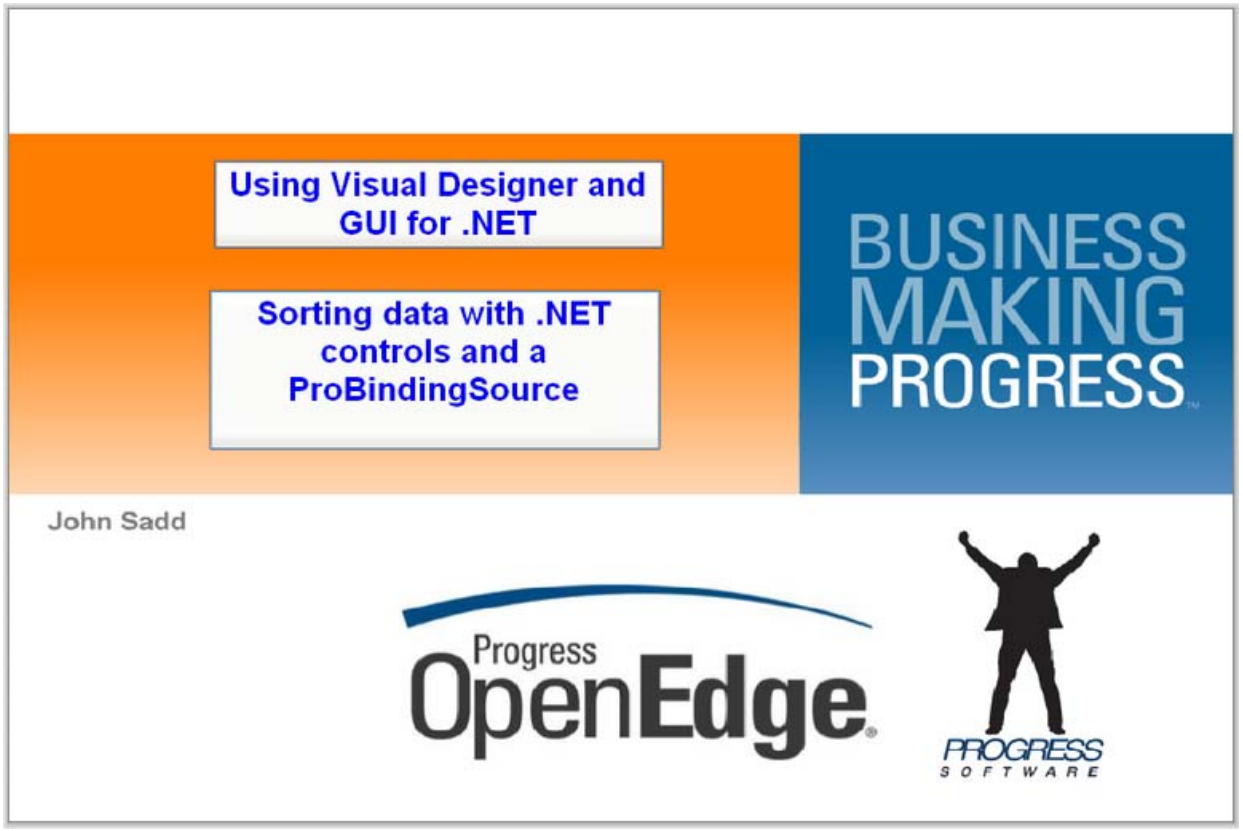
SORTING DATA WITH A PROBINDINGSOURCE AND .NET CONTROLS

John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
December 2009



DISCLAIMER

This document accompanies a series of presentations on sorting data in your .NET user interface using the ProBindingSource control together with Visual Designer and .NET controls. First I put together the supporting code to show some of the capabilities and also some of the issues around sorting data in the user interface as opposed to sorting in the data source. I create a new ABL Interface .cls file for a set of related classes, create a class that uses the Interface, and create a form with an Infragistics grid and a ProBindingSource to manage and display some data.

[This part of the document corresponds to the video presentation that is Part 1 of the Data Sorting series:]

Starting with the usual sequence of **File -> New**, I select **ABL Interface**. An interface is a special type of .cls file that defines a contract that other classes agree to adhere to by implementing the interface. In this case this is a very simple start on a Model to allow me to manage and provide access to different data sources in a consistent way. Typically interface names start with I, so this is IModel. I can enter a Description for what its role is, and when I click finish Architect generates the skeleton of a source file named IModel.cls.

You can see the generated ABL identifies this as an `INTERFACE` rather than a `CLASS`:

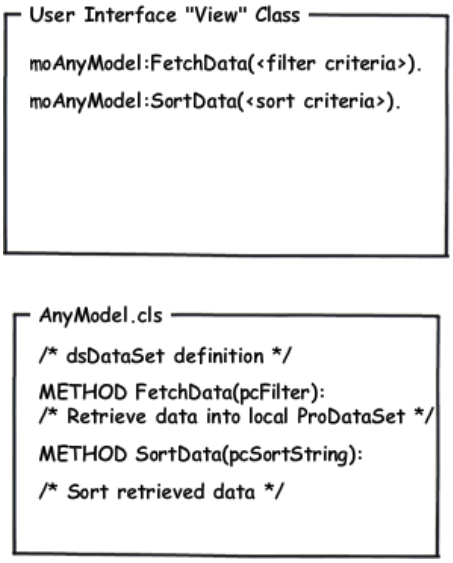
```
/*------------------------------------------------------------------------
    File        : IModel
    Description : Interface for Model classes
  ----------------------------------------------------------------------*/

USING Progress.Lang.*.

INTERFACE IModel:

END INTERFACE.
```

And what I enter in ABL is in effect prototypes of methods that any class that implements this interface must provide. The method definition just identifies the name and signature.

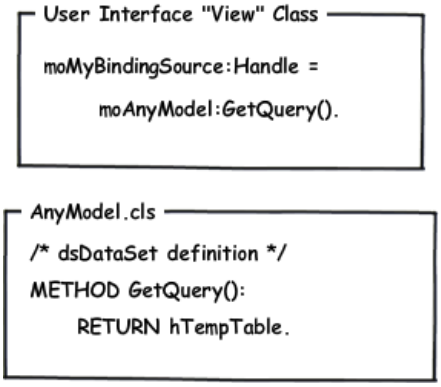
```
USING Progress.Lang.*.

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

END INTERFACE.
```

The first one is **FetchData**, which tells the Model instance to populate itself with some set of data of the type the Model class manages. I just allow for an optional filtering string as an argument.

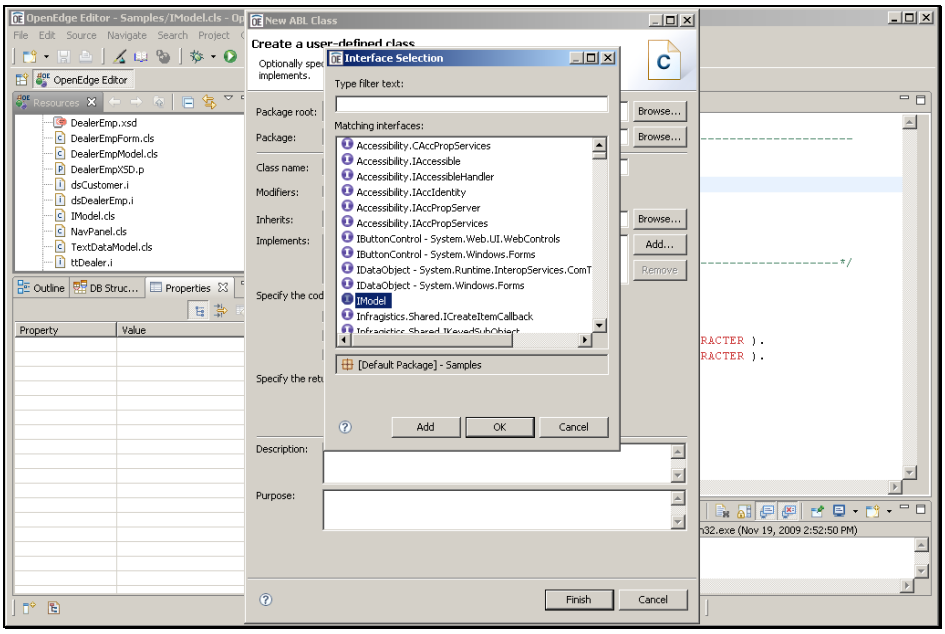
Because this session is about sorting data, I define a separate method **SortData** to communicate sort criteria to the model. This illustration shows how the FetchData and SortData methods in a Model class are expected to be referenced from a user interface View class:



And the class needs to be able to provide access to the data that's been retrieved, in this case the handle to a query on the data, so the method **GetQuery** has a return type of `HANDLE`:



An interface can have definitions for methods, properties, and events. The only restriction is that they all need to be `PUBLIC`. The Interface represents the public contract that a class adheres to so that other classes can reference it consistently and reliably. I save and compile the interface, and then create a new **ABL Class** to use the interface. This is going to be the first class that adheres to the contract defined by the interface, in this case to provide access to Customer data. If I want to specify one or more interfaces that the class implements, I select the Add button next to the interface section:



My new Interface **IModel** is in the list along with all the others that are part of the support for Microsoft .NET controls and Infragistics controls and so forth. In addition I check the boxes in the **Create a user-defined class** wizard that generate a default skeleton for a constructor to execute when the class is run, and a destructor where I could put code to clean up when the class instance is deleted. When I click Finish I can take a look at the code that's been generated for me.

```
/*------------------------------------------------------------------------
    File        : CustomerModel
    Purpose     : Retrieves Customer database data into a ProDataSet
  ----------------------------------------------------------------------*/

USING Progress.Lang.*.

CLASS CustomerModel IMPLEMENTS IModel:
```

You can see that the `CLASS` statement tells the compiler that this class `IMPLEMENTS` the interface **IModel**.

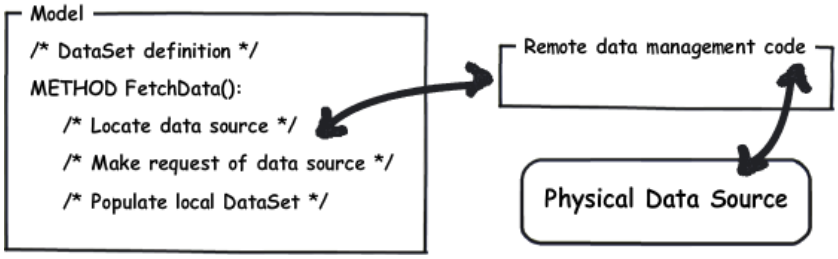
The first element I add to the file is a reference to a ProDataSet that the Model uses to hold data. Now of course data could be retrieved and managed in a number of ways, which is one reason to keep the Model class that knows the details separate from the UI, but this is one specific example of how the interaction can be handled, using a temp-table and dataset definition such as this.

```
    {dsCustomer.i}
```

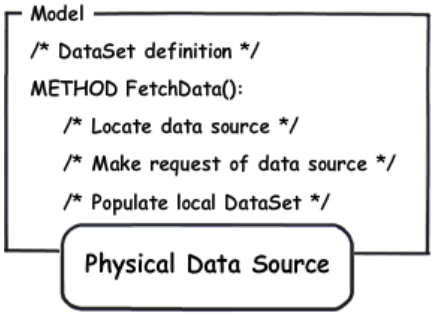
The Model as the top of the data management stack on the client is responsible for interacting with wherever and whatever the actual data source is, and in this case getting the data into the DataSet that the definition in **dsCustomer.i** represents:

```
/*------------------------------------------------------------------------
    File        : dsCustomer.i
    Description : Temp-Table and DataSet definition for the AutoEdge
                  Customer table.
  ----------------------------------------------------------------------*/

DEFINE TEMP-TABLE ttCustomer
    FIELD CustomerFirstName         AS CHARACTER
    FIELD CustomerLastName          AS CHARACTER
    FIELD CustomerBirthCountry      AS CHARACTER
    FIELD CustomerBirthdate         AS DATE
    FIELD CustomerGender            AS LOGICAL
    FIELD CustomerLicenseDate       AS DATE
    FIELD CustomerLicenseExpiryDate AS DATE
    FIELD CustomerID                AS CHARACTER.

DEFINE DATASET dsCustomer FOR ttCustomer.
```

In a complete application there would be a separation between a Model class on the client that holds data locally for the user interface to use, and classes or procedures on the server where the database is connected, as illustrated here:



In this case I leave the basic code that does all the data management work right here in this one simplified class, as shown here:



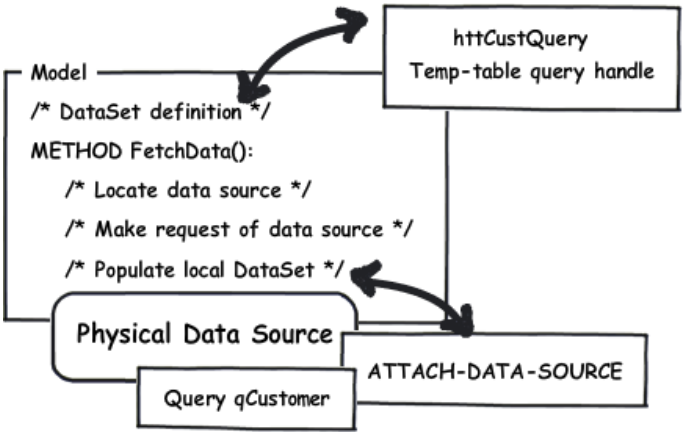
I need a query definition for the actual data source, an ABL `DATA-SOURCE` definition for the DataSet to use, and a variable to hold the handle of a query on the temp-table that the data gets loaded into, to supply back to the user interface.

```
  DEFINE VARIABLE httCustQuery AS HANDLE NO-UNDO.
  DEFINE QUERY qCustomer FOR AutoEdge.Customer.
  DEFINE DATA-SOURCE srcCustomer FOR QUERY qCustomer.
```

Next I need a few statements in the constructor to get the object set up when it's instantiated. So I need to attach the `DATA-SOURCE` to its temp-table, and then set up a dynamic query on that temp-table which is the connecting point to the binding source in the user interface.

```
CONSTRUCTOR PUBLIC CustomerModel (  ):
        SUPER ().
        BUFFER ttCustomer:ATTACH-DATA-SOURCE (DATA-SOURCE srcCustomer:HANDLE).
         CREATE QUERY httCustQuery.
         httCustQuery:SET-BUFFERS(BUFFER ttCustomer:HANDLE).
         httCustQuery:QUERY-PREPARE ("FOR EACH ttCustomer").

END CONSTRUCTOR.
```

This sketch shows the distinction between the query on the database table associated with the DataSource, used to populate the DataSet, and the query on the DataSet's temp-table of data after it's been retrieved.



Next you can see the effects of the default code generation I get from Architect by saying that this class implements **IModel**. Because I have to code an implementation of each of the methods defined in the interface, I'm provided with a starting point to remind me to fill each method in.

```
        METHOD PUBLIC VOID FetchData( INPUT pcFilter AS CHARACTER ):
                UNDO, THROW NEW Progress.Lang.AppError("METHOD NOT IMPLEMENTED").
        END METHOD.

        METHOD PUBLIC HANDLE GetQuery(  ):
                UNDO, THROW NEW Progress.Lang.AppError("METHOD NOT IMPLEMENTED").
        END METHOD.

        METHOD PUBLIC VOID SortData( INPUT pcSort AS CHARACTER ):
                UNDO, THROW NEW Progress.Lang.AppError("METHOD NOT IMPLEMENTED").
        END METHOD.
```

Without some skeleton implementation at least, I'd get a compiler error if I tried to save and compile **`CustomerModel.cls`**, because the compiler is cross-referencing this with the interface definition.

Here's the code that I write for **FetchData**:

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

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

It prepares a query for the DataSet to use to `FILL` the temp-table, and for simplicity's sake, just expects any Filtering information passed in to be a `WHERE` clause for the database query. In real life this would require translation from the user interface understanding of a filter and what the database would require, but this will do for now.

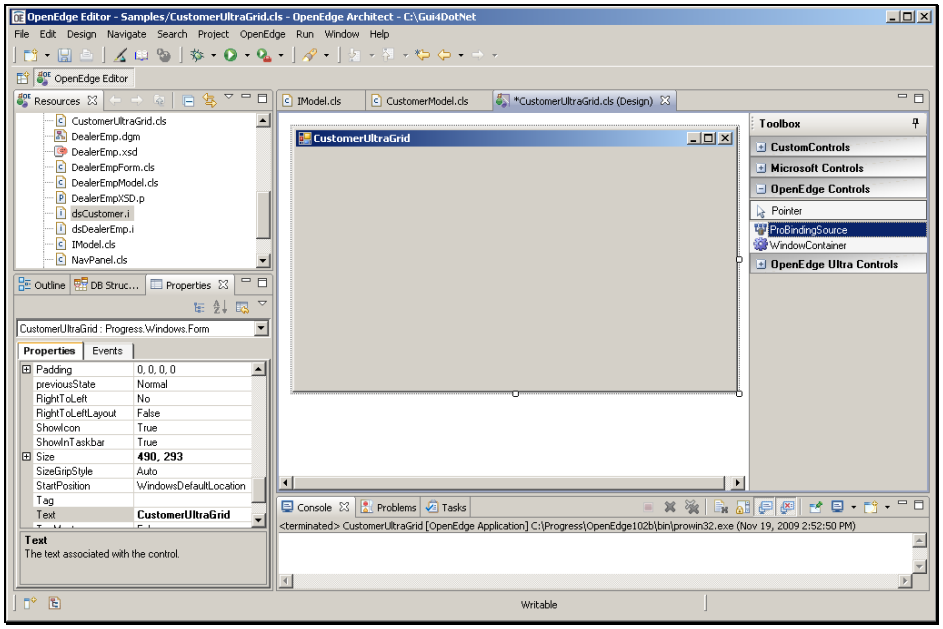
And likewise I have a code for the **GetQuery** method, which just returns the temp-table query handle from the DataSet.

```
METHOD PUBLIC HANDLE GetQuery(  ):

    /* This returns the handle of the ttCustomer query for the DataSet. */
        RETURN httCustQuery.
END METHOD.
```

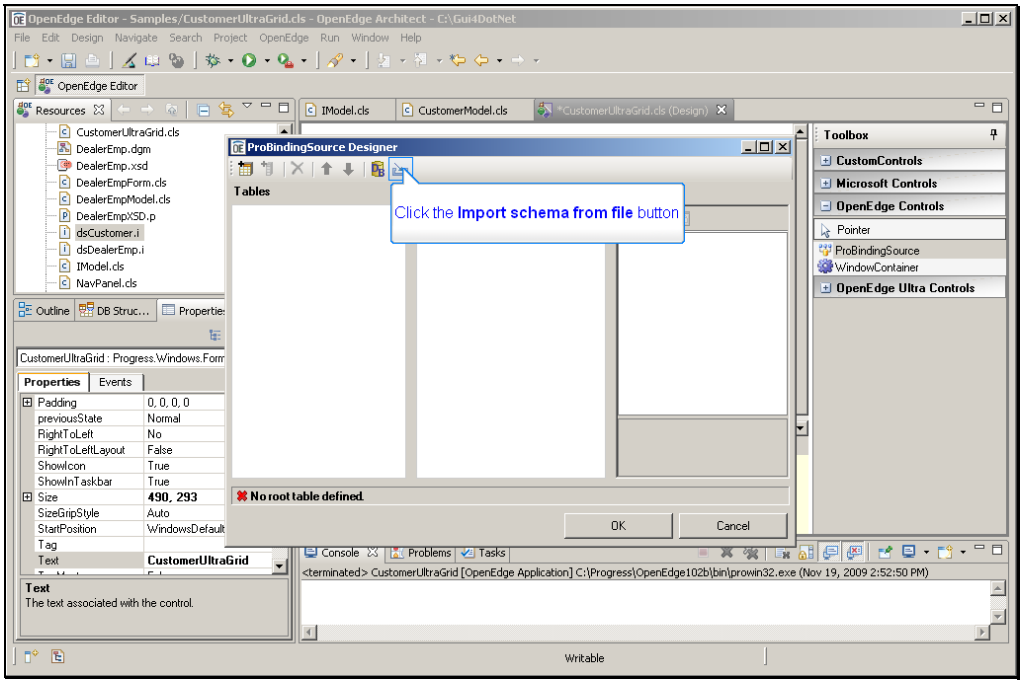
I haven't filled in code for **SortData** yet – that happens later – but because there's at least a skeleton implementation my syntax check is OK, and I can save the class and compile it.

Next I create a form to display the data in. This uses the Infragistics WinGrid control, which we call the **UltraGrid**. This advanced grid control has properties you can set to allow it to sort the data it displays on its own, so it gives me an opportunity to compare letting the user interface handle data sorting for display and having the data source handle it.

There are several alternatives for creating a binding source. In this case I select the **ProBindingSource** directly under the **OpenEdge Controls** in Visual Designer's Toolbox:



In the **ProBindingSource Designer**, I select **Import schema from file**:



This is prepared to use an XSD file, or an ABL source file containing a definition. I select the DataSet definition **dsCustomer.i** to use as a basis for the binding source.

Back in the Design View I want to give the binding source a meaningful name. Since this a definition that will appear in the main block of the class, above any method definitions, it represents what we call a "data member" of the class, so name it **moBSCustomer**, using the naming convention "m" for member and "o" for object, since the variable will hold a reference to a binding source object.

Next I open up the list of **Ultra Controls**, and drop an **UltraGrid** onto the form. In the Quick Start wizard, I set the grid's data source to be the binding source I just created, and that's all I need to set to get started.

As with the binding source, I want to give the variable that holds the object reference for the grid a meaningful name in my source code. That's the object's **Name** property, which I set to **moUltraGridCustomer**.

The **Text** property is the title displayed at the top of the grid, and I set that to **Customer Grid**.

Here in the generated code are the variable definitions for the variables to hold my two object references:

```
/*------------------------------------------------------------------------
    File        : CustomerUltraGrid
    Purpose     : UltraGrid to display and sort Customer data
  ----------------------------------------------------------------------*/

USING Progress.Lang.*.
USING Progress.Windows.Form.
USING Infragistics.Win.UltraWinGrid.*.

CLASS CustomerUltraGrid INHERITS Form:

    DEFINE PRIVATE VARIABLE components AS System.ComponentModel.IContainer NO-UNDO.
    DEFINE PRIVATE VARIABLE moUltraGridCustomer AS
        Infragistics.Win.UltraWinGrid.UltraGrid NO-UNDO.
    DEFINE PRIVATE VARIABLE moBSCustomer AS Progress.Data.BindingSource NO-UNDO.
```

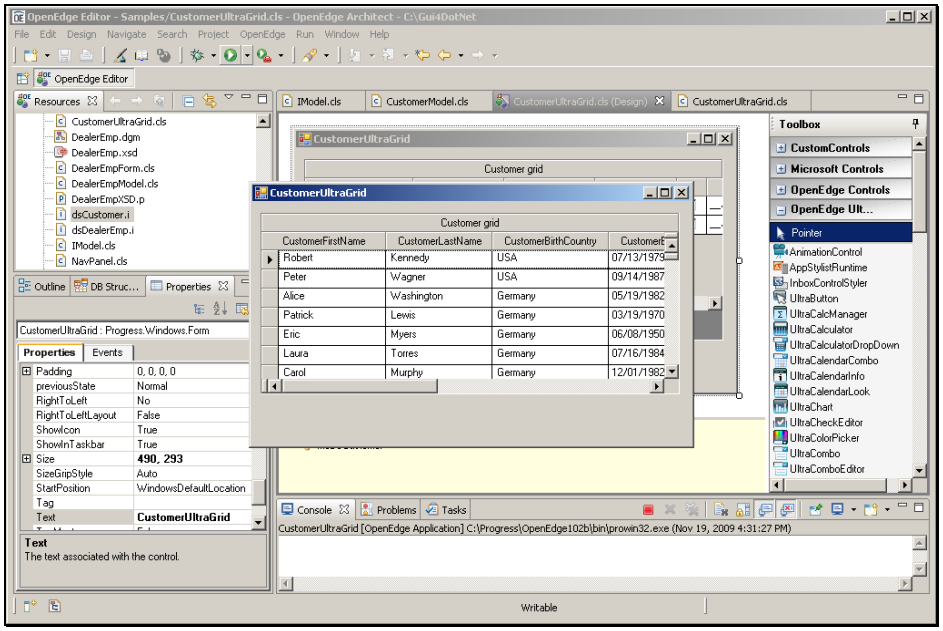
I need another variable to hold a reference to the **CustomerModel** instance that I'll use to provide data to my "View", the user interface class:

```
DEFINE PRIVATE VARIABLE moCustomerModel AS CustomerModel NO-UNDO .
```

Next in the class's constructor, I need to create that **CustomerModel** instance, and then tell it to retrieve all the Customers – for starters I'm not specifying any filtering criteria. And finally, because the binding source needs to have its **Handle** property set to a buffer, a query, or a DataSet, I retrieve the query handle that I set up in the model class.

```
CONSTRUCTOR PUBLIC CustomerUltraGrid (  ):

        SUPER().
        InitializeComponent().
        moCustomerModel = NEW CustomerModel().
        moCustomerModel:FetchData("").
        moBSCustomer:Handle = moCustomerModel:GetQuery().

        CATCH e AS Progress.Lang.Error:
            UNDO, THROW e.
        END CATCH.

END CONSTRUCTOR.
```

Now I save this much so that we can see what happens when I run it. Here's the form with my grid, and you can see it's successfully attached the binding source to the Customer data held by the model.



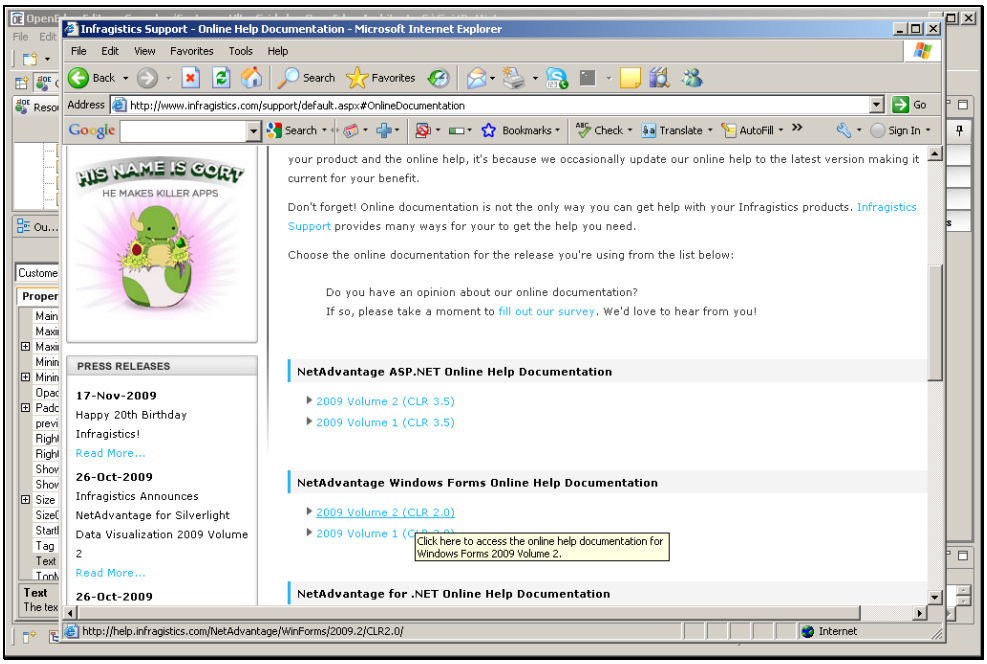
Now I know that somehow the **UltraGrid** is prepared to do data sorting for me, which seems like a nice feature, so I click on a column header, but nothing happens. So I've got to do some investigation to find out how to make this work for me.

[This next part of the document corresponds to the video presentation that is Part 2 of the Data Sorting series:]

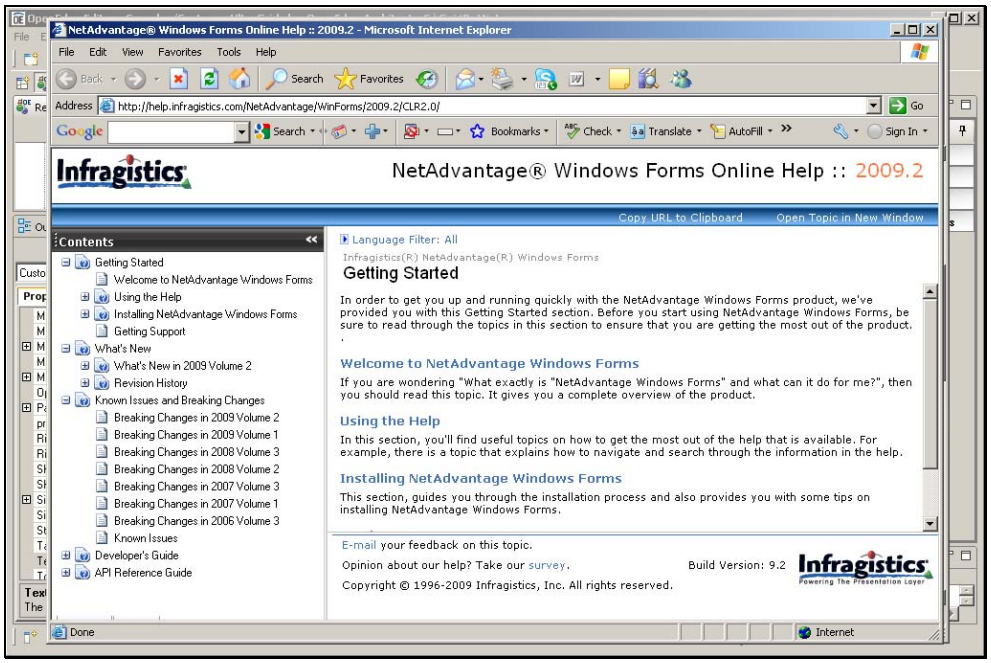
Here's a quick tour of the **Infragistics Online Documentation** as an example of how it can help you understand what properties to set and what code to write for the controls in a form such as the one I'm building, and then use that information in my form to get the grid to show the data in a different sort order than I used to retrieve it from the database. Note that in addition to the online links shown in these examples, the Infragistics documentation is installed locally when you install OpenEdge 10.2B along with the Infragistics controls. You can access it from the Windows Start menu under **Programs->Infragistics->NetAdvantage for .NET->Windows Forms->Documentation->Documentation (local)**.

When I first run the form I discover that the sorting capability of the **UltraGrid** control isn't enabled by default, so I need to investigate what properties of the control have to be set to enable it to sort data for me. And just as I used MSDN in other presentations for information on Microsoft controls, I can go to the Infragistics website to get documentation on all the .NET controls in the **UltraControls** package that are available with OpenEdge 10.2.

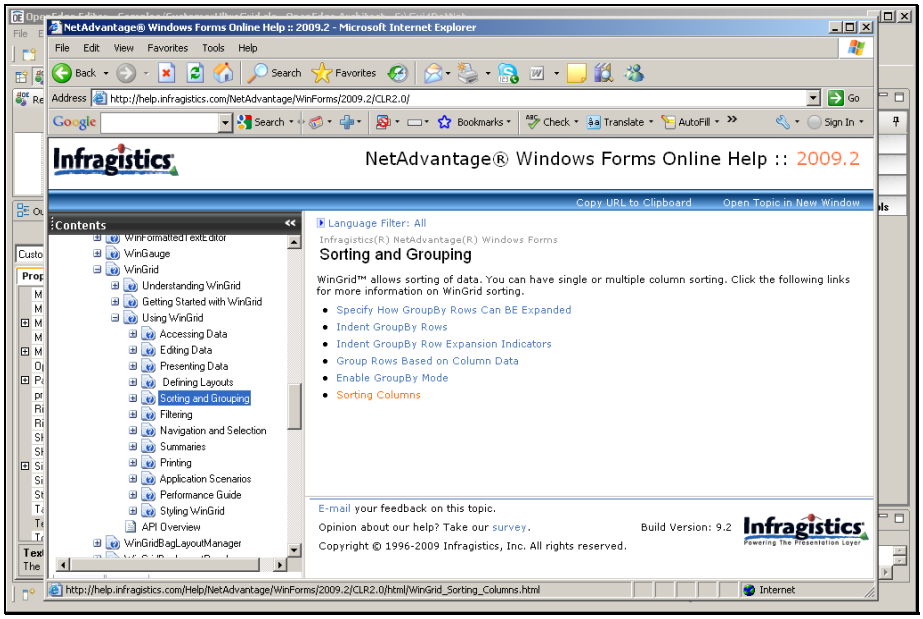
Under the **Support** menu at www.infragistics.com, I can select **Online Documentation**. The controls that are available with OpenEdge 10.2B are identified as **Volume 2 of Windows Forms 2009**, so I select that version to get information on.



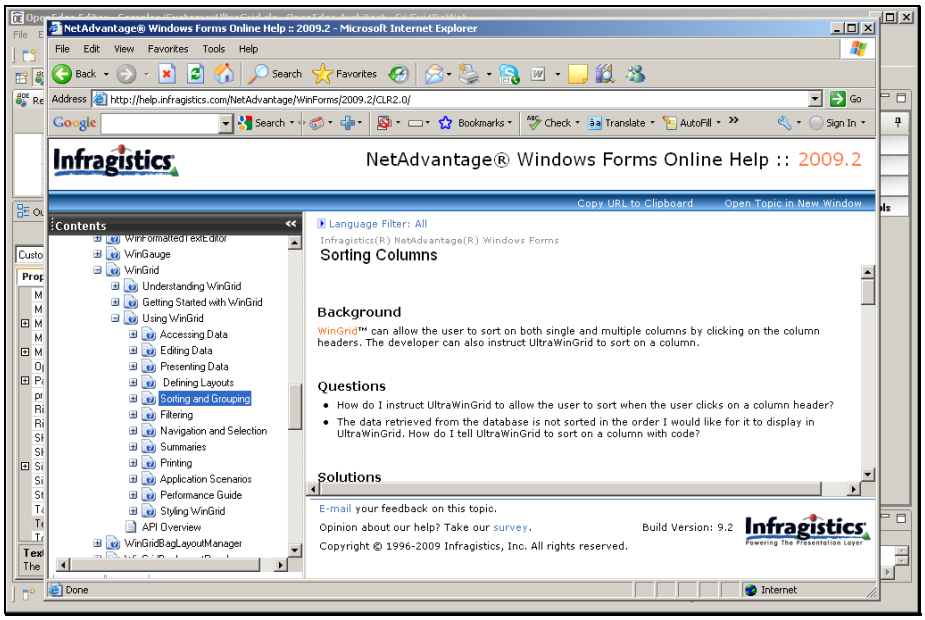
There's a lot of introductory material available that you can start with, but the two major sources of comprehensive information on all the controls are the **Developers Guide** and the **API Reference Guide**.



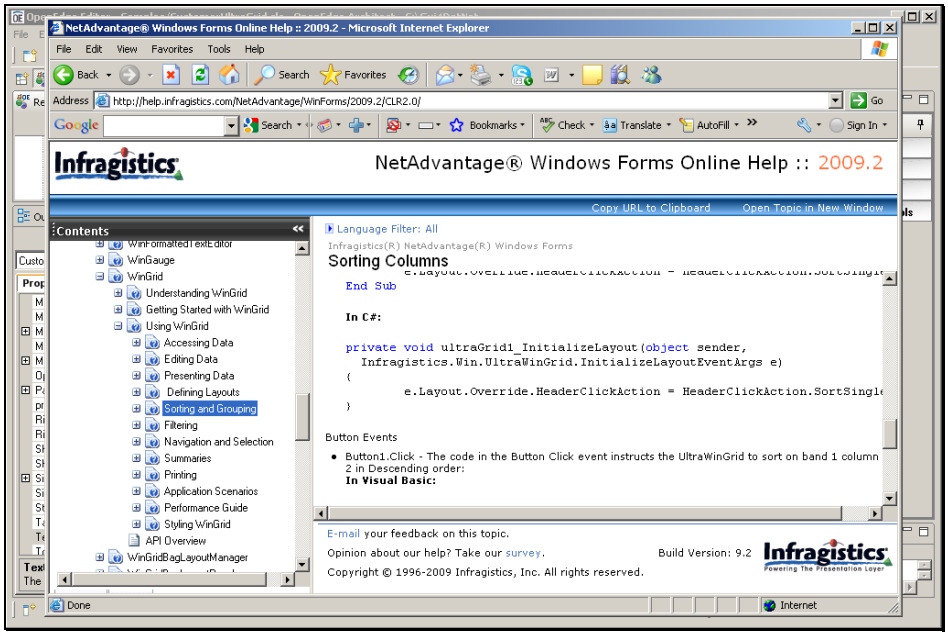
Let me see first what I can find out about the **WinGrid** control, which we call the **UltraGrid,** under the **Developers Guide**. Selecting the **Controls** section of the Guide, you see a list of all the available controls. There's introductory material on every control, but I go to the **Using WinGrid** section.



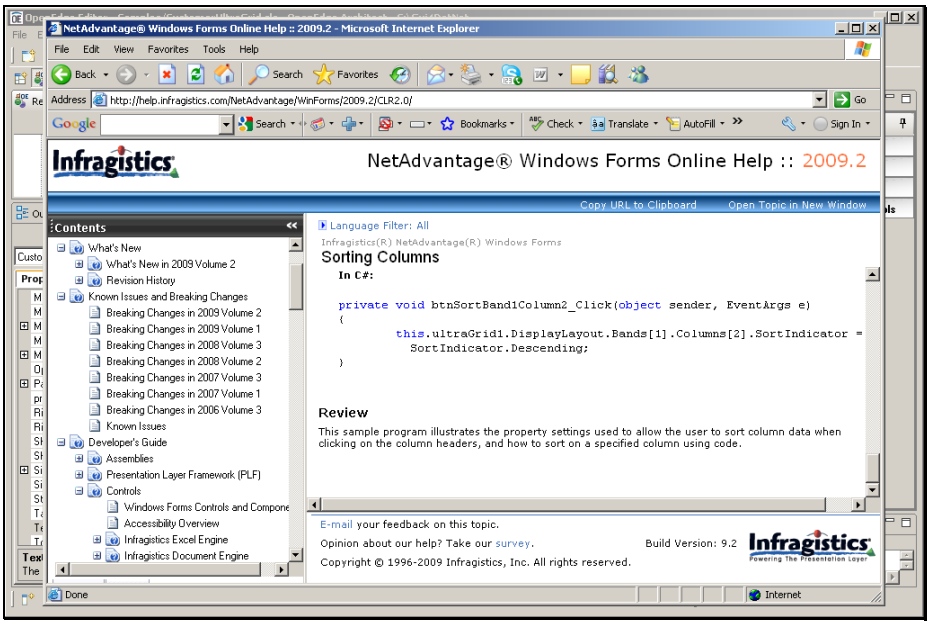
There's the section on **Sorting and Grouping** data with the grid, and within that, a **Sorting Columns** topic, which is what I want to enable on the control. And in that section I find exactly the question I want an answer to, how to enable sorting when the user clicks on a column:



There's information on setting the **HeaderClickAction** property to **SortSingle** or **SortMulti** to enable column sorting, along with some code samples in C# that are pretty close to what the equivalent ABL should look like.

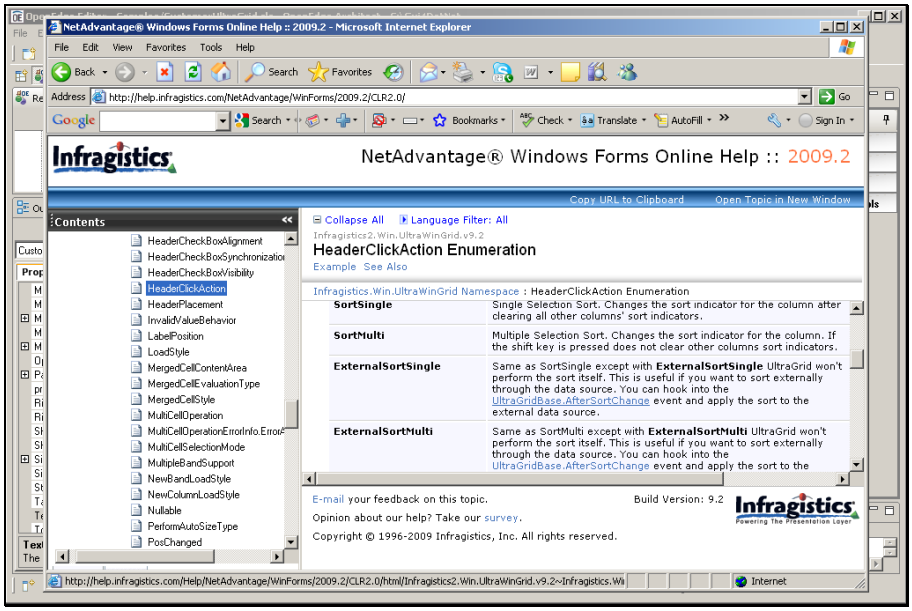


I also find an example of setting the **SortIndicator** to **Descending** if I want to sort the data that way.



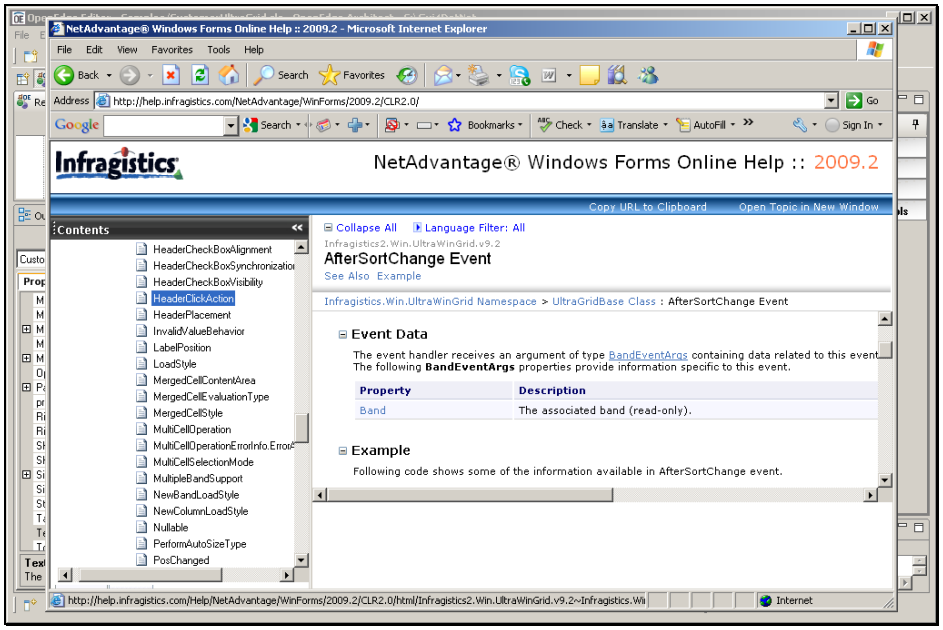
That's the kind of solution information on controls and their properties and events that the **Developers Guide** provides.

The **API Reference Guide** has comprehensive reference documentation on every aspect of every control. Expanding the **WinGrid** and its **Namespace**, there are categories of information on the **Classes** that support each control, the **Enumeration** values that it uses, its **Interfaces**, and related **Delegate** classes that define its event handlers. My **HeaderClickAction** and **SortIndicator** are enumerations, so expanding the **Enumerations** node, I see a description of **HeaderClickAction**:

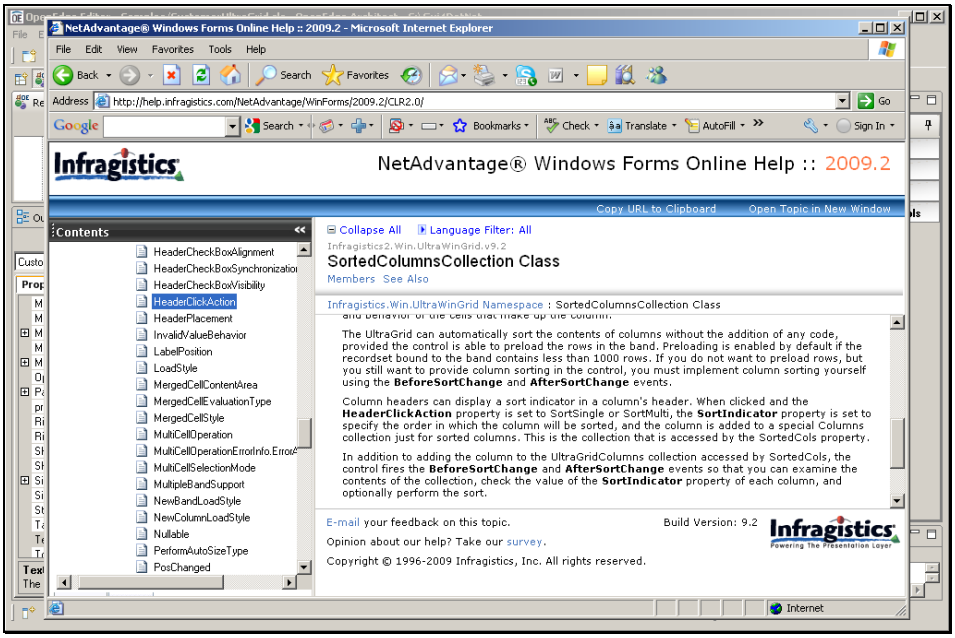


There's information on the **SortSingle** value, which lets the user sort on a single column by clicking on it, and the **SortMulti** value, which lets the user sort on several columns in succession, as well as the **ExternalSortSingle** and **ExternalSortMulti** values that let you capture the sort request yourself, without the grid doing the sorting for you. You take advantage of the ExternalSort… properties by subscribing to the

AfterSortChange event, so I click on the link for that event get some information on it. There I learn that the event takes a subclass of the EventArgs class as input, called **BandEventArgs**.



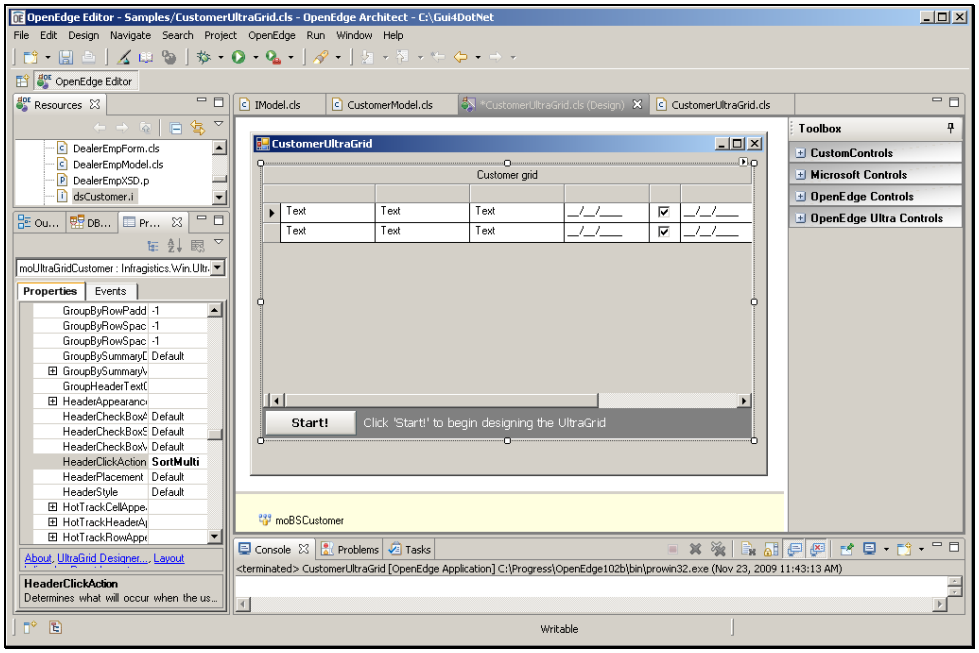
I can open up layers of information on **BandEventArgs** and its sorted columns that I want to understand how to use, and find that **SortedColumns** is a collection of **UltraGridColumn** objects associated with one of the bands of the grid.



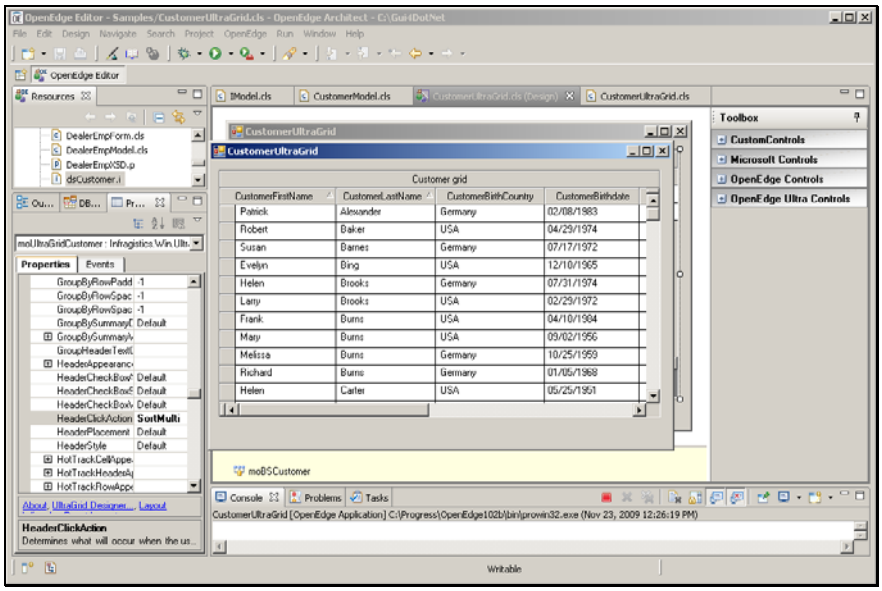
In this topic description there is information on the enumeration values and on using the sort events to capture user actions, and also how the **SortIndicator** is set to indicate whether the sort is descending or not.

I then take a look at the details of the **SortIndicator** enumeration, and see its values. If sorting is enabled for a column, I check for the value **Descending**, to see if the user has requested a descending sort.

I've learned a lot about the enumerations and the events that control sorting, so I return to Visual Designer to try out some of these values. The **DisplayLayout** property of the grid is really an object with a number of contained property values. So as the documentation examples showed me, I expand **DisplayLayout** and then **Override** in the **Properties View**, and here I find the **HeaderClickAction** property. If I drop down its list of values – remember that it's an enumeration with a fixed list of possible values – I can set the property to **SortMulti** to let the grid sort the data when the user clicks on one or more columns.



When I run the form with this property setting, and click on the **CustomerLastName** column, the data is sorted by **CustomerLastName**. I can sort by **CustomerFirstName** within **CustomerLastName** by doing a shift-click on the **CustomerFirstName** column, and see the multiple levels of sorting supported by the grid when **HeaderClickAction** is set to **SortMulti**.



This example shows what seems like a useful feature built into the grid, without my having to do any coding at all. But now that I've seen what the grid can do on its own, I need to consider the fact that letting the user interface take over sorting of my data – in addition to filtering the data and other jobs that are really part of data management – is not necessarily a great idea. I'll next show you how to keep the data

management where it belongs and still let users use the grid control to see the data the way they want to see it.

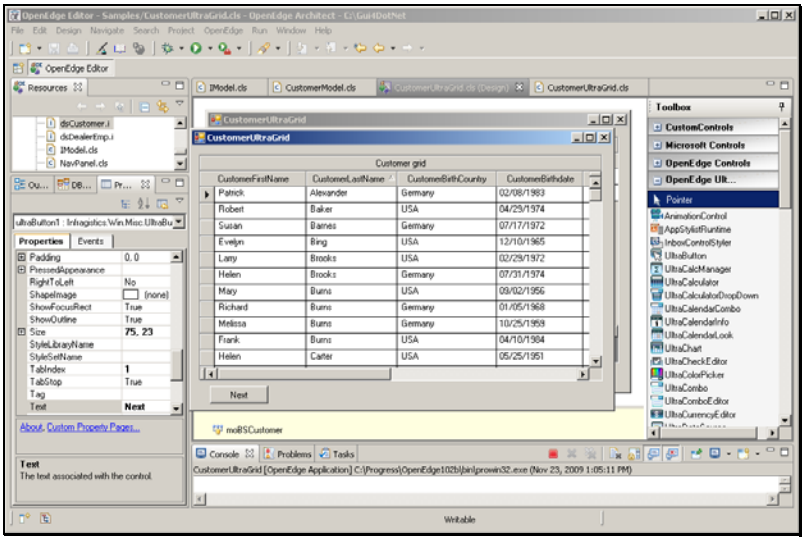
[This next part of the document corresponds to the video presentation that is Part 3 of the Data Sorting series:]

Next this document shows how to intercept the event that fires when the user requests a sort, so that you can code the event handler yourself to apply that sort request to the query that you're managing rather than letting the grid do its own sorting.

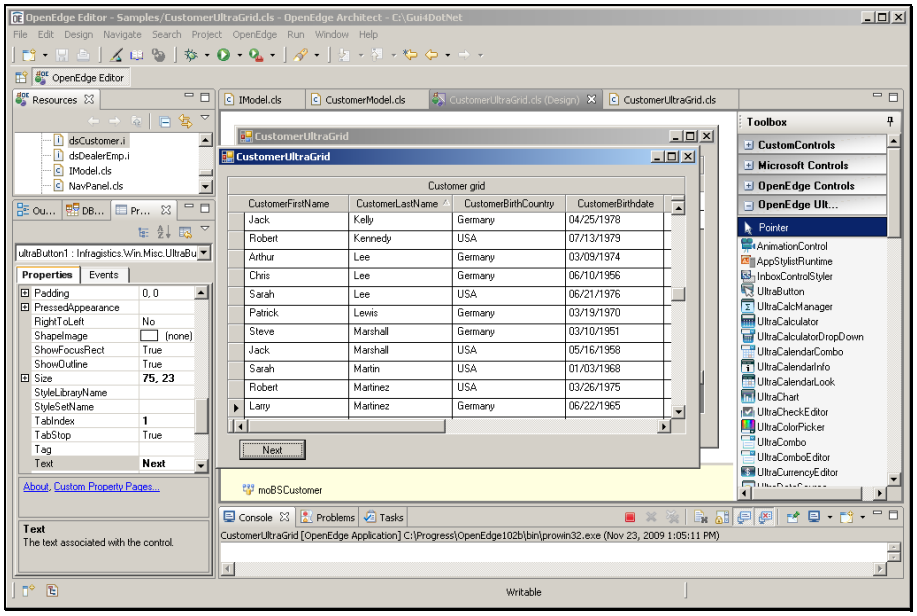
The previous section of this document showed how to enable grid column sorting by setting the **HeaderClickAction** property. To illustrate what I meant by suggesting that letting the grid do the sorting is not necessarily a good idea, I'll add a button to the form and code a simple event handler for it. It's a Next button, to advance the query on the Customer data to the next row. Double-clicking on the control in Visual Designer automatically creates a subscription and a skeleton event handler for the control's default event, in the case of a Button, the **Click** event. I want to increment the **Position** property of the binding source when the button is clicked.

```
@VisualDesigner.
METHOD PRIVATE VOID ultraButton1_Click( INPUT sender AS System.Object,
    INPUT e AS System.EventArgs ):
        moBSCustomer:Position = moBSCustomer:Position + 1.
        RETURN.

END METHOD.
```

Let's see what happens when I run the form containing my button. As before, I can take advantage of the grid's ability to sort the data itself by clicking on a column header such as CustomerLastName. But then I select the first row that's displayed in the grid after the sorting. Note the row selection marker at the beginning of the first row in the grid:

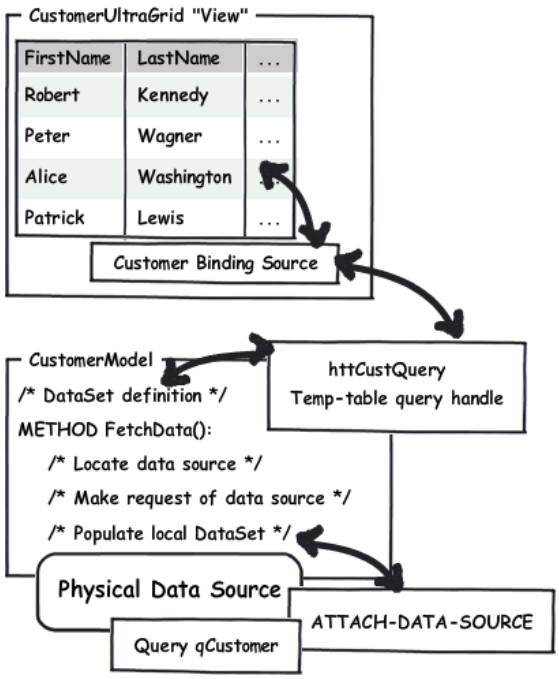


Now I click the Next button.

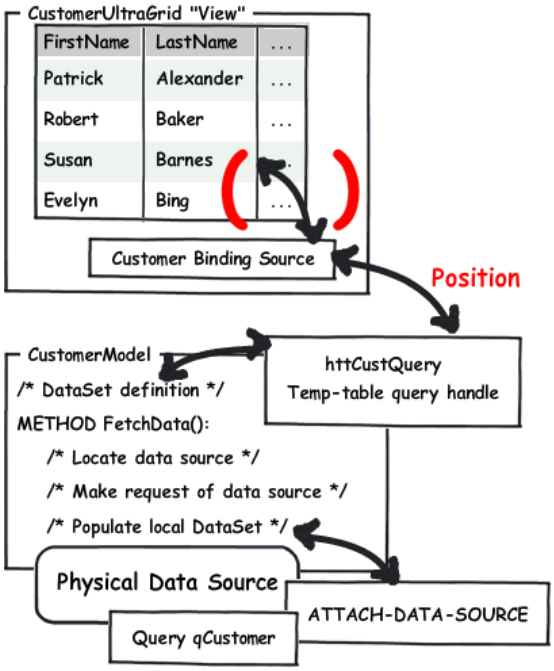


Well, what happened? The grid's position marker jumped to a completely different row. If I click Next again, the same thing happens: the grid jumps to another row that is nowhere near the next row as displayed in the grid. What the Next button is doing has no relation to the data as the grid is displaying it, which is certainly very confusing. Let's take a look at what's happening here.

In this diagram I illustrate that in the Model, there's a database query to retrieve requested data into local storage in a DataSet, and then there's a query on that data for the Customer temp-table. The ProBindingSource is connected to that temp-table query handle, and the ProBindingSource in turn becomes the **DataSource** for the grid, so everything is connected together.



But if I let the grid sort its displayed data independently, then the connection with the data as held in the Model is effectively broken. The Model doesn't know anything about the sort sequence that the user is seeing, so when the Next button advances the **Position** of the binding source, which does a Get-Next on the temp-table query, this is totally out of sync with what the user interface is displaying to the user. Allowing this to happen is obviously not a good thing.



So as nice as this sorting in the grid seems, it's really not a good idea for an application that is in the serious data management business. It's better to think of this grid capability more as demo-ware.

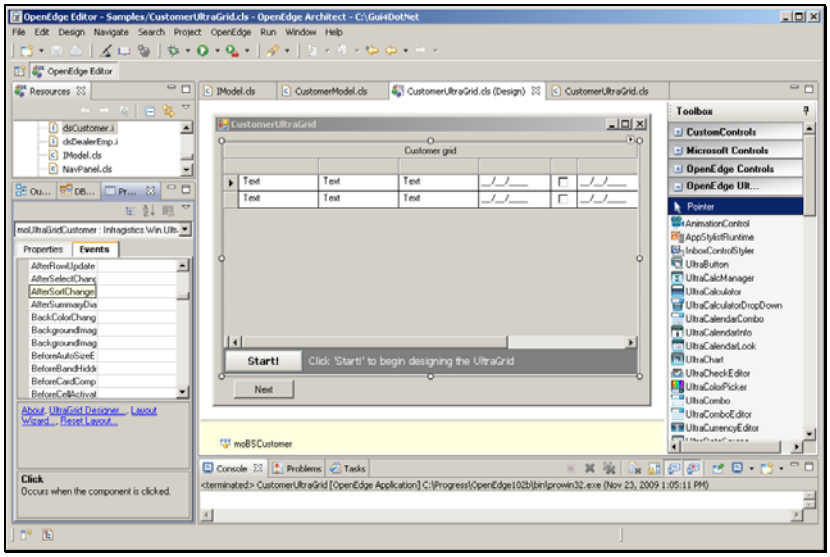
Instead, it's time to write the code to add to the Model to let it control the sorting. Here's some simple code to manage sorting directly on the temp-table that holds the data I've retrieved.

```
METHOD PUBLIC VOID SortData( INPUT pcSort AS CHARACTER ):

    DEFINE VARIABLE iSortField  AS INTEGER   NO-UNDO.
    DEFINE VARIABLE cSortString AS CHARACTER NO-UNDO INIT "".

        DO iSortField = 1 TO NUM-ENTRIES (pcSort) BY 2:
          cSortString = cSortString + " BY " + ENTRY (iSortField,pcSort) + " " +
            ENTRY (iSortField + 1, pcSort).  /* Optional descending qualifier */
        END.
        httCustQuery:QUERY-CLOSE () NO-ERROR .
        httCustQuery:QUERY-PREPARE ("FOR EACH ttCustomer " + cSortString).
        MESSAGE cSortString VIEW-AS ALERT-BOX.
        httCustQuery:QUERY-OPEN ().
END METHOD.
```

Basically this bit of code is doing two things. It's taking a character string passed in as a parameter, which just alternates sort fields and an optional "DESCENDING" qualifier, and turns that into a BY clause for an ABL query. And then it re-prepares the temp-table query on the local data held by the model with that BY clause and re-opens it. This makes the new sort sequence available to the binding source in the form, and through the binding source to the grid. To help you see what's happening, I put in a message statement as well that displays the resulting sort clause in the query.

Now let's look at the other side of this call, which is an event handler in the form class that captures the sort request and constructs the list of columns to sort, passing this as a character string to the **SortData** method. I select the **Events** tab in the **Properties View** to get a list of all the events that the grid

supports. Remember that in the Infragistics documentation I learned that there's an **AfterSortChange** event that fires whenever the user requests a sort by clicking on a column header.



If I just double-click that event name in the Events tab, I get the start of an event handler and a subscription. Here's the code I write to create the sort string to pass to the Model. The first part checks to see if the **HeaderClickAction** is set to either **SortSingle** or **SortMulti**.

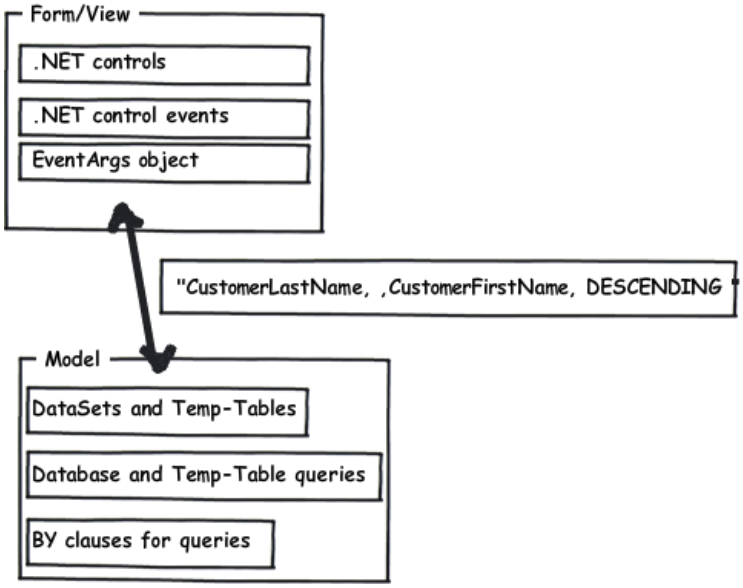
```
@VisualDesigner.
METHOD PRIVATE VOID moUltraGridCustomer_AfterSortChange
  ( INPUT sender AS System.Object,
    INPUT e AS Infragistics.Win.UltraWinGrid.BandEventArgs ):

    DEFINE VARIABLE oSortColumn AS UltraGridColumn NO-UNDO.
    DEFINE VARIABLE iColumn     AS INTEGER         NO-UNDO .
    DEFINE VARIABLE cSortString AS CHARACTER       NO-UNDO INIT "".
    DEFINE VARIABLE oBand       AS UltraGridBand   NO-UNDO.

    IF Progress.Util.EnumHelper:AreEqual
         (moUltraGridCustomer:DisplayLayout:Override:HeaderClickAction,
           HeaderClickAction:SortMulti)
    OR Progress.Util.EnumHelper:AreEqual
         (moUltraGridCustomer:DisplayLayout:Override:HeaderClickAction,
           HeaderClickAction:SortSingle)
    THEN RETURN.
```

The control represents the **HeaderClickAction** value as an enumeration, a set of fixed coded values. ABL doesn't support enumerations directly, but it does provide an **EnumHelper** class with methods like **AreEqual** to let you *work* with enumeration values. Remember too that **SortSingle** and **SortMulti** are the values that tell the grid to do its own sorting, so if that's what it's set to, the event handler just returns and lets the grid do its thing.

But otherwise the method uses some of what we saw in the Infragistics documentation about the **Band** object and its **SortedColumns** property, along with the **SortIndicator** enumeration, to extract the user's request from the event arguments object that is passed in, and turn that into a generic character string that the Model can deal with:

```
        oBand = e:Band.
        DO iColumn = 0 TO oBand:SortedColumns:Count - 1:
            oSortColumn = CAST (oBand:SortedColumns[iColumn],UltraGridColumn).
            cSortString = cSortString + (IF iColumn > 0 THEN "," ELSE "") +
                oSortColumn:KEY + "," +
                (IF Progress.Util.EnumHelper:AreEqual
                  (oSortColumn:SortIndicator, SortIndicator:Descending) THEN
                     " DESCENDING " ELSE " ").
        END.

        moCustomerModel:SortData(INPUT cSortString).

        RETURN.

        END METHOD.
```

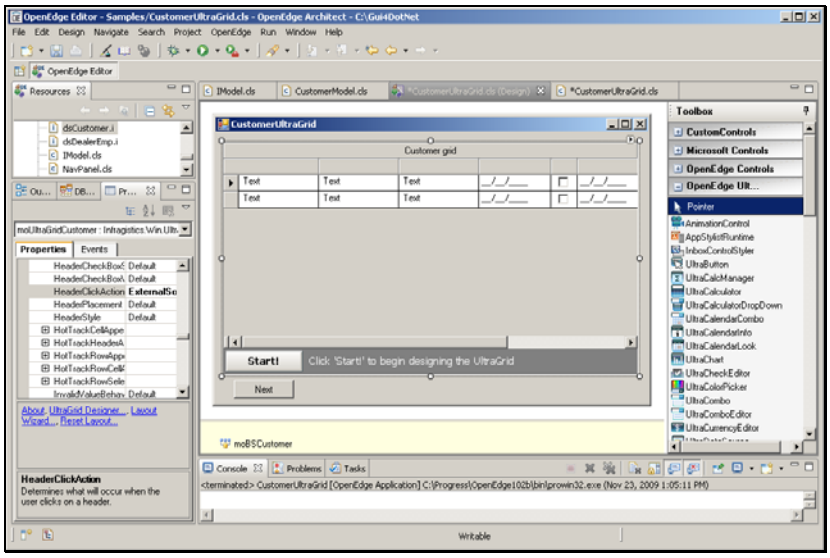
The essential element of this simple code is to make sure that the Model doesn't have to understand anything about the event that initiated the request or the structure of the control elements that provide the information about what columns were clicked. That's the form's job, to deal with the specifics of the controls in the user interface. At the same time, I don't want the form to have to understand how to put together an ABL query to re-sort data; that's the Model's job. So I just put together a list of sort columns and whether they're descending or not, and then run SortData to tell the Model to deal with it. This separation of responsibilities is illustrated by the following diagram:



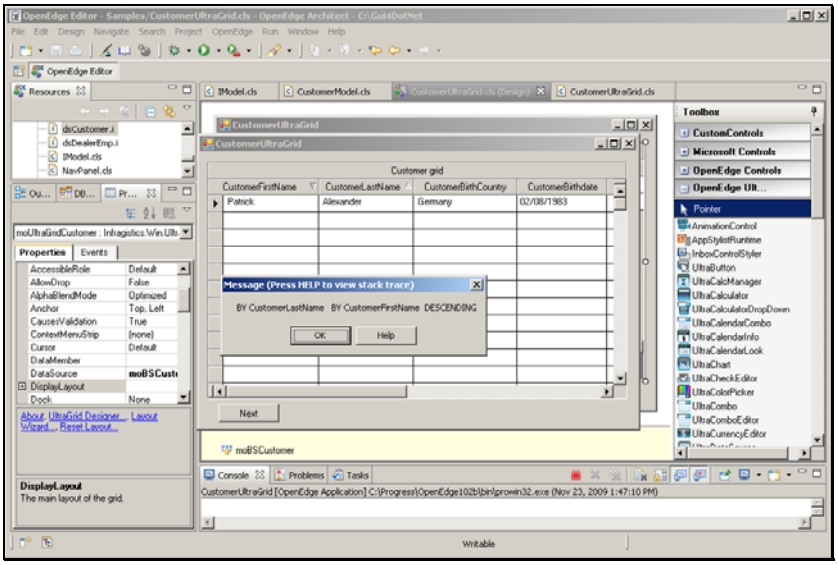
One more thing I have to do up at the top of the form class is to add a **USING** statement for the **UltraWinGrid** so that the compiler will recognize my references to the grid components **UltraGridColumn** and **UltraGridBand** in the **AfterSortRequest** event handler.

```
USING Progress.Lang.*.
USING Progress.Windows.Form.
USING Infragistics.Win.UltraWinGrid.*.
```

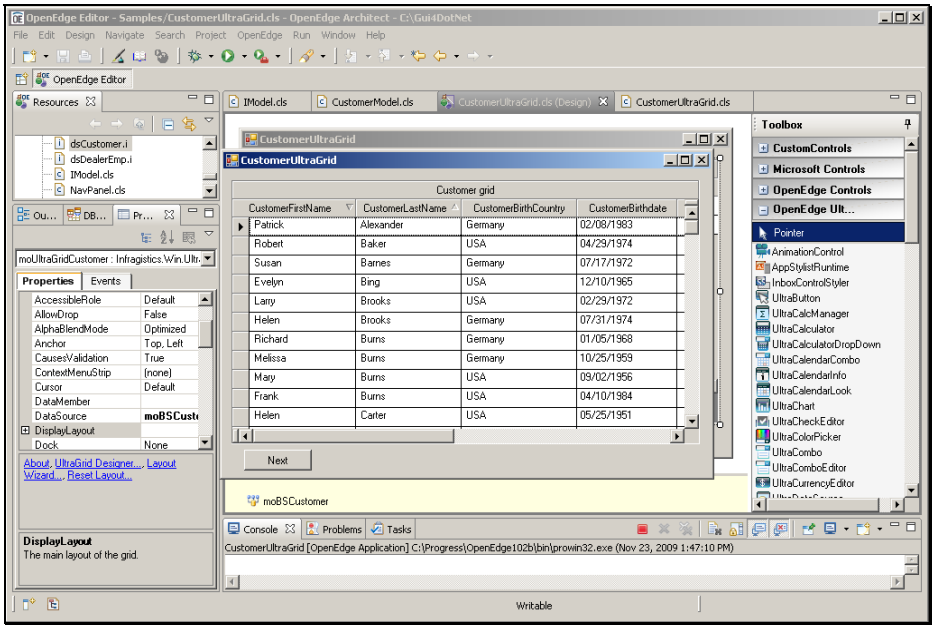
Now I save what I've done, and go back to the design view, where I need to reset the **HeaderClickAction** property to one of the values that tells the grid *not* to do its own sorting. Once again, under **DisplayLayout** and **Override** in the **Properties** View, I find the **HeaderClickAction** property. And as I learned from the online documentation, the values **ExternalSortSingle** and **ExternalSortMulti** tell the grid just to invoke the **AfterSortChange** event and pass the selected columns into it without doing any sorting on its own. So I select **ExternalSortMulti** for multi-column sorting.



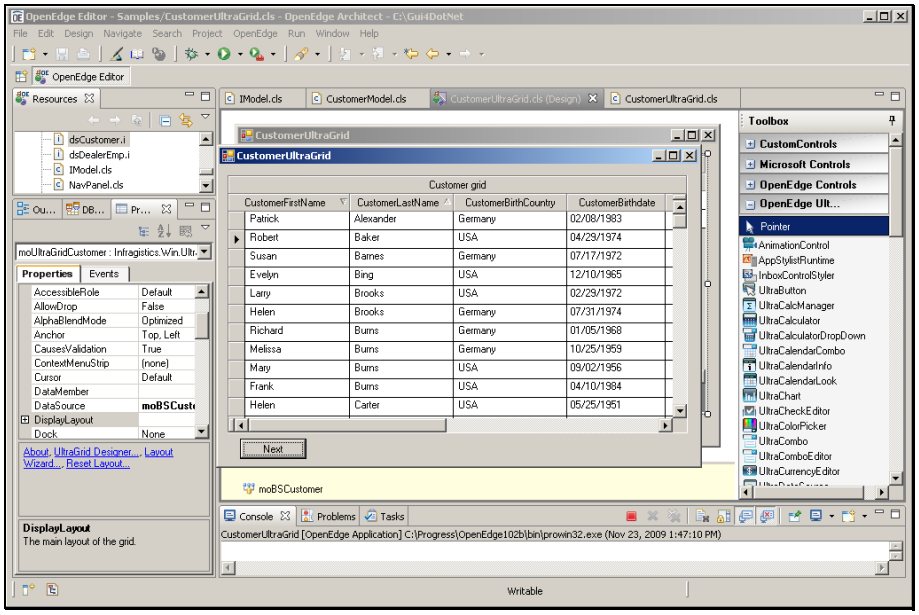
I save and run the form with this new value. When I click on the **CustomerLastName** column, the MESSAGE statement in the Model's **SortData** method appears, to confirm that it was invoked to resort the data, rather than the grid doing the sorting. If I then Shift-Click on the **CustomerFirstName**, that column gets added to the sort. And if I click on the direction arrow in the grid column header, that action gets passed in the **SortIndicator** property as a **Descending** qualifier. The MESSAGE statement alert box now reflects all three of these combined clicks on the **UltraGrid** column headers:



And you can see that re-opening the temp-table query down in the Model makes the data available to the grid through the binding source, with the new sort sequence. What we see initially looks the same as when the grid did its own sorting on its local copy of the data.



But what's significant is that the sort you're seeing is now the same sort order managed by the query in the Model, because the grid is actually using the result of re-preparing and re-opening the query with the BY clause generated by **SortData**. So If I click on the Next button, the position indicator advances in the grid the way you would expect.

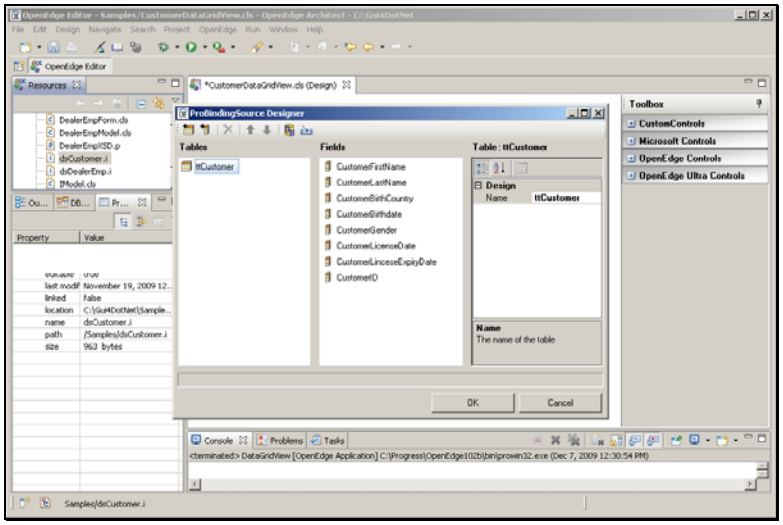


What the user is seeing is in sync with how the Model is actually managing the data, and this is what you want to do in your application to avoid confusion between the user interface and your data management logic. In addition, the external sort option is much faster than having the grid do the sorting, because it can take advantage of OpenEdge database and temp-table indexing, so that's another reason to keep the sorting in the Model where it belongs.

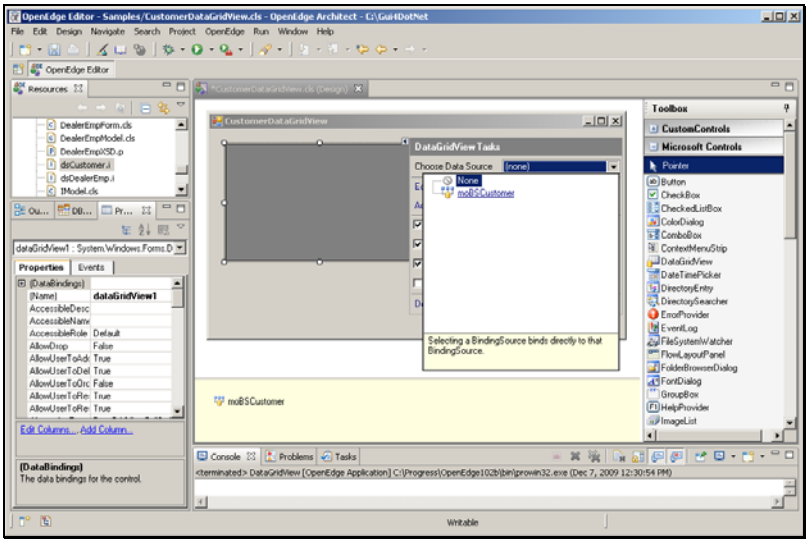
[The next part of the document corresponds to the video presentation that is Part 4 of the Data Sorting series:]

The previous sections of this document show how the binding source interacts with the Infragistics **UltraGrid** control. The Microsoft **DataGridView** control works quite differently, and uses a binding source property and an event that the UltraGrid doesn't. This part of the document shows you the differences.

To illustrate these differences, I create a new form that uses the same Model class for data retrieval that the UltraGrid form uses. I name the form class **CustomerDataGridView**.cls. Remember that one way to create a ProBindingSource is just to drag a source file with a ProDataSet or temp-table definition anywhere onto a form. I do this here by dragging the same definition file, **dsCustomer.i**, that I used in the other example, from the **Resources** View onto the new form. The **ProBindingSource Designer** appears to allow me to edit the definition if I need to:



The new ProBindingSource control gets placed in the non-visual tray below the form itself. I rename the control as I've done elsewhere to **moBSCustomer**. Next I select the Microsoft DataGridView from the **Microsoft Controls** group in the Toolbox and drop it onto the form. The Customer ProBindingSource is available to be chosen as the **DataSource** for the grid.

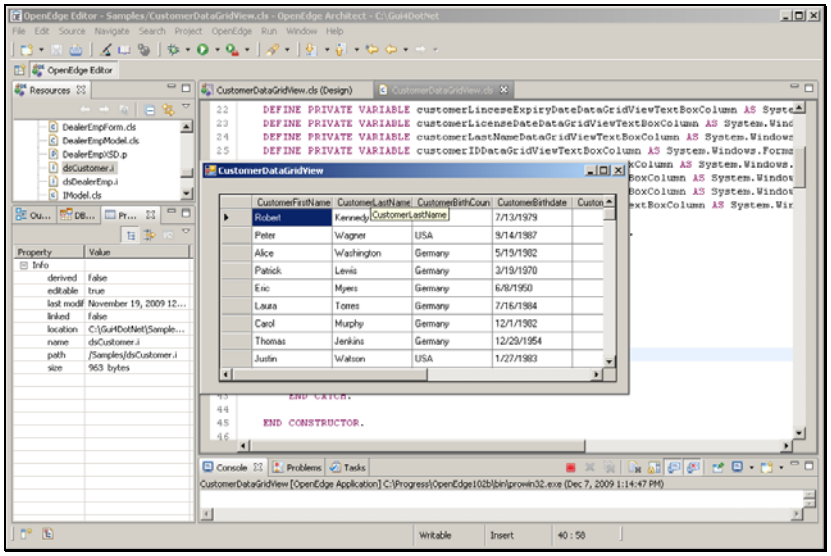


Now I need to go into the code for the form. I want to reuse the same **CustomerModel** class that I created for use with the other form class that displayed data in an UltraGrid. Even though the user interface controls work very differently, I can use the same data management class as before, because I separated the data management code cleanly from the form class that knows the details of how the UI works.

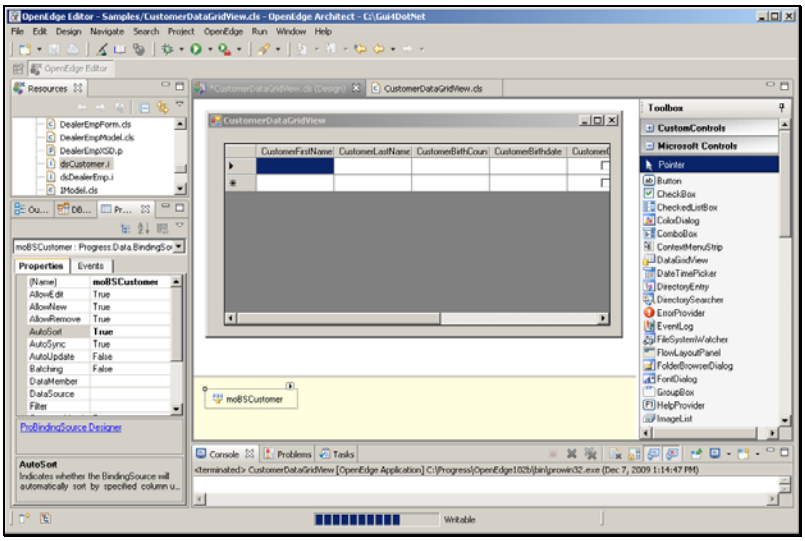
So in the same way as in CustomerUltraGrid.cls, I create an instance of the data management Model class using the **NEW** keyword, tell it to fill its DataSet with all the Customers, and retrieve the temp-table handle to use as the **Handle** property of the ProBindingSource.

```
DEFINE PRIVATE VARIABLE moCustomerModel AS IModel NO-UNDO.

CONSTRUCTOR PUBLIC CustomerDataGridView (  ):

        SUPER().
        InitializeComponent().
        moCustomerModel = NEW CustomerModel().
        moCustomerModel:FetchData("").
        moBSCustomer:HANDLE = moCustomerModel:GetQuery().
        CATCH e AS Progress.Lang.Error:
             UNDO, THROW e.
        END CATCH.

END CONSTRUCTOR.
```

Running the form shows us how this much of the job that we have to do works at this point. This screen capture shows that I've successfully gotten data into the grid, but there's no sorting when I click on a column header:

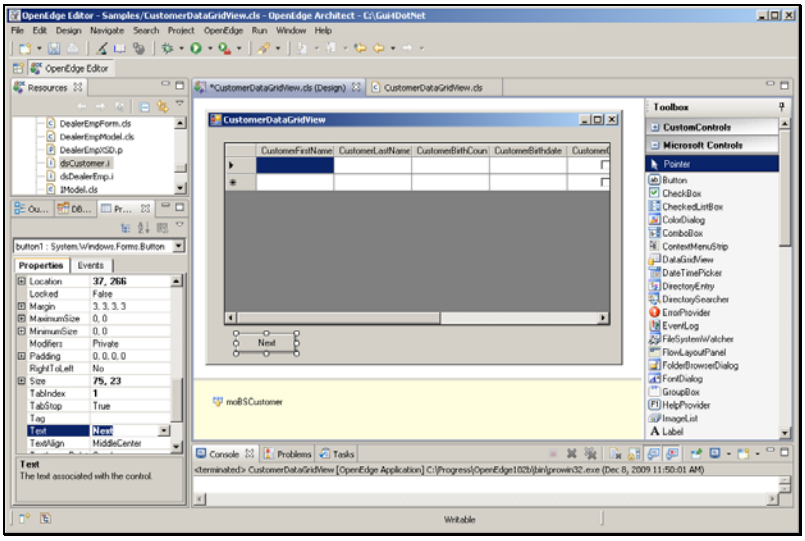


Like the UltraGrid, the Microsoft grid doesn't do any data sorting by default. So I need to find out what to set to enable sorting. If I select the Customer ProBindingSource in the Design view, I see all the properties that a ProBindingSource defines. The details of what these do are all documented in the book ***OpenEdge Development: GUI for .NET Programming*** in the OpenEdge doc set. Not surprisingly, the **AutoSort** property enables the binding source to handle sorting. By default it's **False**, so I reset it to **True**.



Now when I re-run the form and click on a column header, the data gets sorted on the field displayed in that column, as I would expect. Note that the Microsoft grid control doesn't coordinate with the binding source to do more than single column sorting. But because the binding source is doing the sorting in this case instead of the grid, the disadvantage seen in the UltraGrid example – where the sort order of the query and the visible sort order as shown in the grid are out of sync -- isn't a problem.

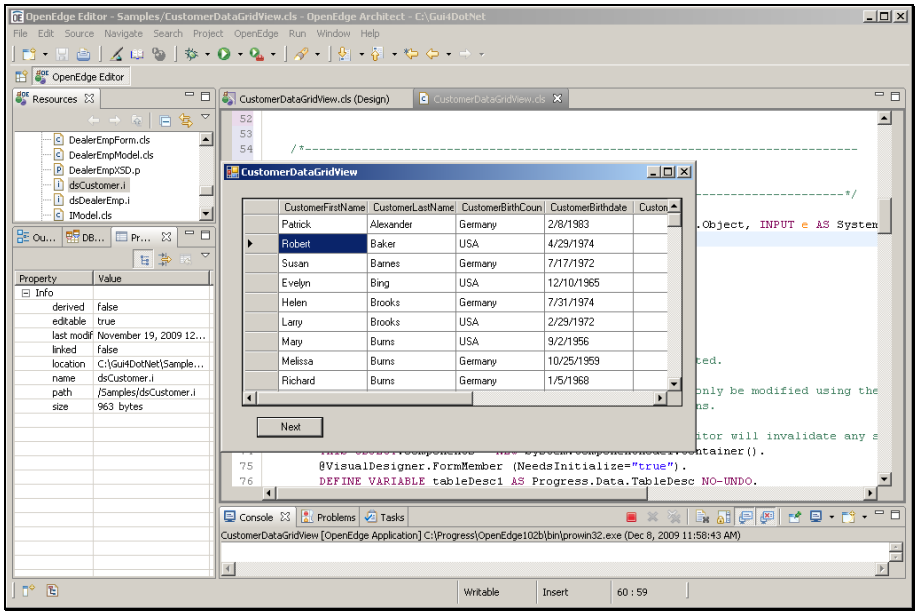
To demonstrate that the sort order shown in the Microsoft grid is the same as the sort order of the underlying query, I again add a **Next** button to the form to advance the query position by incrementing the binding source **Position** property.



I double-click on the button to get a **Click** event handler, and add code to increment the binding source **Position** value, which does a Get-Next on the associated ABL query.

```
@VisualDesigner.
METHOD PRIVATE VOID button1_Click( INPUT sender AS System.Object,
  INPUT e AS System.EventArgs ):
        moBSCustomer:Position = moBSCustomer:Position + 1.
        RETURN.
END METHOD.
```

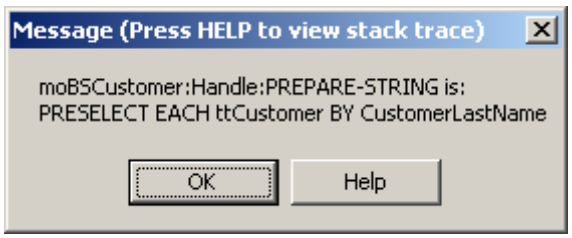
When I re-run the form and click on the CustomerLastName column just as I did before, I see that the data is now sorted properly. And if I click the Next button, the row marker in the grid shows that the data sorting you see in the grid is in sync with the sort sequence of the ttCustomer temp-table query.



I can make this doubly clear by adding a statement to the button's **Click** event handler to display the **PREPARE-STRING** of the temp-table query handle after it has been automatically re-prepared by the effects of setting the **AutoSort** property to **True**:

```
@VisualDesigner.
METHOD PRIVATE VOID button1_Click( INPUT sender AS System.Object,
    INPUT e AS System.EventArgs ):
        moBSCustomer:Position = moBSCustomer:Position + 1.
        MESSAGE "moBSCustomer:Handle:PREPARE-STRING is: " SKIP
           moBSCustomer:Handle:PREPARE-STRING VIEW-AS ALERT-BOX.

        RETURN.

END METHOD.
```

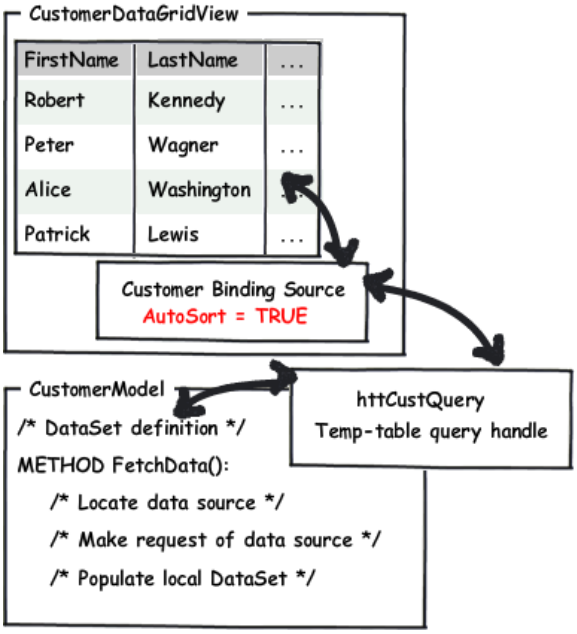
The MESSAGE statement alert box that appears when I run the form with this version of the event handler, and then click on the CustomerLastName column header, shows that the binding source has properly re-prepared the query for me with the appropriate BY clause.



The ProBindingSource, being an OpenEdge-specific .NET control, is able to generate the right BY clause itself, and then re-prepare and re-open the query that its **Handle** property points to, in effect doing the work that I did myself in the UltraGrid example using the **SortData** method. This example reinforces two important facts about the OpenEdge GUI for .NET support.

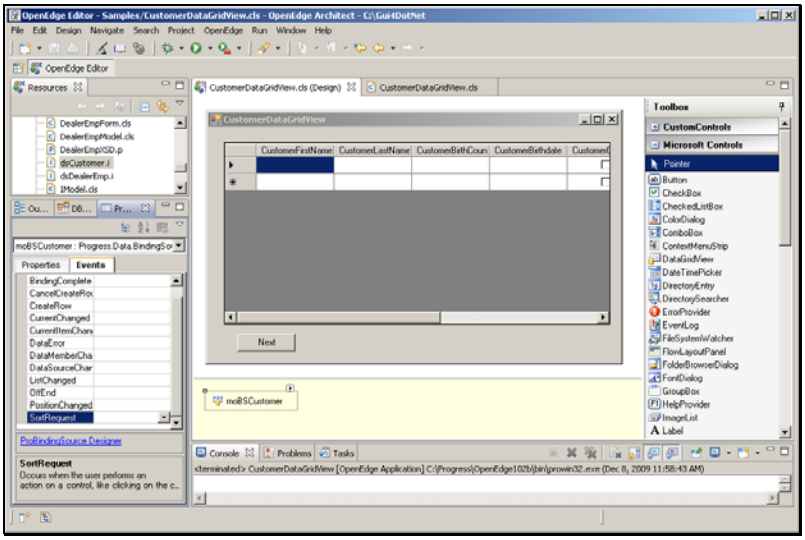
First, it makes it clear that different controls doing basically the same job, such as the Infragistics UltraGrid and Microsoft DataGridView, can work very differently, including in how they interact with the ProBindingSource.

Secondly it shows one of the advantages of having the ProBindingSource as an OpenEdge-specific control, serving as the intermediary between the .NET UI and ABL data management. Just setting **AutoSort** on the ProBindingSource to True enables the ProBindingSource to reopen the query it's attached to with the correct BY clause,as this diagram illustrates:

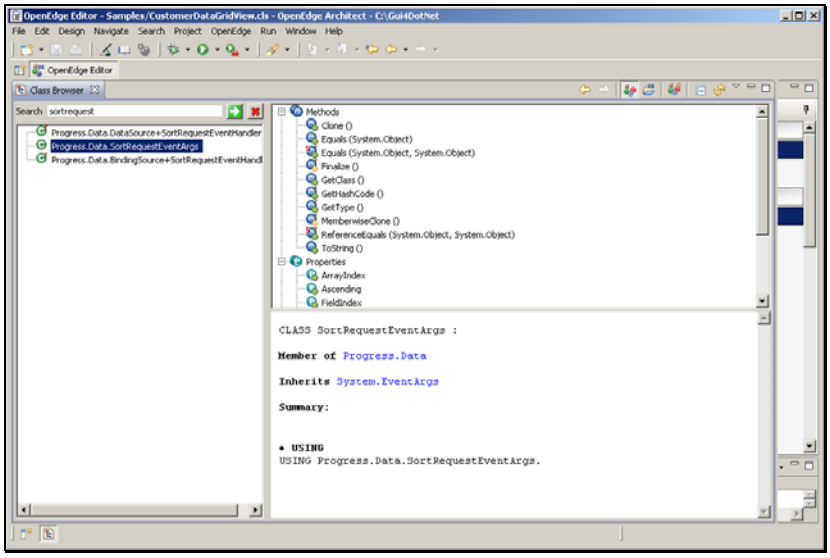


Beyond the automated support for sorting, there may be cases where you want to intercept the sort request yourself much as I did in the UltraGrid sample to do something that goes beyond what AutoSort does, for instance to enable multi-column sorting that the Microsoft grid and the AutoSort property of the binding source don't provide, or to support sorting of data that's being retrieved in multiple batches. The remainder of this document shows how to use that alternative.

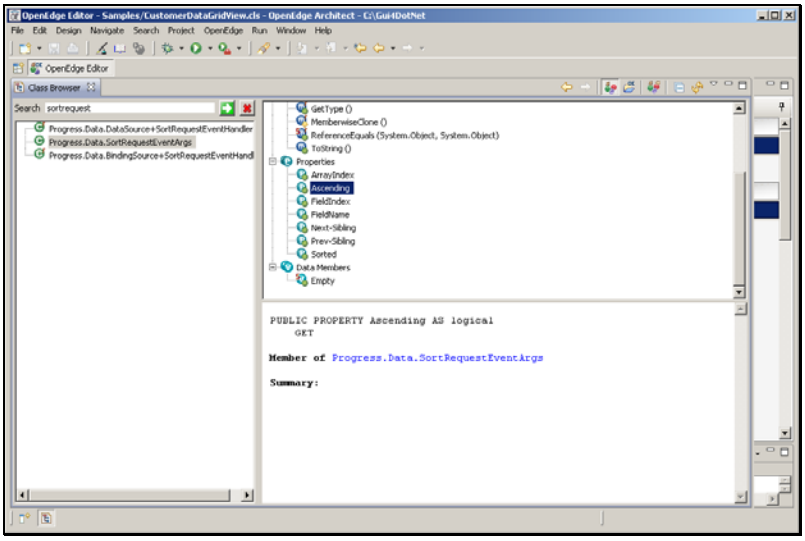
If I look at the events that the ProBindingSource supports, I see a **SortRequest** event:



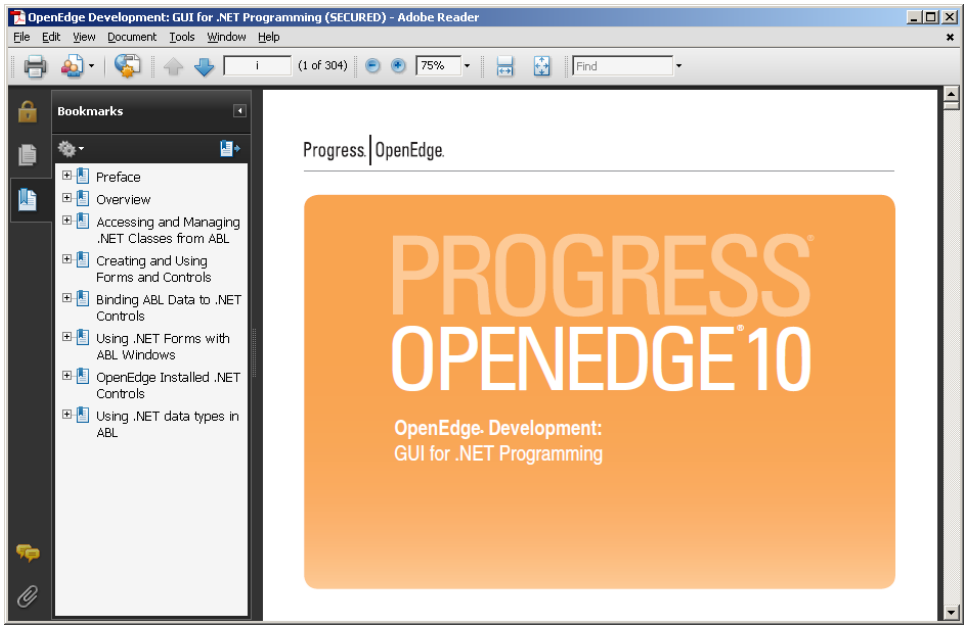
Opening the ClassBrowser to see what I can learn about this event, I enter **SortRequest** as a search string, and select **SortRequestEventArgs**, the EventArgs object that is passed into a SortRequest event handler.



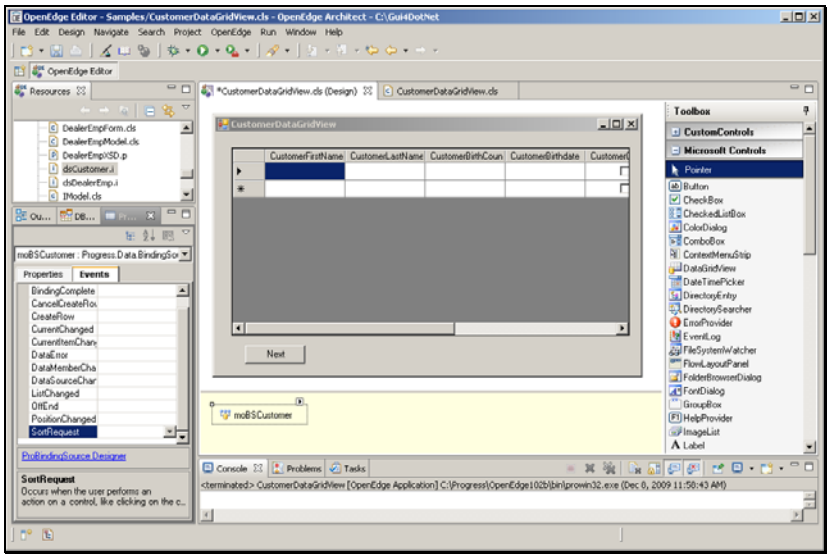
Expanding the list of its properties, I see that there's a **FieldName** property, which holds the unqualified name of the column that the user clicked -- remember that only single-column sorting is supported. There's also a **FieldIndex** property, which holds the position of the field within the list of fields in the binding source, if you want to use that instead of the field name, perhaps if the unqualified **FieldName** is not unique. There's also a logical property named **Ascending** that signals whether the sort is ascending or descending.



Again, you can learn all the details about these properties in the material on data binding in the ***OpenEdge Development: GUI for .NET Programming*** book, which you will find in the Product Documentation available in the OpenEdge section of PSDN on the Progress Communities website.



The list of **SortRequestEventArgs** properties in the **Class Browser** gives me the basic information I need to be able to use the **SortRequest** option in my example form. Since this is an alternative to asking the binding source to **AutoSort** the data itself, I set the **AutoSort** property back to False. Then in the **Events** tab for the binding source control, I double-click the **SortRequest** event to get code generated to subscribe to the event and to provide an event handler.

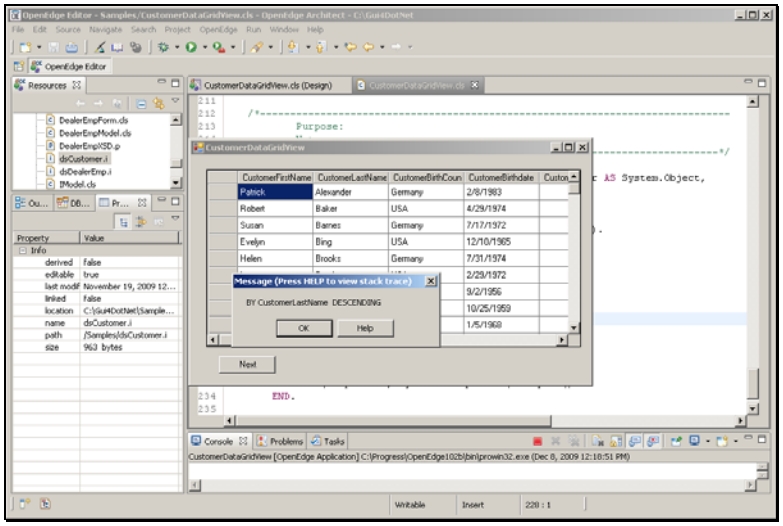


The skeleton event handler that's generated for me shows the **SortRequestEventArgs** parameter being passed in, which in this case is named **args**. Using the properties of the args object, I add code to put together a generic sort string just as I did in the UltraGrid example. There can only be one field to sort on, so I get its name from the **FieldName** property, and append to that the word **DESCENDING** if the value of the **Ascending** property is **False**. And then I pass the string to **SortData**:

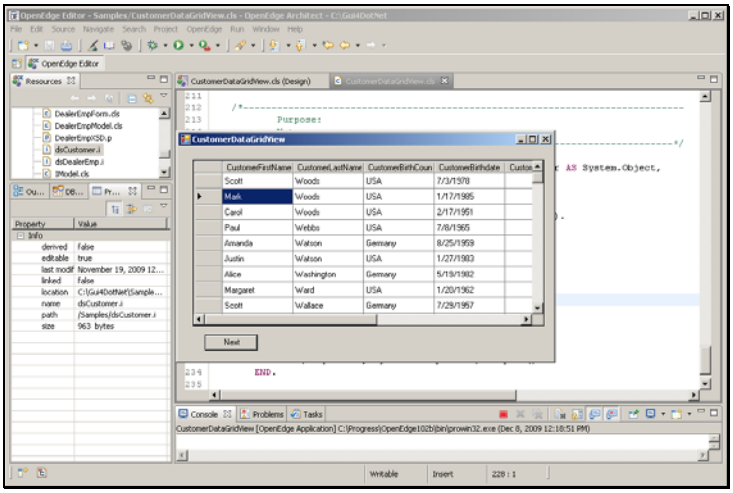
```
@VisualDesigner.
METHOD PRIVATE VOID moBSCustomer_SortRequest( INPUT sender AS System.Object,
    INPUT args AS Progress.Data.SortRequestEventArgs ):
        DEFINE VARIABLE cSortString AS CHARACTER NO-UNDO.
        cSortString = args:FieldName + "," +
          (IF NOT args:ASCENDING THEN " DESCENDING " ELSE " ").
        moCustomerModel:SortData(cSortString).
        RETURN.

END METHOD.
```

Note that this very different user interface form class, containing controls with different properties and events, is using exactly the same sort method in the Model as my other example used, because the code responsibilities are properly separated. When I run the form again and click on the CustomerLastName column, I see the MESSAGE statement that signals that **SortData** was invoked, though in this case the resulting sort is the same as **AutoSort** would have given me automatically.

I click on the column again, which in this control type signals that I want to reverse the sort order, and I see the BY clause MESSAGE statement displayed by **SortData**.



As I noted before, you might want to intercept the **SortRequest** if you wanted the user interface to allow the user to specify multi-column sorting, or some other special treatment of the sort request that **AutoSort** doesn't provide, perhaps checking whether there's an index on the requested sort field. Clicking the Next button confirms that what I see in the grid and the sort order in the query are in sync with each other, whichever sorting option I choose to use.



To review, in this document and the series of video sessions that it accompanies, I showed you how to divide responsibilities between what you can think of as the **View**, your form classes that know about the user interface definition, and the **Model**, the part of the application that manages data access, so that the user interface and the data management don't get out of sync with each other.

I showed how to create an **ABL Interface** that allows you to create a number of classes that consistently implement that interface.

I then created a **Model** data management class that implements the Interface, and added code to fill a local **ProDataSet** with data, and to make the handle of the DataSet's temp-table query available through a method call.

I created a new form class with a **ProBindingSource** -- derived from the same ProDataSet definition used by the Model class -- and added an **UltraGrid** to it, setting its **DataSource** to be the ProBindingSource, then added a Next button to increment the ProBindingSource **Position** property.

I then used the Infragistics documentation to learn about the UltraGrid properties and events, and showed how setting the grid's **HeaderClickAction** property to **SortSingle** or **SortMulti** enables the grid to sort its copy of the data locally.

I used the effects of a Next button to illustrate how the redisplayed data in the grid is not in sync with the sort order of the underlying query that originally populated it. I coded a **SortData** method in the Model class to take a generic sort request and turn it into a BY clause for a re-prepared temp-table query, and coded an **AfterSortChange** event handler in the View to construct the sort request from the members of the **SortRequestEventArgs** parameter.

I reset the grid's **HeaderClickAction** property to **ExternalSortMulti**, and showed how this allows my event handler to interact with the Model class to re-prepare and re-open the Model's query with the new BY clause and use this as the basis for the data displayed in the grid, so that the user interface and the underlying data management class are in sync.

Finally, I showed you how the Microsoft **DataGridView** control works very differently from the **UltraGrid** in how it interacts with the **ProBindingSource**. However, the DataGridView has the advantage that if you ask for automatic sorting support, it's the binding source that's in control, so the sort done for the UI is accomplished by means of the binding source re-preparing and opening the query it's pointing to in the Model, the simple data management class.