Micro Focus Dialog System Runtime for Visual COBOL

Modernizing Dialog System Applications

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
http://www.microfocus.com

2011-11-24

Contents

Modernizing Dialog System Applications

Overview of Modernizing Dialog System Applications

The Dialog System AddPack enables you to modernize Dialog System applications within Visual COBOL for Visual Studio. You upgrade an application to Visual COBOL and from there, you can run the application without change, or modernize it over time. The application runs under the COBOL 2010 Runtime and the Dialog System run-time system.

The first stage is to import the application into Visual COBOL, and there is an import wizard to help. You can then build and run the application from Visual COBOL.

From then on, you can edit and maintain the application from within Visual COBOL. The screensets are referenced in the Visual COBOL project, and you can double-click a screenset to start Dialog System and edit the screenset. In this way, you can continue maintaining your application with Visual COBOL until you are ready to modernize it.

The next stage is to modernize the application gradually, as much or as little as you want, keeping other code unchanged. There is a range of techniques for modernization. For example, you can replace one Dialog System screen with a Windows Form or you can wrap a .NET user control as an ActiveX and use that in Dialog System.

The AddPack provides a number of samples to demonstrate the various modernization techniques, and there is supporting documentation in this Help explaining the significant elements of the code. Some samples use the same code as in Net Express, and have the key difference that they use the Visual COBOL version of the COBOL and Dialog System run-time systems.

Finally, to fully modernize, you use Microsoft tooling, the .NET Framework and Microsoft interoperability techniques.

Note: The Dialog System AddPack is not part of Visual COBOL or COBOL 2010 Runtime. It is separately installable and available from Micro Focus SupportLine

Dialog System AddPack Information and Restrictions

The Dialog System AddPack comprises the Dialog System run-time components, with a subset of the development components. The AddPack comprises:

- Dialog System run-time system and run-time components.
- Panels V2.
- GUI class library and OLE class library. These libraries are needed if you migrate an existing Dialog System application that was extended using those libraries.

  Projects for building the GUI and OLE class libraries from source are also supplied. Additionally, a project file for the Base class library is supplied in Visual COBOL R4 Update 2.
- Visual Studio plug-in to associate screensets in Visual Studio with Dialog System. When you double-click a screenset in Solution Explorer in Visual COBOL, Dialog System starts.
- Sample applications demonstrating a range of modernization techniques.
- Supporting documentation in this Help explaining the significant elements of the sample code.

Prerequisites and Requirements

The Dialog System AddPack requires the following software:

- Visual COBOL for Visual Studio R4 Update 2. This provides full support for developing and running applications.

  Alternatively, since the AddPack contains the Dialog System run-time components, you can run Dialog System applications (not develop them) if you have COBOL 2010 Runtime installed.
- Net Express (which includes Dialog System) for editing the Dialog System screensets

Restrictions and Limitations

The following restrictions apply to the Dialog System AddPack:

- Several Dialog System extensions are not supported, such as DSGRAPH, DSDDE, DSPLAYER and DSONLINE.
- Screenset script animation is not supported.
- To make changes to a screenset, you continue to use the Dialog System painter.
- Only x86 projects are supported, because there is no 64-bit support for Dialog System.
- Unlike in Net Express, debuggable versions of the class libraries are not installed automatically (and there is no debugger switch to enable use of the debug versions as there is in Net Express). Instead, you need to build the libraries yourself. See *Debuggable OO Class Libraries* in the topic *Migrating a Dialog System Application to Visual COBOL*.

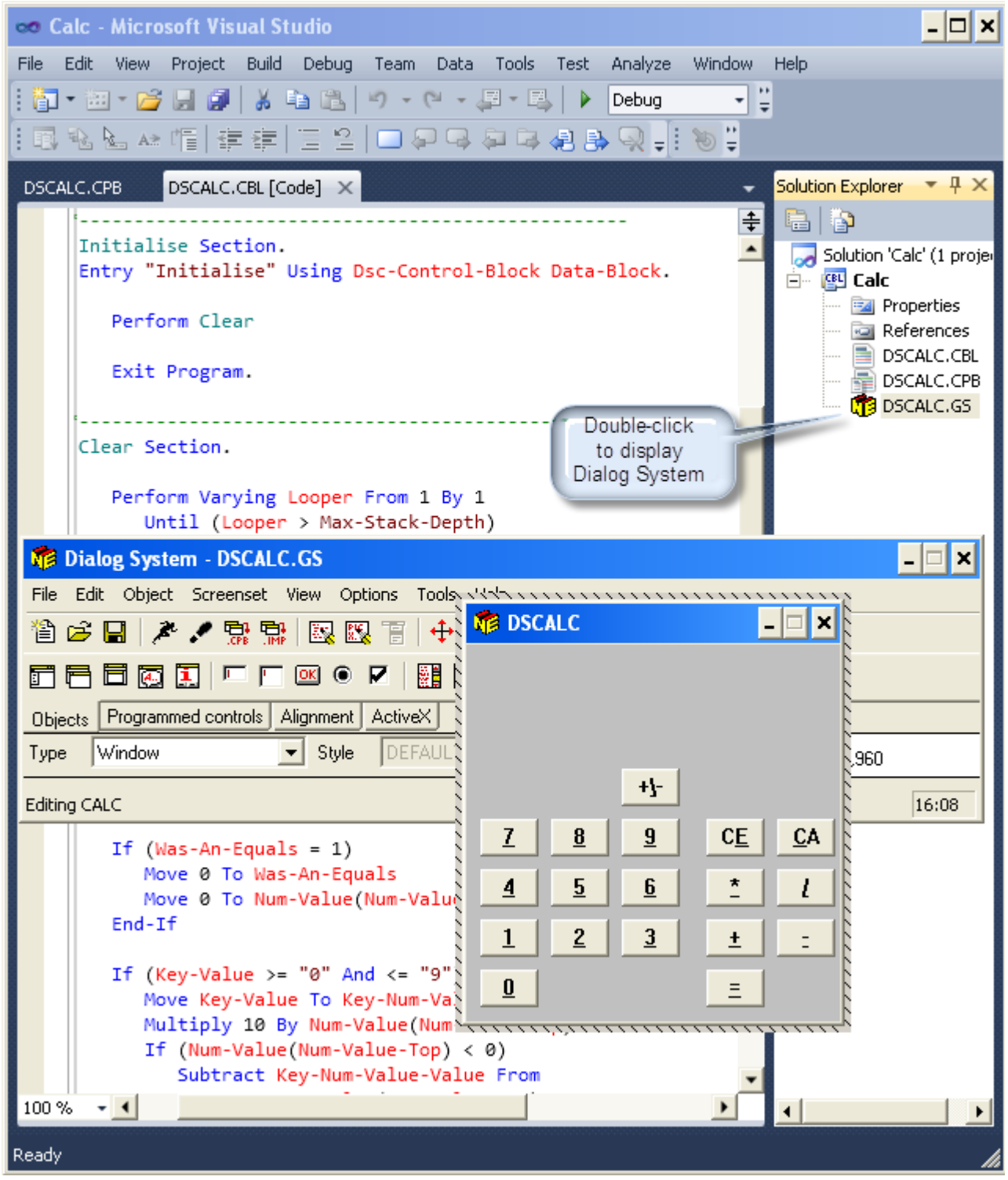
Migrating a Dialog System Application to Visual COBOL

You can use the Net Express Project Import wizard to convert a Net Express project into a Visual COBOL solution, or alternatively you can create a solution manually.

There is one Dialog System specific item that is added to the project and that is the screenset, the `.gs` file. The wizard sets the following properties on the screenset file and you need to do the same if you create the project manually:

- **Copy to Output Directory** is set to **Copy if newer**. This ensures that the screenset is added to the output directory whenever the screenset is changed.
- **Build Action** is set to **Content**. This ensures that the screenset is not compiled and consequently doesn't create spurious warnings or errors.

When the Dialog System application is in Visual COBOL, you can continue to maintain and run it with Visual COBOL, as shown here:



In Visual COBOL, you can:

- Open the screenset, by double-clicking the `.gs` file in Solution Explorer. This starts Dialog System with the screenset open, ready for you to edit.
- Edit a `.cbl` file by double-click it in Solution Explorer.
- Build and debug the application. Although debugging COBOL is supported, the screenset animator is not supported.
- Run the application.

Debuggable OO Class Libraries

If your application uses the native OO COBOL class libraries, and you want to step into those libraries when debugging, you need to build debuggable versions of them. This is different from Net Express, where the libraries were supplied and you could select them from the IDE. To build and use debuggable versions of the libraries:

1. Open the project for the required class library. The projects are located in the `cpylib` folder of your installation, which is by default `%ProgramFiles%\Micro Focus\Visual COBOL 2010`. The class library projects are:

   GUI class library - `guicl\apiguim.cblproj`
   OLE class library - `olecl\olecl.cblproj`
   Base class library - `basecl\baseclm.cblproj`
2. Ensure the Debug configuration is selected and then build the project. The resulting output is stored in the `debug` subfolder of the project folder.
3. In your application, set the COBPATH environment variable to point to the folder containing the debuggable version of the library. To do this, add the variable to the `application.config` file, as follows:

   a. Open your application in Visual COBOL.
   b. Right-click your main project and click **Add** > **New Item** > **Application Configuration File**.
   c. Double-click **Application.config** in Solution Explorer.
   d. In the **Name** field, specify COBPATH.
   e. In the **Value** field, specify the full path of the folder. For example, `%ProgramFiles%\Micro Focus\Visual COBOL 2010\cpylib\guicl\debug`.
   f. Click **Set**.

Note: The Help for the class libraries is available in the file `nxrclr.chm`, which is installed in the Help folder of your installation. The default location is `%ProgramFiles%\Micro Focus\Visual COBOL 2010\Help`.

Modernizing a Dialog System Application

Modernize Dialog System applications by migrating them to Visual COBOL When you are ready to modernize the application, you can choose from a range of techniques. You have the flexibility to modernize as much or as little as you want in your own time. For example, you can replace one Dialog System dialog with a Windows Form or you can wrap a .NET WPF control as an ActiveX control and use that on a Dialog System dialog.

There are distinct advantages to modernizing gradually and using the interoperability techniques to keep the old and new code working together. Microsoft's recommendation is:

"The COM interoperability features of the .NET Framework are very powerful and, in nearly all cases, allow you to continue to use your existing code without migrating it to managed code. As you develop new parts of your application or reuse components of your application from newer managed code applications, in most cases you can simply call your existing components through the COM interoperability functionality provided by .NET."

There are a range of modernization techniques and a multitude of ways to adopt them. The technique to use depends on the application. You first need to understand the application, what it does, how it's structured, how it performs, and its limitations and benefits. You can then identify an area where you can improve things.

Some modernization techniques that you can use are:

- A Windows Forms form replacing a Dialog System dialog, where the form can contain .NET controls. See the Customer + .NET WinForm sample `CustomerWinForm.sln`.

- A Windows Forms control wrapped as an ActiveX control and used on a Dialog System dialog. See the Customer + .NET GridView User Control sample `custgrid.sln`.
- A WPF user control hosted by a Windows Forms user control, which is then exposed as ActiveX ready for use by Dialog System. See the Customer + .NET WPF GridView User Control sample `CustGridWPF.sln`
- A .NET managed code application interacting with Dialog System as native COBOL `.dll` or as . See the Managed Customer sample `ManagedCustomer.sln`.

Your Visual COBOL solution is likely to have some native code projects and some managed code projects. The native code projects will contain the original code, largely unchanged, and also the managed code projects will contain the new modernized elements.

Note: It is not possible to debug both native and managed COBOL in the same session. You can debug them separately by setting the project you want to debug as the StartUp project. To do this, right-click your project, and click **Set as StartUp project**.

Samples of Modernizing Dialog System Applications

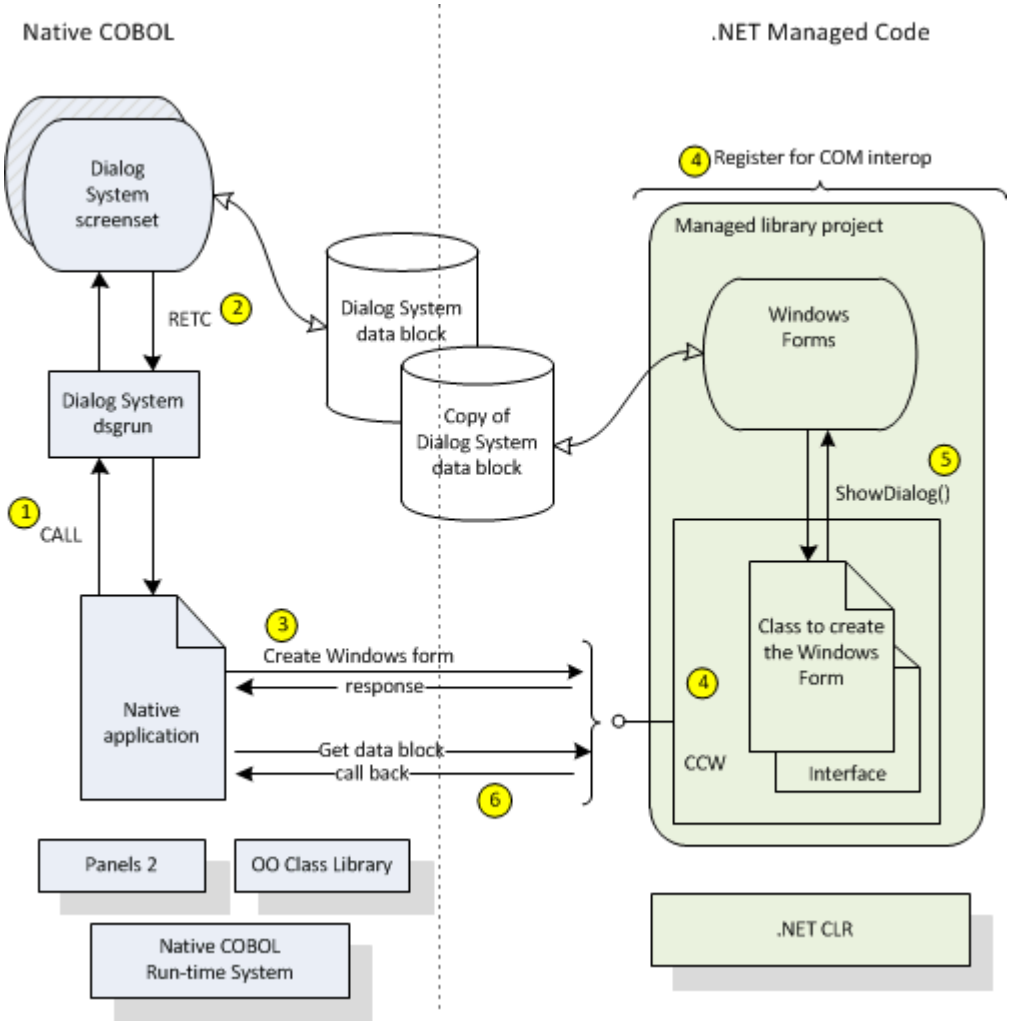
You can access the samples from **Start menu > All Programs > Micro Focus Visual COBOL 2010 > Visual COBOL Samples**. Then click **Dialog System** to list the available samples.

Some samples, Calc, Customer and SplitWindow, use the same code as they did in Net Express. They run the same as before, except the key difference is that they use the Visual COBOL version of the COBOL and Dialog System run-time systems.

Sample: Windows Forms Replacing Dialog System Dialogs

The Customer + .NET WinForm sample `CustomerWinForm.sln` takes the original Customer sample from Dialog System and replaces the Orders dialog with a more modern Windows Forms Orders form. The main Customer application remains largely unchanged, so that the application becomes a native code main program calling managed code as a COM object to handle the Windows Forms Orders form.

The following diagram shows the native code on the left, with Dialog System handling the screenset. The managed code is on the right, with the new Windows Forms form. The image also shows the native code interfacing with a COM Callable Wrapper (CCW) containing the managed code, and shows the data block being passed and returned.



The numbers in the above diagram highlight some key points in the process and are explained below:

1. The native COBOL application calls Dialog System in the traditional way.
2. Instead of calling the now redundant dialog, the dialog returns control to the application, using RETC.
3. The native COBOL invokes the managed code interface, IFormsFactory, using standard native OO COBOL syntax.

   A copy of the data block is created for the managed classes to use.

   An entry point is created in the native code, ready for calling back from the managed code.
4. The managed code library, OrderFormsLibrary is registered for COM Interop. This ensures that a COM Callable Wrapper (CCW) is created, which provides an interface for the native code to use.
5. The managed code class, FormsFactory, instantiates and displays the form, as a modal dialog, handling the events as needed.
6. The managed code calls back to the entry point previously created in the native COBOL, passing back a pointer to the updated data block.

Customer Project in Native COBOL

The solution has two projects: one native and one managed. The native project, Customer, contains the original sources with some changes to handle the Orders dialog differently and to interact with the managed code project.

The native project, Customer, contains:

- Properties - these define how the project is built. For example, it is built as a Windows application called Customer, stored in `.\bin\x86\Debug`.
- `Customer.cbl` - original source code, plus some additional code to interact with the managed COBOL project, which handles the new Windows Forms Orders form.
- `Customer.cpb` - original unchanged copybook generated from the Dialog System data block.
- `Customer.gs` - original screenset, which now handles the Orders button by returning control to the customer.cbl, rather than displaying the Orders dialog
- `Cust.ism` - the original unchanged data file.

To debug the native project, make sure the Customer project is set as the startup project. To do this, right-click the project and click **Set as StartUp Project**.

OrderFormsLibrary Project in Managed COBOL

The managed project, OrderFormsLibrary, contains the new Windows Forms Orders form. It also contains the supporting code to pass the customer data block between managed and native code. It is registered and exposed as a COM object.

The managed project, OrderFormsLibrary, contains:

- Properties - these define how the project is built. Notice that the project is registered as a COM object, thereby exposing it as a COM object and enabling the native project to access it. See **Project** > **Properties** > **COBOL** > **Advanced**.
- References - all the required .NET classes.
- `Calendar.cbl` - implements a calendar editor column for use in the data grid.
- Link to `Customer.cpb` - the same copybook is used and understood by the native and managed COBOL projects. The copybook itself remains in the native project and a link to it is now added in the managed project.
- `FormsFactory.cbl` - implements the CreateOrderForm() method and creates a delegate for the callback function, which is used to get hold of the customer data block.
- `IFormsFactory.cbl` - defines the CreateOrderForm() method, which provides the entry point into managed code from native code.
- `OrderForm.cbl [Design]` - this is the order form, drawn with the designer.
- `OrderForm.cbl [Code]` - this contains the code to handle the form and you can edit this and add your own code.

To debug the managed project, make sure the OrderFormsLibrary project is set as the startup project.

Customer.cbl in Native COBOL

`Customer.cbl`, contains the original code for the business logic and for interaction with the Dialog System screenset. `Customer.cbl` also contains some additional code to interact with the managed COBOL project and to display the Windows Forms Order form. `Customer.cbl` contains additional code to do the following:

1. Set the following Compiler directives, using $SET:

   ```
   $SET ans85 mfoo ooctrl(+P) case
   ```

   - ANS85 remains set as before
   - MFOO enables support for the Micro Focus native OO syntax
   - OOCTRL(+P) enables the run-time system to map COBOL data types to COM data types
   - CASE prevents external symbols (such as Program-ID and names of called programs) being converted to upper case

2. Map the COM class, OrderFormsLibrary.FormsFactory, to an OO COBOL class name, OrderFormFact , in the CLASS-CONTROL paragraph, where the class is in the COM domain, OLE, of the Micro Focus native OO class library:

```
class-control.
OrderFormFact is class"$OLE$OrderFormsLibrary.FormsFactory".
```

3. Evaluate the flag to determine if the Orders button has been pressed and respond accordingly:

```
WHEN customer-orders-flg-true
PERFORM Show-Orders-Form
```

4. Create an instance of the COM object, the new .NET order forms library, which implements a Windows Forms version of the Orders dialog that originally existed in `customer.gs`:

```
invoke OrderFormFact "New" Returning formsLibrary
```

5. Set a procedure pointer that the managed code can use to call the native code:

```
set  pptr  to  entry  "Customer_Callback"
```

6. Get a pointer to the data block, to be used in the callback:

```
Customer-Callback SECTION.
    entry "Customer_Callback" stdcall.
    exit program returning address of CUSTOMER-DATA-BLOCK.
```

7. Invoke and display the new Orders form:

```
invoke formsLibrary "CreateOrderForm" using
    by value pptr-val
```

Customer.GS, the Dialog System Screenset

The screenset, `customer.gs` no longer displays the original Orders dialog, but instead it passes control back to the native COBOL program. The instructions for the Orders button in the Main-Window dialog have changed as follows:

```
SET-FLAG ORDERS-FLG(1)
RETC
* REFRESH-OBJECT DIALOG-BOX
* SET-FOCUS DIALOG-BOX
```

The instructions now set the ORDERS-FLG to indicate that the Orders button has been pressed, so that the native COBOL program can evaluate the flag and respond appropriately. Control is returned to the native COBOL program. The redundant lines are commented out, so that the old Orders dialog is no longer displayed.

IFormsFactory.cbl in Managed COBOL

`IFormsFactory.cbl` defines an interface to create a Windows Forms Order form. `IFormsFactory.cbl` has code to do the following:

1. Declare the interface using the InteropServices classes from the .NET Framework. In addition, establish the COM object support for late binding, by specifying ComInterfaceType:: InterfaceIsDual. Alternatively, you could specify ComInterfaceType::InterfaceIsIDispatch (but not ComInterfaceType::InterfaceIsIUnknown):

```
interface-id OrderFormsLibrary.IFormsFactory
    attribute System.Runtime.InteropServices.InterfaceType
      (type
      System.Runtime.InteropServices.ComInterfaceType::InterfaceIsDual)
```

   Note, the 'Register For COM Interop' property exposes everything as a dual interface, by default.

2. Define the CreateOrderForm() method. This is the definition of the entry point into managed code from native code:

```
method-id CreateOrderForm.
procedure division using by value callback as binary-long.
end method.
```

FormsFactory.cbl in Managed COBOL

`FormsFactory.cbl` creates an instance of the Orders form, when the native program asks for it. `FormsFactory.cbl` has code to do the following:

1. Declare the class, which implements the IFormsFactory interface:
```
class-id OrderFormsLibrary.FormsFactory implements
    type OrderFormsLibrary.IFormsFactory.
```
2. Expose the class to COM, by making it visible:
```
attribute ComVisible(true)
```
3. Specify our interface IFormsFactory as the default interface to expose to COM:
```
attribute ComDefaultInterface
    (type of OrderFormsLibrary.IFormsFactory)
```
4. Set the ComVisible attribute off by default at the assembly level. This means the types we expose must be explicitly marked as ComVisible(true), which gives us a more well-defined type library for clients to use:
```
assembly-attributes.
attribute ComVisible(false).
```
5. Get a delegate for the callback function and use it to get hold of the customer data block. The GetDelegateForFunctionPointer() converts an unmanaged function pointer to a delegate:
```
set  pptr to new System.IntPtr(callback)
set  custCallback to type
    System.Runtime.InteropServices.Marshal::GetDelegateForFunctionPointer
    (pptr, type of CustomerCallback)
    as type  CustomerCallback
```
6. Declare the delegate for the callback function:
```
delegate-id CustomerCallback.
procedure division returning pCustomerDataBlock as type IntPtr.
end delegate.
```
7. Create the Orders form and show it as a modal dialog box:
```
set form to new OrderFormsLibrary.OrderForm(custCallback)
invoke form::ShowDialog()
```

OrderForm.cbl [Design] in Managed COBOL

The order form is drawn with the form designer. See the Windows Forms tutorials.

OrderForm.cbl [Code] in Managed COBOL

`OrderForm.cbl [Code]` is a partial class, also called OrderFormsLibrary.OrderForm and this contains the code to handle the Windows Forms Orders form.

`OrderForm.cbl [Code]` has the following methods:

New()

Call back to native code to retrieve customer data block. We store the pointer in *_pCustomerDataBlock.*:
```
set  _pCustomerDataBlock  to  callback::Invoke()
```

OrderForm_Load()

Unpack the data block to a copy in local storage, which can then be accessed in the same way as in native code. The Marshal::Copy() method copies the pointer *_pCustomerDataBlock* into *pData*:
```
  invoke type Marshal::Copy
  (_pCustomerDataBlock, pData, 0, length  of  CUSTOMER-DATA-
BLOCK)
  set CUSTOMER-DATA-BLOCK to pData
```

PopulateOrder() Initialize the Orders form using the orders information in the CUSTOMER-DATA-BLOCK, with a PERFORM block and statements such as:

```
perform varying row thru OrdersGridView::Rows
move CUSTOMER-ORD-NO(array-ind) to
    row::Cells::get_Item("OrderNo")::Value
move CUSTOMER-ORD-DATE(array-ind) to dt1
...
```

OrderForm_FormClosing() Handle the form closing event and move the locally held Dialog System data block back to the caller. The Marshal::Copy() method copies the pointer *pData* into *_pCustomerDataBlock*:

```
  set pData to CUSTOMER-DATA-BLOCK
  invoke type Marshal::Copy
  (pData, 0, _pCustomerDataBlock, LENGTH OF CUSTOMER-
DATA-BLOCK)
```

OK_Click() Handle the OK button click event.

Delete_Click() Handle the Delete button click event.

OrderForm.Designer.cbl in Managed COBOL

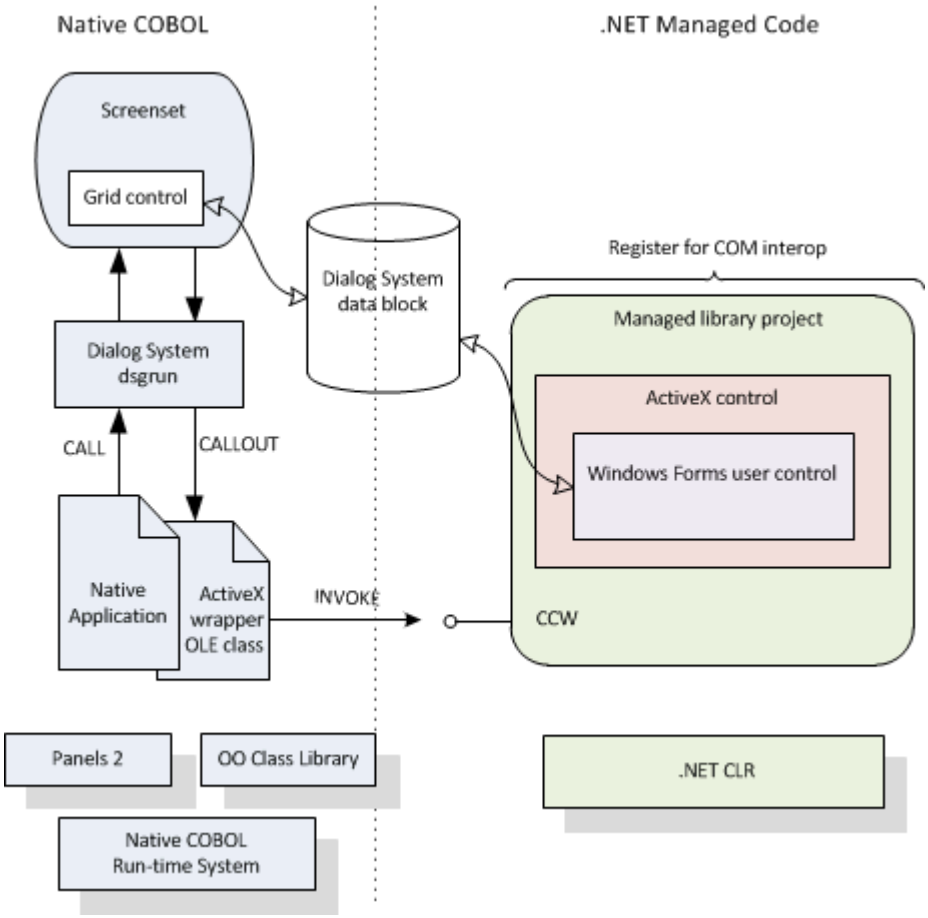
This is the code behind the Windows Forms that defines the form and is generated from the design. See the Windows Forms tutorials.

Sample: Windows Forms Control as ActiveX for a Dialog System Application

The Customer + .NET GridView User Control sample shows a Windows Forms user control wrapped as an ActiveX control. Since Dialog System can host ActiveX controls, the new Windows Forms user control is built to be exposed as an ActiveX control. The sample shows how a user control can be hosted, how it can have methods invoked and how it can fire events, which can be consumed in the Dialog System application.

The sample is based on the original CustGrid sample from Dialog System. It has two projects: one is the original CustGrid sample and the second contains the code for a simple Windows Forms DataGridView user control written in COBOL. The DataGridView control is similar to the one used in the Customer samples (Customer +.NET WPF User Control and Customer + WinForm). Here, in this sample, the DataGridView is a .NET user control instead of a WPF user control or a control contained in Windows Forms.

The following diagram shows the native code on the left, with Dialog System handling the screenset. The managed code is on the right, with the Windows Forms user control wrapped as an ActiveX. The image also shows the native code interfacing with a COM Callable Wrapper (CCW) containing the managed code.



Note:

In this sample, the control is not a complete implementation of a grid control and the application is not a complete implementation. The sample is designed to illustrate the principles only.

CustGrid Project

The CustGrid project contains the original CustGrid project from Dialog System and contains:

- `CustGrid.gs` is the screenset, which now contains the DataGridView control instead of the original ActiveX grid control. In the same way as before, Dialog System communicates with the native code `gridctrl.cbl`, using callouts and callbacks.
- `GridCtrl.cbl` handles the events generated by the DataGridView control. It invokes the COM code in the same way that it invoked the ActiveX code before. For example, the Delete-A-Row Section handles the delete-row event and invokes the COM code, as follows:

```
INVOKE  RowSet  "GetSelect"  USING  theRow  RETURNING  Numeric-Value
INVOKE  RowSet  "Remove"  USING  Numeric-Value
```

  Where `RowSet` is defined in the COM interface for the Windows Forms user control. See the *ControlInterface.cbl* section, later in this topic.

GridViewUserControl Project

The GridViewUserControl project is registered for COM interop. In addition, the COMRegister() method puts the appropriate entries into the registry so that ActiveX containers see this as a control. This is covered in more detail later.

The managed COBOL grid user control is exposed to COM by declaring a class interface, an events interface, and the class itself, as follows:

- Class interface, MicroFocus.VisualCOBOL.ISampleGridView in `ControlInterface.cbl`
- Class SampleGridView in `GridViewControl.cbl`
- Events interface, IGridViewEvents in `GridViewEvents.cbl`

The project contains other classes to complete the implementation:

- UserDataGridViewRow class, which contains the methods to handle the row
- RowSet class, which contains the methods to handle a row of the grid
- CalendarColumn class, which implements a calendar editor column for use in the data grid

ControlInterface.cbl

`ControlInterface.cbl` defines the interface that exposes the methods and properties of the control to COM:

```
interface-id  MicroFocus.VisualCOBOL.ISampleGridView
    attribute  Guid("5A0AED5B-7D47-48C9-9F95-DEE8BC2D4807").
method-id  get  property  RowSet.
...
```

GridViewControl.cbl [code]

`GridViewControl.cbl` contains the class that defines the behavior of the DataGridView control. Its main task is to replicate the behavior of the old ActiveX control in the new DataGridView control.

`GridViewControl.cbl` includes the following code that is of interest:

Class-id SampleGridView

Define the SampleGridView class, which inherits from the Windows Forms UserControl class and implements the ISampleGridView interface declared in `ControlInterface.cbl`. It also implements the ISerializable interface, which enables the class to control its serialization behavior:

```
Class-id SampleGridView
inherits type System.Windows.Forms.UserControl
implements type System.Runtime.Serialization.ISerializable
type MicroFocus.VisualCOBOL.ISampleGridView
```

attribute Serializable()

Enable the class to be serialized. This attribute is required even though the class also implements the ISerializable interface to control the serialization process:

```
attribute Serializable()
```

attribute ComVisible(true)

Make the managed class visible to COM. In other words, exposes the class to COM:

```
attribute ComVisible(true)
```

attribute ProgId

Specify the ProgID for the COM object. This is a human-readable version of the class identifier (CLSID) used to identify COM/ActiveX objects:

```
attribute ProgId
    ("MicroFocus.VisualCOBOL.SampleGridView")
```

attribute ClassInterface

Generate a class interface that supports early and late binding:

```
attribute ClassInterface
    (type ClassInterfaceType::AutoDual)
```

attribute ComDefaultInterface

Specify the interface ISampleGridView as the default interface to expose to COM:

```
attribute ComDefaultInterface
    (type of MicroFocus.VisualCOBOL.ISampleGridView)
```

attribute ComSourceInterfaces

Identify the IGridViewEvents as the interface to be exposed as COM event sources for the library WindowsFormsControlLibrary1.dll (which is built by the GridViewUserControl project):

```
attribute ComSourceInterfaces
    (type of GridViewUserControl.IGridViewEvents)
```

attribute Guid

Specify an explicit GUID. This is useful during the debugging phase, as it avoids a new GUID being generated each time you build and register:

```
attribute Guid("B33E4887-6BC0-4382-883B-EB0015F55D9B")
```

COMRegister()

Adds an entry to the Registry for the specified object. It sets the appropriate entries to register the object as an ActiveX, so that ActiveX containers see this as a control. This enables Dialog System to list the ActiveX as one of the controls available for import.

The project property **Register for COM Interop** registers the object as COM, but does not register it as ActiveX.

COMUnregister()

Removes the entry for the specified object from the Registry.

New()

Create an instance of the SampleGridView.

GetObjectData()

This is used as part of serialization of the control. However, this sample does not have anything to persist so this method is unused.

Event Handling methods

- FireOnChanged() - handles the OnChanged event
- FireRowSelected() - handles the OnRowSelected event
- FireRowDeleted() - handles the OnRowDeleted event
- OnRowEnter() - invokes the FireRowSelected() method

Get property RowSet

Implements the interface ISampleGridView. Creates an instance of the row.

Get property OrderGrid

This enables the Windows Forms user control to access the DataGridView user control.

```
01 OrdersGridView
       type System.Windows.Forms.DataGridView.
method-id. get property OrderGrid.
procedure division
        returning thegrid as type DataGridView.
  set thegrid to self::OrdersGridView
  goback.
end method.
```

OrdersGridView_CellEndEdit()

Handles the cell editing event and invokes FireOnChanged().

```
method-id OrdersGridView_CellEndEdit
    final private.
procedure division using by value
```

```
      sender as object
      e as type
System.Windows.Forms.DataGridViewCellEventArgs.
      invoke self::FireOnChanged
         (e::RowIndex, e::ColumnIndex)
```

Delegates Defines the delegates for our events.

```
01 OnRowSelected type RowSelectedEventHandler event public.
01 OnRowDeleted type RowDeletedEventHandler event public.
01 OnChanged type RowChangedEventHandler event public.
   ...
delegate-id RowSelectedEventHandler.
delegate-id RowDeletedEventHandler.
delegate-id RowChangedEventHandler.
```

GridViewEvents.cbl

`GridViewEvents.cbl` defines an interface for the events and assigns each event a COM dispatch identifier (DispId), as follows:

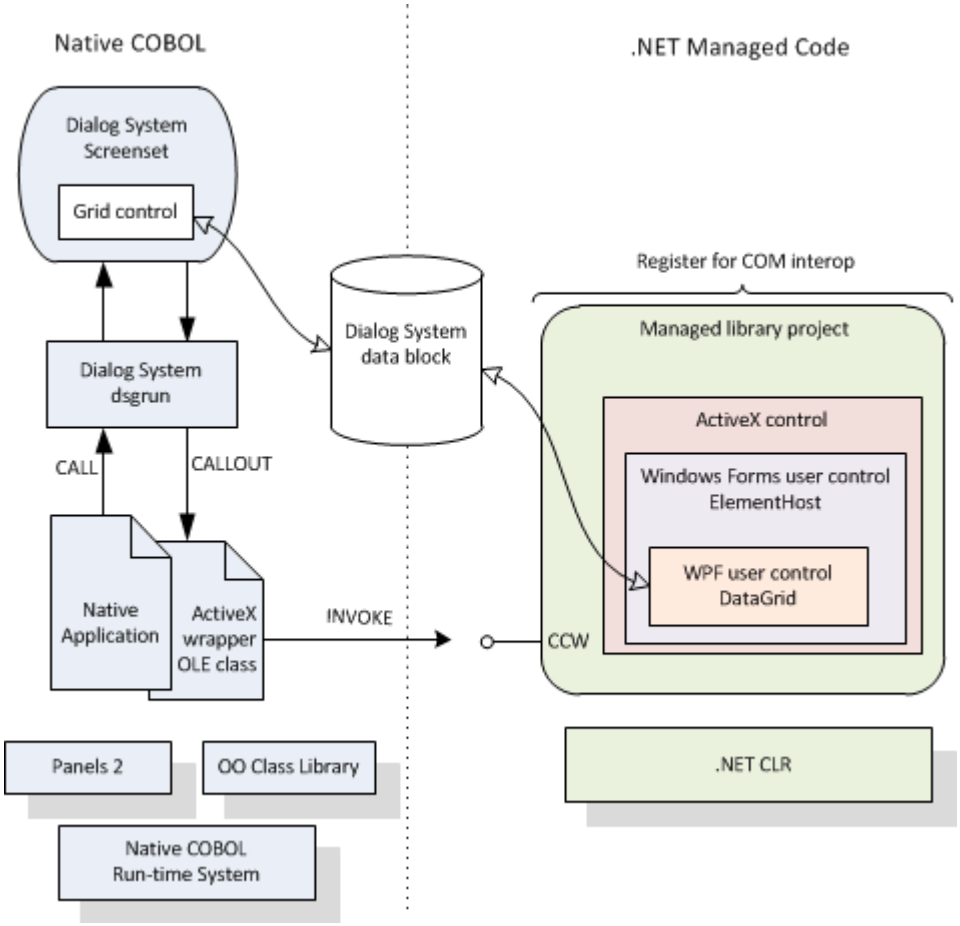
```
method-id OnRowSelected attribute DispId(29).
procedure division using by value
    row as binary-long,
    coln as binary-long.
    ...
method-id OnRowDeleted attribute DispId(18).
method-id OnChanged attribute DispId(6).
```

The events correspond to the delegates defined in the SampleGridView class in `GridViewControl.cbl`.

Note, this interface is not implemented in our solution.

Sample: WPF User Control in a Dialog System Application

The Customer + .NET WPF GridView User Control sample `CustGridWPF.sln` shows a WPF user control used by a Dialog System application. The sample shows how a WPF user control can be hosted, how it can have methods invoked and how it can fire events, which can be consumed in the Dialog System application, as illustrated in the following diagram:



There are two key points about WPF shown in this sample:

- The WPF user control is hosted by a Windows Forms user control, which is then exposed as ActiveX ready for use by Dialog System. This means that the sample has two user controls: one WPF and one Windows Forms. These are both needed because you cannot expose WPF as ActiveX directly.
- The WPF user control uses data binding, that is, it associates the data with the user interface. This is similar to Dialog System associating master fields to controls such as entry fields and list boxes. The sample shows this with the CustomerOrder class and the ObservableCollection of customer orders.

The sample is based on the original CustGrid sample from Dialog System. It has two projects: one is the original CustGrid sample and the second contains the code for a simple WPF DataGrid control written in COBOL. The DataGrid control is similar to the control used in the Customer samples (Customer +.NET User Control and Customer + .NET WinForm). Here, in this sample, the DataGrid is a WPF user control.

Note:

In this sample, the control is not a complete implementation of a grid control and the application is not a complete implementation. The sample is designed to illustrate the principles only.

WPFCustGrid Project

The WPFCustGrid project contains the original CustGrid project from Dialog System.

`CustGrid.gs` is the screenset, which now contains the DataGrid control instead of the original ActiveX grid control. In the same way as before, Dialog System communicates with the native code `gridctrl.cbl`, using callouts and callbacks.

`GridCtrl.cbl` handles the events generated by the DataGrid control. It invokes the COM code in the same way as it invoked the ActiveX code before. For example, the Delete-A-Row Section handles the delete-row event and invokes the COM code, as follows:

```
INVOKE RowSet "GetSelect" USING theRow RETURNING Numeric-Value
INVOKE RowSet "Remove" USING Numeric-Value
```

Where `RowSet` is defined in the COM interface for the WPF user control in `ControlInterface.cbl`, which is described later.

WPFGridViewUserControl Project

The WPFGridViewUserControl project contains two user controls: one WPF and one Windows Forms. Both controls are needed because the WPF user control is hosted as a Windows Forms user control (which is then wrapped as an ActiveX control). The content is actually a WPF control, but it is handled as a Windows Forms user control, just as in the sample Customer + .NET User Control.

To see the two controls, double-click:

- `WPFUserControl.xaml` - The properties show that the control is a WPF control, of type DataGrid. The XAML shows the definition of the DataGrid. The Document Outline shows the control as a DataGrid. Document Outline is available from **View** > **Other Windows**.
- `GridViewControl.cbl [Design]` - The properties show that the control is of type Windows.Forms.ElementHost. The Document Outline shows the control as GridHost of type ElementHost. The Toolbox lists this type of control under WPF Interoperability, as ElementHost.

The project contains the following to complete the implementation:

- CustomerOrder class, which maps to a CustomerOrder group item in the data block, in `CustomerOrder.cbl`
- Class interface, MicroFocus.VisualCOBOL.IWPFSampleGridView in `ControlInterface.cbl`
- Class WPFSampleGridView in `GridViewControl.cbl`
- Events interface, IWPFGridViewEvents in `GridViewEvents.cbl`
- RowSet class, which contains the methods to handle a row of the grid, in `RowSet.cbl`.

The project is registered for COM interop. In addition the COMRegister() method puts the appropriate entries into the registry so that ActiveX containers see this as a control. This method is covered in more detail later.

ControlInterface.cbl

`ControlInterface.cbl` defines the interface that exposes the methods and properties of the control to COM:

```
interface-id  MicroFocus.VisualCOBOL.IWPFSampleGridView
    attribute  Guid("77BECD4D-5504-442D-9DDA-A78C30ADF6A9")
    attribute  InterfaceType(type  ComInterfaceType::InterfaceIsDual)
    attribute  ComVisible(true).
method-id  get  property  RowSet.
...
```

GridViewControl.cbl [code]

`GridViewControl.cbl` contains the class that defines the behavior of the DataGrid control. Its main task is to replicate the behavior of the old ActiveX control in the new DataGrid control.

`GridViewControl.cbl` includes the following code of interest:

WPFSampleGridView Defines the WPFSampleGridView class, which inherits from the Windows Forms UserControl class and implements the IWPFSampleGridView interface declared

in `ControlInterface.cbl`. It also implements the ISerializable interface, which enables the class to control its serialization behavior.

```
Class-id WPFSampleGridView
    inherits type System.Windows.Forms.UserControl
    implements
      type System.Runtime.Serialization.ISerializable
      type MicroFocus.VisualCOBOL.IWPFSampleGridView
```

attribute Serializable()

Enable the class to be serialized. This attribute is required even if though the class also implements the ISerializable interface to control the serialization process.

```
attribute Serializable()
```

attribute ComVisible(true)

Make the managed class visible to COM. Exposes the class to COM:

```
attribute ComVisible(true)
```

attribute ProgId

Specify the ProgID for the COM object. This is a human-readable version of the class identifier (CLSID) used to identify COM/ActiveX objects:

```
attribute ProgId("WPFSampleGridView")
```

attribute ClassInterface

Generate a class interface that supports early and late binding:

```
attribute ClassInterface(type ClassInterfaceType::None)
```

attribute ComDefaultInterface

Specify the interface ISampleGridView as the default interface to expose to COM:

```
attribute ComDefaultInterface
(type of MicroFocus.VisualCOBOL.IWPFSampleGridView)
```

attribute ComSourceInterfaces

Identify the IGridViewEvents as the interface to be exposed as COM event sources for the library WindowsFormsControlLibrary1.dll (which is built by the GridViewUserControl project):

```
attribute ComSourceInterfaces (type of
      MicroFocus.VisualCOBOL.IWPFGridViewEvents)
```

attribute Guid

Specify an explicit GUID. This is useful during the debugging phase, as it avoids a new GUID being generated each time you build and register:

```
attribute Guid("E0A41600-4531-4E55-9B24-6D2F1CF8B106")
```

ObservableCollection of CustomerOrder Objects

Create an observable collection of CustomerOrder objects, which enables the backing data to be updated when the user interface changes (such as when the user edits the data):

```
*> Create an observable collection of CustomerOrder
*> objects so that the data grid receives notifications
*> such as add, delete in order to refresh the list
 01 _customerOrders
     type ObservableCollection[type CustomerOrder]
     value new ObservableCollection[type CustomerOrder]
     public
     property as "CustomerOrders".
```

SampleGridView class

The SampleGridView class provides the following methods and delegate definitions:

COMRegister()

This method adds an entry to the Registry for the specified object. It sets the appropriate entries to register the object as an ActiveX, so that ActiveX containers see this as a control. This enables Dialog System to list the ActiveX as one of the controls available for import.

The project property **Register for COM Interop** registers the object as COM, but does not register it as ActiveX.

COMUnregister()

The reverse of the COMRegister() method.

New()

These methods create an instance of the WPFSampleGridView. The default New() method initializes the instance and adds the specified data by invoking the appropriate methods: OnSelectionChanged(), OnCellEditEnding() and OnSourceUpdated().

OnSourceUpdated()

Handles the OnSourceUpdated event, determines the row updated and invokes the FireOnChanged() method.

OnCellEditEnding()

Handles the OnCellEditEnding event and invokes the FireOnChanged() method.

OnSelectionChanged()

Handles the OnSelectionChanged event, determines the row and invokes the FireRowSelected() method.

GetObjectData()

This is used as part of serialization of the control. However, this sample does not have anything to persist so this method is unused.

Event Handling methods

- FireOnChanged() - handles the OnChanged event
- FireRowSelected() - handles the OnRowSelected event
- FireRowDeleted() - handles the OnRowDeleted event
- OnRowEnter() - invokes the FireRowSelected() method

Get property RowSet

Implements the interface IWPFSampleGridView. Creates an instance of the row.

Get property OrderGrid

This enables the Windows Forms user control to access the WPF DataGrid user control.

```
method-id. get property OrderGrid.
procedure division returning thegrid as
      type System.Windows.Controls.DataGrid.
  set thegrid to self::wpfUserControl1::OrdersGrid
```

WPFSampleGridView_Load()

Handles the cell editing event.

```
method-id OrdersGridView_CellEndEdit final private.
procedure division using by value
    sender as object
    e as type System.EventArgs.
```

Delegate definitions

Defines the delegates for our events:

```
01 OnRowSelected type RowSelectedEventHandler event public.
01 OnRowDeleted type RowDeletedEventHandler event public.
01 OnChanged type CellEditEndingHandler event public.
   ...
delegate-id RowSelectedEventHandler.
delegate-id RowDeletedEventHandler.
delegate-id CellEditEndingHandler.
```

GridViewEvents.cbl

`IWPFGridViewEvents.cbl` defines an interface for the events. The interface has the ComInterfaceType.InterfaceIsIDispatch attribute, which restricts callers to late binding.

```
interface-id MicroFocus.VisualCOBOL.IWPFGridViewEvents
    attribute Guid("3DC45DA1-A580-4280-A2DB-7B6A266032AE")
    attribute InterfaceType
       (type ComInterfaceType::InterfaceIsIDispatch)
    attribute ComVisible(true).
```

The interface defines a method for each event, and assigns each event a COM dispatch identifier (DispId), as follows:

```
method-id OnRowSelected attribute DispId(29).
...
method-id OnRowDeleted attribute DispId(18).
...
method-id OnChanged attribute DispId(6).
```

The events correspond to the delegates defined in the WPFSampleGridView class in `GridViewControl.cbl`.

Note, this interface is not implemented in our solution.

CustomerOrder.cbl

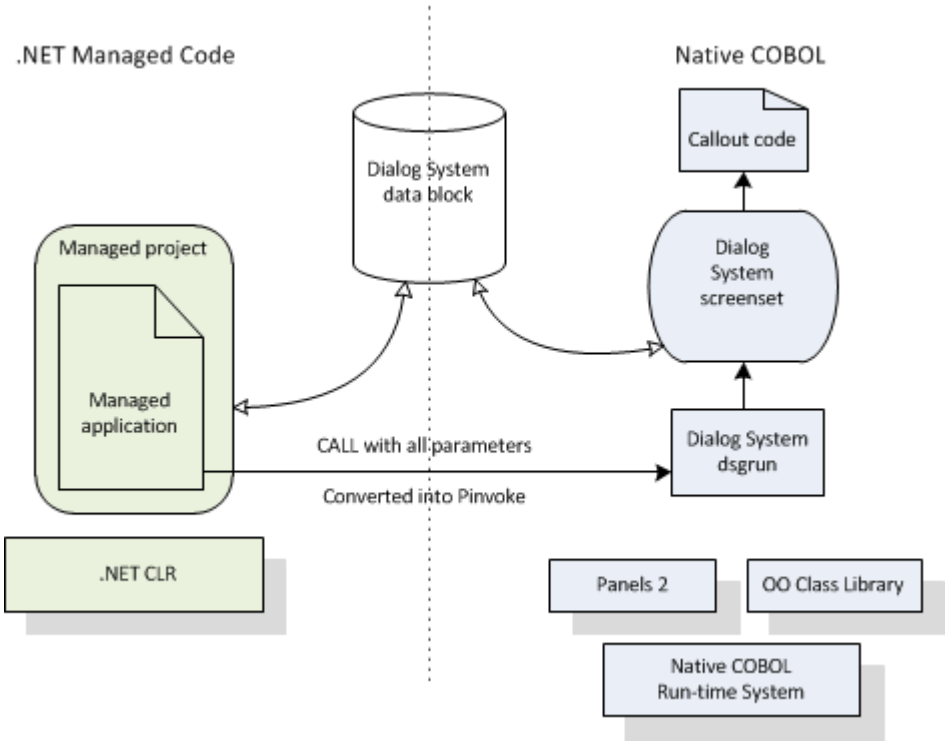
`CustomerOrder.cbl` defines the CustomerOrder class, which contains methods to create and initialize an instance of a customer order, and to return an array of customer orders.

The CustomerOrder class demonstrates a key concept in WPF, data binding. It demonstrates how the data can be associated with the user interface.

The CustomerOrder class maps to a Customer Order group item in the data block, so part of the sample gets data from the data block and creates a list of Customer Orders. This object is then associated with the WPF DataGrid, which renders the control to match the data in the list. The object is defined in `GridViewControl.cbl`, as the type ObservableCollection.

Sample: Managed Dialog System Application

The Managed Customer sample `ManagedCustomer.sln` is broadly the original Customer sample that was supplied with Dialog System. The key difference with this sample is that it is compiled to managed code and run under .NET. The existing Dialog System run-time calls remain but are automatically converted to Platform Invoke calls into the native code. In other words, the user interface and screensets are all still running as native code. The following diagram illustrates this.



This sample is using the Platform Invoke technique, which enables managed code to call unmanaged functions implemented in dynamic link libraries (.dlls). Platform invoke is similar to an API call.

The sample has just one project, a managed code project, comprising:

- Properties - Notice that the project is compiled to managed code. See **Project** > **ManagedCustomer Properties** > **Application** tab, which shows the application is managed because it targets the .NET Framework (4, in our case).
- References - Notice that one of the references is to the Dialog System run-time system, `dsgrun.dll`, which is compiled as native COBOL. This enables the managed COBOL to call into dsgrun, using CALL statements. These CALL statements generate a Platform Invoke call if the CALL matches an entry in the native `.dll`.
- `Customer.cbl` - original source code, plus some additional code to call the native Dialog System run-time system correctly.
- `Customer.cpb` - original copybook generated from the Dialog System data block. This is unchanged.
- `Customer.gs` - original screenset, which is unchanged.
- `Cust.ism` - the original data file, which is unchanged.

Customer.cbl

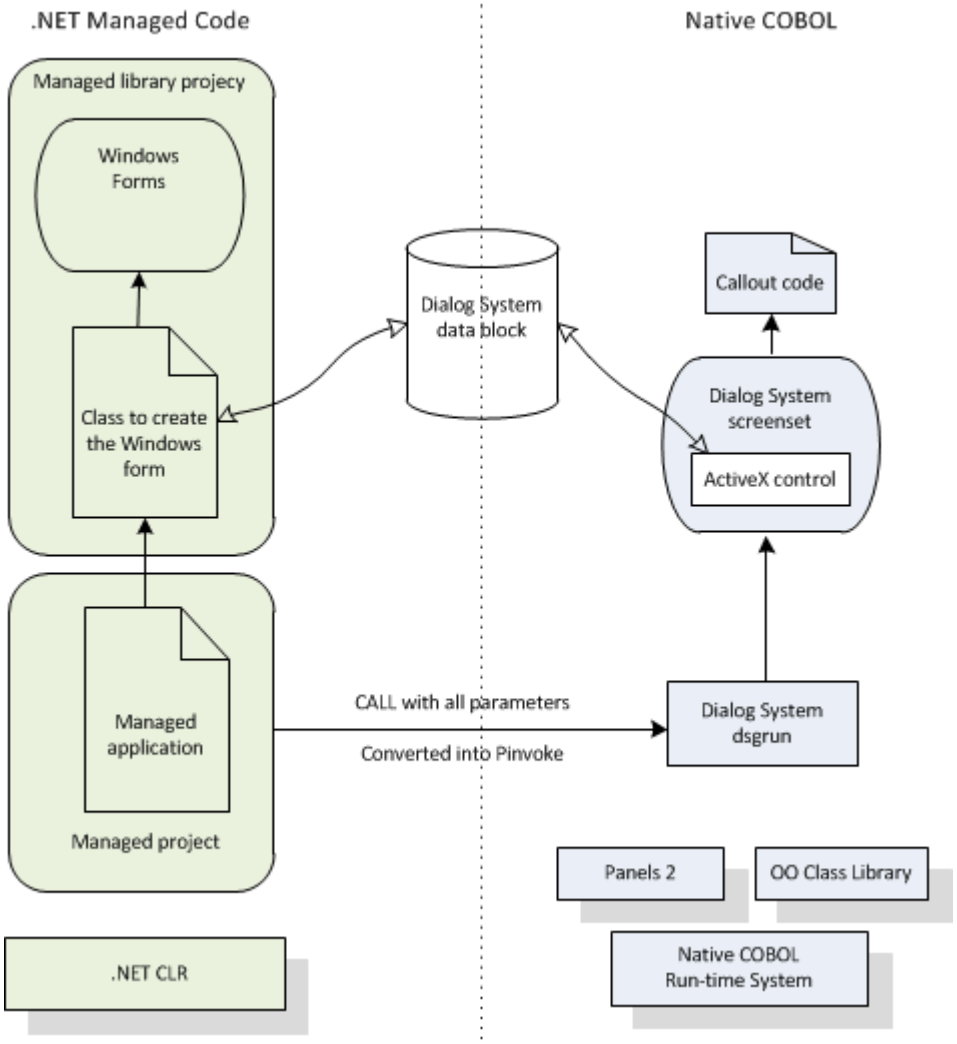
`Customer.cbl` contains the original code for the business logic and for interaction with the Dialog System screenset.

The only change to `Customer.cbl` is to ensure that the managed code calls the native Dialog System run-time system correctly. The code must supply all the parameters that Dialog System might need (or at least signify that we are not passing all the parameters). The original Customer sample uses only two parameters, so the code here needs to add a third parameter as 'omitted'.

```
Call-Dialog-System SECTION.
    CALL dialog-system USING ds-control-block,
                             customer-data-block
                             omitted
```

Sample: Managed Application and Windows Forms

The Managed Customer + .NET WinForm sample `ManagedCustomerWinForm.sln` puts together the Managed Customer sample and the .NET Order Forms dialog that is used in the Customer + WinForm sample. These two together form a sample where all the COBOL is compiled to .NET and the Dialog System user interface remains in native code. The following diagram illustrates this.



The advantage of moving all the code to .NET means that the native/managed interoperation through COM is no longer relevant, which simplifies the sample greatly. Compare this sample with the Customer + Win Form sample.

To create this sample:

- The ManagedCustomer project has a reference to the Dialog System run-time system, `dsgrun`, which you can see if you expand the **References** folder in Solution Explorer.
- The call to Dialog System uses the correct case of the main entry point and supplies all parameters that Dialog System might need (or at least signifies that we are not passing all the parameters). The original Customer sample uses only two parameters, so the code here adds a third parameter as 'omitted'.

```
Call-Dialog-System SECTION.
    CALL dialog-system USING ds-control-block,
                             customer-data-block
                             omitted
```

- The new Order Form dialog written in .NET COBOL is invoked from the main customer using .NET COBOL syntax because this is now compiled to managed code, as follows:

  ```
  invoke type OrderFormsLibrary.FormsFactory::CreateOrderForm
      (by reference CUSTOMER-DATA-BLOCK)
  ```

  This differs from the Customer + Win Form sample which uses native/managed interop through COM to pass the data between the main customer program and the new Order form.

  From the managed side, the Order form no longer has to use .NET marshalling to get the data and can remove the COM interfaces previously used to communicate with native code.
- The project no longer needs to set the project property **Register For COM Interop**.

Index