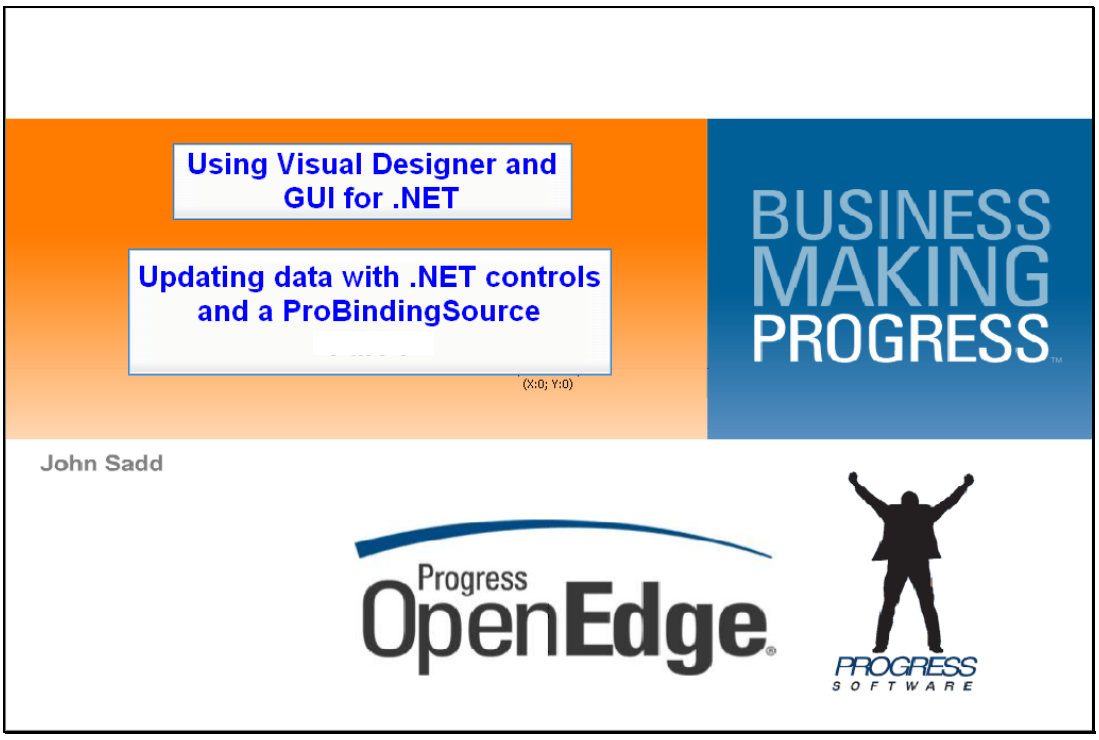
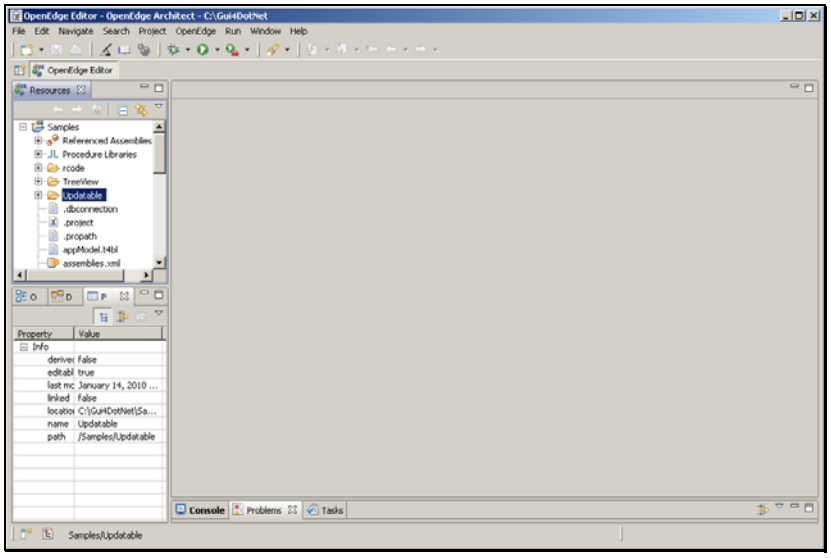
UPDATING DATA WITH .NET CONTROLS AND A PROBINDINGSOURCE

John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
March 2010

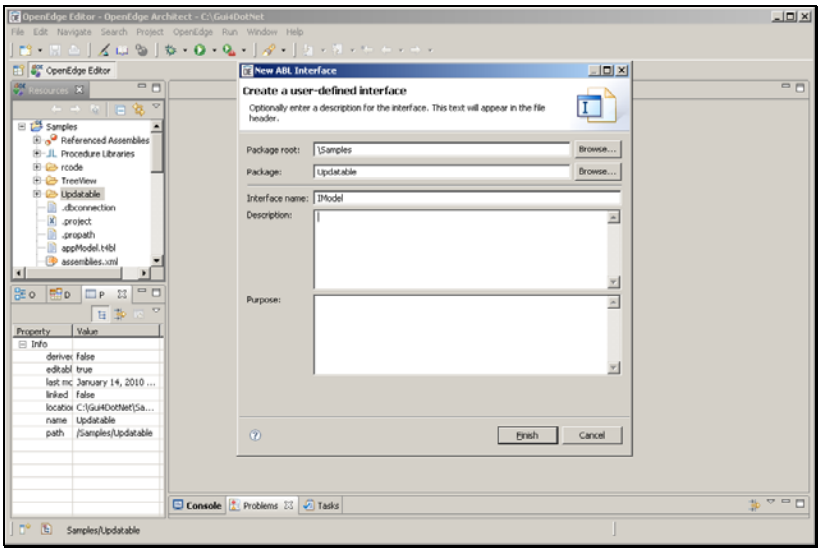


DISCLAIMER

This paper accompanies a two-part presentation that extends other materials on data binding and using the ProBindingSource control, by showing you simple ways to handle update operations. Here I introduce more of the remaining **ProBindingSource** properties, methods and events that support updates, and a few of the basic UI control events that you can interact with as well. As you can see here in the **Resources** View, I've created a new subdirectory in my project called **Updatable** where I'll put my new code samples:



That directory corresponds to the notion of a package, in object-oriented terms, for organizing the code in my project. To show you how to specify the package when you create new source files, I start by creating a new ABL Interface. Because I had selected the **Updatable** folder in the **Resources View**, that package name is already filled in for me as the default. As I've done before, I can just enter the name of the interface. This is a variation on the **IModel** interface that I've used before, so I can use the same name, because it will be stored in a new folder with all the rest of my sample code for trying out update operations:



I start with the Interface code skeleton that Architect generates, paste in the three method definitions used in the read-only operations created in other sessions on the **ProBindingSource**, and then add a new method to support saving data, which takes a buffer name as a parameter, and returns a **LOGICAL** value to tell me whether the save succeeded or not.

```
USING Progress.Lang.*.

INTERFACE Updatable.IModel:
  METHOD PUBLIC VOID FetchData (INPUT pcFilter AS CHARACTER ).
  METHOD PUBLIC VOID SortData  (INPUT pcSort   AS CHARACTER ).
  METHOD PUBLIC HANDLE GetQuery().
  METHOD PUBLIC LOGICAL SaveData(INPUT pcBufferName AS CHARACTER ).

END INTERFACE.
```

Next I create a new class based on the **CustomerModel** class, and place it into my **Updatable** package. The first change I need to make is to qualify the name of the interface it implements so that the compiler finds the right one.

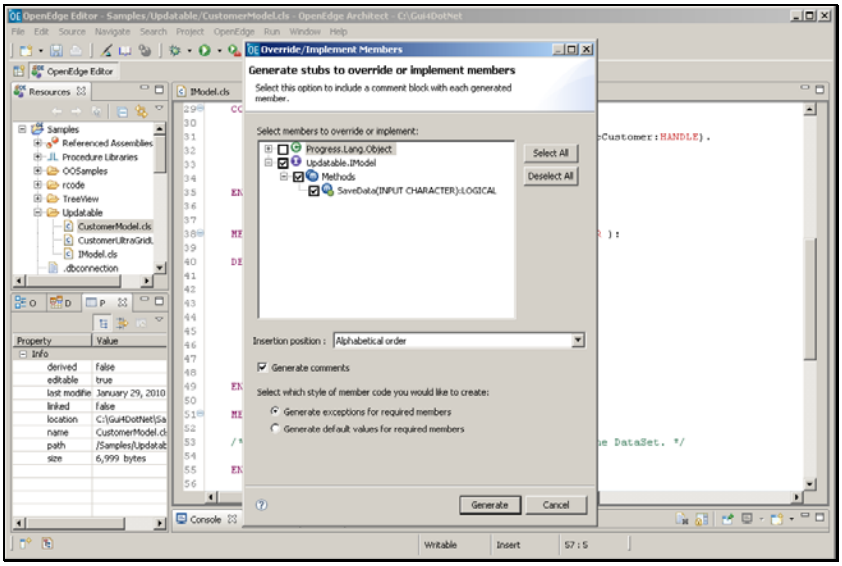
```
CLASS Updatable.CustomerModel IMPLEMENTS Updatable.IModel:
```

Then I need to make a change to the **FetchData** method that tells the model class to populate its ProDataSet. A ProDataSet can keep track of changes you make after it's been filled with data from its data source. But before I fill it I have to make sure that the **TRACKING-CHANGES** property on the **ttCustomer** temp-table is set off. Then after the **FILL** is done, I can set the property to **True** so that any changes made from that point on will be tracked in the **ttCustomer** temp-table's before-table:

```
METHOD PUBLIC VOID FetchData( INPUT pcFilter AS CHARACTER ):

    DEFINE VARIABLE cPrepare AS CHARACTER NO-UNDO.
        TEMP-TABLE ttCustomer:TRACKING-CHANGES = FALSE.
        cPrepare = "FOR EACH AutoEdge.Customer".
        IF pcFilter NE "" THEN
            cPrepare = cPrepare + " WHERE " + pcFilter.
        QUERY qCustomer:QUERY-PREPARE (cPrepare).
        DATASET dsCustomer:FILL().
        httCustQuery:QUERY-OPEN ().
        TEMP-TABLE ttCustomer:TRACKING-CHANGES = TRUE.
    END METHOD.
```

Now I'm ready to accept updates from the user interface. Remember that the **Source** menu in Architect helps you make all sorts of edits to a source file, including new methods, constructors, and event and property definitions for a class. I need to add a new method, but because it's a method defined in my interface **IModel**, I can select the **Override / Implement Members** option to implement the **SaveData** method that I defined in the **Updatable** version of the Interface. I have Architect add it to the source file in alphabetical order, and tell Architect to generate an empty comments block at the top of the method.



To the default code I start out with, as generated by Architect, I first add a couple of variable definitions to point to the temp-table's buffer, and its before-buffer, where changes are kept track of:

```
METHOD PUBLIC LOGICAL SaveData( INPUT pcBufferName AS CHARACTER ):

  DEFINE VARIABLE hCustBuffer   AS HANDLE NO-UNDO.
  DEFINE VARIABLE hBeforeBuffer AS HANDLE NO-UNDO.
```

Also, I put in a sanity check to make sure that the buffer name passed in is the right one. In this case there's only one temp-table in the DataSet, but in other cases there could be more than one, which is why the buffer name parameter is here:

```
    IF pcBufferName NE "ttCustomer" THEN
    DO:
        /* Sanity check -- this is the only buffer in the DataSet */
        MESSAGE "Invalid buffer name " pcBufferName VIEW-AS ALERT-BOX.
        RETURN FALSE.
    END.
```

And then I initialize the two buffer handle variables:

```
    hCustBuffer = BUFFER ttCustomer:HANDLE.
    hBeforeBuffer = hCustBuffer:BEFORE-BUFFER.
```

I want to have some simple logic in the Model to give me a way to show what happens if the user enters invalid data. My table has **CustomerBirthCountry** values of only the USA and Germany, so my check says that if any other value is entered, I reject the change to the temp-table, and return false to signal the error to the View:

```
    IF ttCustomer.CustomerBirthCountry NE "USA" AND
       ttCustomer.CustomerBirthCountry NE "Germany" THEN
    DO:
       MESSAGE "Invalid Birth Country value, must be USA or Germany."
           VIEW-AS ALERT-BOX.
       hBeforeBuffer:REJECT-ROW-CHANGES ().
       RETURN FALSE.
    END.
```

Note that the Model class expects the changes the user has made to have been saved to the DataSet's temp-table already. This is important because you don't want even simple logic like this in the View, which just handles the user interface. And you don't want the Model, which is in charge of the data, to know how to look up into the UI and see the values in the user interface controls. So the UI has to get the values assigned to the temp-table before it runs **SaveData**, and to do this, it uses the ProBindingSource as an intermediary. If there's no error the Model uses the ProDataSet's **SAVE-ROW-CHANGES** method to save data back to the DataSource, the database table it came from. **SAVE-ROW-CHANGES** itself could fail, because it does conflict checks if more than one user is changing the same row at the same time, and there might be underlying database triggers that could fail as well. So if **SAVE-ROW-CHANGES** succeeds, the **ACCEPT-ROW-CHANGES** method keeps the changes in the temp-table and marks the change as complete, and I tell the UI that the save succeeded:

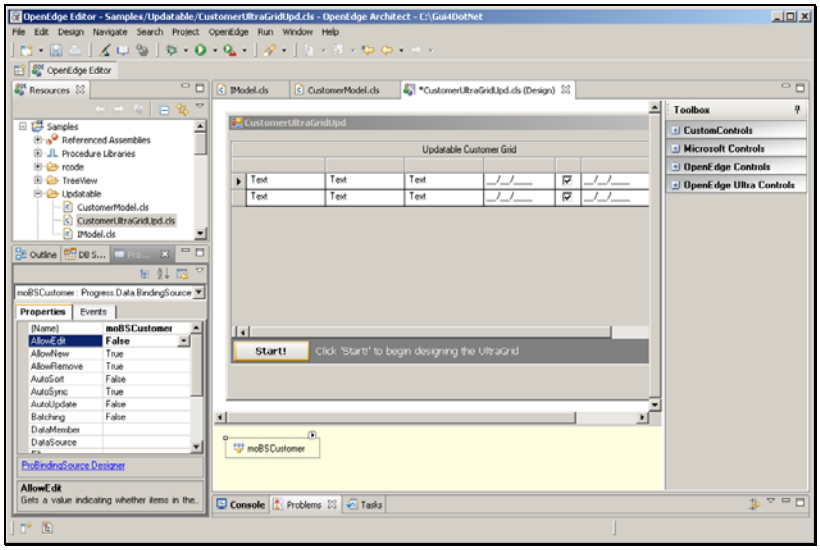
```
/* If we get here all client-side validation succeeded. */
    IF hBeforeBuffer:SAVE-ROW-CHANGES () THEN
    DO:
        hBeforeBuffer:ACCEPT-ROW-CHANGES ().
        RETURN TRUE.
    END.
```

But if **SAVE-ROW-CHANGES** returns an error, then **REJECT-ROW-CHANGES** scrubs the change from the temp-table, and the method returns an error flag.

```
    ELSE DO:
        hBeforeBuffer:REJECT-ROW-CHANGES ().
        RETURN FALSE.
    END.

END METHOD.
```

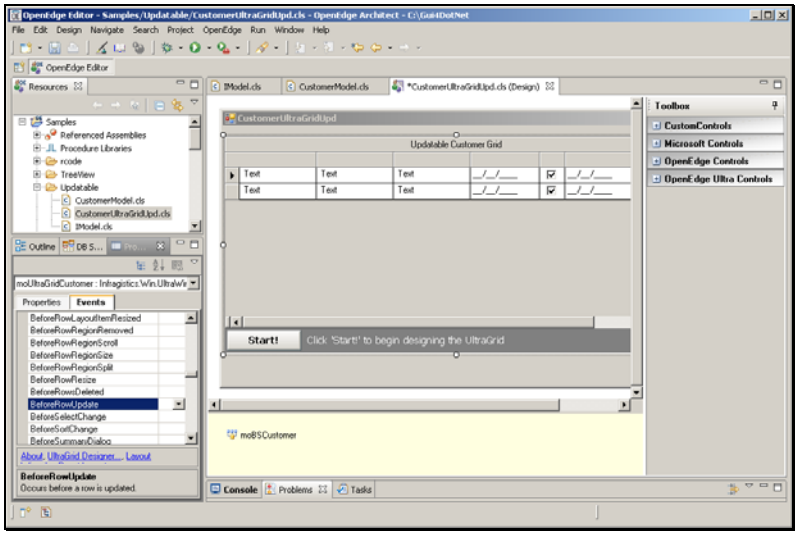
That's a start to the changes to the Model class to handle the data management side of the update. Now let's look at the Form where we'll also make changes, which once again is based on a form used in the earlier data binding examples. If I select its **ProBindingSource** control, I can show you a few more of its properties. The **AllowEdit** property, for instance, determines whether updates through the binding source are enabled or not. You can also enable individual controls in the user interface, but this property lets you control updates programmatically from the binding source. It's true by default, so updates are enabled. but here I set it to false to see how it affects the user interface.



I save the form with that property setting, and run the form to see what happens to my grid. If I select a cell and try to type into it, nothing happens. So the **ProBindingSource** has effectively disabled any controls that are bound to it, which can be a more effective way to manage updates than enabling and disabling individual controls. But it's clear that I want to keep **AllowEdit** set to **True**, so I reset the property to its default.

Another property of the binding source that looks interesting is **AutoUpdate**. This does what the name implies: it will automatically do the binding source **Assign** for you, which in my case would write changes back to the temp-table. But it does not do any error checking or validation or any of the work to get changes back to the actual data source, so it's not recommended except for quick prototyping. That's why it's **False** by default, and I leave it that way.

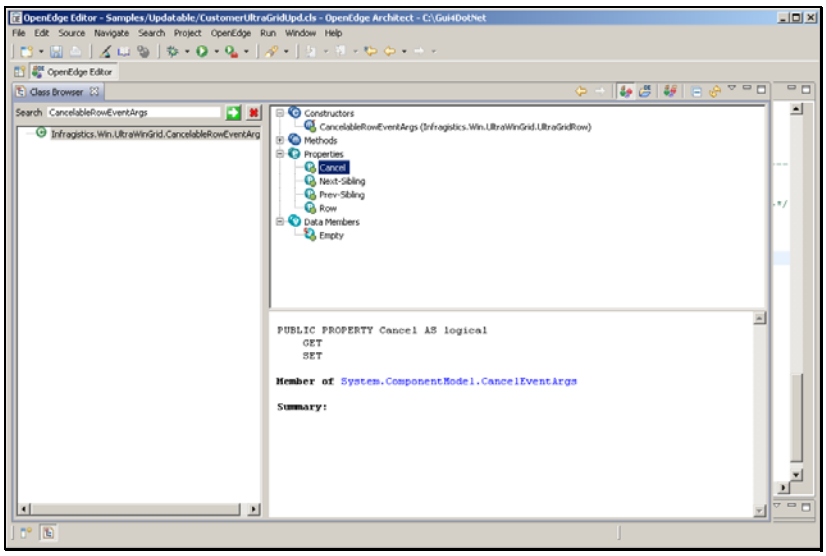
Next let me look at the events on the grid to see what I want to intercept to handle updates. The **UltraWinGrid** supports a whole host of events that you can subscribe to. Most of the names are pretty self-explanatory, and you can learn more about them from the Infragistics documentation. The one I want is **BeforeRowUpdate**, whose meaning should be pretty clear.



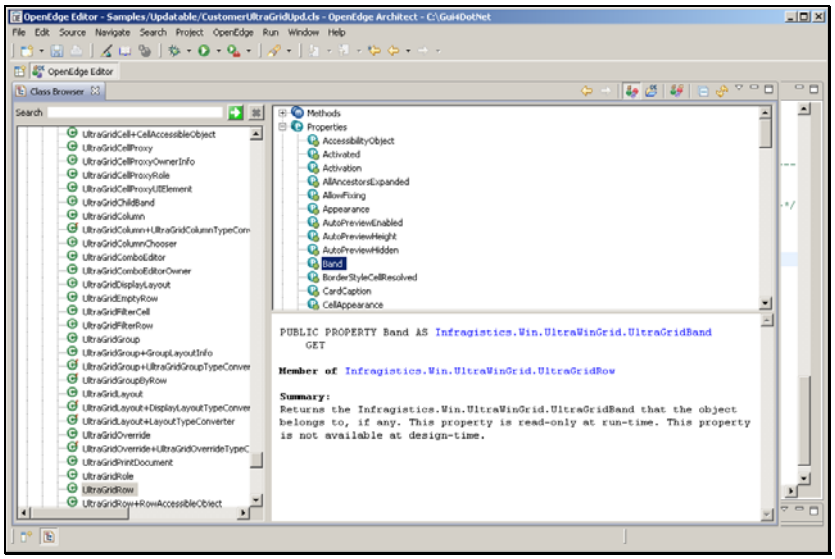
Double-clicking on that event, I get a handler for it. And in the code I see it takes an event args subclass called **CancelableRowEventArgs**.

```
@VisualDesigner.
METHOD PRIVATE VOID moUltraGridCustomer_BeforeRowUpdate
  ( INPUT sender AS System.Object,
    INPUT e AS Infragistics.Win.UltraWinGrid.CancelableRowEventArgs ):
```

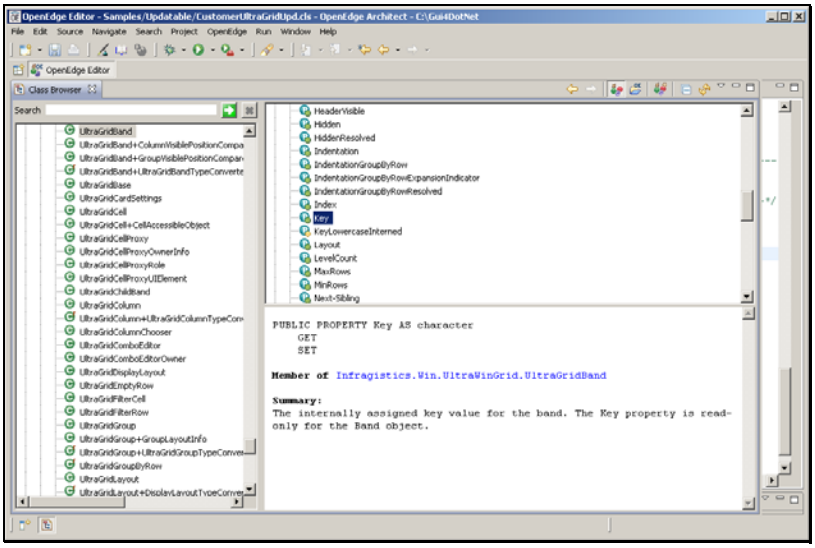
Let me find out what I can about that. In the **Class Browser**, I enter the name of the event args class, and look at its properties. There's a **Cancel** property, first of all, which is of type **Logical**, so I know that if I set it to **True**, then the update will be cancelled.



There's also a **Row** property, and if I drill down into that, and look through *its* properties, I see that it holds a reference to the **Band** in the grid where the selected row is.



And again, looking through the properties, I see a **Key** property.



This is often used to hold the name of the data item that's displayed. For instance, the **Key** property of a **GridColumn** holds the name of the field displayed there, which can be useful. Here it's the name of the buffer whose fields are displayed in this band. That will be useful to me in my event handler.

So I've learned a little about how to manipulate the event args parameter to **BeforeRowUpdate**: I need that buffer name, because I want to be able to tell **SaveData** what buffer was updated, in case there's more than one. And I learned from the **Class Browser** how to drill down into the event args to get the **Row**, then the **Band** for the **Row**, and the **Key** value for the **Band**. That's the buffer name;

```
METHOD PRIVATE VOID moUltraGridCustomer_BeforeRowUpdate
 ( INPUT sender AS System.Object,
   INPUT e AS Infragistics.Win.UltraWinGrid.CancelableRowEventArgs ):

 DEFINE VARIABLE cBuffer   AS CHARACTER NO-UNDO.
 DEFINE VARIABLE cCustName AS CHARACTER NO-UNDO.

  cBuffer = e:Row:Band:Key.
```

There might be circumstances where an event fires when no actual changes were made to a row, so I check the binding source **RowModified** property:

```
  IF moBSCustomer:RowModified THEN
  DO:
```

This property will be true if anything in the row was changed, and it will stay true as long as the grid is positioned to that row. Next comes the key step in the update through the binding source. When I invoke its **Assign** method, the changes in the grid – which you can think of as being like the screen buffer in older ABL terms – are transferred to the underlying record buffer, in this case, the temp-table row that the data came from. If there are any errors in that **Assign**, for instance if the UI let the user type in a value of the wrong data type into a cell, then the **Assign** will fail, so I check for an error return from **Assign**, set the **Cancel** property in the event args that I learned about in the **Class Browser**, and leave the update block:

```
      IF NOT moBSCustomer:Assign() THEN
      DO:
          /* Invalid data was entered in the grid row */
          e:Cancel = TRUE.
          LEAVE.
      END.
```

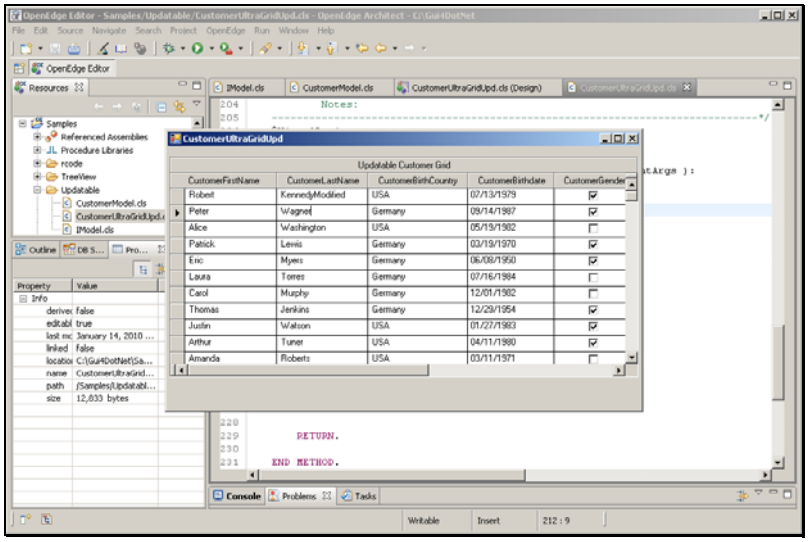
But if the **Assign** succeeds, my changes have been moved to the temp-table, so that the code in the Model can validate them.

This part of the sequence is important. If there were update format masks you wanted to define in the UI itself to help determine what the user enters, to make sure that an integer is entered into an integer field, for instance, or to define a dropdown list of possible values, that's one thing, but you wouldn't want anything else in the way of business logic to execute in the View. In the case of my simple validation of **CustomerBirthCountry** being USA or Germany, I could have defined a grid cell dropdown that I then populated with those selection values. But ABL code that does calculations or validations that constitute real business logic doesn't belong there. My validation check in the **CustomerModel** class is really a placeholder for business logic that would normally execute on the server-side in a distributed application.

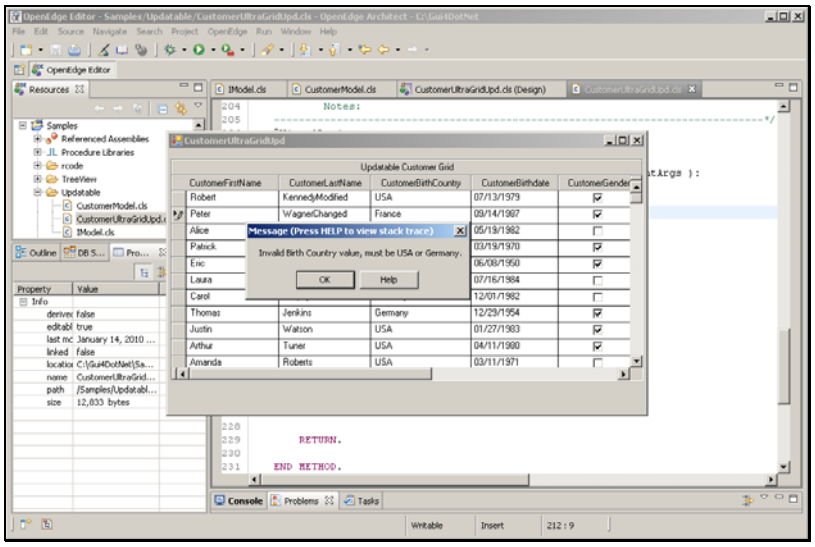
The key thing is that the **Assign** has transferred the updated values to the Model's temp-table, without the UI itself knowing anything about the underlying data storage, and the Model can then execute its validation without knowing anything about the UI. In any case, if **SaveData** returns an error, then I set **Cancel** just as before.

```
IF NOT moCustomerModel:SaveData(cBuffer) THEN
DO:
    e:Cancel = TRUE.
    LEAVE.
END.
```

I can now save this and run it, select a cell, make a change to it, and select another row. Anything I do to leave a row causes **BeforeRowUpdate** to fire, so my update has been executed. There's no visible confirmation of that, though, which I will address later.



Next I change another name, and then tab to the **CustomerBirthCountry** cell, and enter an invalid value. When I leave the row, I see the error that came back from **SaveData**, which is checking the values that were assigned to the temp-table.

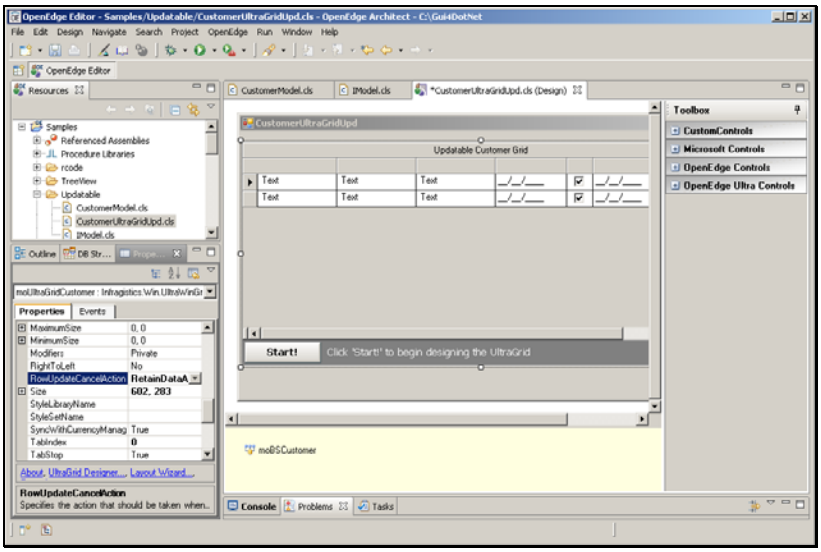


By default the **Cancel** just erases my changes altogether, which is not very friendly, so that's something else I'll improve on before I'm done. I have at least confirmed that a basic update goes through, and that **SaveData** has written it back to the data source, the database table.

[The following part of the document corresponds to the second part of the video presentation.]

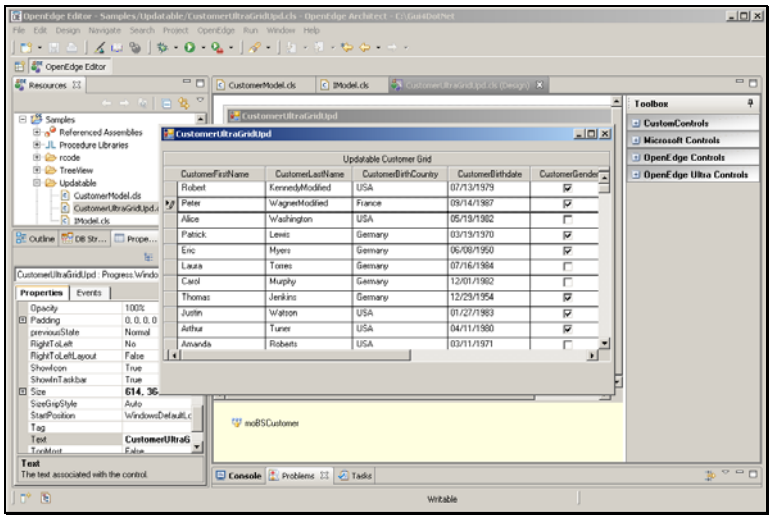
Since there are a few things that I don't like about how my sample grid behaves so far, I'm going to make some changes to show a few of the user interface options that can help improve the form's usability, as well as more of the ProBindingSource properties and methods.

One issue is that if I enter invalid data in a row, such as France for the CustomerBirthCountry name, I get an error message, but then lose my changes when the code sets the **Cancel** property. I could write some ABL code to make it behave differently, but if I select the grid in Visual Designer and look through its properties, I see that there is one called **RowUpdateCancelAction**, which looks like a perfect description of my situation. It's defined in terms of an enumeration, and if I drop down its list of values, I see that the default is **CancelUpdate**, which is what I saw happen, but the other choice is **RetainDataAndActivation**. That's exactly what I want to have happen: to leave the changes in place and leave me on the row with the error – that's what Activation means, that the row is still active – so that I can see and either correct the error or press Escape to cancel the changes myself.



This is a perfect example of both the value of these .NET controls, especially the UltraControls, and also the challenge of using them: On the one hand, there are many properties, methods, and events to sort through to find what you need to solve a problem. On the other hand, there's almost always built-in support for what you need the control to do, so it's worth your while getting to know the controls through the documentation and just reviewing the names of everything the controls support, which are usually clear in telling you what they do. Naming conventions are consistent enough that with a little experience you will get a sense for what to expect and what to look for in a control that's new to you.

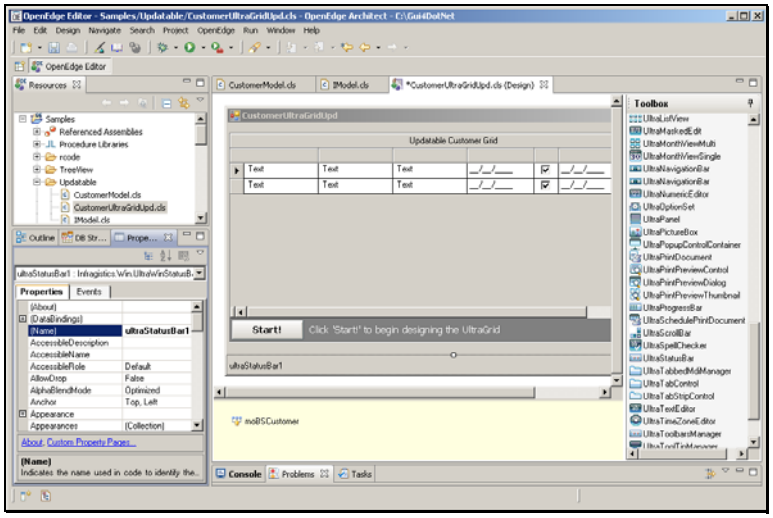
If I run the form again with **RowUpdateCancelAction** changed to **RetainDataAndActivation**, I can make a change to a Customer, and then change the CustomerBirthCountry to an invalid value. Now I see the error message as I did before, but my changes haven't been canceled:



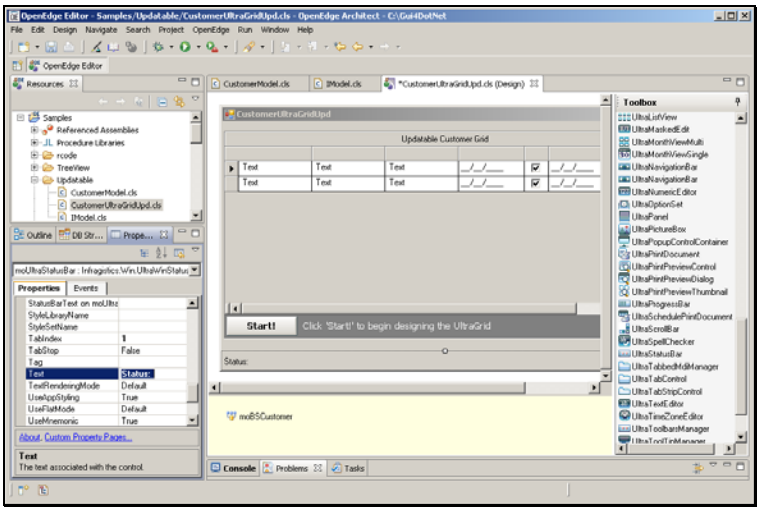
The row in error is still the current row, as indicated by the row edit icon over on the left, so I can click in the cell where the error is, and correct it, and then move out of the row and save all my changes. That's much more user friendly. I try another row, and once again enter an invalid CustomerBirthCountry, and once again see the error message. It's a characteristic of the UltraGrid that I can press **Escape** to cancel an update, so I have that option to go through with the Cancel myself.

I present this as an example of how you can expect that the .NET controls will provide countless variants of behavior that you can capture and take advantage of. You just need to be thorough enough to find and use them.

The next issue I had with my updatable grid was that there was no visible feedback when an update succeeded. As one example of how I can deal with that, I'll add a status bar to the form to display an update status. Here among the Ultra controls is a **StatusBar** that I can drop onto the bottom of the form.



I rename it to be consistent with the other control names I've used. Without going into all the rest of its properties, I initialize its **Text** property to "Status: ", to make that the value initially displayed.



Now I need to add a couple of lines of code to assign a value to display after an update. At the beginning of the **BeforeRowUpdate** event handler. I reset the **Text** property back to its initial value, so that it's reset each time an update happens:

```
DEFINE VARIABLE cBuffer   AS CHARACTER NO-UNDO.
DEFINE VARIABLE cCustName AS CHARACTER NO-UNDO.
    cBuffer = e:ROW:Band:Key.
    moUltraStatusBar:TEXT = "Status: ".
```

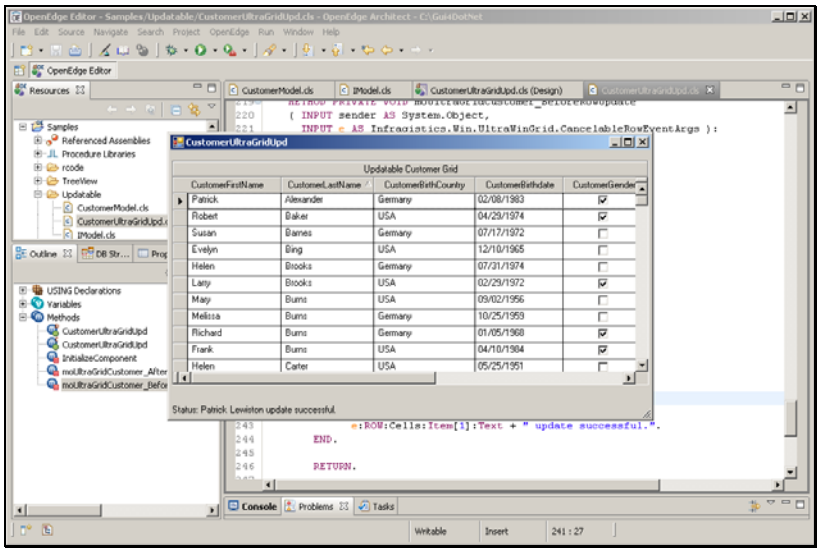
Then at the end of the method I construct a string that displays the first and last name fields:

```
    cCustName = e:ROW:Cells:Item[0]:Text + " " +
                e:ROW:Cells:Item[1]:Text.
```

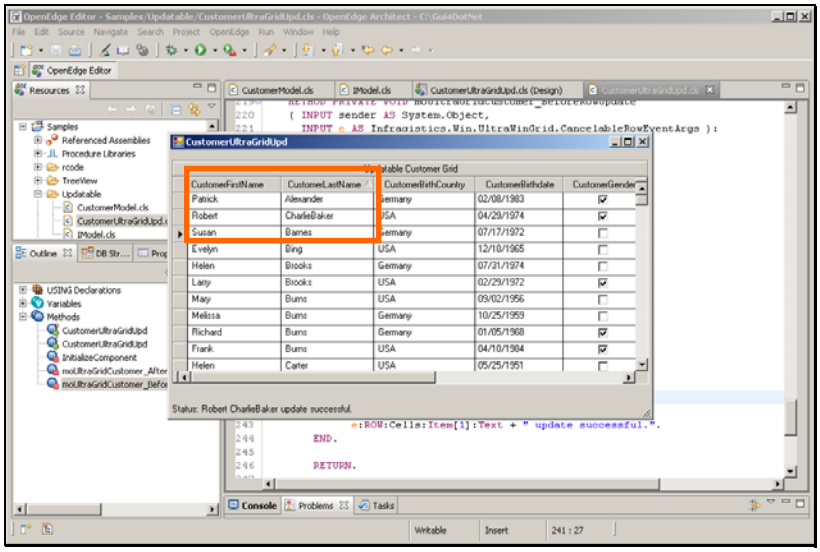
This is worth taking a look at for a moment. Remember that the method's **CancelableRowEventArgs** parameter has a **Row** property. The **Row** contains a **Cells** property, which is a .NET collection of all the

cells in the row. The standard way to access an element of a collection is with the **Item** property, which takes a zero-based index to the cell you want. The **Text** property of each cell is the value it contains. So **Item[0]** and **Item[1]** are the first two columns in the grid, the CustomerFirstName and CustomerLastName fields.

To see how my status bar looks, I can rerun the form, click on a row, change the name, and change the CustomerBirthCountry to a valid value. When I select another row, the **BeforeRowUpdate** handler fires, and I see the status message.



So this is just a simple example of integrating another Ultra control into a form. While I'm running the form and changing values, I can show you another aspect of the update that you could consider an issue. If I click on the CustomerLastName column header, the **SortData** method that I coded in the presentation series on sorting data with the ProBindingSource re-opens the temp-table query, sorted by CustomerLastName. Now if I select a customer, and change the CustomerLastName to a value that should sort somewhere else, the change is saved, but the data isn't resorted, because I didn't ask to reopen the query.

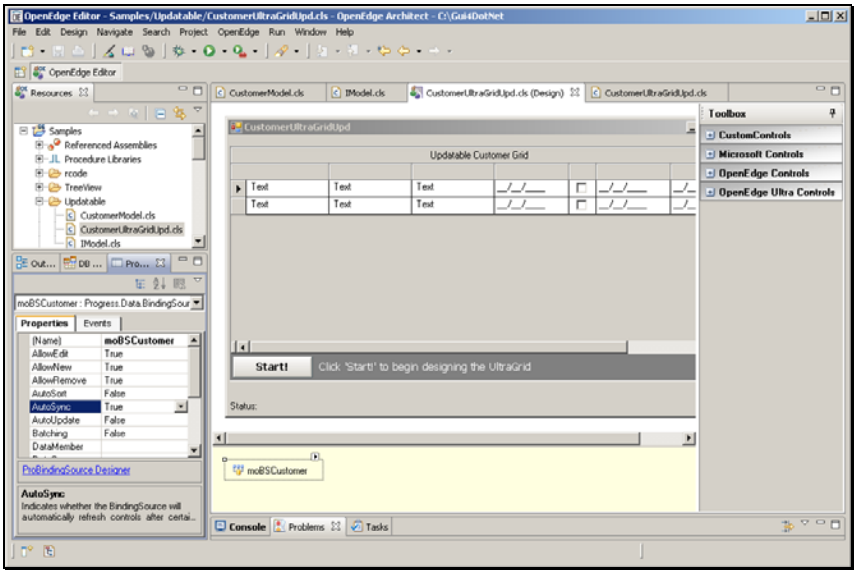


Showing you how to make the resort happen will illustrate another part of the interaction between the user interface, the binding source, and the underlying data. Back in the **CustomerModel** class, I want to make a change to the **SaveData** method. If the save operation succeeds, I want to re-open the current query,

with whatever sort order is defined for it. That way, the data will be resorted properly if I make a change that affects a row's sort sequence in the grid.

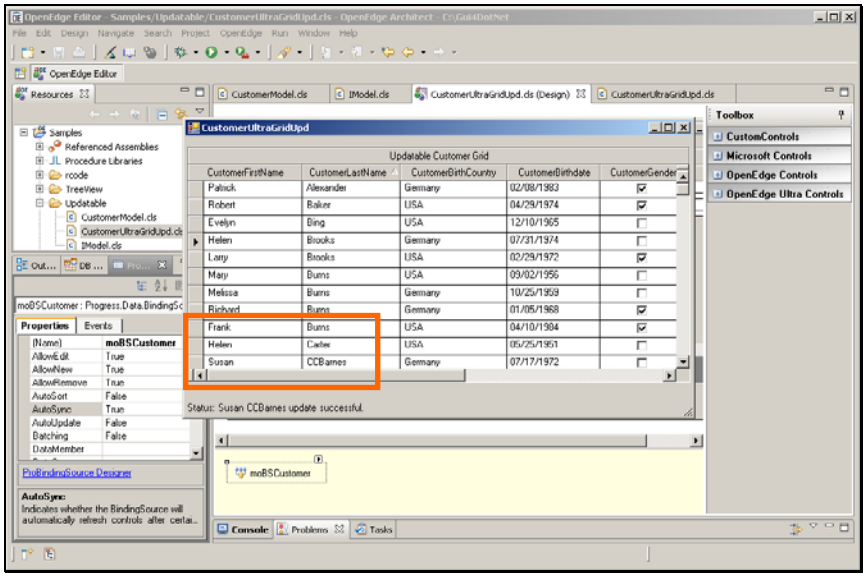
```
/* If we get here all client-side validation succeeded. */
IF hBeforeBuffer:SAVE-ROW-CHANGES () THEN
DO:
    hBeforeBuffer:ACCEPT-ROW-CHANGES ().
    httCustQuery:QUERY-OPEN ().
    RETURN TRUE.
END.
```

Looking at the ProBindingSource properties again makes it clear what's making the resynchronization of data between ProBindingSource and the UI work. The property that plays a key role here is **AutoSync**.



It's **True** by default, which is what supports the behavior I've been showing you. If **AutoSync** is **True**, then any time you re-open a query bound to a ProBindingSource, or use one of the ABL **REPOSITION** statements to change the selected row in the query, the binding source **Position** property is reset automatically, and any user interface controls bound to the binding source are also refreshed to stay in sync with the data. This is probably the behavior you want almost all the time, but if you ever want to control when data gets refreshed yourself, you can set **AutoSync** to **False** and use the binding source **Refresh** method.

To take a look at the result of the query re-open, I save and re-run the form, re-sort the data by CustomerLastName, select a row, and make a change that changes its sort position.



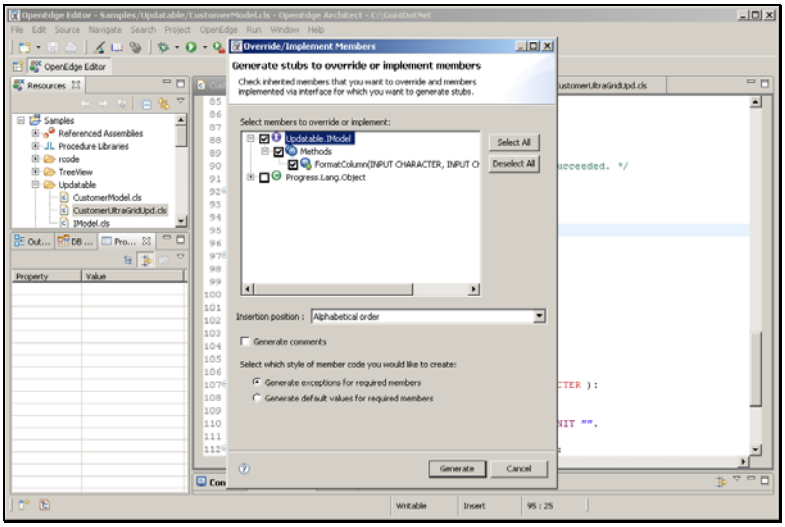
Leaving the updated row, the query is re-opened and all the data re-synchronized with the grid.

Lastly, I'll show you a simple example of using the ProBindingSource **Refresh** method. To do that, I need a new method in the model, so I first add it to the interface. The new method, FormatColumn, takes a column name and its value as parameters, and re-formats the value under certain rules. If the method returns true, then the value has been re-formatted and needs to be re-displayed. That's where **Refresh** comes in.

```
INTERFACE Updatable.IModel:
  METHOD PUBLIC VOID FetchData (INPUT pcFilter AS CHARACTER ).
  METHOD PUBLIC VOID SortData  (INPUT pcSort   AS CHARACTER ).
  METHOD PUBLIC HANDLE GetQuery().
  METHOD PUBLIC LOGICAL SaveData(INPUT pcBufferName AS CHARACTER ).
  METHOD PUBLIC LOGICAL FormatColumn (INPUT pcColumnName AS CHARACTER,
      INPUT pcColumnValue AS CHARACTER).

END INTERFACE.
```

From the Source menu, I add the skeleton code for the new method:



This is the code for the method:

```
METHOD PUBLIC LOGICAL FormatColumn( INPUT pcColumnName AS CHARACTER,
                                    INPUT pcColumnValue AS CHARACTER ):

        CASE pcColumnName:
            WHEN "CustomerBirthCountry" THEN
            DO:
                IF pcColumnValue = "usa" THEN /* in whatever capitalization */
                DO:
                    ttCustomer.CustomerBirthCountry = "USA".
                    RETURN TRUE. /* Force refresh */
                END.
            END.
        END CASE.
        RETURN FALSE. /* No refresh needed. */

END METHOD.
```

If the CustomerBirthCountry column has a value of “usa” with any type of capitalization, the code for that case forces it to be all upper case, and returns **TRUE** to signal that the data value has been changed. Remember that because this is code in the Model, it’s the temp-table value I’m changing, which has no direct effect on the UI.

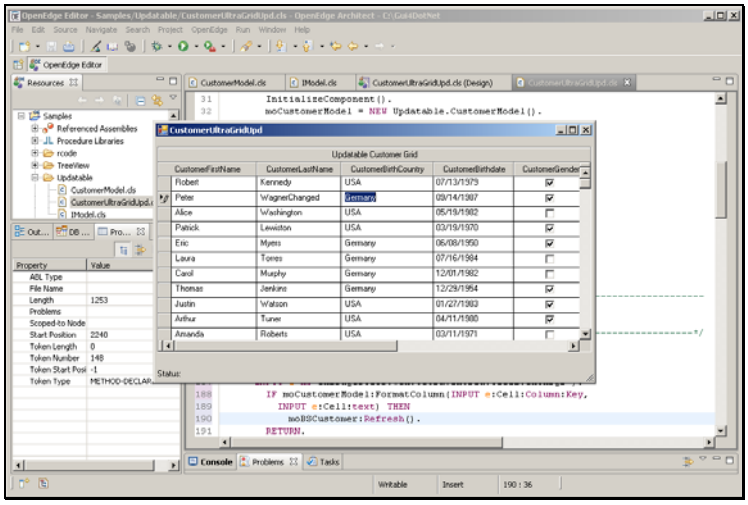
So back in the form I need to invoke **FormatColumn** and check its return value. Looking again through the **UltraGrid** events, I find one named **AfterCellUpdate** that fires after a cell has been updated. A little experience will teach you where to look in these very full-featured controls for support for the behavior you need. The event handler for **AfterCellUpdate** takes an event args class of **CellEventArgs**, and if I were to look that up in the Class Browser, I’d see that, not surprisingly, it has a **Cell** property. The **Column** property of the **Cell** points to the column the cell is in. I have used the Key property once before to get at the buffer name for a band in the grid. Here the **Key** property holds the column name, so that becomes the first parameter to **FormatColumn**. The cell also has a **Text** property, which holds the column value, so that becomes the second parameter. And if **FormatColumn** returns **True**, it has modified the field’s value in the temp-table:

```
METHOD PRIVATE VOID moUltraGridCustomer_AfterCellUpdate(
  INPUT sender AS System.Object,
  INPUT e AS Infragistics.Win.UltraWinGrid.CellEventArgs ):
        IF moCustomerModel:FormatColumn(INPUT e:Cell:Column:Key,
          INPUT e:Cell:Text) THEN
            moBSCustomer:Refresh().
        RETURN.

END METHOD.
```

This is where I invoke the ProBindingSource **Refresh** method to redisplay the row. Because the code hasn’t re-opened the query or re-positioned it, I have to do the **Refresh** myself.

To test this latest change, I save and re-run the form, select a customer, change the CustomerLastName, and then change the CustomerBirthCountry to “usa”, without using all capitals:



When I leave the cell, the **AfterCellUpdate** event fires to run the **FormatColumn** method and refresh the displayed value. That's the last of the changes I'll make to this demonstration of the basics of updating data using the ProBindingSource.

Let's quickly review a few of the things I've shown you in this two-part session:

- Use the **TRACKING-CHANGES** property of a ProDataSet to allow you to **FILL** the DataSet and then keep track of changes to its contents.
- The **SAVE-ROW-CHANGES** method saves changes back to the source database table. **ACCEPT-ROW-CHANGES** marks those changes as being accepted in the dataset itself, and **REJECT-ROW-CHANGES** removes them.
- In the ProBindingSource, the **AllowEdit** property lets you manage whether UI controls that are bound to it are enabled for input or not.
- The **AutoUpdate** property is there just to help you do quick testing of updating data, but should be left **False** for serious development.
- Use the ProBindingSource **RowModified** property to check whether a row in the UI has actually been changed, and the **Assign** method to write changes back to the buffer in the Model that is the binding source's data source.
- Remember that the binding source **AutoSync** property allows an automatic refresh of data when the underlying query is re-opened or repositioned.
- When you need to synchronize a change that doesn't reposition the query, use the ProBindingSource **Refresh** method to push the change out to the UI.
- Use control events like **BeforeRowUpdate** and **AfterCellUpdate** to capture changes that you need to write back to the Model.
- Learn about useful control properties like **RowUpdateCancelAction** to take advantage of built-in behavior that you want in your user interface.

That concludes this two-part session on managing data updates with the ProBindingSource.