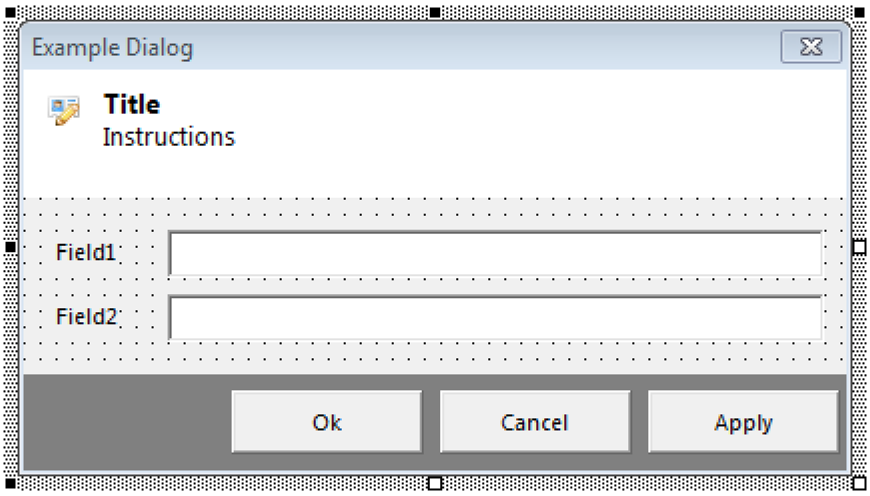
‘Apply’ logic for UserForm dialog

Posted on May 8, 2018May 8, 2018 by Rubberduck VBA
A recent comment on UserForm1.Show (https://rubberduckvba.wordpress.com/2017/10/25/userform1-show/) asked about how to extend that logic to a dialog that would have an “Apply” button. This article walks you through the process – and this time, there’s a download link (https://www.dropbox.com/s/x9sty06m2xh5zb5/ExampleDialog.xlsm?dl=0)!

The dialog is a simple UserForm with two textboxes and 3 buttons:



The **Model** for this dialog is a simple class exposing properties that the two textboxes manipulate – I’ve named the class `ExampleModel`:

```vba
Option Explicit

Private Type TModel
    field1 As String
    field2 As String
End Type

Private this As TModel

Public Property Get field1() As String
    field1 = this.field1
End Property

Public Property Let field1(ByVal value As String)
    this.field1 = value
End Property

Public Property Get field2() As String
    field2 = this.field2
End Property

Public Property Let field2(ByVal value As String)
    this.field2 = value
End Property
```

I also defined a simple `IDialogView` interface, which can be implemented by any other dialog, since it passes the model as an `Object` (i.e. it's not *tightly coupled* with the `ExampleModel` class in any way); the contract is simply "here's your model, now show me a dialog and tell me if I can proceed to consume the model" – in other words, the caller provides an instance of the model, and the implementation returns `True` unless the user cancelled the form.

```vba
Option Explicit

Public Function ShowDialog(ByVal viewModel As Object) As Boolean
End Function
```

The form's code-behind therefore needs to implement the `IDialogView` interface, and somehow store a reference to the `ExampleModel`. And since we have cancellation logic but we're not exposing it (we don't need to – the `IDialogView.ShowDialog` interface handles that concern, by returning `False` if the dialog is cancelled), the `IsCancelled` flag is just internal state.

As far as the "apply" logic is concerned, the thing to note here is the `Public Event ApplyChanges` event, which we *raise* when the user clicks the "apply" button:

```
Option Explicit

Public Event ApplyChanges(ByVal viewModel As ExampleModel)

Private Type TView
    IsCancelled As Boolean
    Model As ExampleModel
End Type
Private this As TView

Implements IDialogView

Private Sub AcceptButton_Click()
    Me.Hide
End Sub

Private Sub ApplyButton_Click()
    RaiseEvent ApplyChanges(this.Model)
End Sub

Private Sub CancelButton_Click()
    OnCancel
End Sub

Private Sub Field1Box_Change()
    this.Model.field1 = Field1Box.value
End Sub

Private Sub Field2Box_Change()
    this.Model.field2 = Field2Box.value
End Sub

Private Sub OnCancel()
    this.IsCancelled = True
    Me.Hide
End Sub

Private Function IDialogView_ShowDialog(ByVal viewModel As Object) As Boole
    Set this.Model = viewModel
    Me.Show vbModal
    IDialogView_ShowDialog = Not this.IsCancelled
End Function

Private Sub UserForm_Activate()
    Field1Box.value = this.Model.field1
    Field2Box.value = this.Model.field2
End Sub

Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
    If CloseMode = VbQueryClose.vbFormControlMenu Then
        Cancel = True
        OnCancel
    End If
End Sub
```

The **Presenter** class does all the fun stuff. Here I've decided to allow the model's data to be optionally supplied as parameters to the `Show` method; the form handles its `Activate` event to make sure the form controls reflect the model's initial values when the form is displayed:

```
Option Explicit
Private WithEvents view As ExampleDialog

Private Property Get Dialog() As IDialogView
    Set Dialog = view
End Property

Public Sub Show(Optional ByVal field1 As String, Optional ByVal field2 As S

    Set view = New ExampleDialog

    Dim viewModel As ExampleModel
    Set viewModel = New ExampleModel
    viewModel.field1 = field1
    viewModel.field2 = field2

    If Dialog.ShowDialog(viewModel) Then ApplyChanges viewModel
    Set view = Nothing

End Sub

Private Sub view_ApplyChanges(ByVal viewModel As ExampleModel)
    ApplyChanges viewModel
End Sub

Private Sub ApplyChanges(ByVal viewModel As ExampleModel)
    Sheet1.Range("A1").value = viewModel.field1
    Sheet1.Range("A2").value = viewModel.field2
End Sub
```

So we have a `Private WithEvents` field that gets assigned in the `Show` method, and we handle the form's `ApplyChanges` event by invoking the `ApplyChanges` logic, which, for the sake of this example, takes the two fields and writes them to A1 and A2 on **Sheet1**; if you've read There is no worksheet (https://rubberduckvba.wordpress.com/2017/12/08/there-is-no-worksheet/) then you know how you can introduce an interface there to decouple the worksheet from the presenter, and then it doesn't matter if you're writing to a worksheet, a text file, or a database: the presenter doesn't *need* to know all the details.

The calling code in `Module1` might look like this:

```
Option Explicit

Public Sub ExampleMacro()
    With New ExamplePresenter
        .Show "test"
    End With
End Sub
```

One problem here, is that the **View** implementation is coupled with the presenter (i.e. the presenter is creating the view): we need the concrete UserForm type in order for VBA to see the events; without further abstraction, we can't quite pass a `IDialogView` implementation to the presenter logic without popping up the actual dialog. Pieter Geerkens has a nice answer on Stack Overflow (https://stackoverflow.com/a/45825831/1188513) that describes how an **Adapter Pattern** can be used to solve this problem by introducing more interfaces, but covering this design pattern will be the subject of another article.

Posted in OOP, tutorials, vbaTagged design-patterns, oop, tutorial, vba

Published by Rubberduck VBA

I'm Mathieu Guindon (Microsoft MVP Office Apps & Services, 2018), you may have known me as "Mat's Mug" on Stack Overflow and Code Review Stack Exchange. I manage the Rubberduck open-source project, whose goal is to bring the Visual Basic Editor (VBE) - VBA's IDE - into the 21st century, by providing features modern IDE's provide. *View all posts by Rubberduck VBA*

11 thoughts on "'Apply' logic for UserForm dialog"

1. **koitaki** *June 10, 2018* *Reply*

Mille merci Mathieu, this is as usual *really* useful stuff.

I would love-love-love to use this MVP approach with Access.
But haven't succeeded with it yet.

For example, with this particular case, when I try to use this approach on Access, it runs into specific difficulties.
Such as, a modal form seems to need to be opened via:
DoCmd.OpenForm view.Name, acNormal, , , , acDialog

1) I don't know if that's actually opening the view form object)
2) This seems to create problems because where this line doesn't work so well:
If Dialog.ShowDialog(viewModel) Then ApplyChanges viewModel
Eg. upon closing the form, the code goes back to the Presenter class, and the form's this.IsCancelled loses its state

I'll keep working my way through these issues, and hopefully get there.
But if in the meantime you have any ideas, I'd be happy to hear them 🙂

Cheers again, really appreciate what you're doing with this!

Chris

Rubberduck VBA *June 10, 2018* *Reply*

Thanks! I'm not very familiar with Access, but I do know that Rubberduck sees Access forms are "document" modules, like a worksheet in Excel. This might explain it. Know that you can still have MSForms UserForm modules in an Access VBA project though: IIRC vanilla-AccessVBE hides the command, but Rubberduck's code explorer doesn't care and exposes the command anyway, so you can add a UserForm in Access if you need one. Hope it helps!

2. **Profex** *September 21, 2018* *Reply*

I decided to convert my current project over to using proper OOP techniques, following your posts. It has a UserForm that has an Import button, Report Button and a Multi-Column-ListBox that lists all the projects from a database among other things. I'm having trouble deciding how far I want to go into defining the Model for the ListBox. It looks like I'm going to implement AddItem, List, Selected, ListCount & ListIndex properties. It's reminding me of my awful 64-bit COM wrapper for the .Net Listview control :(.

How would you deal with ListBox or ComboBox controls in the Model?

Rubberduck VBA *September 21, 2018* *Reply*

Controls themselves belong only in the view; in the model I'd have a getter that gives me the items I need to have in the listbox, and a read/write property for the current/selected item. The code-behind for the listbox would set the model's selecteditem on change, and the setter for the view's "model" property would populate the listbox. I hope it's not too confusing, it's a bit hard to be clear in a comment box.. it's very similar to a ViewModel in WPF/MVVM, except instead of XAML bindings you use the controls' event handlers to set the model properties – like I'm doing here with these textboxes.

1. **Profex** *November 2, 2018*

I ended up creating a model class that mimicked a lot of the the Listbox control properties/methods. If anyone (i.e. SmileyFtW) is interested in it, here is a link:

```
'@Folder("View.Model")
Option Explicit

Private Const MODULE_NAME As String = "ListModel"

Private Type TListModel
   Data() As Variant
   Selected() As Boolean
   Columns As Integer
   Count As Integer
   Index As Integer
End Type

Private this As TListModel

Private Sub Class_Initialize()
   this.Columns = 1
   this.Index = -1
End Sub

Public Sub Clear()
   With this
      Erase .Data
      Erase .Selected
      .Count = 0
      'Columns = 0
      .Index = -1
   End With
End Sub

Public Sub AddItem(Optional Item As Variant, Optional Index As Integer = -1)
Dim r As Integer, C As Integer
   With this
      If Index < -1 Or Index > .Count Then
         Err.Raise 5, , "Invalid argument."
      Else
         ReDim Preserve .Data(.Columns - 1, .Count)
         ReDim Preserve .Selected(.Count)
         If Index >= 0 Then
            ' Move all the data after the Index row, up one row.
            For r = .Count To Index + 1 Step -1
               For C = 0 To .Columns - 1
                  .Data(C, r) = .Data(C, r - 1)
```

```
                    Next
                    .Selected(r) = .Selected(r - 1)
                Next
                ' Clear all the data in the Index row
                For C = 0 To .Columns - 1
                    Set .Data(C, Index) = Nothing
                Next
                .Selected(Index) = False
            Else    ' Set the Index to the next row
                Index = .Count
            End If
            If Not IsMissing(Item) Then .Data(0, Index) = Item
            .Count = .Count + 1
        End If
    End With
End Sub

Public Sub RemoveItem(Index As Integer)
Dim r As Integer, C As Integer
    With this
        If Index < 0 Or Index >= .Count Then
            Err.Raise 5, , "Invalid argument."
        Else
            ' Move all the data after the Index row, up one row.
            For r = Index + 1 To .Count - 1
                For C = 0 To .Columns - 1
                    .Data(C, r - 1) = .Data(C, r)
                Next
                .Selected(r - 1) = .Selected(r)
            Next
            .Count = .Count - 1
            ReDim Preserve .Data(.Columns - 1, .Count - 1)
            ReDim Preserve .Selected(.Count - 1)
        End If
    End With
End Sub

Public Property Get List(Row As Integer, Optional Column As Integer = 0) As Varia
    With this
        If Row < 0 Or Row >= .Count Then
            Err.Raise 381, , "Could not get the List property.  Invalid property-array row
        ElseIf Column < 0 Or Column >= .Columns Then
            Err.Raise 381, , "Could not get the List property.  Invalid property-array colu
        Else
            List = .Data(Column, Row)
        End If
    End With
End Property

Public Property Let List(Row As Integer, Column As Integer, Value As Variant)
    With this
        If Row < 0 Or Row >= .Count Then
            Err.Raise 381, , "Could not get the List property.  Invalid property-array row i
        ElseIf Column < 0 Or Column >= .Columns Then
            Err.Raise 381, , "Could not get the List property.  Invalid property-array colu
        Else
            .Data(Column, Row) = Value
        End If
    End With
End Property

Public Property Get Selected(Index As Integer) As Boolean
    With this
        If Index < 0 Or Index >= .Count Then
            Err.Raise 381, , "Could not get the List property.  Invalid property-array inde
        Else
            Selected = .Selected(Index)
        End If
    End With
End Property

Public Property Let Selected(Index As Integer, Value As Boolean)
    With this
        If Index < 0 Or Index >= .Count Then
            Err.Raise 381, , "Could not get the List property.  Invalid property-array inde
        Else
            .Selected(Index) = Value
        End If
    End With
End Property

Public Property Get ListCount() As Integer
    ListCount = this.Count
End Property

Public Property Get ListIndex() As Integer
    ListIndex = this.Index
End Property

Public Property Let ListIndex(Value As Integer)
```

```
    With this
        If Value < -1 Or Value >= .Count Then
            Err.Raise 5, , "Invalid argument."
        Else
            .Index = Value
        End If
    End With
End Property

Public Property Get ColumnCount() As Integer
    ColumnCount = this.Columns
End Property

Public Property Let ColumnCount(Value As Integer)
Dim NewData() As Variant
Dim r As Integer, C As Integer
    With this
        If Value <= 0 Then
            Err.Raise 5, , "Invalid argument."
        Else
            If .Count > 0 And .Columns <> Value Then
                ' If the columns change, we can't redim the array, we need to create a new l
                ReDim NewData(Value - 1, .Count - 1)
                For r = 0 To .Count - 1
                    For C = 0 To Value - 1
                        NewData(C, r) = .Data(C, r)
                    Next
                Next
                .Data = NewData
                Erase NewData
            End If
            .Columns = Value
        End If
    End With
End Property
```

view raw gistfile1.txt hosted with ❤ by **GitHub**

In the form, I called the following DisplayProjectList routine whenever I needed to update Listbox (Activate/Add/Remove):

```
Private Sub DisplayProjectList()
Dim i As Integer, j As Integer
lstProjects.Clear
With this.Model.ProjectList
For i = 0 To .ListCount – 1
lstProjects.AddItem .List(i, 0)
```

```
For j = 1 To .ColumnCount – 1
lstProjects.List(i, j) = .List(i, j)
Next
lstProjects.Selected(i) = .Selected(i)
Next
lstProjects.ListIndex = .ListIndex
'If lstProjects.ListCount > 0 And lstProjects.ListIndex < 0 Then lstProjects.ListIndex = 0
End With
End Sub
```

Note: I ended up not caring about what was selected on Add (Remove doesn't matter), but because the Model for the ListBox included a Selected property, I also had the following:

```
Private Sub lstProjects_Change()
Dim i As Integer
With this.Model.ProjectList
For i = 0 To lstProjects.ListCount – 1
.Selected(i) = lstProjects.Selected(i)
Next
End With
End Sub
```

For the CobmoBox control, or if you just don't care about preserving the selected items when adding a new item, you would remove all references to the Selected properties.

3. **SmileyFtW** *October 24, 2018* *Reply*

Great stuff. If there were to be an option on the form to show one or more subsets of the list presented in the combo box how might that be done? Say there was a list of 10 values in the data table and the user could select to choose to show only the even ones, the odd ones or all via check boxes or similar, what might that be implemented?

Rubberduck VBA *October 24, 2018* *Reply*

There isn't One True Way, but I think I'd try to keep the presenter responsible for knowing what to do to get the [filtered] data (worksheet, db, hard-coded, whatever), and overwrite a property in the model that, when assigned, raises an event that the view can handle. Or the view could have some 'Refresh' method that clears & re-populates the comboboxes from the updated model contents; view.Refresh would be invoked from the presenter, after it finishes updating the model. So the model needs not only a property for the user-selected combobox value, it also needs a property for the desired filter, and then a property for the available values as per that filter. Basically you just do whatever needs to be done to systematically defer work out of the view and into the presenter. Eventually the model grows too large and confusing, so you keep the "model" stuff (i.e. selected values, user inputs) there and pull the "view model" stuff (i.e. filters, combobox/listbox sources) out into a new class to keep things manageable & clean.

1. **Profex** *November 2, 2018* *Reply*

I posted the code that I used for the basic ListBox above. It includes a Selected property in the model, so you can modify the selected item from either the view or the model. I didn't get to the point where I needed the view.Refresh method that Mathieu mentioned yet, since I refresh the list on most command clicks in the form.

4. **SmileyFtW** *November 23, 2018* *Reply*

   I actually did as Mathieu suggested. Sorta. I get all of the data as a collection from a table on the worksheet using the abstractions in the example. I then convert the collection to an array; I get the subsets at the same time into subset arrays. When the filter is selected a combo box is (re)loaded with the appropriate array. The arrays are mutually exclusive in my situation so each list the combo shows is unique. When an edit of the data is "Applied" then the new model data is re-imported and segregated for display in the combo. I haven't had time to consume Profex's post… I certainly will though!

5. **SmileyFtW** *December 6, 2018* *Reply*

   If there were more than one type of "thing" that could be edited, but only one "thing" a time and the "thing" being edited is one many of those type "things". I am thinking that the ApplyChanges would accept any of the different types of "things" and decide what to do based on what is passed to it. ApplyChanges would also only want (need) to save the one edited "thing" and not the entire set of "things" the individual "thing" belongs to. Assuming the set of "things" is stored as a table (like in the example workbook) and then after the edited item is saved in that table I would think that reloading the affected table would want to be done (as opposed to managing the changes singularly in the presenter) to update the dialog.

   I am thinking that the "ApplyChanges" would determine the type of "thing" passed (using TypeName) and then know how/where to save the changes assuming that each type "thing" has its own table.

   Am I on the right path?

   Rubberduck VBA *December 7, 2018* *Reply*

   Almost. Encapsulate the "things" into a model class, and then all ApplyChanges needs to care about is the model – which the presenter alreary holds a reference to (or it could be passed as an argument of the event) – if applying changes gets non-trivial, consider writing a dedicated, testable class for it – especially if the file system or a database gets involved.