application's instance of the VBOFObjectManager to either True or False. However, note that the code necessary to determine `DebugMode` whether is in effect remains within the Class Modules and is being constantly executed. It may be advisable to remove the generation of all debug-related code -- even that which tests for `DebugMode` being in effect -- at an advanced phase of development or immediately prior to releasing the application.

To eliminate all debug-related code, thus generating for faster running code, specify "NoDebugMode = -1" in the "Conditional Compilation" field of the VB Project's Options Sheet.

Appendix A: Converting from the DataAwareCollection

This section is meaningful to those many users of the DataAwareCollection (which preceded the VB Object Framework), who wish to convert to the VB Object Framework for its additional object-oriented capabilities. While backward compatibility with the DataAwareCollection was a design objective of the VBOF, there have been so many significant changes that this objective could not be met. The author regrets any inconvenience this may have caused.

Due to the many additional features available only in the VB Object Framework, there are several changes to the Class Modules which had been developed to exploit the DataAwareCollection. These are:

1. Applications should change all "DataAwareCollection" references to "VBOFCollection";

2. Applications must instantiate an object of the VBOFObjectManager Class at the beginning of the application, using the following technique:

```
Dim MyObjectManager as New VBOFObjectManager
```

   If necessary, the VBOFObjectManager variable should be declared as `Public`. For an example of instantiating VBOFObjectManager, refer to the `Sub CreateObjectManager()` of the VB Object Framework demonstration package.

3. Except for the VBOFObjectManager object, *applications must never instantiate their own copies of VB Object Framework objects*, as described above, since VBOFObjectManager is responsible for creating all instances of all other VBOF objects. New instances of VBOFCollections should only be requested by the application in Form modules – Class Modules should wrap the VBOFCollection in an appropriately named method, such as "Persons", "Addresses", etc. (refer to Chapter 2, section "Managing Collections within Class Modules" for additional information).

   In order to create a new instance of an VBOFCollection object in a Form module, the applications should specify:

```
Dim MyCollection as VBOFCollection
. . .
Set MyCollection = _
    MyObjectManager.NewVBOFCollection
```

   rather than:

```
Dim MyCollection As New VBOFCollection
```

   For an example of instantiating VBOFCollection, refer to `Sub CreatePersonsCollection()` of the VB Object Framework demonstration package.

4. Collection-controlling methods need to be modified slightly. Collection-controlling methods are used to assist in the implementation of the object containment hierarchy. For example, refer to section "Managing Collections" in this User's Guide.

5. Orphans are no longer automatically deleted from the database when removed from a VBOFCollection.

   Under the DataAwareCollection implementation, objects which were removed from the collection via the Remove method had the corresponding containment information been simultaneously deleted from the database. If there then remained no containing objects for the

object being removed, DataAwareCollection then deemed the object to be an orphan and automatically removed it from the database.

VB Object Framework handles this differently since there is a clear separation of responsibilities between VBOFObjectManager and VBOFCollection. VBOFObjectManager oversees the instantiation and termination of all objects, while VBOFCollection oversees the contents of the collection. Thus, when the `Remove` method of VBOFCollection is invoked, the specified object is removed from the collection and the corresponding containment information is simultaneously deleted from the database. However, there is no automatic deletion of the object from the database, even if it is, in fact, an orphan.

To thoroughly delete an object, the application must invoke the VBOFObjectManager's `RemoveObject` method.

As an alternative, the original behavior of DataAwareCollection regarding the automatic deletion of orphans can be reinstated by setting the VBOFObjectManager property `AutoDeleteOrphans` to `True`, for example:

```
MyObjectManager.AutoDeleteOrphans = True
```

6. The behavior of the `WhereClause:=` parameter to the VBOFCollection differs from its implementation under DataAwareCollection.

   DataAwareCollection had used the specified `WhereClause:=` as the only component of the Where Clause of the generated SQL statement. This left to the application the burden of including within the Where Clause any parameters which would implement the object hierarchy rules.

   VBOFCollection relieves the application of any such responsibility by first generating that portion of the Where Clause which implements the object containment hierarchy, then appending any application-provided Where Clause parameters.

7. Applications no longer need to provide:

   ```
   Public Function ObjectType() As String
   ```

   in each Class Module.

8. All methods of the form:

   ```
   Public Function TableName() As String
   ```

   must be renamed to:

   ```
   Public Function ObjectDataSource() As String
   ```

9. All Class Modules must have the following General Declarations:

   ```
   Public ObjectID As Long
   Public ObjectChanged As Long
   Public ObjectAdded As Long
   Public ObjectDeleted As Long
   Public ObjectParentCount As Long
   Public ObjectManager As VBOFObjectManager
   ```

Note that VB Object Framework assumes the responsibility of controlling each of the above variables, including propagating the single instance of VBOFObjectManager across all objects in the application. Therefore, there is no need for the application to perform any maintenance upon these variables.

10. Applications no longer need to include the Class Module "DataAwareObjectLink".

11. All named parameters `ParentObject:=` have been renamed to `Parent:=`, and all named parameters `SampleObject:=` have been renamed to `Sample:=`.

Appendix B.1: Public Methods of the Class Module VBOFObjectManager

This reference section lists the Public methods of the VBOFObjectManager in alphabetical order and the parameters for each. For contextual about using these methods, refer to "Chapter 2: VBOF Services".

AutoDeleteOrphans (Property)

The `AutoDeleteOrphans` Get and Let Property methods control the AutoDeleteOrphans property of the VBOF. This property is necessary because the DataAwareCollection product, which was released prior to the VBOF, operated in a mode where "orphaned" data rows were automatically deleted from the database. Under the VBOF, this is not always desirable and the default is not to automatically remove orphans. This property is provided for backward compatibility with the DataAwareCollection.

Get AutoDeleteOrphans example:

```
MyBoolean = MyObjectManager.AutoDeleteOrphans
```

Let AutoDeleteOrphans example:

```
MyObjectManager.AutoDeleteOrphans = True
```

Database (Property)

The `Database` Get and Set Property methods control the `Database` property of the VBOFObjectManager. The `Database` property must be set to a valid VB-accessible database by the application before any database-oriented functions can be performed by the VBOF.

For usage information, refer to Chapter 2, "Starting VBOF Services".

Usage example:

The application program must set the Database property prior to receiving any database-related support functions

Get Database example:

```
Set MyDatabase = MyObjectManager.Database
```

Set Database example:

```
Set MyObjectManager.Database = MyDatabase
```

DebugMode (Property)

The `Debug` Get and Set Property methods control whether or not debug ("trace") information is generated by the VBOF objects as they perform their functions.

By default, this feature is disabled. To enable the generation of the debug code, the application must specify "NoDebugMode = 0" in the "Conditional Compilation" field of the VB Project's Options Sheet. That alone does not actually enable the debug trace – it just enables the generation of the code to actually generate the code. At run-time, the application must also set the VBOFObjectManager's DebugMode Property to True.

For more information about this feature, refer to Chapter 5, "Using "NoDebugMode" to Suppress Generation of Debugging Code"

Get DebugMode example:

```
MyBoolean = MyObjectManager.DebugMode
```

Set DebugMode example:

```
MyObjectManager.DebugMode = MyBoolean
```

Form_UnloadQuery (Method)

The `Form_UnloadQuery` method is provided as a single service point for any VBOF-supported Form which is about to close. In this method, the VBOFObjectManager closes all VBOF Wrappers instances and severs any Event Notifications currently registered to the Form or its Wrappers.

It is designed to be the only VBOF-related statement in the Form's `Form_UnloadQuery` event procedure (although other non-VBOF statements can also be there, if needed.)

For additional information about the `Form_UnloadQuery` method, refer to Chapter 3, section "Forms".

Returns:

Nothing.

Parameters:

`Form`

(Required, Type is Form, as a positional parameter, *not as a Named Parameter*) Identifies the Form which is about to be closed.

`Wrappers`

(Optional, Types are VBOF Wrappers, as positional parameters, *not as Named Parameters*) Identifies the Wrappers which are defined to the Form. There can be any number of specified Wrappers.

Form_QueryUnload example:

```
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As
Integer)
     ObjectManager.Form_QueryUnload _
        Me, _
        pvtPersonsListBoxWrapper, _
        pvtAddressesListBoxWrapper, _
              . . .
```

InitializeObject (Method)

The `InitializeObject` method is provided to allow the application to introduce into VBOF services an object of its own origination. However note that some VBOF services are not fully available to such objects because VBOF did not instantiate them from any data source.

For additional information about the `InitializeObject` method and restrictions upon objects introduced in this manner, refer to Chapter 2, section "Initializing Objects".

Returns:

A boolean, indicating whether or not the `InitializeObject` method was executed successfully.

Parameters:

`Object:=`

(Required, Variant) Identifies the object which is to be initialized for VBOF services.

InitializeObject example:

```
Dim MyObject As MyClassModule
 . . .
```

```
ObjectManager.InitializeObject _
    Object:=MyObject
```

ManageCollection (Method)

The `ManageCollection` method completely manages a VBOFCollection on behalf of the application and returns either the entire VBOFCollection or a specific object, depending on whether or not the `ObjectID:=` parameter has been specified (refer to the Visual Basic Programmer's Guide, Chapter 7, section "Using Visual Basic Collection Objects" for details about this behavior).

For additional information about the `ManageCollection` method, refer to Chapter 2, section "Populating Collections" and "Managing Collections within Class Modules".

Returns:

A populated instance of the class VBOFCollection (if the `ObjectID:=` parameter has not been specified);

or,

An instantiated object of the same class as the Class Module provided in the `Sample:=` parameter, initialized with the contents of the row retrieved from the appropriate data source, as indicated by the `ObjectID:=` parameter.

Parameters:

`Collection:=`

(Required, VBOFCollection) Identifies the VBOFCollection to be managed.

`Parent:=`

(Required (*), Variant) Identifies the containing object. The simplest technique is to specify `Parent:=Me`.

(*) If the `Parent:=` parameter had been specifically provided in the `NewVBOFCollection` method, it is not necessary to specify it in this method.

`Sample:=`

(Required, Variant) Identifies a temporary instance of the Class Module of the same type to be instantiated by VBOF.

`ANSISQL:=`

(Optional, Boolean) Indicates whether or not VBOF should use ANSI-compliant SQL when generating the SQL statement which is used by VBOF to retrieve the appropriate row from the data source. The default for this is True.

`Database:=`

(Optional, Database) Identifies the Database in which the data source is located.

If the VBOFObjectManager's `Database` property had been specified before the `NewVBOFCollection` method had been executed for this VBOFCollection, and if the VBOFCollection's `Database` property should be the same as the VBOFObjectManager's `Database` property at that time, then this parameter can be eliminated.

Also, if the `Database:=` parameter had been specifically provided in the `NewVBOFCollection` method, it is not necessary to specify it in this method.

`ObjectID:=`

(Optional, Long) Identifies the `ObjectID` property of the desired object. If provided, the `ManageCollection` method returns only the requested object. If this parameter is not provided, the entire VBOFCollection is returned.

`ODBCPassThrough:=`

(Optional, Boolean) Indicates whether or not VBOF should use the `ODBCPassThrough` option when executing the SQL statement which is used by VBOF to retrieve the appropriate row from the data source. The default for this is False.

`OrderByClause:=`

(Optional, String) Identifies an SQL Order By Clause which is to be applied to the SQL statement which is used by VBOF to retrieve the appropriate row from the data source. This can be used when the application needs to have the objects appear in a specific order.

`SQL:=`

(Optional, String) Identifies an SQL statement which is to be used by VBOF to retrieve the appropriate row from the data source. Because of VBOF's implementation, such a value is typically not needed.

`WhereClause:=`

(Optional, String) Identifies an SQL Where Clause which is to be applied to the SQL statement which is used by VBOF to retrieve the appropriate row from the data source. Because of VBOF's implementation, such a value is typically not needed.

ManageCollection example:

```
Public Function Persons(Optional ObjectID As Variant) As Variant
    Dim tempNewPerson As New Person
    Set Persons = _
        ObjectManager. _
            ManageCollection( _
                Collection:=myPersonsCollection, _
                Parent:=Me, _
                ObjectID:=ObjectID, _
                Sample:=tempNewPerson, _
                Database:=MyDatabase, _
                OrderByClause:="LastName ASC, FirstName ASC"
```

NewObject (Method)

The `NewObject` method returns an instantiated new Object which is singly contained within another object (versus being part of a collection of objects which are contained within another object). This is typical for contained objects such as an employee's manager (e.g., the Employee.Manager property), or the state object within an address object (e.g., the Address.State property), etc.

For additional information about the `NewObject` method, refer to Chapter 2, section "Implementing Contained Objects".

Returns:

An instantiated object of the same Class as the Class Module provided in the `Sample:=` parameter, initialized with the contents of the row retrieved from the appropriate data source, as indicated by the `ObjectID:=` parameter.

Parameters:

`Sample:=`

(Required, Variant) Identifies a temporary instance of the Class Module of the same type to be instantiated by VBOF.

`ObjectID:=`

(Required, Long) Identifies the `ObjectID` property of the desired object. This value is used by VBOF to retrieve the correct row from the data source.

`ANSISQL:=`

(Optional, Boolean) Indicates whether or not VBOF should use ANSI-compliant SQL when generating the SQL statement which is used by VBOF to retrieve the appropriate row from the data source. The default for this is True.

`Database:=`

(Optional, Database) Identifies the Database in which the data source is located. If some other mechanism has already been used to establish this property within the VBOFObjectManager (such as setting the VBOFObjectManager's `Database` property), this parameter can be eliminated.

`ODBCPassThrough:=`

(Optional, Boolean) Indicates whether or not VBOF should use the `ODBCPassThrough` option when executing the SQL statement which is used by VBOF to retrieve the appropriate row from the data source. The default for this is False.

`SQL:=`

(Optional, String) Identifies an SQL statement which is to be used by VBOF to retrieve the appropriate row from the data source. Because of VBOF's implementation, such a value is typically not needed.

`WhereClause:=`

(Optional, String) Identifies an SQL Where Clause which is to be applied to the SQL statement which is used by VBOF to retrieve the appropriate row from the data source. Because of VBOF's implementation, such a value is typically not needed.

NewObject example 1 (using `ObjectID:=`):

```
Public Function ObjectInitializeFromRecordSet(Optional RecordSet
As Variant) As Person

    Dim NewPerson as New Person
    . . .
' copy values from the RecordSet (not important for this example)
    . . .
' pick-up the contained Manager object
    If Not IsNull(RecordSet("ManagerObjectID")) Then
        Set pvtManager = _
            ObjectManager.NewObject( _
                Sample:=NewPerson, _
                ObjectID:=CStr(RecordSet("ManagerObjectID")))
    End If
```

NewObject example 2 (using `WhereClause:=`):

```
Public Function ObjectInitializeFromRecordSet(Optional RecordSet
As Variant) As Address

    Dim NewState as New State
    . . .
```

```
    pvtStateCode = RecordSet("StateCode")
    . . .
' pick-up the contained State object
    If Not IsNull(RecordSet("StateCode")) Then
        Set pvtState = _
            ObjectManager.NewObject( _
                Sample:=NewState, _
                WhereClause:="StateCode = '" & pvtStateCode &
"'")
    End If
```

NewVBOFCollection (Method)

The `NewVBOFCollection` method returns a new, properly instantiated VBOFCollection object.

For additional information about the `NewVBOFCollection` method, refer to Chapter 2, section "Populating Collections".

Returns:

A VBOFCollection.

Parameters:

`Parent:=`

(Optional, Variant) Identifies the object which contains the VBOFCollection. If the `Parent:=` parameter is not provided in this method, then it must be provided in the `ManageCollection` method, as it is used.

`Database:=`

(Optional, Database) Identifies the Database in which the data source is located. If some other mechanism has already been used to establish this property within the VBOFObjectManager (such as setting the VBOFObjectManager's `Database` property), this parameter can be eliminated.

NewVBOFCollection example:

```
Private pvtPersons As VBOFCollection
. . .
' Example 1:  not specifying optional parameters
Set pvtPersons = _
    ObjectManager.NewVBOFCollection

' Example 2:  specifying optional parameters
Set pvtPersons = _
    ObjectManager.NewVBOFCollection ( _
        Parent:=Me, _
        Database:=MyDatabase)
```

NewVBOFDataWrapper (Method)

The `NewVBOFDataWrapper` method returns a new, properly instantiated VBOFDataWrapper object, bound to the specified VBOFCollection and VB Data control.

For additional information about the `NewVBOFDataWrapper` method, refer to Chapter 2, section "Wrapping the Data Object" and Appendix D.

Returns:

A VBOFDataWrapper.

Parameters:

`Collection:=`

(Required, VBOFCollection) Identifies the VBOFCollection which is to be bound to the VBOFDataWrapper.

`DataControl:=`

(Required, VB Data) Identifies the VB Data object which is to be bound to the VBOFDataWrapper

NewVBOFDataWrapper example:

```
Private Sub Form_Load()
    Set pvtPersonsDataWrapper = _
        ObjectManager.NewVBOFDataWrapper( _
            Collection:=publicCompany.Persons, _
            DataControl:=Data1)
```

NewVBOFDBGridWrapper (Method)

The `NewVBOFDBGridWrapper` method returns a new, properly instantiated VBOFDBGridWrapper object, bound to the specified VBOFCollection and VB DBGrid control.

For additional information about the `NewVBOFDBGridWrapper` method, refer to Chapter 2, section "Wrapping DBGrids" and Appendix E.

Returns:

A VBOFDBGridWrapper.

Parameters:

`Collection:=`

(Required, VBOFCollection) Identifies the VBOFCollection which is to be bound to the VBOFDBGridWrapper.

`DBGrid:=`

(Required, VB Data) Identifies the VB DBGrid object which is to be bound to the VBOFDBGridWrapper

NewVBOFDBGridWrapper example:

```
Private Sub Form_Load()
    Set pvtPersonsDBGridWrapper = _
        ObjectManager.NewVBOFDBGridWrapper( _
            Collection:=pvtPersons, _
            DBGrid:=DBGrid1)
```

NewVBOFListBoxWrapper (Method)

The `NewVBOFListBoxWrapper` method returns a new, properly instantiated VBOFListBoxWrapper object, bound to the specified VBOFCollection and VB ListBox or ComboBox control.

For additional information about the `NewVBOFListBoxWrapper` method, refer to Chapter 2, section "Wrapping List Boxes and Combo Boxes" and Appendix F.

Returns:

A VBOFListBoxWrapper.

Parameters:

`Collection:=`

(Required, VBOFCollection) Identifies the VBOFCollection which is to be bound to the VBOFListBoxWrapper.

`ListBox:=`

(Required, VB Data) Identifies the VB ListBox or ComboBox object which is to be bound to the VBOFListBoxWrapper

NewVBOFListBoxWrapper example:

```
Private Sub Form_Load()
    Set pvtPersonsListBoxWrapper = _
        ObjectManager.NewVBOFListBoxWrapper( _
            Collection:=pvtPersons, _
            ListBox:=ListBox1)
```

NewVBOFRecordSetWrapper (Method)

The `NewVBOFRecordSetWrapper` method returns a new, properly instantiated VBOFRecordSetWrapper object, bound to the specified.

For additional information about the `NewVBOFRecordSetWrapper` method, refer to Chapter 2, section "Wrapping the RecordSet Object" and Appendix G.

Returns:

A VBOFRecordSetWrapper.

Parameters:

`Collection:=`

(Required, VBOFCollection) Identifies the VBOFCollection which is to be bound to the VBOFRecordSetWrapper.

NewVBOFRecordSetWrapper example:

```
Private Sub Form_Load()
    Set pvtPersonsRecordSetWrapper = _
        ObjectManager.NewVBOFRecordSetWrapper( _
            Collection:=publicCompany.Persons)
```

RegisterForCollectionEvent (Method)

The `RegisterForCollectionEvent` method registers the specified RegisterObject as a target of notification when the specified VBOFCollection triggers an event. In addition, the application can minimize the scope of the notifications to only the event named in `TriggerEvent:=` parameter.

For additional information about the `RegisterForCollectionEvent` method, refer to Chapter 4, section "Using Events in Conjunction with OO Programming".

Returns:

A Boolean, indicating whether or not the registration occurred correctly.

Parameters:

`Collection:=`

(Required, VBOFCollection) Identifies the VBOFCollection whose triggered events are to be monitored.

`RegisterObject:=`

(Required, Variant) Identifies the object which is to receive notification whenever a qualifying event is triggered by the VBOFCollection specified in the `Collection:=` parameter.

`TriggerEvent:=`

(Optional, String) Identifies the specific event, as triggered by the VBOFCollection, in which the RegisterObject has an interest.

RegisterForCollectionEvent example:

```
Private Sub Form_Load()
    ObjectManager.RegisterForCollectionEvent( _
        Collection:=pvtPersons, _
        RegisterObject:=Me)
```

RegisterForObjectEvent (Method)

The `RegisterForObjectEvent` method registers the specified RegisterObject as a target of notification when an event is triggered by the specified TriggerObject (if specified), or any object of a given TriggerObjectType (if specified). In addition, the application can minimize the scope of the notifications to only the event named in `TriggerEvent:=` parameter.

For additional information about the `RegisterForObjectEvent` method, refer to Chapter 4, section "Using Events in Conjunction with OO Programming".

Returns:

A Boolean, indicating whether or not the registration occurred correctly.

Parameters:

`RegisterObject:=`

(Required, Variant) Identifies the object which is to receive notification whenever a qualifying event is triggered by the TriggerObject, TriggerObjectType, as specified.

`TriggerObject:=`

(Optional, Variant) Identifies the object whose triggered events are to be monitored.

`TriggerObjectType:=`

(Optional, String) Identifies the name of the Class Module whose object instances are to be monitored for triggered events.

`TriggerEvent:=`

(Optional, String) Identifies the specific event, as triggered by the TriggerObject or TriggerObjectType, in which the RegisterObject has an interest.

RegisterForObjectEvent example:

```
Private Sub Form_Load()
    ObjectManager.RegisterForObjectEvent( _
        Collection:=pvtPersons, _
        RegisterObject:=Me, _
        TriggerObjectType:="Person", _
        TriggerEvent:="Changed)
```

RemoveCollection (Method)

The `RemoveCollection` method empties and removes the specified VBOFCollection object and all of its contained objects. By default, any VBOF-maintained object containment links are severed by this operation between the contained object and the containing object in which the VBOFCollection is defined. Refer to the `NoDelete:=` parameter to control this behavior.

Returns:
A Boolean, indicating whether or not the RemoveCollection function was executed successfully.

Parameters:
`Collection:=`
(Required, VBOFCollection) Identifies the VBOFCollection to be emptied

`NoDelete:=`
(Optional, Boolean) Specifies whether or not the whether or not the VBOF-maintained object containment links are to be severed by this operation between the contained object and the containing object in which the VBOFCollection is defined.

RemoveCollection example:

```
MyObjectManager.RemoveCollection _
    pvtPersons
```

TerminateForm (Method)

The `TerminateForm` method performs the same service as the `Form_QueryUnload` method. The only difference is the polymorphic benefit this name has when associated with the `TerminateObject` method.

Refer to the description of the `Form_QueryUnload` method for details regarding this method.

TerminateObject (Method)

The `TerminateObject` method provides a mechanism for removing objects from VBOF services.

Returns:
A Boolean, indicating whether or not the object termination occurred correctly.

Parameters:
`Object:=`
(Required, Variant) Identifies the object which is to be terminated.

TerminateObject example:

```
Private Sub Form_Load()
    ObjectManager.TerminateObject( _
        Object:=pvtPerson)
```

TriggerObjectEvent (Method)

The `TriggerObjectEvent` method causes the specified triggered event to be broadcast across the VBOF service are to all registered recipient objects.

For additional information about the `TriggerObjectEvent` method, refer to Chapter 4, section "Using Events in Conjunction with OO Programming".

Returns:
A Boolean, indicating whether or not the registration occurred correctly.

Parameters:
`Event:=`
(Required, String) Identifies the event which is being triggered.

`Object:=`
(Optional, Variant) Identifies the object which is triggering the event.

TriggerObjectEvent example:

```
Public Function ObjectHasChanged()
' Mark this object as "Changed" and trigger the
'   "Changed" event

    ObjectChanged = True

    If Not ObjectManager Is Nothing Then
        ObjectManager.TriggerObjectEvent _
            Event:="Changed", _
            Object:=Me
    End If
End Function
```

Verbose (Property)

The `Verbose` property control whether or not certain warning messages are displayed in MessageBoxes or are ignored.

Get Verbose example:

```
MyBoolean = MyObjectManager.Verbose
```

Let Verbose example:

```
MyObjectManager.Verbose = True
```

Workspace (Property)

The `Workspace` Get and Set Property methods control the Workspace property of the VBOFObjectManager. The Workspace property must be set to a valid VB-accessible Workspace by the application before any database-oriented functions can be performed by the VBOF.

For usage information, refer to Chapter 2, “Starting VBOF Services”.

Get Database example:

```
Set MyWorkspace = MyObjectManager.Workspace
```

Set Database example:

```
Set MyObjectManager.Workspace = Workspaces(0)
```

Appendix B.2: Events Triggered by the Class Module VBOFObjectManager

Instantiated

The “Instantiated” event is triggered when the VBOFObjectManager determines that a given object is, in fact, unique across the environment, and has allowed the instantiation to proceed.

The “Instantiated” event is delivered only to the instantiated object.

Appendix C.1: Public Methods of the Class Module VBOFCollection

This section outlines the public methods of the VBOFCollection and the parameters for each.

Add (Method)

The `Add` method allows the application to add an object to the VBOFCollection. VBOF automatically stores object containment information in its private data store. If the VBOFObjectManager finds the object to be unique across the application, the object is automatically inserted into its data store.

Returns:

A Variant, which is the object which was actually added to the VBOFCollection.

> Note: The returned object is not necessarily the same object which is provided by the application in the `Item:=` parameter. This is because the VBOFObjectManager may have found that the provided object is actually a duplicate of another instance of the same object. Under this condition, the original object would have been added to the VBOFCollection and the application-provided object would have been discarded. It is important that the application receive the returned object and begin using that object from that point forward. There is no guarantee that the originally provided object continues to exist after having invoked this method.

Parameters:

`Item:=`

(Required, Variant) Identifies the object to be added to the VBOFCollection.

`Key:=`

(Optional, Long) Identifies the key of the object to be added to the VBOFCollection. If not provided, VBOF uses the object's `ObjectID` property.

Add example:

```
Set MyObject = _
    MyVBOFCollection.Add _
        (Item:=MyObject)
```

AutoDeleteOrphans (Property)

The `AutoDeleteOrphans` Get and Let Property methods control the `AutoDeleteOrphans` property of the VBOFCollection. This property is necessary because the DataAwareCollection product, which was released prior to the VBOF, operated in a mode where "orphaned" data rows were automatically deleted from the database. Under the VBOF, this is not always desirable and the default is not to automatically remove orphans. This property is provided for backward compatibility with the DataAwareCollection.

Note: The VBOFObjectManager also has the `AutoDeleteOrphans` Property. The VBOFCollection's `AutoDeleteOrphans` Property is initialized to value of the VBOFObjectManager's `AutoDeleteOrphans` Property at the time that the VBOFCollection is instantiated (through the VBOFObjectManager's `NewVBOFCollection` method).

Get AutoDeleteOrphans example:

```
MyBoolean = MyVBOFCollection.AutoDeleteOrphans
```

Let AutoDeleteOrphans example:

```
MyVBOFCollection.AutoDeleteOrphans = True
```

Collection (Method)

The `Collection` method returns VB Collection object used by VBOFCollection to actually contain the objects.

Returns:
A VB Collection.

Parameters:
None.

Collection example:

```
Dim MyCollection As Collection
. . .
Set MyCollection = _
    MyVBOFCollection.Collection
```

CollectionIndex (Method)

The `CollectionIndex` method returns the index of the specified object within the VBOFCollection.

Returns:
A Long, which is index of the specified object within the VBOFCollection.

Parameters:

`Item:=`
(Optional, Variant) Identifies the object whose the index within the VBOFCollection is to be returned.

`Key:=`
(Optional, Variant) Identifies the Collection Key value of the object whose the index within the VBOFCollection is to be returned.

`WhereClause:=`
(Optional, String) Identifies an SQL Where Clause whose resolution identifies the object whose the index within the VBOFCollection is to be returned.

`FindFirst:=`
(Optional, Boolean) Indicates whether or not VBOF should execute the SQL Where Clause identified in the `WhereClause:=` after having executed the `FindFirst` method over the underlying RecordSet.

Note: The `WhereClause:=` parameter is also required for this parameter to be valid.

`FindLast:=`
(Optional, Boolean) Indicates whether or not VBOF should execute the SQL Where Clause identified in the `WhereClause:=` after having executed the `FindLast` method over the underlying RecordSet.

Note: The `WhereClause:=` parameter is also required for this parameter to be valid.

`FindNext:=`
(Optional, Boolean) Indicates whether VBOF should apply the SQL Where Clause identified in the `WhereClause:=` to the RecordSet's `FindNext` method.

Note: The `WhereClause:=` parameter is also required for this parameter to be valid.

`FindPrevious:=`

(Optional, Boolean) Indicates whether VBOF should apply the SQL Where Clause identified in the `WhereClause:=` to the RecordSet's `FindPrevious` method.

Note: The `WhereClause:=` parameter is also required for this parameter to be valid.

CollectionIndex examples:

```
Dim MyCollection as VBOFCollection

MyIndex = _
    MyCollection.CollectionIndex _
        (Item:=MyObject)

MyIndex = _
    MyCollection.CollectionIndex _
        (Key:=MyKey)

MyIndex = _
    MyCollection.CollectionIndex _
        (WhereClause:="LastName = 'Jones'")

MyIndex = _
    MyCollection.CollectionIndex _
        (WhereClause:="LastName = 'Jones'", _
        FindFirst:=True)
```

Count (Method)

The `Count` method returns number of objects currently contained in the VBOFCollection.

Returns:

A Long.

Parameters:

None.

Count example:

```
Dim MyLong As Long
. . .
MyLong = MyVBOFCollection.Count
```

Database (Property)

The `Database` Get and Set Property methods control the `Database` property of the VBOFCollection. The `Database` property must be set to a valid VB-accessible database by the application before any database-oriented functions can be performed by the VBOFCollection.

Note: The VBOFObjectManager also has the `Database` Property. The VBOFCollection's `Database` Property is initialized to value of the VBOFObjectManager's `Database` Property at the time that the VBOFCollection is instantiated (through the VBOFObjectManager's `NewVBOFCollection` method).

Get Database example:

```
Set MyDatabase = MyVBOFCollection.Database
```

Set Database example:

```
Set MyVBOFCollection.Database = MyDatabase
```

MostRecentlyAddedObject (Property)

The `MostRecentlyAddedObject` Property method allows the application to retrieve the object most recently added to the VBOFCollection.

Returns:
A Variant

Parameters:
None.

MostRecentlyAddedObject example:

```
Set MyObject= _
    MyVBOFCollection.MostRecentlyAddedObject
```

MostRecentlyAddedObjectIndex (Property)

The `MostRecentlyAddedObjectIndex` Property method allows the application to retrieve the index within the VBOFCollection of the object most recently added to the VBOFCollection.

Returns:
A Long.

Parameters:
None.

MostRecentlyAddedObjectIndex example:

```
Set MyLong = _
    MyVBOFCollection.MostRecentlyAddedObjectIndex
```

OrderByClause (Property)

The `OrderByClause` Get and Set Property methods control the `OrderByClause` property of the VBOFCollection. Through this method, the application can set the desired SQL Order By clause to be applied anytime the VBOFCollection retrieves data from the data source. By default, VBOF does not apply an Order By clause to the SQL statements in internally generates.

Note: The `PopulateCollection` and `ManageCollection` methods have the `OrderByClause` parameter, which can be used, as well as this technique of setting the `OrderByClause` property.

Get OrderByClause example:

```
MyString = MyVBOFCollection.OrderByClause
```

Set OrderByClause example:

```
MyVBOFCollection.OrderByClause = MyString
```

Parent (Property)

The `Parent` Get and Set Property methods control the `Parent` property of the VBOFCollection. Through this method, the application can set the appropriate `Parent` object for the VBOFCollection.

Note: The `PopulateCollection` and `ManageCollection` methods have the `Parent` parameter, which can be used, as well as this technique of setting the `Parent` property.

Get Parent example:

```
Set MyObject = MyVBOFCollection.Parent
```

Set Parent example:

```
Set MyVBOFCollection.Parent = MyObject
```

PopulateCollection (Method)

The `PopulateCollection` method returns an VBOFCollection which has been populated with the appropriate instantiated objects.

Note: It is recommended that either the `Database:=` or `RecordSet:=` parameter be specifically provided. *If neither is provided, any previously established* `Database:=` *value is automatically substituted and the VBOFCollection proceeds as if the* `Database:=` *parameter had been provided*.

For additional information about the `PopulateCollection` method, refer to Chapter 2, section "Populating Collections".

Returns:

A VBOFCollection

Parameters:

`Sample:=`

(Required, Variant) Identifies a temporary instance of the Class Module of the same type to be instantiated by VBOFCollection and placed into the collection.

`Parent:=`

(Required (*), Variant) Identifies the containing object. The simplest technique is to specify `Parent:=Me`.

(*) If the `Parent:=` parameter had been specifically provided in the `NewVBOFCollection` method, it is not necessary to specify it in this method.

`ANSISQL:=`

(Optional, Boolean) Indicates whether or not VBOF should use ANSI-compliant SQL when generating the SQL statement which is used by VBOF to retrieve the appropriate row from the data source. The default for this is True.

`Database:=`

(Optional, Database) Identifies the Database in which the data source is located.

If the VBOFObjectManager's `Database` property had been specified before the `NewVBOFCollection` method had been executed for this VBOFCollection, and if the VBOFCollection's `Database` property should be the same as the VBOFObjectManager's `Database` property at that time, then this parameter can be eliminated.

Also, if the `Database:=` parameter had been specifically provided in the `NewVBOFCollection` method, it is not necessary to specify it in this method.

`ODBCPassThrough:=`

(Optional, Boolean) Indicates whether or not VBOF should use the `ODBCPassThrough` option when executing the SQL statement which is used by VBOF to retrieve the appropriate row from the data source. The default for this is False.

`OrderByClause:=`

(Optional, String) Identifies an SQL Order By Clause which is to be applied to the SQL statement which is used by VBOF to retrieve the appropriate row from the data source. This can be used when the application needs to have the objects appear in a specific order.

`RecordSet:=`

(Optional, RecordSet) Identifies the application-provided RecordSet object already containing the data rows which are to be converted to objects and placed into the VBOFCollection.

`SQL:=`

(Optional, String) Identifies an SQL statement which is to be used by VBOF to retrieve the appropriate row from the data source. Because of VBOF's implementation, such a value is typically not needed.

`WhereClause:=`

(Optional, String) Identifies an SQL Where Clause which is to be applied to the SQL statement which is used by VBOF to retrieve the appropriate row from the data source. Because of VBOF's implementation, such a value is typically not needed.

PopulateCollection example:

```
Dim pubPersons as VBOFCollection
Dim tempNewPerson as Person
. . .
Set pubPersons = _
    ObjectManager.NewVBOFCollection
. . .
pubPersons.PopulateCollection _
    Sample:=tempNewPerson, _
    Parent:=MyCompany
```

RecordSet (Method)

The `RecordSet` method returns the underlying RecordSet object in use by the VBOFCollection.

Returns:

A RecordSet object

Parameters:

None.

RecordSet example (setting a local RecordSet object):

```
Dim MyRecordSet As RecordSet
. . .
Set MyRecordSet = _
    MyVBOFCollection.RecordSet
```

RecordSet example (setting a DataControl object's RecordSet property):

```
Set MyDataControl.RecordSet = _
    MyVBOFCollection.RecordSet
```

Refresh (Method)

The `Refresh` method causes the VBOFCollection to refresh its entire content of contained objects. The VBOFCollection begins its `Refresh` processing based on the state of its properties at that time, such as it `Database`, `Parent`, `Sample`, `OrderByClause` and `WhereClause`

Returns:

A VBOFCollection

Parameters:
None.

Refresh example:
```
Dim pvtPersons As VBOFCollection
. . .
pvtPersons.Refresh
```

Remove (Method)

The `Remove` method causes the VBOFCollection to remove the object specified by the `Item:=` or `Key:=` parameter.

It also severs the VBOF-maintained object containment information it internally maintains. If this action results in the object becoming an orphan (i.e., there are no more know parent objects of the object) then the VBOFCollection refers to its `AutoDeleteOrphans` property for direction as to whether or not the object's associated data row should be deleted from its data source.

Returns:
A VBOFCollection after having removed the specified `Item:=` object.

Parameters:
`Item:=`
(Optional, Variant) Identifies the object to be removed from the VBOFCollection.

Either the `Item:=` or `Key:=` must be provided.

`Key:=`
(Optional, Long) Identifies the key of the object to be removed from the VBOFCollection.

Either the `Item:=` or `Key:=` must be provided.

`NoDelete:=`
(Optional, Boolean) Controls whether or not the object's associated data row should be deleted from its data source is the object is found to be an orphan. If this parameter is not provided and the object is found to have become an orphan, the VBOFCollection takes action based on its `AutoDeleteOrphans` property.

Remove example:
```
Dim pvtPersons As VBOFCollection
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    pvtPersons(1)
. . .
pvtPersons.Remove _
    Item:=pvtPerson
```

Replace (Method)

The `Replace` method causes the VBOFCollection to replace the object specified by the `Item:=` parameter with the object specified by the `ReplaceWith:=` parameter.

The associated data row in the underlying data source is also replaced, as such.

Returns:

A VBOFCollection after having replaced the specified `Item:=` object with the specified `ReplaceWith:=` object.

Parameters:

`Item:=`

(Required, Variant) Identifies the object to be replaced within the VBOFCollection.

`ReplaceWith:=`

(Required, Variant) Identifies the object to replace the object specified by the `Item:=` parameter within the VBOFCollection.

Replace example (updating an object in-place):

```
Dim pvtPersons As VBOFCollection
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    pvtPersons(1)
. . .
pvtPerson.LastName = "Jones"
. . .
pvtPersons.Replace _
    Item:=pvtPerson, _
    ReplaceWith:=pvtPerson
```

Replace example (replacing with a different object):

```
Dim pvtPersons As VBOFCollection
Dim pvtPerson as Person
Dim pvtDifferentPerson as Person
. . .
Set pvtPerson = _
    pvtPersons(1)
. . .
pvtDifferentPerson.LastName = "Jones"
pvtDifferentPerson.FirstName = "Bob"
. . .
pvtPersons.Replace _
    Item:=pvtPerson, _
    ReplaceWith:=pvtDifferentPerson
```

WhereClause (Property)

The `WhereClause` Get and Set Property methods control the `WhereClause` property of the VBOFCollection. Through this method, the application can set the desired SQL Where clause addendum to be applied anytime the VBOFCollection retrieves data from the data source. By default, VBOF does not apply an addendum to the Where clause to the SQL statements in internally generates.

Note: The `PopulateCollection` and `ManageCollection` methods have the `WhereClause` parameter, which can be used, as well as this technique of setting the `WhereClause` property.

Get WhereClause example:

```
MyString = MyVBOFCollection.WhereClause
```

Set WhereClause example:

```
MyVBOFCollection.WhereClause = "LastName Like 'Jone*'"
```

Appendix C.2: Events Triggered by the Class Module VBOFCollection

AddedItem (Collection Event)

The "Added" Collection event is publicly triggered by the VBOFCollection `Add` method when the VBOFCollection has successfully added an object to the collection

PopulatedFromDatabase (Collection Event)

The "PopulatedFromDatabase" Collection event is publicly triggered by the VBOFCollection `PopulateCollection` method when the VBOFCollection has been successfully populated from the database.

PopulatedFromRecordSet (Collection Event)

The "PopulatedFromRecordSet" Collection event is publicly triggered by the VBOFCollection `PopulateCollection` method when the VBOFCollection has been successfully populated from the user-specified RecordSet object.

Refreshed (Collection Event)

The "Refreshed" Collection event is publicly triggered by the VBOFCollection `Refresh` method when the VBOFCollection has been successfully refreshed.

RemovedItem (Collection Event)

The "RemovedItem" Collection event is publicly triggered by the VBOFCollection `Remove` method when an object has been successfully removed from the VBOFCollection.

RemovedItem (Object Event)

The "RemovedItem" Object event is publicly triggered by the VBOFCollection `Remove` method when an object has been successfully removed from the underlying data source.

ReplacedItem (Collection Event)

The "ReplacedItem" Object event is publicly triggered by the VBOFCollection `Replace` method when an object has been successfully replaced with another in the VBOFCollection.

Appendix D: Public Methods of the Class Module VBOFDataWrapper

AbsolutePosition (Property)

The `AbsolutePosition` Property Get and Set methods control the `AbsolutePosition` property of the underlying RecordSet object within the VBOFCollection to which the VBOFDataWrapper is bound.

Get AbsolutePosition example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim MyLong As Long
. . .
MyLong = MyDataWrapper.AbsolutePosition
```

Set AbsolutePosition example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim MyLong As Long
. . .
MyDataWrapper.AbsolutePosition = MyLong
```

Refer to the `AbsolutePositionObject` method for a more object-oriented technique than the numeric-oriented `AbsolutePosition` method.

AbsolutePositionObject (Property)

The `AbsolutePositionObject` Property Get and Set methods interface with the `AbsolutePosition` property of the underlying RecordSet object within the VBOFCollection to which the VBOFDataWrapper is bound according to the desired object.

Get AbsolutePositionObject example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim MyPerson As Person
. . .
Set MyPerson = MyDataWrapper.AbsolutePositionObject
```

Set AbsolutePositionObject example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim MyPerson As Person
. . .
Set MyDataWrapper.AbsolutePositionObject = MyPerson
```

BOF (Method)

The `BOF` method returns the `BOF` property of the underlying RecordSet object within the VBOFCollection to which the VBOFDataWrapper is bound.

Returns:

A Boolean.

Parameters:

None

BOF example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim MyBoolean as Boolean
```

```
. . .
MyBoolean = MyDataWrapper.BOF
```

Clone (Method)

The `Clone` method returns a RecordSet which has been cloned from the underlying RecordSet object within the VBOFCollection to which the VBOFDataWrapper is bound.

Returns:
A RecordSet object.

Parameters:
None

Clone example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim MyRecordSet as RecordSet
. . .
Set MyRecordSet = MyDataWrapper.Clone
```

CloseRecordSet (Method)

The `CloseRecordSet` method closes the underlying RecordSet object within the VBOFCollection to which the VBOFDataWrapper is bound.

Returns:
A Long, containing the value of the VB Err object reflecting the state of the `Close` method of the underlying RecordSet object.

Parameters:
None

CloseRecordSet example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim MyLong as Long
. . .
MyLong = MyDataWrapper.CloseRecordSet
```

EOF (Method)

The `EOF` method returns the `EOF` property of the underlying RecordSet object within the VBOFCollection to which the VBOFDataWrapper is bound.

Returns:
A Boolean.

Parameters:
None

EOF example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim MyBoolean as Boolean
. . .
MyBoolean = MyDataWrapper.EOF
```

FindFirst (Method)

The `FindFirst` method executes a search forward across the underlying RecordSet object within the VBOFCollection to which the VBOFDataWrapper is bound, beginning at the start of the RecordSet, using the search criteria specified in the `SearchCriteria:=` parameter.

Returns:

A Variant, which is the object which is the equivalent of the first data row retrieved which qualifies for the search criteria specified in the `SearchCriteria:=` parameter, or `Nothing`, if a suitable object for the `SearchCriteria:=` parameter is not found.

Parameters:

`SearchCriteria:=`

(Required, String) Identifies the search criteria to be used to locate the first qualifying row.

FindFirst example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyDataWrapper.FindFirst
        (SearchCriteria:="LastName = 'Jones'")
```

FindLast (Method)

The `FindLast` method executes a search backward across the underlying RecordSet object within the VBOFCollection to which the VBOFDataWrapper is bound, beginning at the end of the RecordSet, using the search criteria specified in the `SearchCriteria:=` parameter.

Returns:

A Variant, which is the object which is the equivalent of the first data row retrieved which qualifies for the search criteria specified in the `SearchCriteria:=` parameter, or `Nothing`, if a suitable object for the `SearchCriteria:=` parameter is not found.

Parameters:

`SearchCriteria:=`

(Required, String) Identifies the search criteria to be used to locate the first qualifying row.

FindLast example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyDataWrapper.FindLast
        (SearchCriteria:="LastName = 'Jones'")
```

FindNext (Method)

The `FindFirst` method executes a search forward across the underlying RecordSet object within the VBOFCollection to which the VBOFDataWrapper is bound, beginning at the current position of the RecordSet, using the search criteria specified in the `SearchCriteria:=` parameter.

Returns:

A Variant, which is the object which is the equivalent of the next data row retrieved which qualifies for the search criteria specified in the `SearchCriteria:=` parameter, or `Nothing`, if a suitable object for the `SearchCriteria:=` parameter is not found.

Parameters:

`SearchCriteria:=`

(Required, String) Identifies the search criteria to be used to locate the first qualifying row.

FindNext example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyDataWrapper.FindNext
        (SearchCriteria:=”LastName = 'Jones'”)
```

FindPrevious (Method)

The `FindLast` method executes a search backward across the underlying RecordSet object within the VBOFCollection to which the VBOFDataWrapper is bound, beginning at the current position of the RecordSet, using the search criteria specified in the `SearchCriteria:=` parameter.

Returns:

A Variant, which is the object which is the equivalent of the next data row retrieved which qualifies for the search criteria specified in the `SearchCriteria:=` parameter, or `Nothing`, if a suitable object for the `SearchCriteria:=` parameter is not found.

Parameters:

`SearchCriteria:=`

(Required, String) Identifies the search criteria to be used to locate the first qualifying row.

FindLast example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyDataWrapper.FindPrevious
        (SearchCriteria:=”LastName = 'Jones'”)
```

MoveFirst (Method)

The `MoveFirst` method positions the underlying RecordSet object within the VBOFCollection to which the VBOFDataWrapper is bound, to the beginning of the RecordSet.

Returns:

A Variant, which is the object which is the equivalent of the first data row in the underlying RecordSet object within the VBOFCollection, or `Nothing`, if no objects are found in the VBOFCollection.

Parameters:

None

MoveFirst example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyDataWrapper.MoveFirst
```

MoveLast (Method)

The `MoveLast` method positions the underlying RecordSet object within the VBOFCollection to which the VBOFDataWrapper is bound, to the last row of the RecordSet beyond the current.

Returns:

A Variant, which is the object which is the equivalent of the last data row in the underlying RecordSet object within the VBOFCollection, or `Nothing`, if no objects are found in the VBOFCollection.

Parameters:

None

MoveFirst example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyDataWrapper.MoveLast
```

MoveNext (Method)

The `MoveNext` method positions the underlying RecordSet object within the VBOFCollection to which the VBOFDataWrapper is bound, to the next row of the RecordSet beyond the current.

Returns:

A Variant, which is the object which is the equivalent of the next data row in the underlying RecordSet object within the VBOFCollection, or `Nothing`, if no more objects are found in the VBOFCollection.

Parameters:

None

MoveNext example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyDataWrapper.MoveNext
```

MoveToObject (Method)

The `MoveToObject` method positions the underlying RecordSet object of the VBOFCollection to which the VBOFDataWrapper is bound, to the row which is the equivalent of the object specified in the `Object:=` parameter.

Returns:

A Variant, which is the object specified in the `Object:=` parameter or `Nothing`, if no suitable object for the specified `Object:=` parameter is not found.

Parameters:

`Object:=`

(Required, Variant) Identifies the object to be used to position the underlying RecordSet object.

FindNext example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim pvtPerson as Person
```

```
. . .
Set pvtPerson = _
    MyDataWrapper.MoveToObject
        (Object:=pvtPerson)
```

MoveToRecordNumber (Method)

The `MoveToRecordNumber` method positions the underlying RecordSet object of the VBOFCollection to which the VBOFDataWrapper is bound, to the row whose RecordNumber is specified in the `RecordNumber:=` parameter.

Returns:

A Variant, which is the object equating to the specified `RecordNumber:=` parameter or `Nothing`, if no suitable object for the specified `RecordNumber:=` parameter is not found.

Parameters:

`RecordNumber:=`

(Required, Variant) Identifies the RecordNumber to be used to position the underlying RecordSet object.

FindNext example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim MyLong as Long
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyDataWrapper.MoveToRecordNumber
        (RecordNumber:=MyLong)
```

Rebind (Method)

The `Rebind` method must be invoked if either the bound VBOFCollection or DataControl are significantly altered.

Returns:

A Boolean indicating whether or not the `Rebind` was successful.

Parameters:

`Collection:=`

(Optional, VBOFCollection) The VBOFCollection object which is to be bound to the VBOFDataWrapper.

`DataControl:=`

(Optional, DataControl) The DataControl object which is to be bound to the VBOFDataWrapper.

Rebind example:

```
MyDataWrapper.Rebind _
    Collection:=pvtPersons, _
    DataControl:=Data1
```

RecordCount (Method)

The `RecordCount` method returns the number of rows which are contained in the underlying RecordSet object within the VBOFCollection to which the VBOFDataWrapper is bound.

Returns:

A Long.

Parameters:

None

RecordCount example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim MyLong as Long
. . .
MyLong = MyDataWrapper.RecordCount
```

RecordSet (Method)

The `RecordSet` method returns the underlying RecordSet object in use by the VBOFCollection.

Returns:

A RecordSet object.

Parameters:

None.

RecordSet example:

```
Dim MyDataWrapper as VBOFDataWrapper
Dim MyRecordSet as RecordSet
. . .
Set MyRecordSet = _
    MyDataWrapper.RecordSet
```

Refresh (Method)

The `Refresh` method can be invoked if the application needs to refresh the DataControl display, perhaps after the bound VBOFCollection as been significantly changed.

Returns:

A RecordSet object which is the RecordSet object contained within the bound VBOFCollection, reflecting any changes which may have occurred due to the `Refresh` method

Parameters:

`DisplayOnly:=`

(Optional, Boolean) Indicates whether or not the application requests that only the DataControl is to be refreshed. The default action is False, to refresh the both underlying VBOFCollection and the DataControl display.

Refresh example:

```
MyDataWrapper.Refresh
```

Unbind (Method)

The `Unbind` method releases the underlying VBOFCollection and DataControl from the VBOFDataWrapper. This is necessary because Visual Basic does not release the memory consumed by any given object until all references to the object have been severed. The `Unbind` method releases the reference to the VBOFCollection and DataControl once established by the VBOFDBGridWrapper.

Note: This method is not typically executed by the application. Rather, the application Form modules should invoke the VBOFObjectManager's `Form_QueryUnload` event procedure. The `Form_QueryUnload` method is a more thorough and encapsulated implementation than if the application were to attempt to perform a clean-up process.

Unbind example (indirectly):

```
Private Sub Form_QueryUnload(. . .)

    MyObjectManager.Form_QueryUnload ( _
        Me, _
        MyDataWrapper
```

Appendix E: Public Methods of the Class Module VBOFDBGridWrapper

This section outlines the public methods of the VBOFDBGridWrapper and the parameters for each.

Bookmark (Property)

The `Bookmark` Property Get and Set methods control the Bookmark attribute of the bound DBGrid. The DBGrid's Bookmark value is roughly the character representation of the numeric index of the row, although adding and removing rows after the initial object population can effect this value. The Bookmark value can be Get or Set at any time after the VBOFDBGridWrapper has been instantiated and bound (see the VBOFObjectManager's method `NewVBOFDBGridWrapper`).

In the DBGrid_RowColChange event procedure, the Bookmark can be retrieved so the application program can keep track of the Bookmark of the selected object in the DBGrid. Elsewhere, the Bookmark can be programmatically set to a specific variant value.

Get Bookmark example:

```
Dim MyDBGridWrapper as VBOFDBGridWrapper
Dim MyBookMark As Variant
. . .
MyBookMark = _
    MyDBGridWrapper.Bookmark
```

Set Bookmark example:

```
Dim MyDBGridWrapper as VBOFDBGridWrapper
Dim MyBookMark As Variant
. . .
MyDBGridWrapper.Bookmark = _
    MyBookMark
```

Note: For a more object-oriented technique, refer to the `BookmarkObject` methods.

BookmarkObject (Property)

The `BookmarkObject` Property Get and Set methods control the Bookmark attribute of the bound DBGrid in an object-oriented manner by referring to an object, versus a Bookmark variant. The BookmarkObject value can be Get or Set at any time after the VBOFDBGridWrapper has been instantiated and bound (see the VBOFObjectManager's method `NewVBOFDBGridWrapper`).

In the DBGrid_RowColChange event procedure, the BookmarkObject can be retrieved so the application program can keep track of the selected object in the DBGrid. Elsewhere, the BookmarkObject can be programmatically set to a specific object.

Get BookmarkObject example:

```
Dim MyObject As Person ' or Variant, or other Class type
. . .
Set MyObject = _
    MyDBGridWrapper.BookmarkObject
```

Set BookmarkObject example:

```
Dim MyObject As Person ' or Variant, or other Class type
. . .
Set MyDBGridWrapper.BookmarkObject = _
    MyObject
```

Rebind (Method)

The `Rebind` method allows either the underlying VBOFCollection or DBGrid to be significantly altered, then rebound to continue processing with the new components.

If the application needs to significantly change either the underlying VBOFCollection or DBGrid after the VBOFDBGridWrapper has been instantiated by the VBOFObjectManager, the Rebind method must be executed for continued processing. Examples of significant changes are if the VBOFDBGridWrapper is currently bound to the VBOFCollection of "Joe's" Addresses (instances of Address objects), then the VBOFCollection is reprocessed to be a VBOFCollection of "Bill's" Addresses (instances of Address objects) or if the VBOFCollection of "Joe's" Addresses is reprocessed as a VBOFCollection of "Joe's" Phone objects.

Parameters:

`Collection:=`

(Optional, VBOFCollection) Identifies the VBOFCollection which has been significantly altered, or which replaces the previously used VBOFCollection.

`DBGrid:=`

(Optional, DBGrid) Identifies the DBGrid which has been significantly altered, or which replaces the previously used DBGrid.

Rebind example.

The following example shows the VBOFCollection instance MyAddresses being significantly altered. Assume that AddressesDBGridWrapper had already been bound to the collection of Addresses of some other Person object:

```
Private Sub PersonsDBGrid_RowColChange(. . .)
    Dim MyPerson As Person
    Dim MyAddresses As VBOFCollection
     . . .
    Set MyPerson = _
        PersonsDBGridWrapper.BookmarkObject
    Set MyAddresses = _
        MyPerson.Addresses
    AddressesDBGridWrapper.Rebind _
        Collection:=MyAddresses
```

Refresh (Method)

The `Refresh` method causes the displayed contents of the underlying DBGrid to be refreshed.

Note: There is no need to execute the `Refresh` method after having executed the `Rebind` method because VBOFDBGridWrapper automatically refreshes the DBGrid during the `Rebind` operation.

Refresh example:

```
MyDBGridWrapper.Refresh
```

Unbind (Method)

The `Unbind` method releases the underlying VBOFCollection and DBGrid from the VBOFDBGridWrapper. This is necessary because Visual Basic does not release the memory consumed by any given object until all references to the object have been severed. The `Unbind` method releases the reference to the VBOFCollection and DBGrid once established by the VBOFDBGridWrapper.

Note: This method is not typically executed by the application. Rather, the application Form modules should invoke the VBOFObjectManager's `Form_QueryUnload` event procedure. The

`Form_QueryUnload` method is a more thorough and encapsulated implementation than if the application were to attempt to perform a clean-up process.

Unbind example (indirectly):

```
Private Sub Form_QueryUnload(. . .)

    MyObjectManager.Form_QueryUnload ( _
        Me, _
        MyDBGridWrapper
```

UnboundAddData (Method)

The DBGrid's `UnboundAddData` event procedure should be coded within the Form module if the DBGrid is intended to allow new rows to be added.

Returns:

The new object, populated with the data from the new row.

Parameters:

`RowBuf:=`

(Required) The identical `RowBuf` parameter passed to the `DBGrid_UnboundAddData` event procedure.

`NewRowBookmark:=`

(Required) The identical `NewRowBookmark` parameter passed to the `DBGrid_UnboundAddData` event procedure.

`Sample:=`

(Required, Variant) A temporary, "throw-away" object of the class to be instantiated and populated with the data from the new row.

`Parent:=`

(Optional, Variant) The container object of the underlying VBOFCollection. If not presented, the VBOFDBGridWrapper uses the `Parent:=` value previously defined to the VBOFCollection, if available.

UnboundAddData example:

```
Private Sub DBGrid1_UnboundAddData(ByVal RowBuf As RowBuffer,
NewRowBookmark As Variant)
    Dim tempSample as New MyClass ' (not literally "MyClass")
    MyDBGridWrapper.UnboundAddData _
        RowBuf:=RowBuf, _
        NewRowBookmark:=NewRowBookmark, _
        Sample:=tempSample
End Sub
```

UnboundDeleteRow (Method)

The DBGrid's `UnboundDeleteRow` event procedure should be coded within the Form module to have VBOF automatically manage the deletion of objects through the DBGrid.

Returns:

A Long which contains the number of rows currently in the DBGrid

Parameters:

`Bookmark:=`

(Required) The identical `Bookmark` parameter passed to the `DBGrid_UnboundDeleteRow` event procedure.

UnboundDeleteRow example:

```
Private Sub DBGrid1_UnboundDeleteRow(Bookmark As Variant)
    pvtPersonsDBGridWrapper. _
        UnboundDeleteRow _
            Bookmark:=Bookmark
End Sub
```

UnboundReadData (Method)

The DBGrid's `UnboundReadData` event procedure should be coded within the Form module to allow VBOF to automatically populate the DBGrid with values from the objects contained within the associated VBOFCollection.

Returns:

A Long, which contains the number of rows added to the DBGrid.

Parameters:

`RowBuf:=`

(Required) The identical `RowBuf` parameter passed to the `DBGrid_UnboundReadData` event procedure.

`StartLocation:=`

(Required) The identical `StartLocation` parameter passed to the `DBGrid_UnboundReadData` event procedure.

`ReadPriorRows:=`

(Required) The identical `ReadPriorRows` parameter passed to the `DBGrid_UnboundReadData` event procedure.

UnboundReadData example:

```
Private Sub DBGrid1_UnboundReadData(ByVal RowBuf As RowBuffer,
StartLocation As Variant, ByVal ReadPriorRows As Boolean)
    pvtPersonsDBGridWrapper. _
        UnboundReadData _
            RowBuf:=RowBuf, _
            StartLocation:=StartLocation, _
            ReadPriorRows:=ReadPriorRows
End Sub
```

UnboundWriteData (Method)

The DBGrid's `UnboundWriteData` event procedure should be coded within the Form module if the DBGrid is intended to allow updates to rows.

Returns:

A Variant, which is the object which was updated by the user.

Parameters:

`RowBuf:=`

(Required) The identical `RowBuf` parameter passed to the `DBGrid_UnboundWriteData` event procedure.

`WriteLocation:=`

(Required) The identical `WriteLocation` parameter passed to the `DBGrid_UnboundWriteData` event procedure.

UnboundWriteData example:

```
Private Sub DBGrid1_UnboundWriteData(ByVal RowBuf As RowBuffer,
WriteLocation As Variant)
    pvtPersonsDBGridWrapper. _
        UnboundWriteData _
            RowBuf:=RowBuf, _
            WriteLocation:=WriteLocation
```

Appendix F: Public Methods of the Class Module VBOFListBoxWrapper

AddItems (Method)

The `AddItems` method causes VBOF to populate the bound ListBox with values from the objects contained in the bound VBOFListBoxWrapper.

Note: The Class Modules for objects which are to be supported by the VBOFListBoxWrapper must have the method `ObjectListBoxValue`.

Returns:
A Boolean indicating whether or not the items were successfully added to the ListBox.

Parameters:
None.

AddItems example:

```
pvtPersonsListBoxWrapper.Clear
pvtPersonsListBoxWrapper.AddItems
```

ListCount (Method)

The `ListCount` method returns the number of objects which are currently in the ListBox.

Returns:
A Long which contains the number of objects currently in the ListBox.

Parameters:
None.

ListCount example:

```
Dim MyLong As Long
. . .
MyLong = _
    pvtPersonsListBoxWrapper.ListCount
```

ListIndex (Property)

The VBOFListBoxWrapper provides the `ListIndex` Get and Set Property methods as part of the object-oriented management properties. In "Get" mode, this method returns the ListBox's `ListIndex` property for the object which selected in the ListBox. In "Set" mode, this method sets the ListBox's `ListIndex` property to the specified number.

Get ListIndex example:

```
Dim MyLong As Long
. . .
MyLong = _
    pvtPersonsListBoxWrapper.ListIndex
```

Set ListIndex example:

```
Dim MyLong As Long
. . .
pvtPersonsListBoxWrapper.ListIndex = _
 MyLong
```

See also the `ListIndexObject` method for a more object-oriented technique of accomplishing the same task.

ListIndexObject (Property)

The VBOFListBoxWrapper provides the `ListIndexObject` Get and Set Property methods as part of the object-oriented management properties in a more object-oriented technique than the `ListIndex` method. In "Get" mode, this method returns the object which is referenced by the current value of the ListBox's `ListIndex` property. In "Set" mode, this method sets the ListBox's `ListIndex` property to the object which is specified by the application.

Get ListIndexObject example:

```
Dim pvtCurrentPerson as Person
. . .
Set pvtCurrentPerson = _
    pvtPersonsListBoxWrapper.ListIndexObject
```

Set ListIndexObject example:

```
Dim pvtCurrentPerson as Person
. . .
pvtPersonsListBoxWrapper.ListIndexObject = _
    pvtCurrentPerson
```

Rebind (Method)

The VBOFListBoxWrapper's `Rebind` method must be invoked if either the bound VBOFCollection or ListBox are significantly altered.

Returns:

A Boolean indicating whether or not the `Rebind` was successful.

Parameters:

`Collection:=`

(Optional, VBOFCollection) The VBOFCollection object which is to be bound to the VBOFListBoxWrapper.

`ListBox:=`

(Optional, ListBox) The ListBox which is to be bound to the VBOFListBoxWrapper.

Rebind example:

```
pvtPersonsListBoxWrapper.Rebind _
    Collection:=pvtPersons, _
    ListBox:=ListBox1
```

Refresh (Method)

The VBOFListBoxWrapper's `Refresh` method can be invoked if the application needs to refresh the ListBox display, perhaps after the bound VBOFCollection as been significantly changed.

Returns:

A Long which contains the number of objects currently in the ListBox.

Parameters:

`DisplayOnly:=`

(Optional, Boolean) Indicates whether or not the application requests that only the ListBox is to be refreshed. The default action is False, to refresh the both underlying VBOFCollection and the ListBox display.

Refresh example:

```
pvtPersonsListBoxWrapper.Refresh
```

RemoveItem (Method)

The VBOFListBoxWrapper's `RemoveItem` method can be invoked when the application needs to remove an object from the ListBox and knows (or can determine) the ListIndex of the object to be removed.

Returns:

A Boolean which indicates whether or not the object was successfully removed from the VBOFCollection and the ListBox.

Parameters:

`ListIndex:=`

(Required, Long) References the ListIndex value of the item to be removed from the ListBox and the underlying VBOFCollection..

RemoveItem example:

```
pvtPersonsListBoxWrapper.RemoveItem _
    ListIndex:= 1
```

See also the `RemoveObject` method for a more object-oriented technique of accomplishing the same task.

RemoveObject (Method)

The VBOFListBoxWrapper's `RemoveObject` method can be invoked when the application needs to remove an object from the ListBox and the application knows the object to be removed. This is a more object-oriented technique than the `RemoveItem` method.

Returns:

A Boolean which indicates whether or not the object was successfully removed from the VBOFCollection and the ListBox.

Parameters:

`Object:=`

(Required, Variant) References the object which is to be removed from the ListBox and the underlying VBOFCollection..

RemoveItem example:

```
pvtPersonsListBoxWrapper.RemoveObject _
    pvtCurrentPerson
```

SelectObject (Property)

The VBOFListBoxWrapper provides the `SelectObject` Get and Set Property methods as part of the object-oriented management properties. The "Get" mode of this method returns the object in the ListBox which is currently selected by the user. The "Set" mode selects the specified object in the ListBox.

Get SelectObject example:

```
Dim pvtCurrentPerson as Person
. . .
```

```
Set pvtCurrentPerson =
    pvtPersonsListBoxWrapper.SelectObject
```

Set SelectObject example:

```
Set pvtPersonsListBoxWrapper.SelectObject = _
    pvtCurrentPerson
```

SelectObjects (Property)

The VBOFListBoxWrapper provides the `SelectObjects` Get and Set Property methods as part of the object-oriented management properties. The "Get" mode of this method returns a VB Collection of the objects in the ListBox which are currently selected by the user. The "Set" mode receives a VB Collection of objects to be selected in the ListBox.

Get SelectObjects example:

```
Dim MyCollection As Collection
 . . .
Set MyCollection =
    pvtPersonsListBoxWrapper.SelectObjects
```

Set SelectObjects example:

```
Dim MyCollection As Collection
 . . .
MyCollection.Add MyPerson1
MyCollection.Add MyPerson2
 . . .
Set pvtPersonsListBoxWrapper.SelectObjects = _
    MyCollection
```

TopIndex (Property)

The VBOFListBoxWrapper provides the `TopIndex` Get and Set Property methods to wrap the `TopIndex` methods of the VB ListBox in an object-oriented manner. The "Get" method returns `TopIndex` property of the ListBox bound to the VBOFListBoxWrapper. The "Set" method sets `TopIndex` property of the ListBox bound to the VBOFListBoxWrapper.

Get TopIndex example:

```
Dim MyTopIndex As Long
 . . .
MyTopIndex = _
    pvtPersonsListBoxWrapper. _
        TopIndex
```

Set TopIndex example:

```
Dim MyTopIndex As Long
 . . .
pvtPersonsListBoxWrapper.TopIndex = _
    MyTopIndex
```

See also the `TopObject` methods for a more object-oriented technique of accomplishing the same task.

TopObject (Property)

The VBOFListBoxWrapper provides the `TopObject` Get and Set Property methods as part of the object-oriented management properties. The "Get" method returns the object which is currently positioned at the top of the visible area of the ListBox bound to the VBOFListBoxWrapper. The "Set" method sets the object which is to appear at the top.

Get TopObject example:

```
Dim MyTopObject As Person
 . . .
Set MyTopObject = _
    pvtPersonsListBoxWrapper.TopObject
```

Set TopObject example:

```
Dim MyTopObject As Person
 . . .
Set pvtPersonsListBoxWrapper.TopObject = _
    MyTopObject
```

Unbind (Method)

The `Unbind` method releases the underlying VBOFCollection and ListBox from the VBOFListBoxWrapper. This is necessary because Visual Basic does not release the memory consumed by any given object until all references to the object have been severed. The `Unbind` method releases the reference to the VBOFCollection and ListBox once established by the VBOFListBoxWrapper.

Note: This method is not typically executed by the application. Rather, the application Form modules should invoke the VBOFObjectManager's `Form_QueryUnload` event procedure. The `Form_QueryUnload` method is a more thorough and encapsulated implementation than if the application were to attempt to perform a clean-up process.

Unbind example (indirectly):

```
Private Sub Form_QueryUnload(. . .)

    MyObjectManager.Form_QueryUnload ( _
        Me, _
        MyListBoxWrapper
```

Appendix G: Public Methods of the Class Module VBOFRecordSetWrapper

AbsolutePosition (Property)

The `AbsolutePosition` Property Get and Set methods control the `Bookmark` property of the underlying RecordSet object within the VBOFCollection.

Get AbsolutePosition example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim MyLong As Long
. . .
MyLong = MyRecordSetWrapper.AbsolutePosition
```

Set AbsolutePosition example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim MyLong As Long
. . .
MyRecordSetWrapper.AbsolutePosition = MyLong
```

AbsolutePositionObject (Property)

The `AbsolutePositionObject` Property Get and Set methods interface with the `AbsolutePosition` property of the underlying RecordSet object within the according to the desired object.

Get AbsolutePositionObject example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim MyPerson As Person
. . .
Set MyPerson = MyRecordSetWrapper.AbsolutePositionObject
```

Set AbsolutePositionObject example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim MyPerson As Person
. . .
Set MyRecordSetWrapper.AbsolutePositionObject = MyPerson
```

BOF (Method)

The `BOF` method returns the `BOF` property of the underlying RecordSet object within the VBOFCollection to which the VBOFRecordSetWrapper is bound.

Returns:

A Boolean.

Parameters:

None

BOF example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim MyBoolean as Boolean
. . .
MyBoolean = MyRecordSetWrapper.BOF
```

Clone (Method)

The `Clone` method returns a RecordSet which has been cloned from the underlying RecordSet object within the VBOFCollection to which the VBOFRecordSetWrapper is bound.

Returns:
A RecordSet object.

Parameters:
None

Clone example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim MyRecordSet as RecordSet
. . .
Set MyRecordSet = MyRecordSetWrapper.Clone
```

CloseRecordSet (Method)

The `CloseRecordSet` method closes the underlying RecordSet object within the VBOFCollection to which the VBOFRecordSetWrapper is bound.

Returns:
A Long, containing the value of the VB Err object reflecting the state of the `Close` method of the underlying RecordSet object.

Parameters:
None

CloseRecordSet example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim MyLong as Long
. . .
MyLong = MyRecordSetWrapper.CloseRecordSet
```

EOF (Method)

The `EOF` method returns the `EOF` property of the underlying RecordSet object within the VBOFCollection to which the VBOFRecordSetWrapper is bound.

Returns:
A Boolean.

Parameters:
None

EOF example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim MyBoolean as Boolean
. . .
MyBoolean = MyRecordSetWrapper.EOF
```

FindFirst (Method)

The `FindFirst` method executes a search forward across the underlying RecordSet object within the VBOFCollection to which the VBOFRecordSetWrapper is bound, beginning at the start of the RecordSet, using the search criteria specified in the `SearchCriteria:=` parameter.

Returns:

A Variant, which is the object which is the equivalent of the first data row retrieved which qualifies for the search criteria specified in the `SearchCriteria:=` parameter, or `Nothing`, if a suitable object for the `SearchCriteria:=` parameter is not found.

Parameters:

`SearchCriteria:=`

(Required, String) Identifies the search criteria to be used to locate the first qualifying row.

FindFirst example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyRecordSetWrapper.FindFirst
        (SearchCriteria:="LastName = 'Jones'")
```

FindLast (Method)

The `FindLast` method executes a search backward across the underlying RecordSet object within the VBOFCollection to which the VBOFRecordSetWrapper is bound, beginning at the end of the RecordSet, using the search criteria specified in the `SearchCriteria:=` parameter.

Returns:

A Variant, which is the object which is the equivalent of the first data row retrieved which qualifies for the search criteria specified in the `SearchCriteria:=` parameter, or `Nothing`, if a suitable object for the `SearchCriteria:=` parameter is not found.

Parameters:

`SearchCriteria:=`

(Required, String) Identifies the search criteria to be used to locate the first qualifying row.

FindLast example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyRecordSetWrapper.FindLast
        (SearchCriteria:="LastName = 'Jones'")
```

FindNext (Method)

The `FindFirst` method executes a search forward across the underlying RecordSet object within the VBOFCollection to which the VBOFRecordSetWrapper is bound, beginning at the current position of the RecordSet, using the search criteria specified in the `SearchCriteria:=` parameter.

Returns:

A Variant, which is the object which is the equivalent of the next data row retrieved which qualifies for the search criteria specified in the `SearchCriteria:=` parameter, or `Nothing`, if a suitable object for the `SearchCriteria:=` parameter is not found.

Parameters:

`SearchCriteria:=`

(Required, String) Identifies the search criteria to be used to locate the first qualifying row.

FindNext example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyRecordSetWrapper.FindNext
        (SearchCriteria:="LastName = 'Jones'")
```

FindPrevious (Method)

The `FindLast` method executes a search backward across the underlying RecordSet object within the VBOFCollection to which the VBOFRecordSetWrapper is bound, beginning at the current position of the RecordSet, using the search criteria specified in the `SearchCriteria:=` parameter.

Returns:

A Variant, which is the object which is the equivalent of the next data row retrieved which qualifies for the search criteria specified in the `SearchCriteria:=` parameter, or `Nothing`, if a suitable object for the `SearchCriteria:=` parameter is not found.

Parameters:

`SearchCriteria:=`

(Required, String) Identifies the search criteria to be used to locate the first qualifying row.

FindLast example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyRecordSetWrapper.FindPrevious
        (SearchCriteria:="LastName = 'Jones'")
```

MoveFirst (Method)

The `MoveFirst` method positions the underlying RecordSet object within the VBOFCollection to which the VBOFRecordSetWrapper is bound, to the beginning of the RecordSet.

Returns:

A Variant, which is the object which is the equivalent of the first data row in the underlying RecordSet object within the VBOFCollection, or `Nothing`, if no objects are found in the VBOFCollection.

Parameters:

None

MoveFirst example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyRecordSetWrapper.MoveFirst
```

MoveLast (Method)

The `MoveLast` method positions the underlying RecordSet object within the VBOFCollection to which the VBOFRecordSetWrapper is bound, to the last row of the RecordSet beyond the current.

Returns:

A Variant, which is the object which is the equivalent of the last data row in the underlying RecordSet object within the VBOFCollection, or `Nothing`, if no objects are found in the VBOFCollection.

Parameters:
None

MoveFirst example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyRecordSetWrapper.MoveLast
```

MoveNext (Method)

The `MoveNext` method positions the underlying RecordSet object within the VBOFCollection to which the VBOFRecordSetWrapper is bound, to the next row of the RecordSet beyond the current.

Returns:
A Variant, which is the object which is the equivalent of the next data row in the underlying RecordSet object within the VBOFCollection, or `Nothing`, if no more objects are found in the VBOFCollection.

Parameters:
None

MoveNext example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyRecordSetWrapper.MoveNext
```

MoveToObject (Method)

The `MoveToObject` method positions the underlying RecordSet object of the VBOFCollection to which the VBOFRecordSetWrapper is bound, to the row which is the equivalent of the object specified in the `Object:=` parameter.

Returns:
A Variant, which is the object specified in the `Object:=` parameter or `Nothing`, if no suitable object for the specified `Object:=` parameter is not found.

Parameters:
`Object:=`
(Required, Variant) Identifies the object to be used to position the underlying RecordSet object.

FindNext example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyRecordSetWrapper.MoveToObject
        (Object:=pvtPerson)
```

MoveToRecordNumber (Method)

The `MoveToRecordNumber` method positions the underlying RecordSet object of the VBOFCollection to which the VBOFRecordSetWrapper is bound, to the row whose RecordNumber is specified in the `RecordNumber:=` parameter.

Returns:

A Variant, which is the object equating to the specified `RecordNumber:=` parameter or `Nothing`, if no suitable object for the specified `RecordNumber:=` parameter is not found.

Parameters:

`RecordNumber:=`

(Required, Variant) Identifies the RecordNumber to be used to position the underlying RecordSet object.

FindNext example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim MyLong as Long
Dim pvtPerson as Person
. . .
Set pvtPerson = _
    MyRecordSetWrapper.MoveToRecordNumber
        (RecordNumber:=MyLong)
```

Rebind (Method)

The `Rebind` method must be invoked if either the bound VBOFCollection or DataControl are significantly altered.

Returns:

A Boolean indicating whether or not the `Rebind` was successful.

Parameters:

`Collection:=`

(Optional, VBOFCollection) The VBOFCollection object which is to be bound to the VBOFRecordSetWrapper.

`DataControl:=`

(Optional, DataControl) The DataControl object which is to be bound to the VBOFRecordSetWrapper.

Rebind example:

```
MyRecordSetWrapper.Rebind _
    Collection:=pvtPersons, _
    DataControl:=Data1
```

RecordCount (Method)

The `RecordCount` method returns the number of rows which are contained in the underlying RecordSet object within the VBOFCollection to which the VBOFRecordSetWrapper is bound.

Returns:

A Long.

Parameters:

None

RecordCount example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim MyLong as Long
. . .
MyLong = MyRecordSetWrapper.RecordCount
```

RecordSet (Method)

The `RecordSet` method returns the underlying RecordSet object in use by the VBOFCollection.

Returns:

A RecordSet object.

Parameters:

None.

RecordSet example:

```
Dim MyRecordSetWrapper as VBOFRecordSetWrapper
Dim MyRecordSet as RecordSet
. . .
Set MyRecordSet = _
    MyRecordSetWrapper.RecordSet
```

Refresh (Method)

The `Refresh` method can be invoked if the application needs to refresh the DataControl display, perhaps after the bound VBOFCollection as been significantly changed.

Returns:

A RecordSet object which is the RecordSet object contained within the bound VBOFCollection, reflecting any changes which may have occurred due to the `Refresh` method

Parameters:

`DisplayOnly:=`

(Optional, Boolean) Indicates whether or not the application requests that only the DataControl is to be refreshed. The default action is False, to refresh the both underlying VBOFCollection and the DataControl display.

Refresh example:

```
MyRecordSetWrapper.Refresh
```

Unbind (Method)

The `Unbind` method releases the underlying VBOFCollection and DataControl from the VBOFRecordSetWrapper. This is necessary because Visual Basic does not release the memory consumed by any given object until all references to the object have been severed. The `Unbind` method releases the reference to the VBOFCollection and DataControl once established by the VBOFDBGridWrapper.

Note: This method is not typically executed by the application. Rather, the application Form modules should invoke the VBOFObjectManager's `Form_QueryUnload` event procedure. The `Form_QueryUnload` method is a more thorough and encapsulated implementation than if the application were to attempt to perform a clean-up process.

Unbind example (indirectly):

```
Private Sub Form_QueryUnload(. . .)
```

```
MyObjectManager.Form_QueryUnload ( _
    Me, _
    MyRecordSetWrapper
```