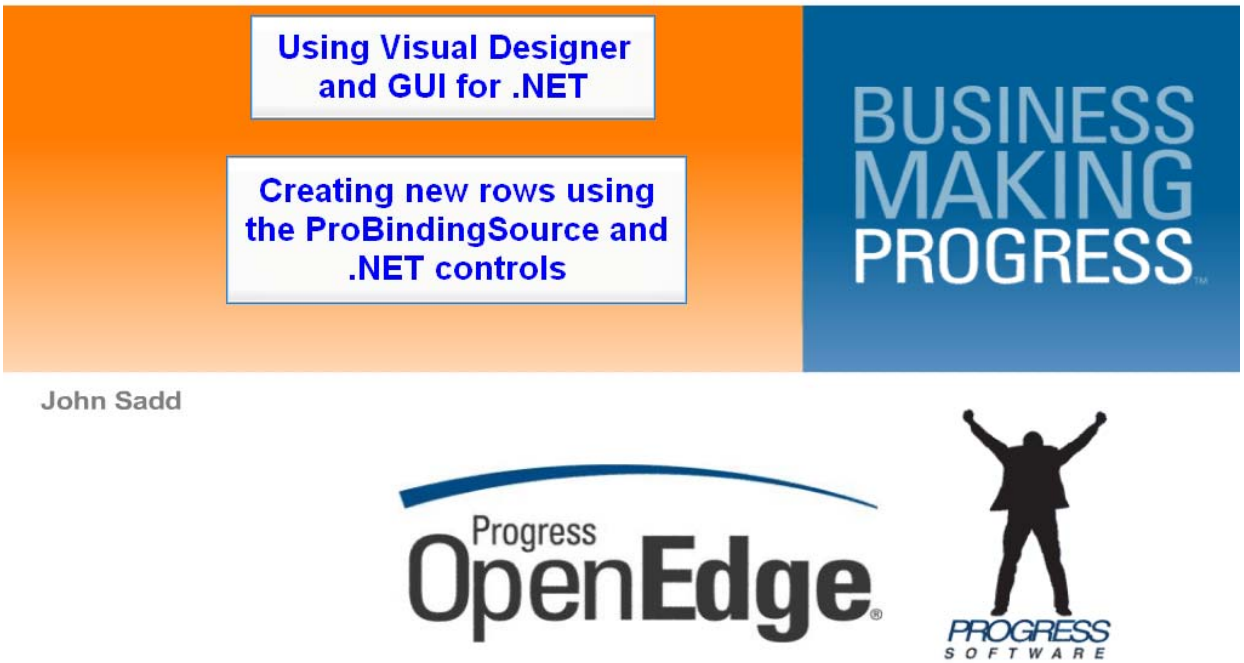
CREATING NEW ROWS WITH THE PROBINDINGSOURCE AND .NET CONTROLS

John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
March 2010

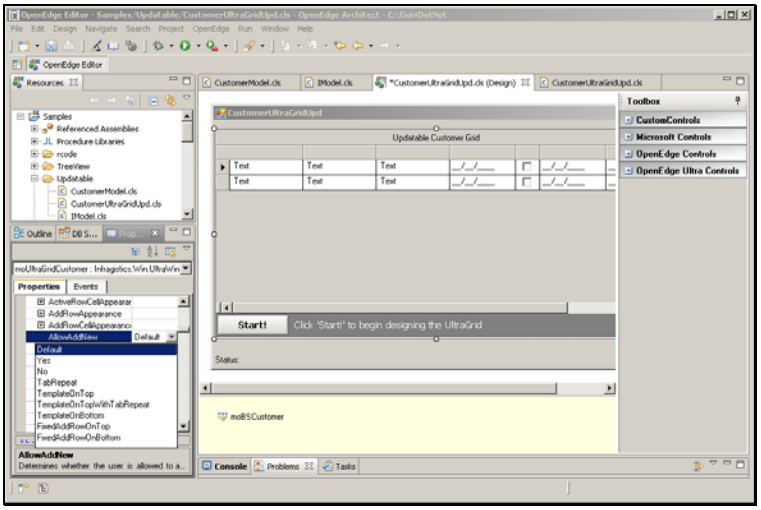


DISCLAIMER

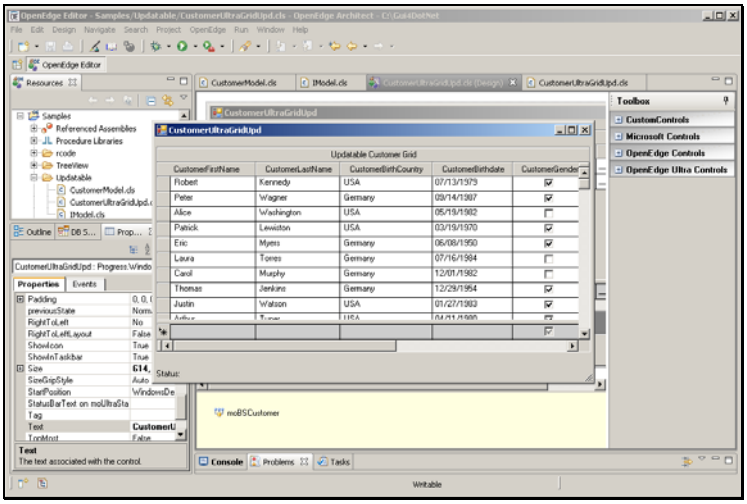
This document accompanies the video presentation that continues the series on updating data using the ProBindingSource. This session shows how to use binding source events to add new rows to a table whose data is displayed in an UltraGrid control in a .NET user interface.

Along with the **AllowEdit** property that determines whether data entry into UI controls will be allowed, If you look at the ProBindingSource properties in the Properties View of Visual Designer, you see that there's a property called **AllowNew** that you can set in the binding source to have it manage whether UI controls that are bound to it will allow a request to create a new row. As with **AllowEdit**, it's **True** by default, so it doesn't need to be changed for this session's example, but you'd want to set it to **False** for a read-only user interface or one that supports updates but not creates.

The other side of supporting creates in the user interface is a key property in the **UltraGrid** that needs to be set to enable the user interface to respond to a request for a new row. This is under **DisplayLayout**, and then **Override** in the **Properties View** for the **UltraGrid**. The property is called **AllowAddNew**. As the name implies, this determines whether – from the grid control's perspective – the user can request a new row through the control, and also how that new row is represented in the grid.



As you can see, there are a number of different options in its enumeration. You can learn about these in the Infragistics documentation, but one that presents a reasonable UI affordance for inserting new rows is **FixedAddRowOnBottom**. Like other Infragistics names and enumeration values, this one states pretty clearly what it does. If you save and compile the form with this property setting, and run the form, you can see below that there's a permanent overlay at the bottom of the grid that lets you request a new record just by clicking in this special row. This seems like a reasonable UI that lets you add a row independent of where it will wind up sorting in the grid itself.



You could type in values for a new row here and save it, but that won't do anything meaningful yet. There is some measure of default behavior, but as with the **AutoUpdate** property, it won't get your new row back to the database or execute any other initialization code, so it's necessary to go back to Visual Designer to add some supporting code to provide the behavior needed.

The object-oriented approach to building an application really requires you to work from the bottom up in your design, which is a good discipline, so this means going back to the Interface that defines all the methods a Model class uses in data management. Here I add a method definition named **CreateModelRow** for the code that will handle a request for a new row in any class that manages data. It passes in a buffer name, in case the Model manages data in more than one table.

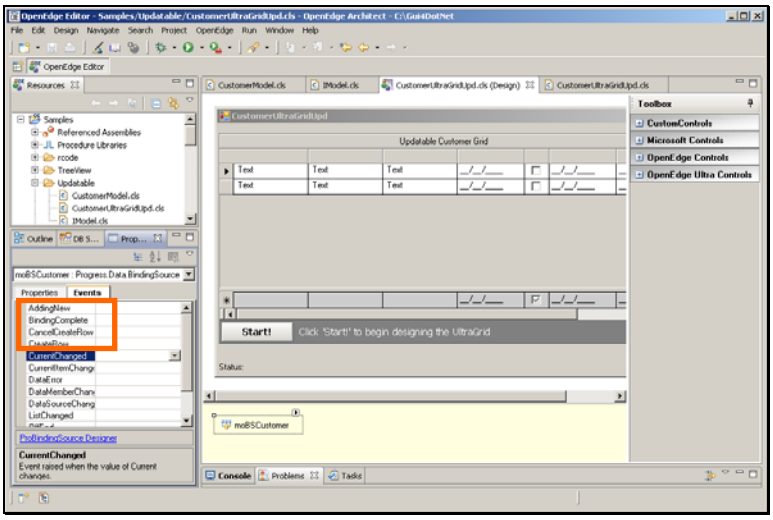
```
INTERFACE Updatable.IModel:
  METHOD PUBLIC VOID FetchData (INPUT pcFilter AS CHARACTER ).
  METHOD PUBLIC VOID SortData  (INPUT pcSort   AS CHARACTER ).
  METHOD PUBLIC HANDLE GetQuery().
  METHOD PUBLIC LOGICAL SaveData(INPUT pcBufferName AS CHARACTER ).
  METHOD PUBLIC LOGICAL FormatColumn (INPUT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

END INTERFACE.
```

And now I can implement that method in my one example Model class. As before, from the **Source** menu, I can select **Override / Implement Methods** to get the matching skeleton code generated for me. The binding source is coordinated with its underlying data source to make the method's implementation very straightforward.

```
METHOD PUBLIC LOGICAL CreateModelRow( INPUT pcBufferName AS CHARACTER ):
  DEFINE VARIABLE cCustomerID AS CHARACTER NO-UNDO.

    DO TRANSACTION ON ERROR UNDO, LEAVE :
        CREATE ttCustomer.
        ASSIGN ttCustomer.CustomerFirstName          = "First Name"
               ttCustomer.CustomerLastName           = "Last Name"
               ttCustomer.CustomerBirthCountry       = "USA"
               ttCustomer.CustomerBirthdate          = 1/1/1950
               ttCustomer.CustomerGender             = TRUE /* = "male" */
               ttCustomer.CustomerLicenseDate        = 1/1/1950
               ttCustomer.CustomerLicenseExpiryDate = 1/1/1950
               ttCustomer.CustomerID                 = GUID.  /* Unique char ID */

        httCustQuery:CREATE-RESULT-LIST-ENTRY ().
    END. /* transaction */
    RETURN TRUE.  /* No checks for errors yet. */
END METHOD.
```

First the code creates the new row in the temp-table – the same would work for a database table, but I'm illustrating a separation of the client-side model from the database itself. Then the method assigns whatever initial values are needed. This could include calculating a line number or any other value for the user. In fact, you can see that one line uses the **GUID** built-in ABL function to generate that long character string that the AutoEdge database uses as a unique key field in every table. Finally, the method just executes the ABL **CREATE-RESULT-LIST-ENTRY** function on the query that the binding source is using. This works very neatly, because when you add a new row through a ProBindingSource, it automatically positions the query to the new row being added, so **CREATE-RESULT-LIST-ENTRY** adds it to the end of the query without having to reopen it. For this reason, you shouldn't do anything fancier than this – don't try to explicitly reposition the query or the binding source; just do what is shown here and it will all work properly.

Next we have to go back to the ProBindingSource and its events. First let me point out an event you *don't* want to use.



The first one in the list here is called **AddingNew**, which sounds like exactly what you would want to use to create a new row. But remember that our OpenEdge-specific ProBindingSource is an extension of a standard .NET binding source object, so it inherits that object's events and properties. Some of those are useful, and some don't really apply to an OpenEdge application. **AddingNew** is one of those that don't apply, because if you look at its **EventArgs**, you'll see that it has a **NewObject** property which is an instance of any System.Object, appropriate for dealing with object-oriented data, perhaps, but not the relational data in your database. We decided not to hide these inherited events and properties from you, and under certain very limited circumstances they might be useful, but generally you should stick to the ones that support OpenEdge applications, as they're described in the OpenEdge documentation and materials such as this.

The event you *do* want to use to handle an add request is **CreateRow**. Let's take a look at its parameters. As always, the ClassBrowser will give us a quick overview of the properties that get passed in. And that's of type **CreateRowEventArgs**.

There are several properties you can use in your event handler. The first is **BandIndex**. If your data is in a grid that supports multiple display bands that display data from different tables, this integer property gives you the zero-based index into the list of bands. That's one way you can identify which table the request is for.

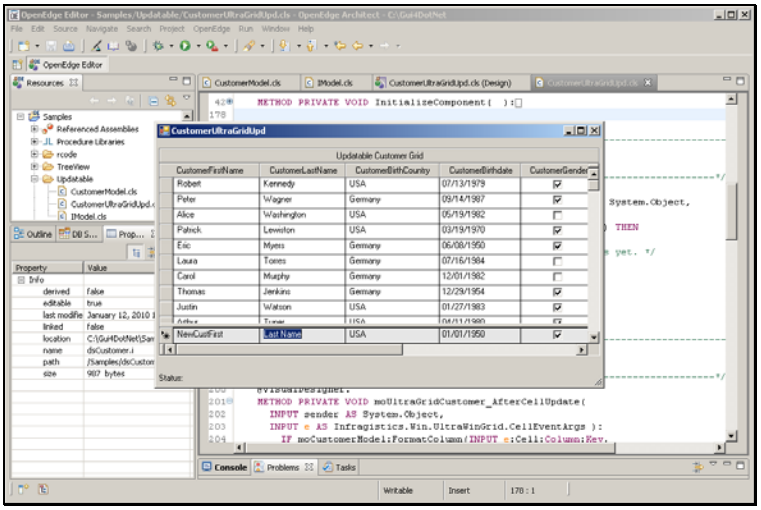
Alternatively, there's a **BufferHdl** property that holds the handle to the buffer for the table, and a **BufferName** property that holds its name. You could use either of these as well.

The last property to look at is **Created**, a logical flag that you set to signal whether the underlying create in your temp-table or database table succeeded or not. This tells the binding source whether to tell the UI controls to cancel the create or not on their end. So all the event handler method has to do is invoke the

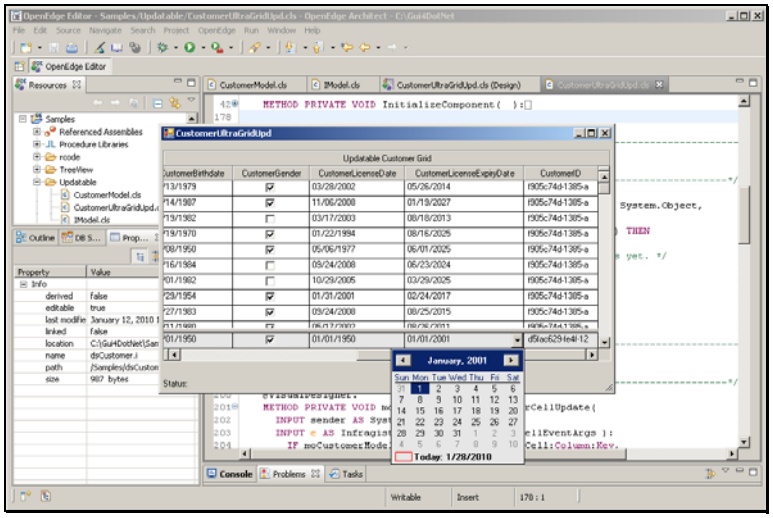
Model's **CreateModelRow** method. The call passes in the **BufferName** property value as a placeholder, though remember that in my case I have only one buffer. and if that method returns **True** the caller sets the **Created** property to **True** to signal success. Note that this simplified example doesn't check for specific errors.

```
METHOD PRIVATE VOID moBSCustomer_CreateRow( INPUT sender AS System.Object,
    INPUT args AS Progress.Data.CreateRowEventArgs ):
        IF moCustomerModel:CreateModelRow(INPUT args:BufferName) THEN
            args:Created = TRUE.
        ELSE args:Created = FALSE.  /* No real checks for errors yet. */

        RETURN.

END METHOD.
```

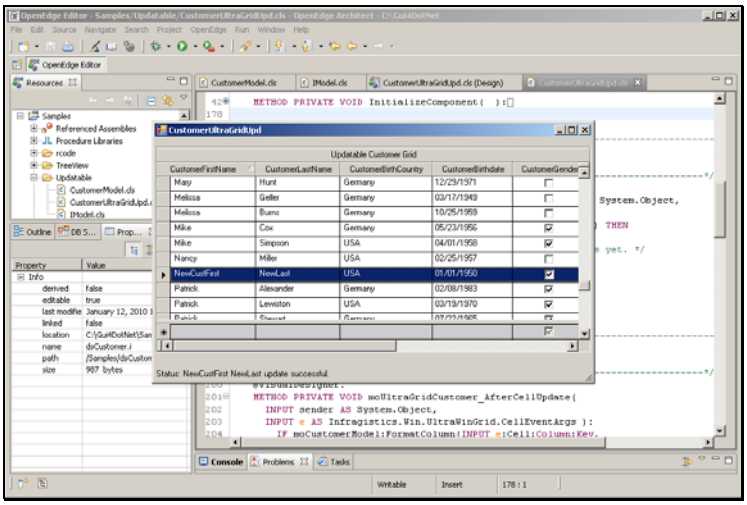
Now all the code is in place to try out the support for new rows. If run the form again and click in the row that's reserved for new row creates, the first thing I see is that the initial values are automatically transferred by the binding source up to the grid to display. I can select any of the grid's cells and override the default value.



If I scroll across the cells, you can see in the next screenshot the rest of the initial values, including the **GUID** object id field. I can change another value such as one of the dates, and see the calendar control that is automatically applied to a grid column that's bound to a field defined as a date.



Selecting another row causes the **BeforeRowUpdate** event to fire just as it does for any other update, and that saves the changes I made to my new row back to the temp-table and from there to the database table. You can see the update message here in the **StatusBar**, and if I re-sort the data by clicking on a column header, I can scroll down and find my new row in its correct sort position.



So the support for adding new rows through the grid requires just a few lines of code.

Next I need to look at the code to allow me to cancel the create of a new row. If I were to start to enter a new row and then press Escape to cancel it, the binding source isn't able to do everything automatically that I might need in order to remove the row I no longer want, so I need to put in explicit code to handle that event.

First I add another method to the **IModel** interface to handle a cancel of a new row create, and call it **CancelCreateModelRow**. The new method uses the same buffer name parameter used in the **CreateModelRow** method:

```
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).
  METHOD PUBLIC LOGICAL CreateModelRow (INPUT pcBufferName AS CHARACTER ).
  METHOD PUBLIC LOGICAL CancelCreateModelRow (INPUT pcBufferName AS CHARACTER ).
END INTERFACE.
```

In the **CustomerModel** class, I then implement that method. The code needed is very simple, and parallel to the statements in **CreateModelRow**. It explicitly deletes the temp-table row that corresponds to the row in the grid, the same row that **CreateModelRow** just created. And to remove the row from the result list, it uses the query's **DELETE-RESULT-LIST-ENTRY** method, returning **True** to signal that the delete succeeded.

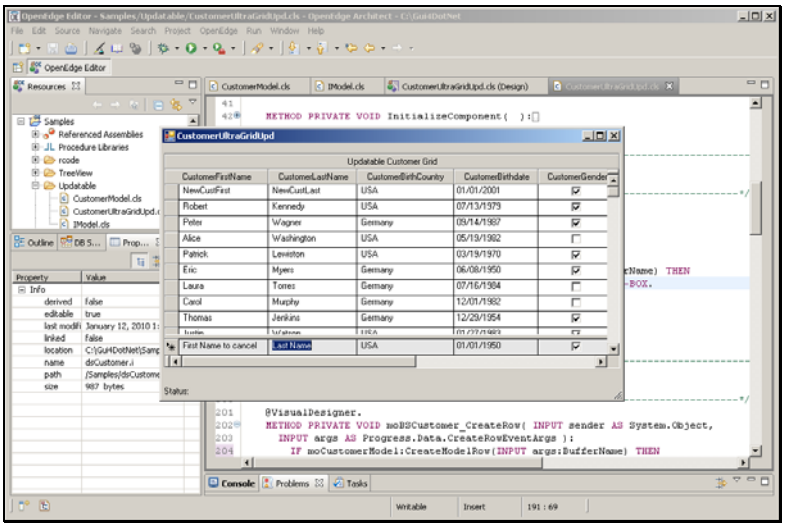
```
METHOD PUBLIC LOGICAL CancelCreateModelRow( INPUT pcBufferName AS CHARACTER ):

        DELETE ttCustomer. /* That's the only buffer. */
        httCustQuery:DELETE-RESULT-LIST-ENTRY ().
        RETURN TRUE.

END METHOD.
```

Next I need to return to the form and create an event handler for **CancelCreateRow**. Double-clicking on the event in the **Events** tab of the Visual Designer **Properties View** generates the event handler method skeleton code. Inspecting the parameters that are passed in, I see the **EventArgs** class is another subclass called **CancelCreateRowEventArgs**. This supports the same **BandIndex**, **BufferHdl**, and **BufferName** properties, just as **CreateRowEventArgs** does, but of course no **Created** property, because the create of the row is exactly what I'm canceling out of.

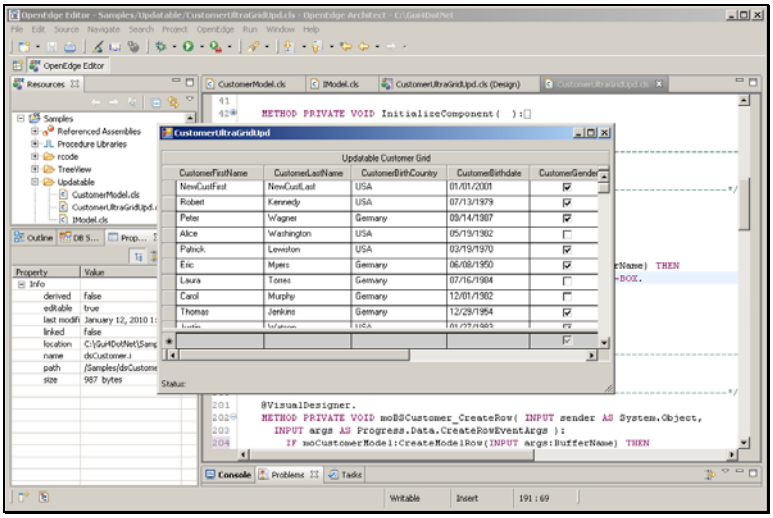
Without adding any special code for unexpected conditions, the event handler just invokes the new cancel method in the model, and display a message if for some reason it doesn't succeed.

```
METHOD PRIVATE VOID moBSCustomer_CancelCreateRow(
    INPUT sender AS System.Object,
    INPUT args AS Progress.Data.CancelCreateRowEventArgs ):

        IF NOT moCustomerModel:CancelCreateModelRow (args:BufferName) THEN
          MESSAGE "Row cancel did not succeed." VIEW-AS ALERT-BOX.

        RETURN.

END METHOD.
```

To try this out, I rerun the form and start the add of a new row. Remember that as soon as I click in this special row at the bottom of the grid, the row is added to the temp-table. I can enter a few values before I decide this isn't what I wanted to do.



The way the **UltraGrid** works is that the first time you press **Escape** any change to the value for the current cell is undone. And the next time you press **Escape**, the whole row is undone. This is when the **CancelCreateRow** event fires. So after pressing **Escape** twice, the newly added row is gone, not just from the new row line in the grid, but from the temp-table as well.



If I had completed the add operation and saved the row, then of course I would have needed support for a row delete operation to undo it at that point. And that is the subject of the next session in this series on ProBindingSource support for row deletes.