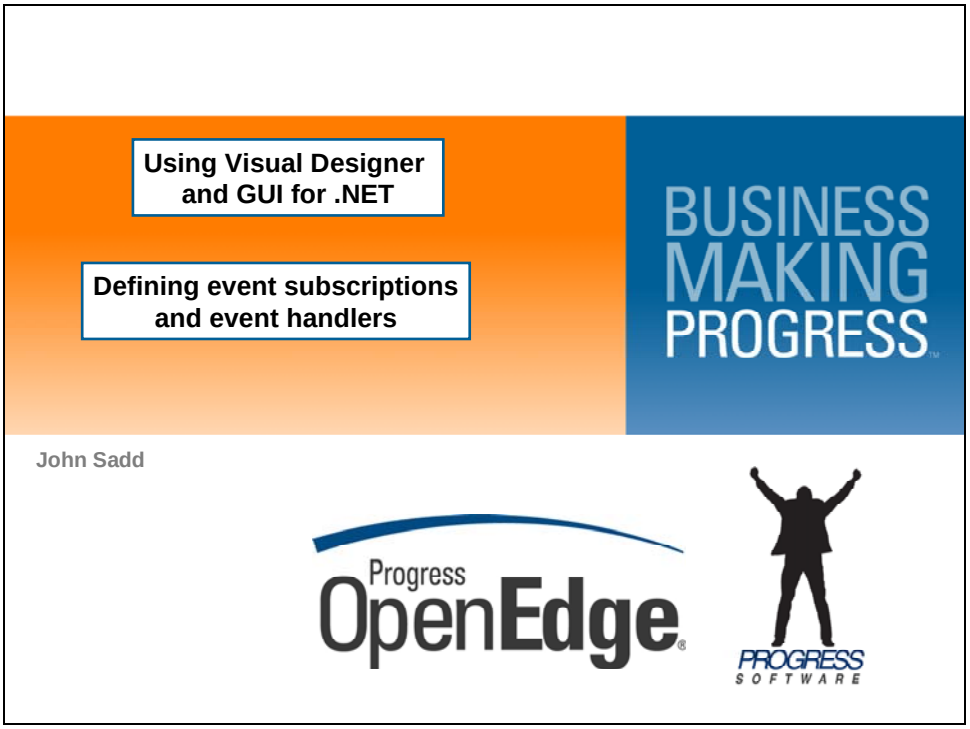
DEFINING EVENT SUBSCRIPTIONS AND EVENT HANDLERS

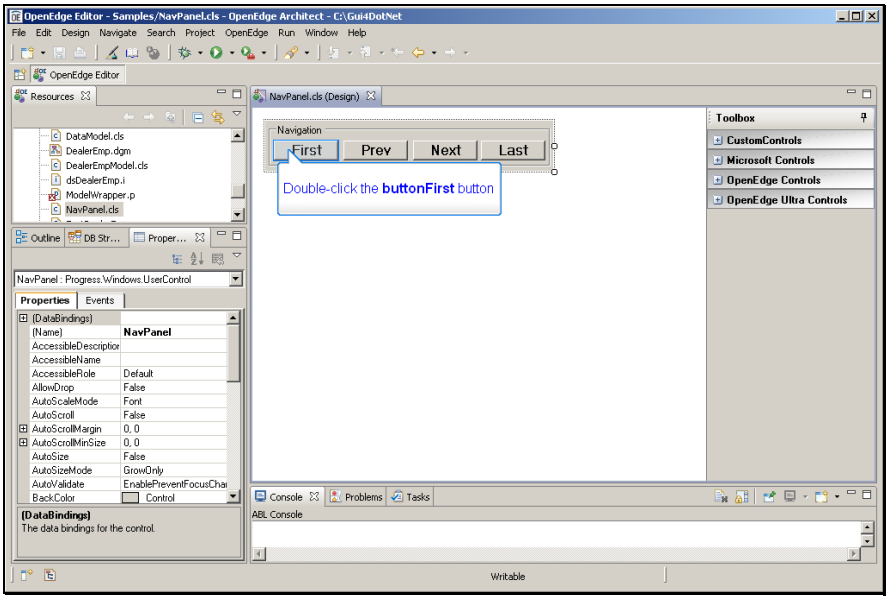
John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
November 2009



DISCLAIMER

This document accompanies a series of presentations on using Visual Designer in OpenEdge Architect, and support for GUI for .NET in OpenEdge 10.2. The material covered by this document is derived from a two-part video session that adds event handlers to the **Navigation Panel User Control** built in the session titled **Building a Navigation Panel as an ABL User Control**. The event handlers extend the navigation panel to cause it to respond to the click event on the navigation buttons. Code added to the event handlers repositions the binding source that drives the grid in a Customer form.

The **NavPanel** class is an **ABL User Control** containing a panel of buttons. This document shows how to create code that will respond to the user clicking on each of the four buttons. Each control type can generate many different kinds of events that your application may want to respond to in different situations. But every control has a default event that is the one you are expected to use most often. In the case of a simple control like a button, identifying the default event is usually obvious; for a button it's the **Click** event. All you have to do in Visual Designer is double-click the control in the Design view to have the basic code generated for that default event. For example, I can double-click on the First button in the NavPanel, as shown here:



When I do that I'm automatically brought into the editor to see the code Visual Designer has generated.

There are two pieces to the generated code. What you see first in the code box that follows is the skeleton for the event handler itself, the method that gets run when the event occurs.

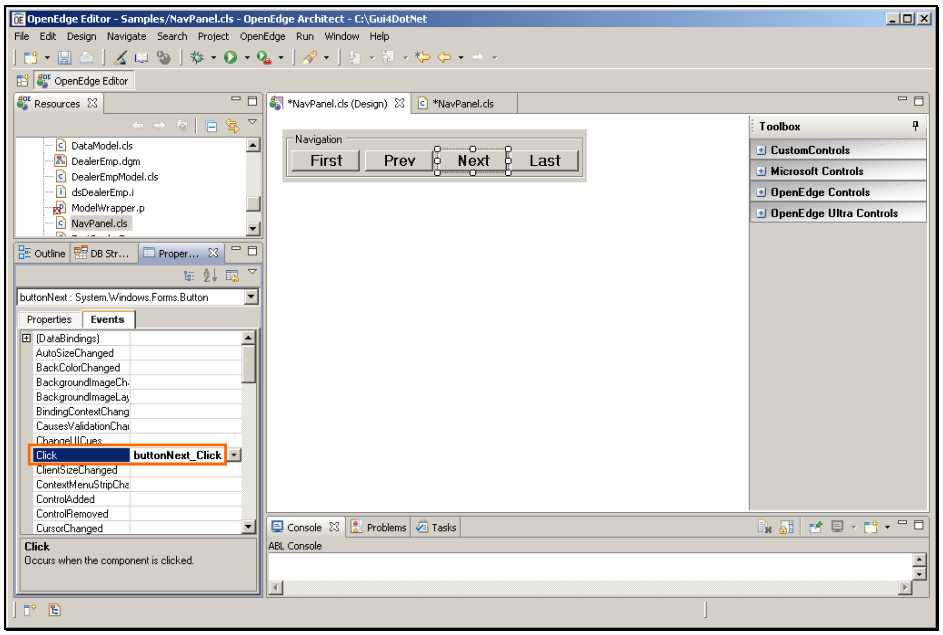
```
@VisualDesigner.
METHOD PRIVATE VOID buttonFirst_Click( INPUT sender AS System.Object,
    INPUT e AS System.EventArgs ):
        RETURN.
END METHOD.
```

You can see that the default name for the method is the control name plus the event name. You can also see that event handlers always take two parameters. The first, usually named **sender**, is the object reference to the control that generated the event. The second, typically just named **e**, is an object containing whatever information the control provides to identify the specifics of the event. Sometimes you will care about these parameters, and sometimes you don't really need them, because you just need to know that the event occured. Later I'll fill this method in with code to respond to a click on the **First** button.

Next let's take a look at the other piece of code that was generated. As part of the code in **InitializeComponent** that sets properties for the **First** button, there's a new statement that subscribes to the **Click** event for the **First** button. The argument to the **Subscribe** method specifies the ABL method we just looked at as the one to run when the event occurs. This statement makes the connection between the control event and the event handler method that supplies the logic to handle the event.

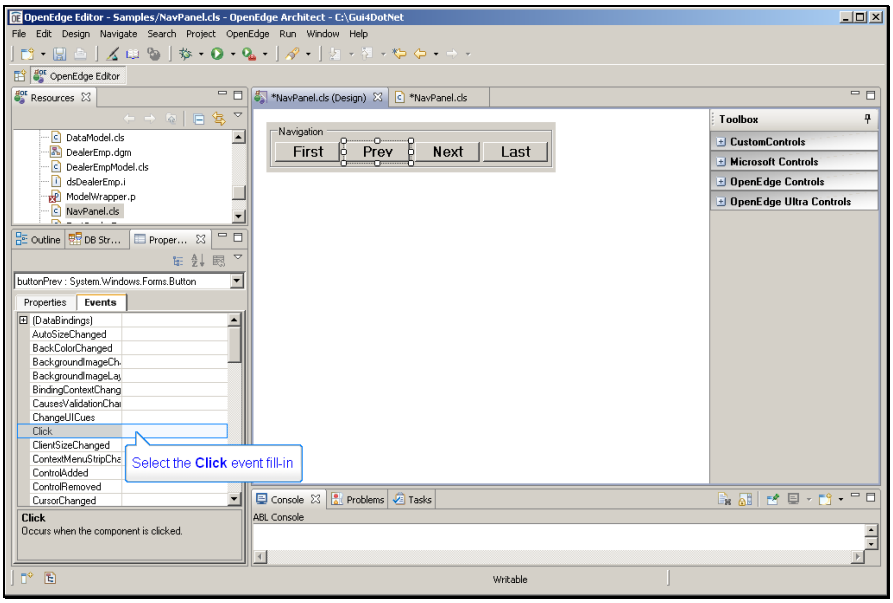
```
THIS-OBJECT:buttonFirst:Click:Subscribe(THIS-OBJECT:buttonFirst_Click).
```

Double-clicking on a control gives you a default event handler for the default event. You can also get an event handler for any event the control supports by selecting the control and then selecting the **Events** tab in the **Properties View**. If I select the **Next** button, for example, I can generate an event handler for any event the control supports by double-clicking on the event name. I can get a **Click** event handler in this way, too, as an alternative to double-clicking the control itself. The following screen capture shows the generated event handler name assigned when I double-click the event name in the Events tab:

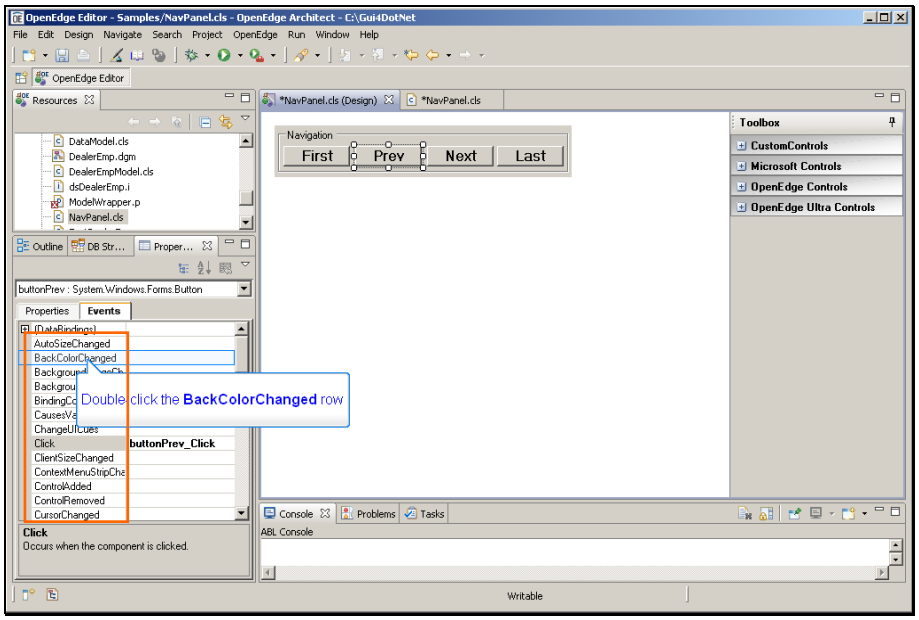


Visual Designer then takes you back to the code view so that you can edit code into the event handler.

There's a third variation for generating event handlers. If I simply click the fill-in next to the event name for any event in the Properties View, as shown below, I can type the name of the method and tab out of the field. This can be useful I don't want the default method name.



The examples so far have shown different ways to get an event handler and subscription for the default event for a control. You can get an event handler for *any* event by double-clicking on its name in the Properties View (to get an event handler with the default method name), or by selecting it and typing in the event handler name you want. Just by selecting a control in your form and then selecting the Events tab, you can see what a variety of events even a simple control like a button supports if they should be of use to you. To show how to generate an event handler for an event other than the default event, I can just pick one such as **BackColorChanged** and double-click on it.

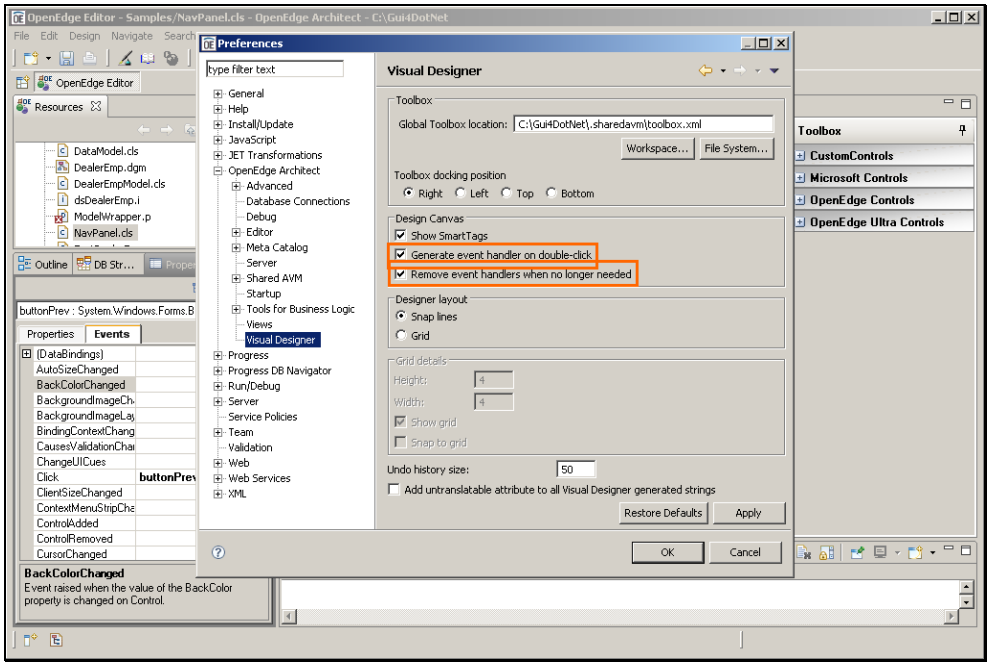


Here's the skeleton for an event handler that would be executed if the background color of the button were changed.

```
@VisualDesigner.
METHOD PRIVATE VOID buttonPrev_BackColorChanged( INPUT sender AS System.Object,
    INPUT e AS System.EventArgs ):
        RETURN.
END METHOD.
```

Having created this event handler that I don't really want, I can then show you how to get rid of one. To get rid of an event handler and its subscription, just erase the name and tab out of the event name field in the Properties View. If I do this for the BackColorChanged event its event handler and subscription are now gone.

You can imagine that you might not always want that to have event handlers deleted automatically. If you have put a lot of code into an event handler you wouldn't want to risk losing it just because you removed the subscription or you inadvertently erased the handler name. Understanding how to control this aspect of Visual Designer's behavior provides a good opportunity to look at the **Preferences** dialog for Visual Designer. It's under **OpenEdge Architect** -> **Visual Designer** in the **Window -> Preferences** menu.



The first option of interest here is the checkbox labelled **Generate Event Handler on double-click**. This is the default behavior the examples in this document have been showing you. If for some reason you didn't want the event handler to be generated automatically when you double-click on a control, you could check this option off.

The next check box is labelled **Remove event handlers when no longer needed**. If you don't want Visual Designer to delete an event handler method just because you erase or change its name, you can check this off. Then the event handler code is still there for you to reuse or re-activate later. For instance, if you rename an event handler in the Events tab, by default the old event handler method is deleted as the one under the new name is created. That's one reason why you might want to check off this option to remove the handler automatically. That said, it's advisable to design your application so that the logic for dealing with events is in a separate class from the actual UI control event handler methods, which would eliminate any chance of valuable event handler logic in the UI class being deleted inadvertently, but at least now you know where you can maintain the options that you have.

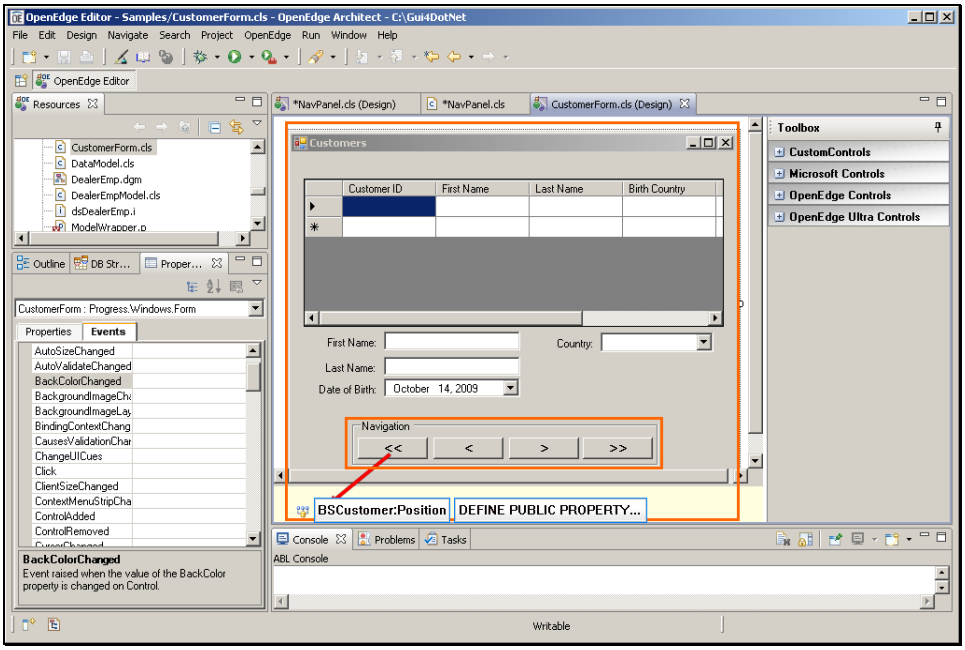
Finally, just to reinforce the default behavior one more time, if I double-click on the **Last** button in the Design view, I get an event handler and subscription for it:

```
@VisualDesigner.
METHOD PRIVATE VOID buttonLast_Click( INPUT sender AS System.Object,
    INPUT e AS System.EventArgs ):
        RETURN.
END METHOD.
```

Since I was just showing you how to prevent Visual Designer from deleting event handler methods that it thinks are not needed, there's a way to do that for an individual method as well. The annotation line above each method definition that says **`@VisualDesigner`** identifies the method as being managed by Visual Designer. If you want to protect a particular method from deletion, you can delete the annotation line above the method definition. Be cautioned that this is the *only* case where deleting Visual Designer annotations is recommended. Otherwise you should never touch this or other code generated by Visual Designer.

Next I add the code to the event handlers to reposition the binding source that drives the grid in a Customer Form.

Now that I've generated the basic event handlers and subscriptions, let's think about the code that has to go into them. My **NavPanel User Control** class has to communicate with the binding source in the containing form, in my example the **CustomerForm** class. With CustomerForm.cls open, let's take a look at what the event handlers are going to have to do. Remember that the panel of buttons is in a separate class file from the grid and its binding source. Navigating is a matter of setting the **Position** property of the binding source. When I change that, the position in the grid and the row values displayed in the fields reflect the change in the binding source position. So I need to make the binding source object in the form into a public property so the navigation panel button event handlers can change its **Position** property, as shown here:



Following the variable definitions in the **CustomerForm** class I need to define a property to hold a reference to the form's binding source so that the **NavPanel** class can access it. It's a **`PUBLIC`** property to make it accessible from another class, and it's of type **`Progress.Data.BindingSource`**. It needs a default **`GET`**ter so it can be read. But I make the **`SET`**ter private so that no other class can change its value.

```
DEFINE PUBLIC PROPERTY FormBindingSource AS PROGRESS.Data.BindingSource
        GET.
        PRIVATE SET.
```

Down in the constructor that's run when the class is instantiated, I add a line of code assigning the value of the binding source object reference to the new property, and save and compile this change to the form.

```
        FormBindingSource = THIS-OBJECT:BSCustomer.
```

Now I go back into the navigation panel and fill in the code for the button **Click** event handlers.

The first thing I need to do is tell the compiler to search for any unqualified control references in the namespace **System.Windows.Forms**, by adding this third **USING** statement to the two generated for me:

```
USING Progress.Lang.*.
USING Progress.Windows.UserControl.
USING System.Windows.Forms.*.
```

Doing this saves me from having to fully qualify every reference to my Button controls, for instance. This is a bit like extending the ProPath for ABL classes and procedures.

Now I can enter the code for the **First** button event handler. Remember that the first argument to an event handler is the object that generated the event, called **sender**. That's my **Button**, and I'm going to reference that object as a **Button** object, so I define a variable to hold the reference.

```
  DEFINE VARIABLE EventButton AS Button.
```

Next I want a variable to hold a reference to the parent form where the binding source is. In a more serious application I would certainly create a super class for my forms where the **FormBindingSource** property would be defined, and an Interface to assure consistent access for a Customer form, an Order form, and other forms that should all work the same way. In this simplified case the property is just defined directly in the **CustomerForm** class, so I make my reference of that type.

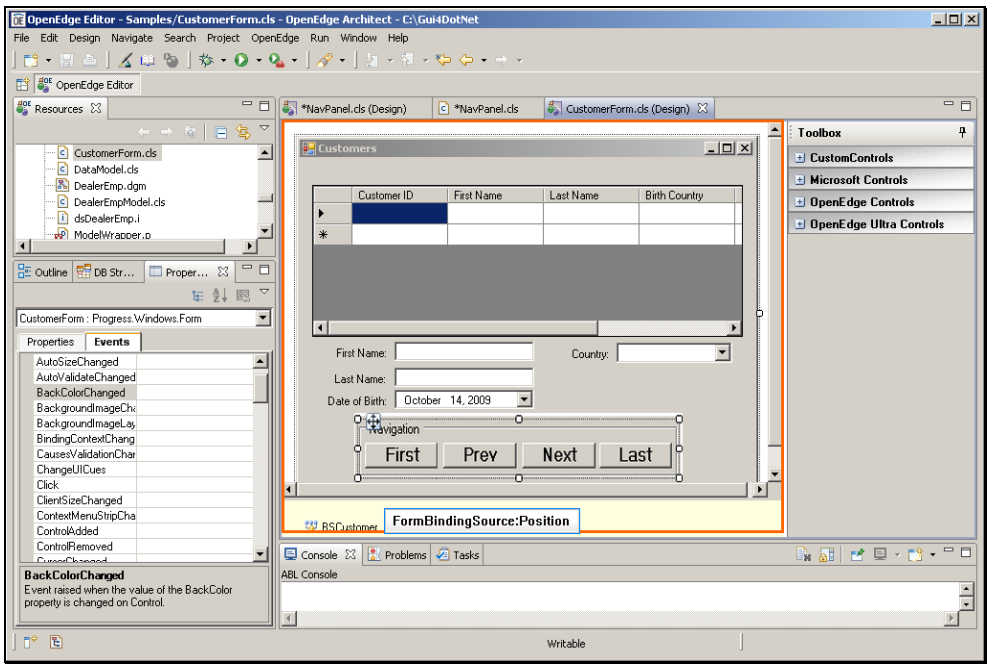
```
  DEFINE VARIABLE ParentForm AS CustomerForm.
```

These last two statements illustrate how the class-based extensions to ABL let you write exactly the same kinds of language statements to reference .NET objects (**System.Windows.Forms.Button** in this case) as well as instances of ABL classes you define (such as **CustomerForm**).

The **CAST** function in the next statement tells the compiler to view the **sender** parameter as a **Button**. This is necessary because I am going to reference its **Parent** property. **System.Object** – which is how **sender** is defined in the default event handler signature -- is such a minimal high-level class that it doesn't even define the **Parent** property, so I have to make the reference more specific to tell the compiler to allow my reference to the **Parent** property in the object reference passed into the handler.

```
  EventButton = CAST(sender,Button).
```

To understand how the code is going to walk up the chain of parents to get to the **CustomerForm**, let's take another look at the chain of objects. Every control has a **Parent** property. In this case, the immediate parent of each button is the **GroupBox**. The **Parent** of the **GroupBox** is the **NavPanel** User Control. The **Parent** of the **NavPanel** is the containing **CustomerForm**. So the code has to go up three levels of parents to get to the form, in order to reference the form's binding source property.



So here's the statement that locates the **CustomerForm** and puts it in the **ParentForm** variable:

```
ParentForm = CAST(EventButton:Parent:Parent:Parent,CustomerForm).
```

In effect, the statement says: Start with the **EventButton** object which I've associated with the **sender** input parameter to the event handler method. Retrieve the object held by its **Parent** property. (This is the **GroupBox** object.) Then retrieve *its* **Parent** (the **NavPanel** class instance.) Then in turn retrieve *its* **Parent** object (the **CustomerForm** that holds the **NavPanel**). Treat this object reference as an instance of the class **CustomerForm**. Assign that object reference to the local **ParentForm** variable.

As it happens, there's a method called **FindForm** in the .NET **Control** namespace that locates the parent form more flexibly, returning the first **Parent** object it finds in the chain that is a form, so you could also express the statement that locates the **ParentForm** like this:

```
ParentForm = CAST(FindForm(),CustomerForm).
```

Now the code for the **First** button event handler can set the value of the binding source **Position** property to zero, the first row. When I press **Ctrl-Shift-C** to do a syntax check, the compiler is examining both of these classes -- the **NavPanel** class I'm coding and the **CustomerForm** class that it referenced -- to make sure that all the references make sense, for instance that the **CustomerForm** class that I'm **CAST**ing to contains a **FormBindingSource** property, and that that property holds a reference to an object that has a **Position** property.

Here's the complete code for the **First** button event handler:

```
@VisualDesigner.
METHOD PRIVATE VOID buttonFirst_Click( INPUT sender AS System.Object,
    INPUT e AS System.EventArgs ):
  DEFINE VARIABLE EventButton AS Button.
  DEFINE VARIABLE ParentForm AS CustomerForm.
  EventButton = CAST(sender,Button).
  ParentForm = CAST(EventButton:Parent:Parent:Parent,CustomerForm).
  /* or: ParentForm = CAST(FindForm(),CustomerForm). */
  ParentForm:FormBindingSource:Position = 0.

  RETURN.
END METHOD.
```

Next is the similar code for the **Last** button event handler. The **Count** property of the binding source is the number of rows the binding source is managing. It's a one-based count, and the **Position** is a zero-based property, so the **Position** of the last row is one less than the row count.

```
@VisualDesigner.
METHOD PRIVATE VOID buttonLast_Click( INPUT sender AS System.Object,
    INPUT e AS System.EventArgs ):
  DEFINE VARIABLE EventButton AS Button.
  DEFINE VARIABLE ParentForm AS CustomerForm.
  EventButton = CAST(sender,Button).
  ParentForm = CAST(EventButton:Parent:Parent:Parent,CustomerForm).
  ParentForm:FormBindingSource:Position =
                    ParentForm:FormBindingSource:Count - 1.

  RETURN.
END METHOD.
```

The **Next** button event handler increments **Position**, checking that it's not already at the end:

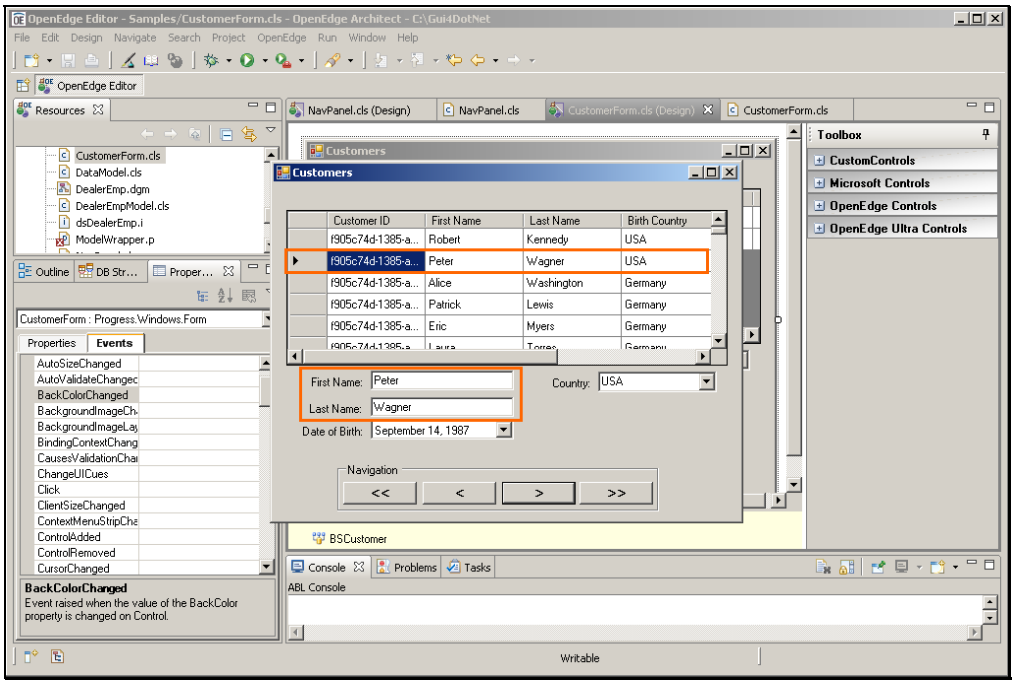
```
@VisualDesigner.
METHOD PRIVATE VOID buttonNext_Click( INPUT sender AS System.Object,
    INPUT e AS System.EventArgs ):
  DEFINE VARIABLE EventButton AS Button.
  DEFINE VARIABLE ParentForm AS CustomerForm.
  EventButton = CAST(sender,Button).
  ParentForm = CAST(EventButton:Parent:Parent:Parent,CustomerForm).
  IF ParentForm:FormBindingSource:Position <
                    ParentForm:FormBindingSource:Count - 1 THEN
                      ParentForm:FormBindingSource:Position =
                        ParentForm:FormBindingSource:Position + 1.

  RETURN.
END METHOD.
```

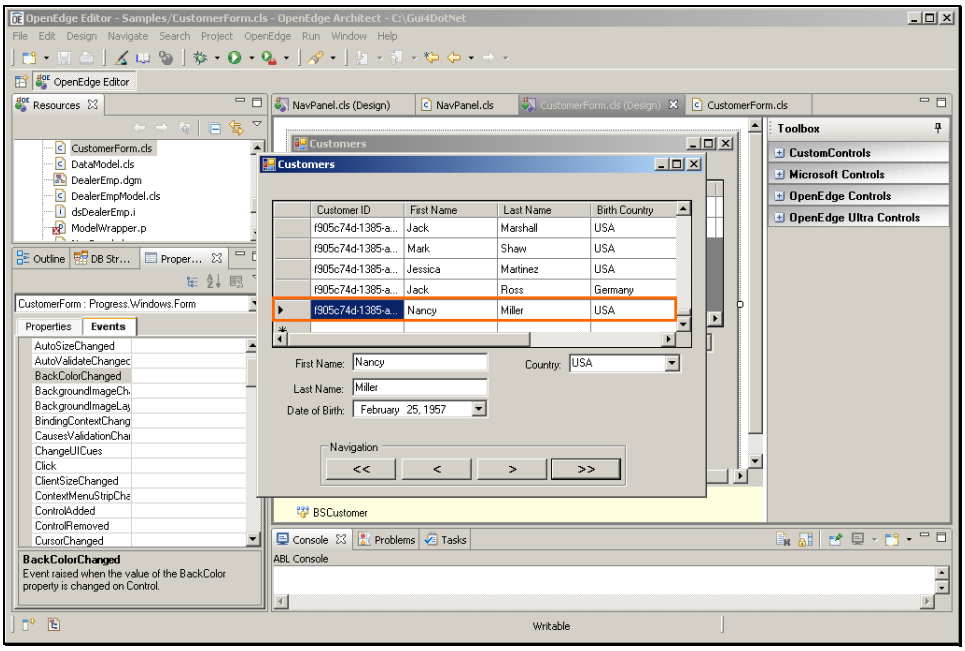
Finally, the **Prev** button event handler decrements the value of **Position**:

```
@VisualDesigner.
METHOD PRIVATE VOID buttonPrev_Click( INPUT sender AS System.Object,
    INPUT e AS System.EventArgs ):
  DEFINE VARIABLE EventButton AS Button.
  DEFINE VARIABLE ParentForm AS CustomerForm.
  EventButton = CAST(sender,Button).
  ParentForm = CAST(EventButton:Parent:Parent:Parent,CustomerForm).
  IF ParentForm:FormBindingSource:Position > 0 THEN
                    ParentForm:FormBindingSource:Position =
                      ParentForm:FormBindingSource:Position - 1.
  RETURN.
END METHOD.
```

Now I can save all of this new ABL, go back to the **CustomerForm** and run it to see if the new code all works. If I click the **Next** button, both the grid and the field-level controls reposition in response to the change to the binding source **Position** value:



I can try out the **First** button. and the **Last** button, and it all seems to be working.



Now that everything works, I'll do what programmers always do and change the code. ☺

You will have noticed that there was a lot of duplicated code in the four button **Click** event handlers. In fact, it's perfectly OK to use the same event handler method for multiple events. These are the steps I must go through to unify the code into a single method:

First I can generalize the name of the event handler to just **Button_Click**. Here's a case where you would want to think about the default Visual Designer behavior of deleting an event handler when it's no longer used. If I change the name of the event handler for the **Click** event on the **First** button in the **Properties View**, the **ButtonFirst_Click** event handler is gone, and I've got an empty new one in its place. So be sure you don't lose code when you make a change such as this. After consolidating the logic from all four individual click event handlers, here's the code for a single event handler for all four events:

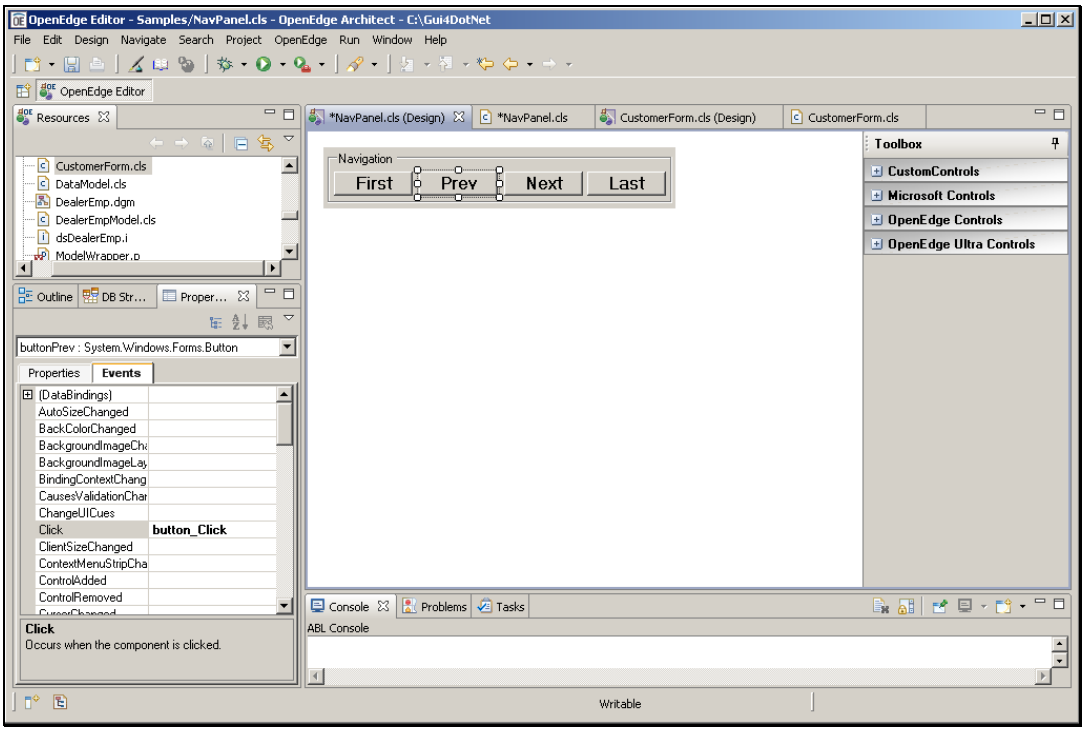
```
@VisualDesigner.
METHOD PRIVATE VOID button_Click( INPUT sender AS System.Object,
    INPUT e AS System.EventArgs ):
        DEFINE VARIABLE ParentForm AS CustomerForm.
        DEFINE VARIABLE EventButton AS Button.
        EventButton = CAST(sender,Button).
        ParentForm = CAST(EventButton:Parent:Parent:Parent,CustomerForm).
        CASE EventButton:Name:
            WHEN "buttonFirst" THEN
                ParentForm:FormBindingSource:Position = 0.
            WHEN "ButtonPrev" THEN
                IF ParentForm:FormBindingSource:Position > 0 THEN
                    ParentForm:FormBindingSource:Position =
                      ParentForm:FormBindingSource:Position - 1.
            WHEN "ButtonNext" THEN
                IF ParentForm:FormBindingSource:Position <
                     ParentForm:FormBindingSource:Count - 1 THEN
                       ParentForm:FormBindingSource:Position =
                         ParentForm:FormBindingSource:Position + 1.
            WHEN "ButtonLast" THEN
                ParentForm:FormBindingSource:Position =
                    ParentForm:FormBindingSource:Count - 1.
            END CASE.

        RETURN.

END METHOD.
```

As you can see in the **CASE** statement, the method first identifies which button generated the event – remember that this got passed in as the **sender** parameter -- by looking at the **EventButton**'s **Name** property. Then I can put all four logic variations into the same event handler.

To finish up, I change the **Click** event for the other three buttons in the **Events** tab of the **Properties View** to run the same unified event handler. When I do this, the **Subscribe** statements are adjusted and the old event handlers cleaned up automatically. For each of the other three buttons, I select its **Click** event handler name, and change the name to my unified event handler. Each time I'm placed back into the code view, but of course there are no further code changes to be made.



If I scroll down through the code when I'm done, I see that the old event handler methods have all been removed, and the **button_Click** method has taken their place:

```
CONSTRUCTOR PUBLIC NavPanel (  ):

        SUPER().
        InitializeComponent().
        CATCH e AS Progress.Lang.Error:
            UNDO, THROW e.
        END CATCH.

END CONSTRUCTOR.




/*------------------------------------------------------------------------------
        Purpose:
        Notes:
------------------------------------------------------------------------------*/
        @VisualDesigner.
        METHOD PRIVATE VOID button_Click( INPUT sender AS System.Object, INPUT e AS
System.EventArgs ):
```

Scrolling down to where the button properties are set, I see that the **Subscribe** methods have been adjusted to run the new unified event handler for all the **Click** events.

```
/*  */
/* buttonLast */
/*  */
THIS-OBJECT:buttonLast:Font = NEW System.Drawing.Font
   ("Microsoft Sans Serif", 12, System.Drawing.FontStyle:Bold,
      System.Drawing.GraphicsUnit:Point, System.Convert:ToByte(0)).
THIS-OBJECT:buttonLast:Location = NEW System.Drawing.Point(247, 19).
        . . .
THIS-OBJECT:buttonLast:Click:Subscribe(THIS-OBJECT:button_Click).
/*  */
/* buttonNext */
/*  */
        . . .
THIS-OBJECT:buttonNext:Click:Subscribe(THIS-OBJECT:button_Click).

        . . .
THIS-OBJECT:buttonPrev:Click:Subscribe(THIS-OBJECT:button_Click).
        . . .
THIS-OBJECT:buttonFirst:Click:Subscribe(THIS-OBJECT:button_Click).
```

If I save and compile these changes to the class, I'm able to test out my changes to make sure the event handler now is run for all four events.

That's all for this description of the two-part video session on defining event subscriptions and event handlers, which introduced you to event subscriptions and event handlers and how Visual Designer supports you in generating and using them.