Browse Form Manger

NextAge
CONSULTING

NextAge Consulting
Pete Halsted
110 East Center St. #1035
Madison, SD 57042
pete@thenextage.com
www.thenextage.com
www.thenextage.com/wordpress

Table of Contents

BSD 3 License

NextAge Consulting - Pete Halsted

Pete Halsted has been developing custom business management applications for small to medium-sized companies, since 1987. His focus is on client/server, distributed and cloud based development utilizing WinDev, WebDev, and PostgreSQL. Pete is a Clarion Certified Developer with 25 years in the industry, has spoken at several Developers conferences, and provided Developer training and mentoring on a one on one basis. He has served companies both large and small as Project Manager, Lead Architect, Lead Developer and Chief Technology Officer. Pete tours the country full-time by motor home with his wife and dog, enjoying the freedom provided by cloud based technologies. Pete is available for Project Management, Custom Design, Development, Training, and Speaking assignments. For more information please visit www.thenextage.com or follow his blog at www.thenextage.com/wordpress

General Information

The Browse Form Manager Class was created to provide some advanced interfaces in my applications, without having to include a lot of code to manage the interface in each window. Let me start by saying that from an Academic standpoint this Class breaks all the “rules”. As the Academic “Rule” is that class should not deal with the interface. However, I am not create a class that will be graded by a college professor somewhere, I am trying to get my job done, quicker and easier and follow the rule of when possible reuse code.

This class does manage the adding, changing and deleting of records, but there is nothing fancy in that code, it is just standardized code that I had on all my add, change, delete buttons, that has been made generic enough to be reusable. The real purpose and power of this class is the unique interfaces that it provides and manages so an advanced user interface can be provided without a lot of hand coding, for the most part, once the screen is designed, all the controls laid out on the window, then its a simple matter of adding 10-20 lines of class calls to the window to allow the class to manage the window. The interface styles supported by this class are:

1. Standard Browse and Form on separate windows. The table can be set to allow multiple selections and the class can manage several open forms at one time.
2. Peek-A-Boo Browse/Form Combination
    a. A browse with an integrated form that is automatically displayed and hidden as needed and learns the users preference of either only seeing the form while in edit mode, or always seeing the form.
3. Edit in Place Browse
    a. Browse with editable fields and no form
4. Tabbed Interface
    a. This is similar to the latest interfaces used by Microsoft Outlook and similar programs, the browse is displayed in a tab, and as records are edited that are opened as separate forms, contained in additional tabs. Logically this is very similar to the Standard Separate windows, but it appears to be one window with many tabs to the end user
5. There is also a extension class to assist with the management of child tables

There are many advanced properties and methods that allow you to extend the functionality of the window and still have the class manage the basic interface. This document will start by covering the basic setup of each of the interface styles, and then cover the advanced methods and properties that can be used to extend the class.

Peek-A-Boo Browse/Form Combination

Manages a browse with an integrated form that is automatically displayed and hidden as needed. There is also logic that "Learns" the users preference of either only seeing the form while in edit mode, or if they prefer to always seeing the from.

1. Include BFMPeek Code Brick in the window's *Global Declarations Event*
    a. `BFM is BrowseFormManager`
        i. Instantiates a copy of the Browse Form Manager class
    b. `BFM.TheTableControl = TBL_xxx..FullName`
        i. Name of the Table Control
    c. `BFM.PeekaBooContainerControl = SC..FullName`
        i. Name of the container control used for the form, likely a Super Control or a Tab Control
    d. `BFM.AddGrayControl(SC..FullName)`
        i. Adds a control to the list of controls to set the state to Gray when not in edit mode, adding the container control will automatically affect all the controls within the container.
        ii. Multiple calls can be made to to add additional controls to be managed.
        iii. Optionally *AddDisableControl* can be called instead which will set the state to inactive, but not Gray it.
        iv. If desired a slightly different interface can be achieved by Using AddDisableControl for the container, and AddGrayControl for the Form Buttons, which will make the form fields read only but does not gray them, making them them easier to read. The graying of the Form Buttons, provides feedback to the user that the Form is not in Edit Mode.
    e. `BFM.FileToUpdate = xxx..Name`
        i. The name of the File being maintained
    f. `BFM.FilePrimaryKey = xxx.xxx..Name`
        i. The name of the File Field that is the primary key of the File
    g. `BFM.TablePrimaryKey = TBL_xxx.COL_xxx..FullName`
        i. The name of the column in the table that holds the value of the primary key for the File. This column does not have to be visible
    h. `BFM.ScreenType = BrowseFormManager.PeekaBoo`
        i. Sets the Class management mode for Peek-A-Boo
    i. `BFM.FocusControl = SC.EDT_xxx..FullName`
        i. The control that will gain focus when entering edit mode
    j. `BFM.ControlToHide = SC..FullName`
        i. This is generally the container control that will be hidden, the size of this control is used to adjust the size of the table control to create the Peek-A-Boo effect.
    k. `BFM.AddControlToMove(TPL_PeekABooBrowseButtons..FullName)`

        i. This is generally the template control that contains the Add, Change and Delete buttons so that move up and down as the browse grows and shrinks with the Peek-A-Boo effect.
    l. `BFM.PeekABooState = DM.GetDefault(DefaultManager.UserWindow,"PeekABooState",False)`
        i. This sets the Initial State of the form as either visible or not. The sample code is using the Default Manager Class to store this value for each user, which means the application “Learns” if the user prefers to see the form all the time or only when it is in edit mode.
        ii. This is optional and could be hard coded if you prefer you application to always work the same way.
2. The container control needs to be marked as either disabled or grayed in the IDE so that the initial state of the control matches the Class Configuration.
3. Include a call to the BrowseRowSelect method In the table’s *Row Selection Event*
    a. `BFM.BrowseRowSelect()`
    b. This call refreshes the form as the user moves from row to row of the table
4. In the Closing Event of the window, the users preference for the form display mode can be stored:
    a. `DM.SaveDefault(DefaultManager.UserWindow,"PeekABooState",BFM.PeekABooState)`
    b. Again this is option and can be left out if you will not be using a User preference to control the initial Peek-A-Boo state.
5. Browse Buttons - Add, Change, Delete and Show/Hide Form
    a. A control template is provide to populate these buttons and their class calls, and generally will not be to be changed, they are documented here so you can understand what calls are being made incase you need to extend the functionality for your situation. These single line calls to the class take care of all the management of the interface, priming records etc.
    b. When placing the template on the window WinDev names it something similar to CTPL_NoName, depending on your program charter. Renaming it to TPL_PeekABooBrowseButtons, will make it match what the other templates and code bricks expect.
    c. Add Button
        i. `BFM.AddButton()`
    d. Change Button
        i. `BFM.ChangeButton()`
    e. Delete Button
        i. `BFM.DeleteButton()`
    f. Show/Hide Form Button
        i. `BFM.ShowHideForm()`
6. Form Buttons - Save, Cancel
    a. A control template is provide to populate these buttons and their class calls, and generally will not be to be changed, they are documented here so you

can understand what calls are being made incase you need to extend the functionality for your situation. These single line calls to the class take care of all the management of the interface, saving records, resetting the table, etc.

b. Save Button
    i. `BFM.SaveButton()`
c. Cancel Button
    i. `BFM.CancelButton()`

7. Using with the Business Rules Class
    a. If using the Business Rules Class, a call to the ValidateAllFields method needs to be performed prior to saving the record, the easiest way to do this with the is to use the one of the “Advanced Properties” of the Browse Form Manager - (BeforeUpdateProcedure)
    b. This would be set in the Global Declarations process of the window:
        i. `BFM.BeforeUpdateProcedure = "WindowName.BeforeUpdate"`
    c. A local procedure for the Window would be created as follows,
        i.
```
PROCEDURE BeforeUpdate()

IF BFM.ActionType <> BrowseFormManager.DeleteRecord
THEN
           BadField is string =
BR.ValidateAllFields(TPL_BusinessRuleEditor.WindowNumb
erForBr)
           IF BadField <> "" THEN
              SetFocus({BadField,indControl})
              RESULT False
           END
END
RESULT True
```

    d. See Advanced Properties and the Business Rules Class document for additional information.

Browse with Separate Form

Manages a browse with a separate form window. This is the more traditional style of data entry interface. The browse can be multi select and can manage several open forms at one time.

1. Include *BFMSepBrowse* Code Brick in the window's *Global Declarations Event*
    a. `BFM is BrowseFormManager`
        i. Instantiates a copy of the Browse Form Manager class
    b. `BFM.TheTableControl = TBL_xxx..FullName`
        i. Name of the Table Control
    c. `BFM.FileToUpdate = xxx..Name`
        i. The name of the File being maintained
    d. `BFM.FilePrimaryKey = xxx.xxx..Name`
        i. The name of the File Field that is the primary key of the File
    e. `BFM.TablePrimaryKey = TBL_xxx.COL_xxx..FullName`
        i. The name of the column in the table that holds the value of the primary key for the File. This column does not have to be visible
    f. `BFM.UpdateWindow = "FRM_xxx"`
        i. The name of the window procedure that will be the update form. ***Note: This is surrounded with quotes, it is easier to type the procedure name without quotes, allowing intellisense to assist with the procedure name and then wrap it with quotes afterwards.***
    g. `BFM.ScreenType = BrowseFormManager.Separate`
        i. Sets the Class management mode for a Separate Form
2. Browse Buttons - Add, Change, Delete
    a. A control template is provide to populate these buttons and their class calls, and generally will not be to be changed, they are documented here so you can understand what calls are being made incase you need to extend the functionality for your situation. These single line calls to the class take care of all the management of the interface, priming records etc.
    b. Add Button
        i. `BFM.AddButton()`
    c. Change Button
        i. `BFM.ChangeButton()`
    d. Delete Button
        i. `BFM.DeleteButton()`
    e. ***Note: The Class calls are identical to those made by the Peek-A-Boo buttons, the ScreenType property tells these methods how they should handle the request***
3. Set Up the Form Window, this window will be a traditional style edit window.
    a. Populate the edit controls on the window, and set their links to the database file fields.

b. Include BFMSepForm Code Brick in the window's *Global Declarations Process*
    i. `PROCEDURE FRM_xxx(inSysid,LOCAL BFM is BrowseFormManager)`
        1. Redeclares the procedure definitional with two parameters, the primary key of the record being added, and an instance of the Browse Form Manager Class. This version of the class is instantiated with all the setting of the class managing the browse, but becomes its own copy, that is part of what allows multiple forms to be open at one time.
    ii. `BFM.FormSysId = inSysid`
        1. `Tells this instance of the class the value of the primary key`
    iii. `BFM.FormInit()`
        1. Performs all the initialization of the fetching the record, priming values on inserts and setting up the form for editing.

4. Form Buttons - Save, Cancel
    a. A control template (SeparateFormButtons) is provide to populate these buttons and their class calls, and generally will not be to be changed, they are documented here so you can understand what calls are being made incase you need to extend the functionality for your situation. You will notice that there are several lines of code instead of the one line class call that is used by the Peek-A-Boo Version. The Peek-A-Boo version follows nearly the same actual steps of calling the before and after update procedures its just that they are exposed via *Advanced Class Properties* instead of directly in the form as they are for the separate from.
    b. Save Button
        i. `BFM.SaveFormPrep()`
            1. Resyncs the database record and then moves the screen fields to the file fields.
        ii. `IF BeforeUpdateProcedure() THEN`
            1. Calls a local procedure to provide a place to do validation code. This is coded identical to how it is code for Peek-A-Boo using the *BeforeUpdateProcedure Advance Property*.
        iii. `BFM.SaveButton()`
            1. Does the actual updating of the database.
        iv. `AfterUpdateProcedure()`
            1. Calls a local procedure to provide a place to perform operations after the record is saved such as updating total files etc.
        v. `BFM.RefreshTableLine(BFM.FormSysId,BFM.ActionType)`
            1. Communicates back to the calling browse and updates and either adds the line on an insert or updates the line on an edit
        vi. `Close()`
            1. Closes the Form
        vii. `END`

            1. End statement that corresponds to the IF statement at the beginning of the code
    c. Cancel Button
        i. `Close()`
            1. There is no need for special code on a cancel since have done no data updating
5. Add the BeforeUpdateProcedure and AfterUpdateProcedure
    a. Since the control template is coded to call these local procedures, they must exist even if there is no code in them. They are coded the exact same as the procedures that are referenced by the *Advance Class Properties* with matching names.

Edit in Place Browse

Manages a browse that allows edit in place editing. Child Tables managed by the Child Manager class are setup slightly different and will be documented with the Child Manager Class.

1. Set the Properties of the Browse to allow Edit in Place
    a. Set Initial Status on GUI tab to Editable
    b. On the details tab
        i. Check Cascading Input On
        ii. Check Save During Row Exit On
2. Include *BFMEIP* Code Brick in the window's *Global Declarations Event*
    a. `BFM is BrowseFormManager`
        i. Instantiates a copy of the Browse Form Manager class
    b. `BFM.TheTableControl = TBL_xxx..FullName`
        i. Name of the Table Control
    c. `BFM.FileToUpdate = xxx..Name`
        i. The name of the File being maintained
    b. `BFM.FilePrimaryKey = xxx.xxx..Name`
        i. The name of the File Field that is the primary key of the File
    c. `BFM.TablePrimaryKey = TBL_xxx.COL_xxx..FullName`
        i. The name of the column in the table that holds the value of the primary key for the File. This column does not have to be visible
    d. `BFM.FocusControl = SC.EDT_xxx..FullName`
        i. The control that will gain focus when entering edit mode
    e. `BFM.ScreenType = BFM.eSeperate`
        i. Sets the Class management mode for Edit in Place
3. Edit In Place Buttons - Add, Change, Delete
    a. A control template is provide to populate these buttons and their class calls, and generally will not be to be changed, they are documented here so you can understand what calls are being made incase you need to extend the functionality for your situation. These single line calls to the class take care of all the management of the interface, priming records etc.
    b. Add Button
        i. `BFM.AddButtonEIP()`
    c. Change Button
        i. `BFM.SetControlFocus()`
    d. Delete Button
        i. `BFM.DeleteButton()`

Tabbed Interface

Manages a browse that is hosted inside a tab control, and the forms will be opened as additionals tabs. The browse can be multi select and can manage several open forms at one time. Much of the setup of the Tabbed Interface is similar to the Separate Browse Form because technically they are separate, as the forms are really a separate internal window being displayed inside the tab control. It may seem a little daunting to setup the first time but once you have set up one, you can create a "template" that contains most of the controls required. Currently the class supports having 5 forms (tabs) open at once. When the window is first displayed the five form tabs are hidden. When a form is requested one of the tabs is made visible and the form is displayed inside an internal window on that tab, the caption of the tab is set to one of the columns of the browse. If all 5 tabs are already displayed a window will give the user the opportunity to replace one of the open tabs with the new requested record. When saving or canceling a form, the tab will be hidden again. The demo application has a fully functioning example.

1. The entire "window" is hosted inside a tab control.
    a. The first tab will have all the control that would normally be on a "Separate Browse"
    b. Create 5 additional tabs, on each of these tabs add an internal window control that uses the entire area of the tab.
2. Include *BFMTabBrowse* Code Brick in the window's *Global Declarations Event*
    a. `BFM is BrowseFormManager`
        i. Instantiates a copy of the Browse Form Manager class
    b. `BFM.TheTableControl = TBL_xxx..FullName`
        i. Name of the Table Control
    c. `BFM.FileToUpdate = xxx..Name`
        i. The name of the File being maintained
    d. `BFM.FilePrimaryKey = xxx.xxx..Name`
        i. The name of the File Field that is the primary key of the File
    e. `BFM.TablePrimaryKey = TBL_xxx.COL_xxx..FullName`
        i. The name of the column in the table that holds the value of the primary key for the File. This column does not have to be visible
    f. `BFM.UpdateWindow = "IFRM_xxx"`
        i. The name of the window procedure that will be the update form. ***Note: This is surrounded with quotes, it is easier to type the procedure name without quotes, allowing intellisense to assist with the procedure name and then wrap it with quotes afterwards.***
        ii. This must be an internal window, it will be hosted inside the internal window controls that you placed on the 2 - 6th tabs of the tab control.
    g. `BFM.ScreenType = BrowseFormManager.Tabbed`
        i. Sets the Class management mode for a Tabbed Interface

h. `BFM.TabControl = TAB_BFM..FullName`
    i. The name of the tab control
i. `BFM.SetInternalWindowControl(1,IW_CustomerForm2..FullName)`
j. `BFM.SetInternalWindowControl(2,IW_CustomerForm3..FullName)`
k. `BFM.SetInternalWindowControl(3,IW_CustomerForm4..FullName)`
l. `BFM.SetInternalWindowControl(4,IW_CustomerForm5..FullName)`
m. `BFM.SetInternalWindowControl(5,IW_CustomerForm6..FullName)`
    i. These 5 lines set the name of the Internal Window Controls on tabs 2 - 6.
n. BFM.ColumnForCaption = TBL_xxx.COL_xxx..FullName
    i. The column from the browse that will be used to set the caption of a tab when a form is added and the tab is displayed.

3. Browse Buttons - Add, Change, Delete
    a. A control template is provide to populate these buttons and their class calls. This is the same Control template as the SeparateBrowseButtons and generally will not be to be changed, they are documented here so you can understand what calls are being made incase you need to extend the functionality for your situation. These single line calls to the class take care of all the management of the interface, priming records etc.
    b. Add Button
        i. `BFM.AddButton()`
    c. Change Button
        i. `BFM.ChangeButton()`
    d. Delete Button
        i. `BFM.DeleteButton()`
4. Set Up the Form Window, this window will be a traditional style edit window, except it will be an Internal Window
    a. Populate the edit controls on the window, and set their links to the database file fields.
    b. Include BFMSepForm Code Brick in the window's *Global Declarations Process.* Note this is the Separate Code brick because the code is the same.
        i. `PROCEDURE FRM_xxx(inSysid,LOCAL BFM is BrowseFormManager)`
            1. Redeclares the procedure definitional with two parameters, the primary key of the record being added, and an instance of the Browse Form Manager Class. This version of the class is instantiated with all the setting of the class managing the browse, but becomes its own copy, that is part of what allows multiple forms to be open at one time.
        ii. `BFM.FormSysId = inSysid`
            1. Tells this instance of the class the value of the primary key
        iii. `BFM.FormInit()`
            1. Performs all the initialization of the fetching the record, priming values on inserts and setting up the form for editing.
5. Form Buttons - Save, Cancel

a. A control template (TabbedFormButtons) is provide to populate these buttons and their class calls, and generally will not be to be changed, they are documented here so you can understand what calls are being made incase you need to extend the functionality for your situation. The code is identical to the code for the SeparateFormButtons, except the code to close the tab, instead of the window..
b. Save Button
    i. `BFM.SaveFormPrep()`
        1. Resyncs the database record and then moves the screen fields to the file fields.
    ii. `IF BeforeUpdateProcedure() THEN`
        1. Calls a local procedure to provide a place to do validation code. This is coded identical to how it is code for Peek-A-Boo using the *BeforeUpdateProcedure Advance Property*.
    iii. `BFM.SaveButton()`
        1. Does the actual updating of the database.
    iv. `AfterUpdateProcedure()`
        1. Calls a local procedure to provide a place to perform operations after the record is saved such as updating total files etc.
    v. `BFM.RefreshTableLine(BFM.FormSysId,BFM.ActionType)`
        1. Communicates back to the calling browse and updates and either adds the line on an insert or updates the line on an edit
    vi. `BFM.CloseTheTab(BFM.CurrentTab)`
        1. Closes the Form and hide the tab
    vii. `END`
        1. End statement that corresponds to the IF statement at the beginning of the code
c. Cancel Button
    i. `BFM.CloseTheTab(BFM.CurrentTab)`
        1. There is no need for special code on a cancel since have done no data updating

6. Add the BeforeUpdateProcedure and AfterUpdateProcedure
    a. Since the control template is coded to call these local procedures, they must exist even if there is no code in them. They are coded the exact same as the procedures that are referenced by the *Advance Class Properties* with matching names.

Properties

Note: Private Properties are not documented.

ActionType
The current file action of the class, uses constants eAdd, eChange, eDelete, eViewOnly

AfterUpdateProcedure
If specified this procedure will be executed after to saving the record. This provides a great place to perform special processing of related file, etc..

BeforeUpdateProcedure
If specified this procedure will be executed prior to saving the record. If it returns true then the save will be performed, If it returns false then the save will not be performed and the form will remain in edit mode. This provides a great place to perform validation prior to saving.

BlankTestColumn
Table Control column to be tested when using Edit in Place. If this column is blank then a record will not be added. This prevents getting blank records when a user enters into a new row before saving

ChildLookupsProcedure
If specified this procedure will be executed after the primary record of the table control is fetched, during a row selection. This provides a great place records related lookup table buffers if needed.

ColumnForCaption
Table column used when creating the caption for the from tabs of the tabbed interface.

ControlToHide
Used with Peek-a-Boo Form. This is the control containing the form that will be hidden during the Peek-a-Boo effect. Generally this will be a Super Control or Tab control.

ControlToHide
Used with Peek-a-Boo Form. Control that should be moved when Peek-a-Boo is in effect. Generally this will be the control template containing the Add, Change and Delete Buttons.

CurrentTab
The tab that the current form is hosted in when using the tabbed interface.

FilePrimaryKey
The database table field that is the primary key of the file.

FileToUpdate
The database table that will be maintained by the class

FocusControl
Edit control that will receive focus when the Peek-a-Boo form is in edit mode.

FormSysId
The primary key value to be used by the separate form. The separate form manages its own context of the record, so that multiple forms can be opened at once, and the browse can be operated while forms are open. This will be used by the to retrieve the correct record on the form. A value of 0 triggers an insert action.

InternalWindowControl
Used with Tabbed interface. The internal control that will host the internal window for the form.

PeekaBooContainerControl
Used with Peek-a-Boo Form. This is the control will be reset when inserting a new record. General this will be the same as the ControltoHide, but not always. For instance if displaying Child Tables in Tabs as part of the Peek-a-Boo form, the tab control would be the ControlToHide, but you would need to wrap the edit fields of the form inside a Super Control and use that as the PeekaBooContainerControl so that they can be reset on Insert. Reseting the tab control would interfere with the performance of the child tables in the other tabs.

PeekABooState
Controls the visibility of the Peek-a-Boo form.

PrimeProcedure
If specified this procedure will be executed when setting up a form for insert. This is a great place to setup default values, etc.

RefreshStyle
How the table refreshes, after an update, either one of the WinDev constants for TableDisplay (taInit,taCurrentRecord,taCurrentSelection, etc.) or None for no refresh. The default value depends on the type of browse and if QBE is present or not, unless you are trying to achieve a specific effect you will likely not need to manually set this property.

ScreenType
The type of interface being managed, uses the following equates: PeekaBoo, EIP, Separate, Tabbed

TabControl
Used with Tabbed interface. This is the tab control that is used for the browse and the forms.

TablePrimaryKey
This is the table control column that contains the value of the primary key for the table being maintained. The column can be hidden.

TheTableControl
The table control being managed by the class.

UpdateWindow
The name of the window that contains the form. For the Tabbed interface this is an internal window. For the separate form interface this is a true window.

Methods

Note: Private methods are not documented.

AddButton
Starts the Add action on Separate, Tabbed, and Peek-a-Boo interfaces. Called from the Insert button.

AddButtonEIP
Starts the Add action on Edit in Place interface. Called from the Insert button.

AddChildManager
Lets the Browse Form Manager know that that it will be using an instance of the Child Manager Class. Multiple Child Manage Classes can be added by making more than one call to the method.

AddControlToMove
Adds a control to the list of controls to move when Peek-a-Boo form is hidden and revealed.

AddDisableControl
Adds a control to the list of controls that should be disabled when the Peek-a-Boo form is not in Edit Mode.

AddGrayControl
Adds a control to the list of controls that should be grayed when the Peek-a-Boo form is not in Edit Mode.

AddHideControl
Adds a control to the list of controls that should be hidden when the Peek-a-Boo from is hidden. This is in addition to the ControlToHide property for the main container control.

ApplyChanges
Processes all the edits and writes them to the database for Edit in Place. Called from the Apply Button.

BrowseRowSelect
Should be called in the Row Select event of the Table Control when using the Peek-a-Boo form.

CancelButton
Cancels the Edit Mode of a Peek-a-Boo Form.
ChangeButton

Starts the Edit action on Separate, Tabbed, and Peek-a-Boo interfaces. Called from the Change button.

CloseTheTab
Close a Form Tab when using the Tabbed Interface.

DeleteButton
Starts the Delete action on all interfaces. Called from the Delete button.

FormInit
Called from the form of the Separate and Tabbed Interface initialize the record buffer.

RefreshTableLine
Refreshes the line in the table control when the separate or tabbed form is saved.

SaveButton
Save the record on all interfaces. Called from the Save button.

SaveFormPrep
Called to resync the database record prior to performing the validation and save logic on separate and tabbed interfaces.

SetControlFocus
Sets Focus to the Control specified by the related property for Peek-a-Boo forms. Also sets focus the table column for Edit in Place interfaces.

ShowHideForm
Toggles the Peek-a-Boo Forms visibility status.

Equates

PeekaBoo
Sets the ScreenType to the PeekaBoo Interface

EIP
Sets the ScreenType to the Edit in Place Interface

Separate
Sets the ScreenType to the Separate Browse/Form Interface

Tabbed
Sets the ScreenType to the Tabbed Interface

AddRecord
Record Add Mode

ChangeRecord
Record Change Mode

DeleteRecord
Record Delete Mode

ViewRecord
Record View Mode

Code Bricks

BFMEIP - BFM Edit in Place Declaration
Instantiates a copy of the Browse Form Manager class and sets it up to manage an Edit in Place Browse

BFMPeek - BFM Peek-a-Boo Declaration
Instantiates a copy of the Browse Form Manager class and sets it up to manage a Peek-A-Boo Browse/Form Combination Window

BFMSaveDefault - BFM Save PeekABooState Default
Uses the default manager to store the users preference for the initial mode of the Peek-A-Boo form. Should be place in the *Closing Event* of the window.

BFMSepBrowse - BFM Seperate Browse Declaration
Manages a browse with a separate form window. This is the more traditional style of data entry interface.

BFMSepForm - BFM Seperate Form Declaration
Manages the form portion of the separate browse form interface. This is the more traditional style of data entry interface.

BFMTabBrowse - BFM Tabbed Browse Declaration
Manages a browse with a forms hosted in a tab control.

BFMRowSelect - BFM Browse Row Select
Calls the BrowseRowSelect method to refresh the form as the user moves from row to row of the table. Should be place in the *Row Selection Event* of the table when using the PeekABoo interface.

Control Templates

EIPButtonsFull
Add, Change, and Delete buttons and class calls for Browse that uses an Edit in Place Interface.

EIPPopup
Adds a Popup Menu and Class calls for a Browse that uses an Edit in Place Interface.

PeekABooBrowseButtons
Add, Change, Delete and Show/Hide Form button and class calls for Peek-A-Boo windows

PeekABooFormButtons
Save and Cancel buttons and class calls for Peek-A-Boo Windows

SeparateFormButtons
Save and Cancel Buttons and class calls for Forms used with the Separate Browse/Form Interface.

SeparateBrowseButtons
Add, Change, and Delete buttons and class calls for Browse that uses a separate form window.

TabbedFormButtons
Save and Cancel Buttons and class calls for Forms used with the Tabbed Interface.

Child Manager Extension Class

The Child Manager Extension Class assist with managing child tables on a window that is being managed by the Browse Form Manager Class. This is mainly use to assist with small Edit in Place child tables (such as a child table of phone numbers for a customer). The child records are maintained in memory until they are applied or the parent is saved, this provides some very nice features like the ability to add child records during the insert of the parent without worry about Referential Integrity Issues and the ability to cancel out of child table changes. Traditionally we have always had to either force the parent to be inserted first, use temporary "shadow" tables for the details, or some other workaround to manage child records correctly all leading to lots of hand code that needs to be maintained.

Follow these steps to add a child table to a window being managed.

1. Add the child table to the form
    a. Both the primary key and the key to the parent table need to be columns in the table. They can be invisible if desired but they must be included.
    b. The table needs to be marked as "Loaded in memory"
    c. On the details tab of the child table be sure that "Cascading input" and "Save during row exit" are both unchecked.
    d. The Table needs to be marked as Editable or Grayed, depending on what the inital state should be. By default tables are created as Selection (without edit)
    e. The columns that will be edited need to be marked as editable
2. In the Global Declaration Section of the window, after the Browse Form Manager is configured, configure the Child Manager class. The Child Setup Code Brick will assist with this.
    a. `CM is ChildManager`
        i. Instantiates a copy of the Child Manager class
    b. `CM.TheTableControl = TBL_xxx..Name`
        i. Name of the Table Control
    c. `CM.FocusControl = TBL_xxx.COL_xxx..Name`
        i. Name of the Column to receive focus when entering edit mode
    d. `CM.FileToUpdate = xxx..Name`
        i. The name of the child file to be udpated
    e. `CM.FilePrimaryKey = xxx.xxx..Name`
        i. The child files primary key field
    f. `CM.TablePrimaryKey = TBL_xxx.COL_xxx..FullName`
        i. The name of the column that contains the primary key for the child file
    g. `CM.BlankTestColumn = TBL_xxx.COL_xxx..FullName`
        i. If this column is blank then the row is considered to be blank and is not inserted. Edit in Place tends to create a blank record at the bottom of the table depending on how the use navigates it, this feature is to keep from treating that row as a real row.

h. `CM.ParentColumn = TBL_xxx.COL_xxx..FullName`
    i. Name of the Column that contains the ID field to the Parent
i. `CM.ParentFilePrimaryKey = "xxxx.xxxx"`
    i. Name of the file field from the parent table, the Parent column will be set to this value when inserting the rows into the database. Note: Notice this is in quotes, a trick is to enter the file name and field name without quotes, which lets intellisense assist you, then surround it with quotes afterwards.
j. `CM.ChildFileParentField = xxx.xxx..Name`
    i. The field in the child file that references the parent file
k. `BFM.AddChildManager(CM)`
    i. Lets the Browse Form Manager know that it will be managing the table, this allows it to make calls to the Child Manager Class.

3. Child Add Change Delete Buttons - Add, Change, Delete and Apply Changes
    a. A control template is provide to populate these buttons and their class calls, and generally will not be to be changed, they are documented here so you can understand what calls are being made incase you need to extend the functionality for your situation. These single line calls to the class take care of all the management of the interface, priming records etc.
    b. Add Button
        i. `CM.AddButtonEIP()`
    c. Change Button
        i. `CM.SetControlFocus()`
    d. Delete Button
        i. `CM.DeleteButton()`
    e. Apply Changes Button
        i. `CM.ApplyChanges()`
4. In the init code of the table
    a. `CM.SetChildFilter()`

Note: Multiple child files can be managed, by simply instantiating multiple versions of the class, such as CM2 is ChildManager. All the calls would then be CM2. instead of CM. And the control template code would need to be overridden to make CM2. calls as well.

Properties of Child Manager Class

Note: Only Properties that are not the same as those of the Browse Form Manager are documented.

ChildFileParentField
The field in the ChildTable that will hold the value of the parent table's primary key value.

ParentColumn
The column of the child table control that contains the value of the parent table's primary key value.

ParentFilePrimaryKey
The parent table's primary key field.

VarialbeForFilter
Generally not required, as ParentFilePrimaryKey is used to set the child filters, but when using the separate or tabbed interfaces, where the file buffer can not be trusted, this needs to be set to the variable used when calling the form.

Methods of Child Manager Class

Note: Only Methods that are not the same as those of the Browse Form Manager are documented.

RefreshTable
Refreshes the Child Table. Called from the Browse Form Manager when a parent record is changed.

SetChildFilter
Performs a HFilter to limit the child table to records linked to the Parent Record.

Control Templates of Child Manager Class

ChildElPPopUpWithApply
Popup Menu with Add, Change, Delete, and Apply Changes for Edit in Place Child Browse.

ChildElPButtonsWithApply
Add, Change, Delete, and Apply Changes Buttons for Edit in Place Child Browse.

Code Bricks of Child Manager Class

BFMChild - BFM Child Browse Declaration
Instantiates a copy of the Child Manager class and sets it up to manage a child table included on a form being managed by the Browse Form Manager.

Change Log

1.0 - January 16, 2013

Initial Release