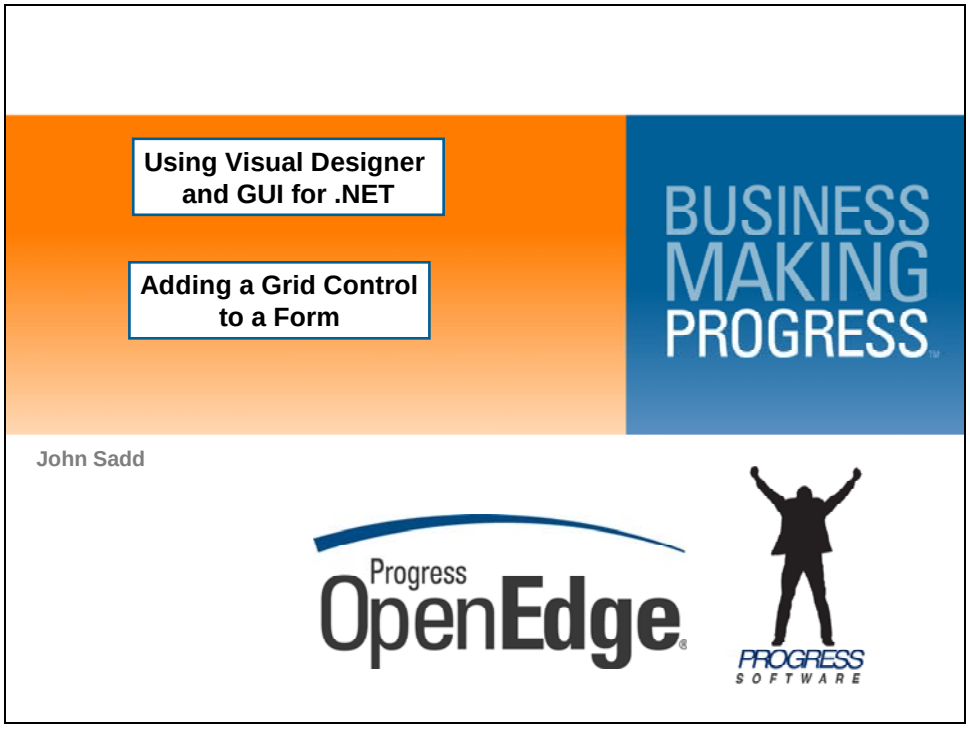
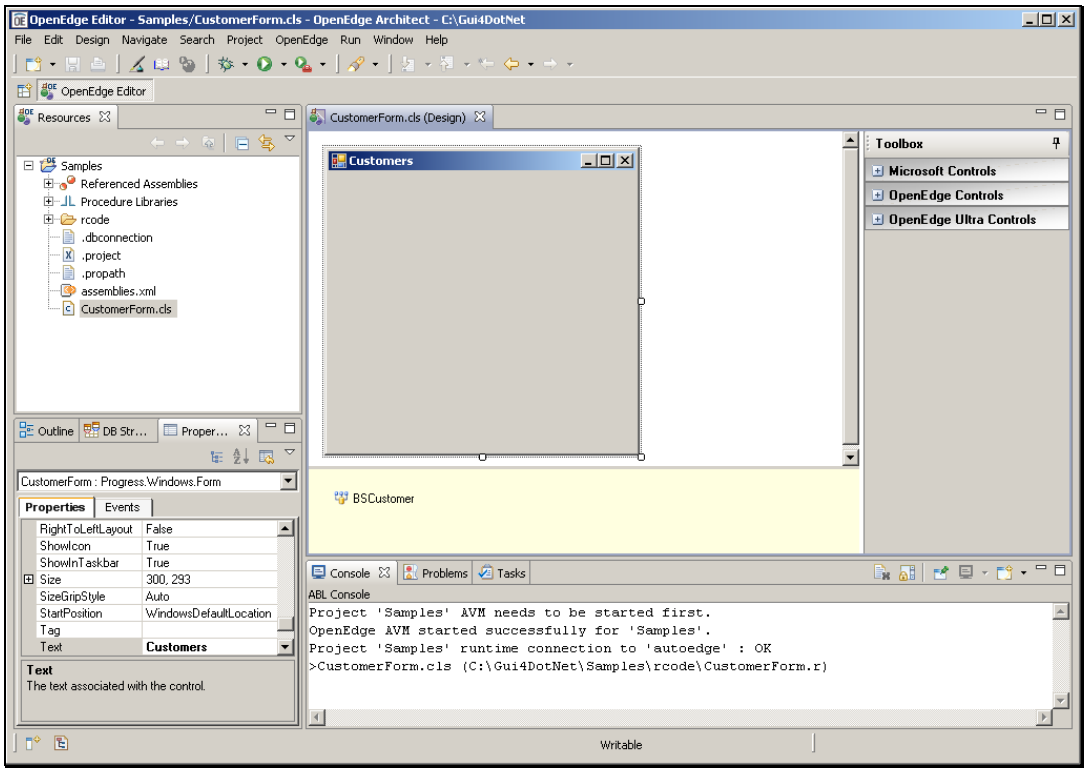
ADDING A GRID CONTROL TO A FORM

John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
November 2009

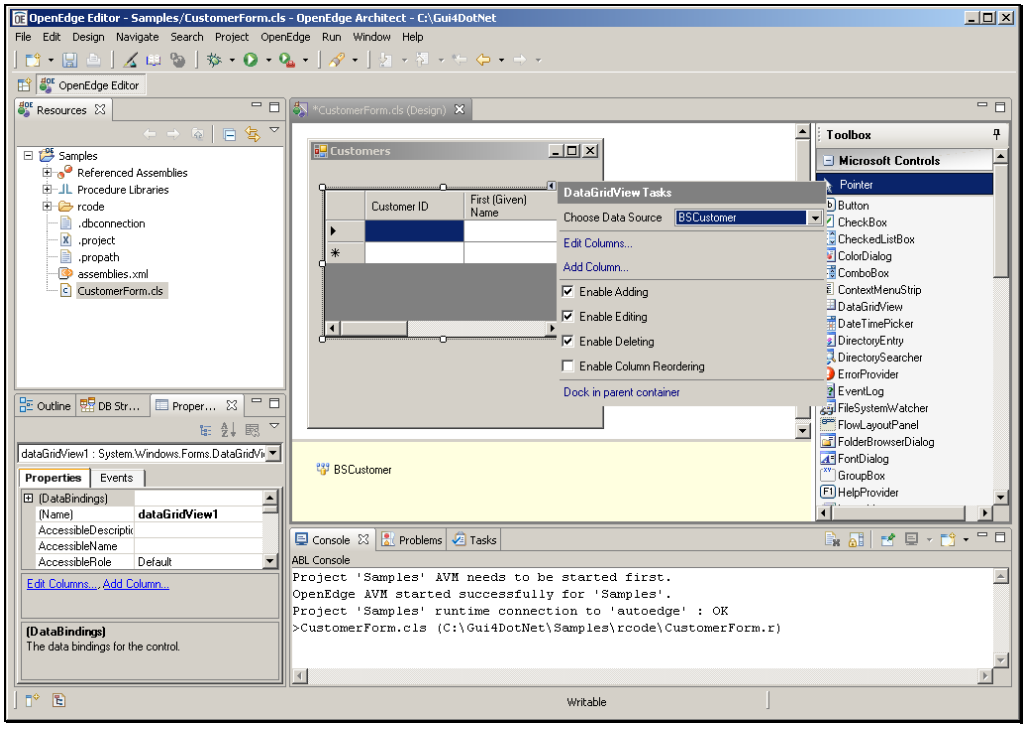


DISCLAIMER

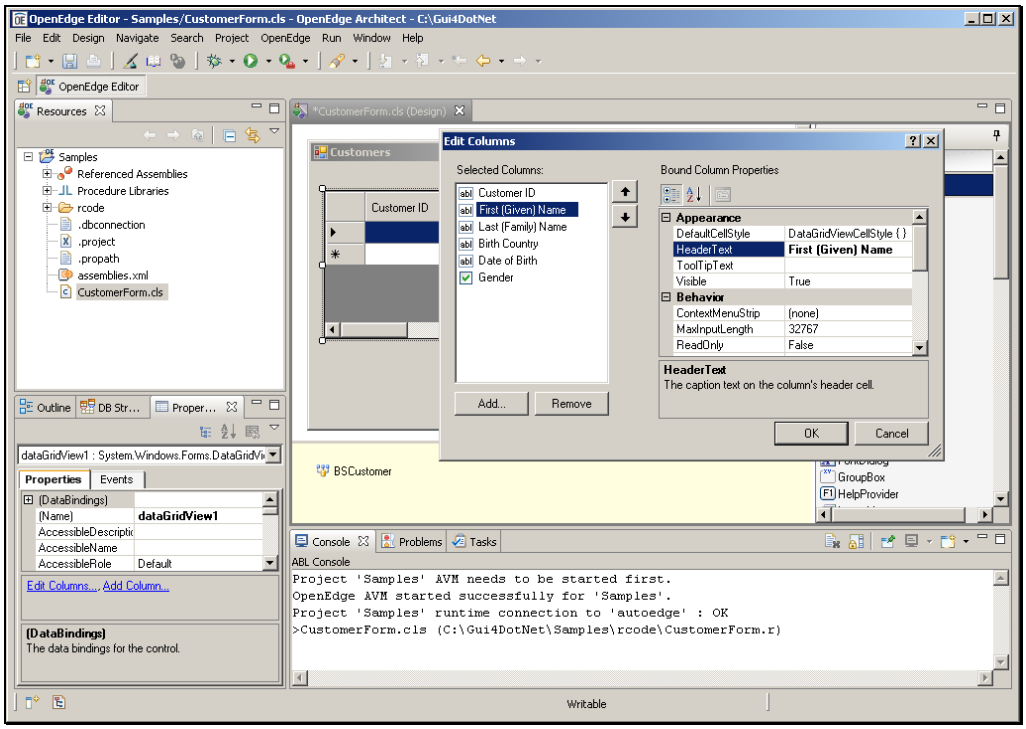
This is the second in a series of documents that accompany a series of video sessions to introduce you to the Visual Designer in OpenEdge Architect and the support for GUI for .NET in OpenEdge 10. It's the second of two parts on **Creating a Form and a ProBindingSource.** In this session I add a grid control to a form and connect it up with fields in a table that was defined for a ProBindingSource control previously. In the session on **Defining an ABL Form and Binding Source**, I created a form class called CustomerForm and added a ProBindingSource control to it, which you can see here as **BSCustomer**, to manage data from the Customer table in the AutoEdge sample database.



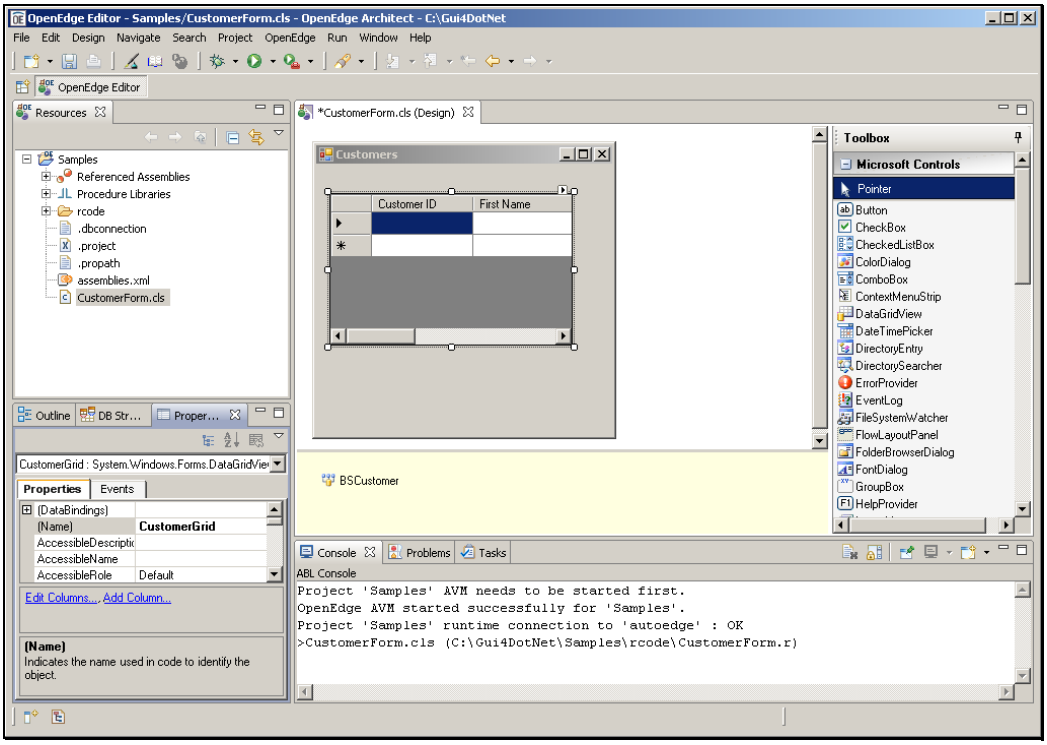
Now I'm going to add a visual control to the form. There are two main collections of these. The first is controls created by Microsoft as part of .NET that are included with OpenEdge. The second is controls from Infragistics identified as **OpenEdge Ultra Controls**, which are a separately purchasable and installable item. The Ultra Controls have many great features, but I'll focus first on the Microsoft controls that are "in the box". I select the **DataGridView**, roughly the .NET equivalent of the ABL Browse widget, to display customer fields. I drag the control onto the form, and immediately its **SmartTag** property sheet pops up to let me pick a data source for the grid, and I'll choose the **Customer ProBindingSource**. This is why it made sense to define the binding source first, so that the visual controls then have something to point to to get their data from.



The grid picks up the field datatypes and column labels from the binding source, which in turn got them from the database schema. But I can make changes to those values by selecting **Edit Columns**. For instance, I can pick a column such as the first name and scroll through its properties, and adjust the **HeaderText** property, which is the column label. and do the same for the Last Name column.



And likewise I can go through the grid's properties in the **Properties View**, and rename the control to **CustomerGrid**.



Now that I've added the grid, I can take a look at the code that got generated.

Once again, the definitions of object variables that Visual Designer generates go here in the main block of the class, and you need to be careful not to disturb them when you add your own code to the class. The other thing you'll notice is that there's a variable to hold the reference to the grid itself, and also one for each of the columns. So each column in the grid is its own object. Most of them are of the type **TextBoxColumn**, though **Gender**, which is defined in the database schema as a LOGICAL field, is represented as a **CheckBoxColumn**.

```
CLASS CustomerForm INHERITS Form   :

    DEFINE PRIVATE VARIABLE BSCustomer AS Progress.Data.BindingSource NO-UNDO.
    DEFINE PRIVATE VARIABLE components AS System.ComponentModel.IContainer NO-UNDO.
    DEFINE PRIVATE VARIABLE customerLastNameDataGridViewTextBoxColumn AS
        System.Windows.Forms.DataGridViewTextBoxColumn NO-UNDO.
    DEFINE PRIVATE VARIABLE customerIDDataGridViewTextBoxColumn AS
        System.Windows.Forms.DataGridViewTextBoxColumn NO-UNDO.
    DEFINE PRIVATE VARIABLE customerGenderDataGridViewCheckBoxColumn AS
        System.Windows.Forms.DataGridViewCheckBoxColumn NO-UNDO.
    DEFINE PRIVATE VARIABLE customerFirstNameDataGridViewTextBoxColumn AS
        System.Windows.Forms.DataGridViewTextBoxColumn NO-UNDO.
    DEFINE PRIVATE VARIABLE customerBirthdateDataGridViewTextBoxColumn AS
        System.Windows.Forms.DataGridViewTextBoxColumn NO-UNDO.
    DEFINE PRIVATE VARIABLE customerBirthCountryDataGridViewTextBoxColumn AS
        System.Windows.Forms.DataGridViewTextBoxColumn NO-UNDO.
    DEFINE PRIVATE VARIABLE CustomerGrid AS System.Windows.Forms.DataGridView
        NO-UNDO.
```

In **InitializeComponent** you can see the **NEW** statements that create instances of the grid and its columns.

```
        METHOD PRIVATE VOID InitializeComponent(  ):

        THIS-OBJECT:CustomerGrid = NEW System.Windows.Forms.DataGridView().
        THIS-OBJECT:customerIDDataGridViewTextBoxColumn = NEW
            System.Windows.Forms.DataGridViewTextBoxColumn().
        THIS-OBJECT:customerFirstNameDataGridViewTextBoxColumn = NEW
            System.Windows.Forms.DataGridViewTextBoxColumn().
        THIS-OBJECT:customerLastNameDataGridViewTextBoxColumn = NEW
            System.Windows.Forms.DataGridViewTextBoxColumn().
        THIS-OBJECT:customerBirthCountryDataGridViewTextBoxColumn = NEW
            System.Windows.Forms.DataGridViewTextBoxColumn().
        THIS-OBJECT:customerBirthdateDataGridViewTextBoxColumn = NEW
            System.Windows.Forms.DataGridViewTextBoxColumn().
        THIS-OBJECT:customerGenderDataGridViewCheckBoxColumn = NEW
            System.Windows.Forms.DataGridViewCheckBoxColumn().
        CAST(THIS-OBJECT:BSCustomer,
            System.ComponentModel.ISupportInitialize):BeginInit().
        CAST(THIS-OBJECT:CustomerGrid,
            System.ComponentModel.ISupportInitialize):BeginInit().
```

In the next section of InitializeComponent, you can see the grid has a property called **AutoGenerateColumns** that's explicitly set to false. This is another example of a property that, on the one hand, isn't displayed in the Properties View, but is assigned an explicit initial value. You could set this property to true programmatically if you wanted a grid that would generate columns dynamically at runtime based on the columns in its data source.

```
        /*  */
        /* CustomerGrid */
        /*  */
        THIS-OBJECT:CustomerGrid:AutoGenerateColumns = FALSE.
```

You can also see that the grid column objects are placed into an array and then the **AddRange** method is used to add them as a collection to the **Columns** property of the grid. So this is how they're all tied together.

```
        DEFINE VARIABLE arrayvar2 AS System.Windows.Forms.DataGridViewColumn
            EXTENT 6 NO-UNDO.
        arrayvar2[1] = THIS-OBJECT:customerIDDataGridViewTextBoxColumn.
        arrayvar2[2] = THIS-OBJECT:customerFirstNameDataGridViewTextBoxColumn.
        arrayvar2[3] = THIS-OBJECT:customerLastNameDataGridViewTextBoxColumn.
        arrayvar2[4] = THIS-OBJECT:customerBirthCountryDataGridViewTextBoxColumn.
        arrayvar2[5] = THIS-OBJECT:customerBirthdateDataGridViewTextBoxColumn.
        arrayvar2[6] = THIS-OBJECT:customerGenderDataGridViewCheckBoxColumn.
        THIS-OBJECT:CustomerGrid:Columns:AddRange(arrayvar2).
```

The grid's **Location** is set using a **Point** object that defines the coordinates of a point, and the **Size** property is set using a **Size** object that defines its two dimensions. And most important, the grid's **DataSource** property is set to the binding source instance. DataSource is a common property of all data bound .NET controls , and the OpenEdge-specific ProBindingSource control was created to satisfy the expectations for a data source in a way that supports ABL data management.

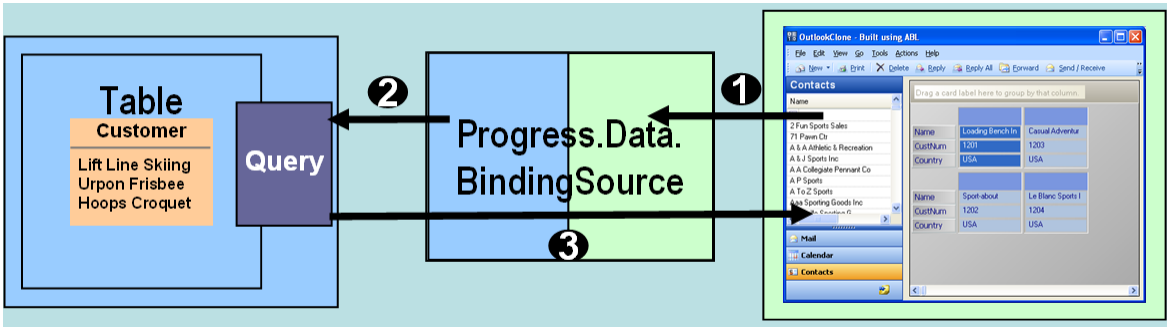
```
        THIS-OBJECT:CustomerGrid:DataSource = THIS-OBJECT:BSCustomer.
        THIS-OBJECT:CustomerGrid:Location = NEW System.Drawing.Point(13, 28).
        THIS-OBJECT:CustomerGrid:Name = "CustomerGrid".
        THIS-OBJECT:CustomerGrid:Size = NEW System.Drawing.Size(450, 226).
        THIS-OBJECT:CustomerGrid:TabIndex = 0.
```

Then the code assigns basic properties to each of the column objects, including the **HeaderText** that represents the column labels.

```
        /*  */
        /* customerIDDataGridViewTextBoxColumn */
        /*  */
        THIS-OBJECT:customerIDDataGridViewTextBoxColumn:DataPropertyName =
"CustomerID".
        THIS-OBJECT:customerIDDataGridViewTextBoxColumn:HeaderText = "Customer ID".
        THIS-OBJECT:customerIDDataGridViewTextBoxColumn:Name =
"customerIDDataGridViewTextBoxColumn".
        …
        /*  */
        /* CustomerForm */
        /*  */
        THIS-OBJECT:ClientSize = NEW System.Drawing.Size(475, 266).
        THIS-OBJECT:Controls:Add(THIS-OBJECT:CustomerGrid).
        THIS-OBJECT:Name = "CustomerForm".
        THIS-OBJECT:Text = "Customers".
        CAST(THIS-OBJECT:BSCustomer,
            System.ComponentModel.ISupportInitialize):EndInit().
        CAST(THIS-OBJECT:CustomerGrid,
            System.ComponentModel.ISupportInitialize):EndInit().
        THIS-OBJECT:ResumeLayout(FALSE).
        CATCH e AS Progress.Lang.Error:
            UNDO, THROW e.
        END CATCH.
       END METHOD.

…

END CLASS.
```



Now I have the grid control attached to the binding source control as its DataSource, as shown in step one of this illustration. I defined the binding source in terms of the Customer database table, but now I need to actually bind it to a query on the customer table, as shown as step 2, in order that the grid control can actually display that data, as shown in step 3.

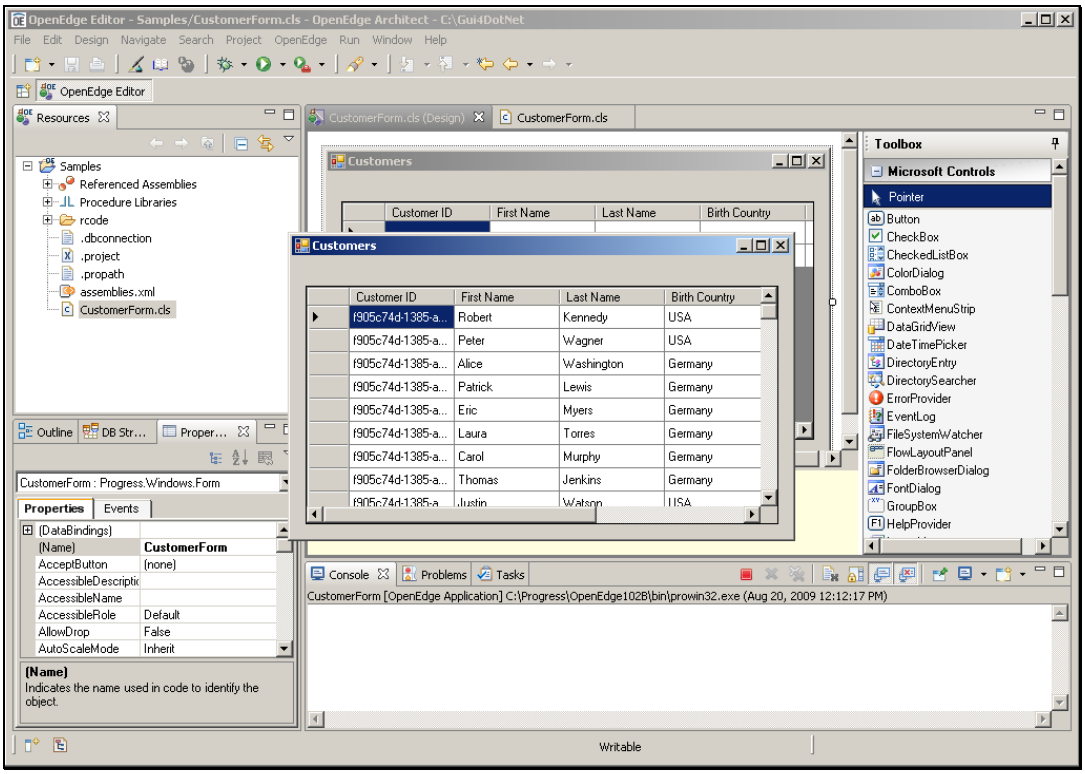
I add a `HANDLE` definition to the class's main block. Note that because this definition is in the main block of the class, it is a *data member* that can have an access qualifier of PUBLIC, PROTECTED, or PRIVATE, and the default is PUBLIC, which would make the variable accessible from other classes. Because it will be accessed only within this class, I define it as PRIVATE. In general, if you want a value to be read and/or set from other classes, you should define it as a property instead of a variable, so that you can define GETter and SETter access code to control its use, and keep your data member variables PRIVATE or PROTECTED.

In the constructor, following the statement that invokes **InitializeComponent**, I create an ABL query, set it to to use the Customer buffer for the database table, prepare the query to retrieve all the AutoEdge Customers, and open the query. Now I need to make that query handle the **Handle** property of the binding source. A ProBindingSource can manage data from a database query, a buffer holding a single row, a temp-table, or even an entire ProDataSet, depending on what you set its Handle property to.

```
DEFINE PRIVATE VARIABLE hCustQuery AS HANDLE NO-UNDO.

        CONSTRUCTOR PUBLIC CustomerForm (  ):
            SUPER().
            InitializeComponent().
            CREATE QUERY hCustQuery.
            hCustQuery:SET-BUFFERS(BUFFER Customer:HANDLE).
            hCustQuery:QUERY-PREPARE("FOR EACH Customer").
            hCustQuery:QUERY-OPEN ().
            BSCustomer:HANDLE = hCustQuery.
            CATCH e AS Progress.Lang.Error:
                UNDO, THROW e.
            END CATCH.
        END CONSTRUCTOR.
```

The next screen capture shows the DataViewGrid with all of the AutoEdge customers in it:



Just by defining a ProBindingSource to hold schema for data from my database, then dropping a grid control onto the form, then making the binding source the data source for the grid, and creating a standard ABL query to supply data to the binding source, I have a simple but fully functional .NET form to display my customer data. In this simple test, there's no specific Run Configuration for Architect to use to run my form. But you can't NEW a class directly as the top-level r-code file to run in an OpenEdge session. There has to be a procedural wrapper around it. So what happens when I just ask Architect to run my form? We can find the answer *while the form is running*, because there's a temporary procedure file that's been generated for me that will be deleted when the run completes.

The temporary ABL procedure is placed into this folder relative to the project:

`.metadata\.plugins\.com.openedge.pdt.project\temp.dir.`

Looking in that folder, you can see a procedure whose name begins **RunClass** with my form name in its name. Here is the ABL procedure that's acting as a wrapper around my class file.

```
USING System.Windows.Forms.Application FROM ASSEMBLY.
DEFINE VARIABLE rTemp AS CLASS CustomerForm NO-UNDO.
DO ON ERROR  UNDO, LEAVE
        ON ENDKEY UNDO, LEAVE
        ON STOP   UNDO, LEAVE
        ON QUIT   UNDO, LEAVE:
        rTemp = NEW CustomerForm ( ) .
        WAIT-FOR System.Windows.Forms.Application:Run ( rTemp ).
        CATCH e1 AS Progress.Lang.AppError:
               MESSAGE e1:ReturnValue VIEW-AS ALERT-BOX BUTTONS OK TITLE "Error".
        END CATCH.
        CATCH e2 AS Progress.Lang.Error:
          DEFINE VARIABLE i AS INTEGER NO-UNDO.
          DO i = 1 TO e2:NumMessages:
            MESSAGE e2:GetMessage(i) VIEW-AS ALERT-BOX BUTTONS OK TITLE "Error".
          END.
        END CATCH.
END.
FINALLY.
IF VALID-OBJECT(rTemp) THEN DELETE OBJECT rTemp NO-ERROR.
END FINALLY.
```

Significantly, it defines a variable to hold an instance of my form class, creates a NEW instance of that class, which runs the r-code for the class, and then uses the same WAIT-FOR statement that has always been part of the ABL GUI to wait for, in this case, the completion of the Run event for the form. Then it does error checking and cleans up. It's important to be aware of the fact that this procedure is here for you, because your finished application will have to have a similar startup procedure that creates instances of your classes and does a WAIT-FOR similar to this one.

So that's all for this session. When I close the customer form, that satisfies the WAIT-FOR in the wrapper procedure, and Architect cleans up by deleting that from the temp directory.