# How to customise the TWebBrowser user interface (part 3 of 6)

*Developing a reusable base class*

## Developing a reusable base class

Recall that this class, *TNulWBContainer*, will have all the features of a suitable container that hosts *TWebBrowser*. It will:

- Implement *IOleClientSite*.
- Implement *IDocHostUIHandler* in a neutral way.
- Implement *IUnknown* without reference counting.
- Register the container with the web browser control.
- Expose the hosted web browser control as a property.

## Getting Started

We will begin with the basics – getting hold of the hosted browser control and registering the container as its host. Declare a new class as per *Listing 2*:

```
type
  TNulWBContainer = class(TObject,
    IUnknown, IOleClientSite, IDocHostUIHandler
  )
  private
    fHostedBrowser: TWebBrowser;
    // Registration method
    procedure SetBrowserOleClientSite(const Site: IOleClientSite);
  public
    constructor Create(const HostedBrowser: TWebBrowser);
    destructor Destroy; override;
    property HostedBrowser: TWebBrowser read fHostedBrowser;
  end;
```

*Listing 2*

This is quite straightforward. We define the new type and list the interfaces we are to support. The private *SetBrowserOleClientSite* method is used to register / un-register the container with the hosted browser control. We then declare a constructor that takes a reference to the browser control and make it available via a property. A destructor is also declared.

*Listing 3* shows how the constructor is implemented.

```
constructor TNulWBContainer.Create(const HostedBrowser: TWebBrowser);
begin
  Assert(Assigned(HostedBrowser));
  inherited Create;
  fHostedBrowser := HostedBrowser;
  SetBrowserOleClientSite(Self);
end;
```

*Listing 3*

Here we simply check that the browser control passed to the constructor is not nil and record a reference to it in a field. Our container object then registers itself as the web browser's host by calling *SetBrowserOleClientSite* and passing a reference to the container's *IOleClientSite* interface.

The destructor simply un-registers the container from the browser by passing *nil* to *SetBrowserOleClientSite*, as *Listing 4* shows.

```
destructor TNulWBContainer.Destroy;
begin
  SetBrowserOleClientSite(nil);
  inherited;
end;
```

*Listing 4*

Having discussed the purpose of *SetBrowserOleClientSite* we can now look at its implementation. See *Listing 5*:

```
procedure TNulWBContainer.SetBrowserOleClientSite(
  const Site: IOleClientSite);
var
  OleObj: IOleObject;
begin
  Assert((Site = Self as IOleClientSite) or (Site = nil));
  if not Supports(
    fHostedBrowser.DefaultInterface, IOleObject, OleObj
  ) then
    raise Exception.Create(
      'Browser''s Default interface does not support IOleObject'
    );
  OleObj.SetClientSite(Site);
end;
```

*Listing 5*

In this method we retrieve the browser's *IOleObject* interface and then its *SetClientSite* method, passing the *Site* parameter. This parameter must be either a reference to our object's *IOleClientSite* interface, which registers the container with the *TWebBrowser* control, or it must be *nil*, to un-register the container.

> ***IOleObject***
>
> To learn about *IOleObject* see the Windows SDK help file that ships with Delphi.

# Implementing a non reference counted object

We will now move on to implement the required interfaces, starting with *IUnknown*. As already noted, the interface will be implemented so that the object is not reference counted.

*IUnknown* is defined in the `System` unit. We begin by adding its methods to a protected section of *TNulWBContainer*'s declaration. *Listing 6* shows the changes needed.

```
type
  TNulWBContainer = class(TObject,
    IUnknown, IOleClientSite, IDocHostUIHandler)
  private
    ...
  protected
    { IUnknown }
    function QueryInterface(const IID: TGUID; out Obj): HResult;
      stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    ...
  end;
```

*Listing 6*

We implement these methods as shown in *Listing 7*.

```
function TNulWBContainer.QueryInterface(
  const IID: TGUID; out Obj): HResult;
begin
  if GetInterface(IID, Obj) then
    Result := S_OK
  else
    Result := E_NOINTERFACE;
end;

function TNulWBContainer._AddRef: Integer;
```

```
begin
  Result := -1;
end;

function TNulWBContainer._Release: Integer;
begin
  Result := -1;
end;
```

*Listing 7*

*Listing 7* is quite straightforward. We copy the *QueryInterface* code from Delphi's implementation of *TInterfacedObject* except that we substitute *S_OK* for the 0 returned by *TInterfacedObject*'s code.

To ensure the object is not reference counted, *_AddRef* and *_Release* each return -1. Furthermore, unlike in *TInterfacedObject*, our version of *_Release* never frees the object. This leaves us to manage the object's lifetime ourselves.

# Implementing IOleClientSite

Next we move on to look at how our object will implement *IOleClientSite*.

> ***IOleClientSite***
>
> *IOleClientSite* is declared in the `ActiveX` unit. The Delphi Windows SDK help provides a full description of the interface.

Remember that, in our constructor, we register our object's *IOleClientSite* interface with the browser control. We know that browser control calls our *QueryInterface* method via the registered *IOleClientSite* interface to try to gain access to our *IDocHostUIHandler* implementation. Because the browser object doesn't require any other functionality of *IOleClientSite* we can get away with providing a minimal, "do nothing" implementation. Therefore we will just stub out the methods.

Once again we begin by adding the interface's methods to the protected section of our class declaration. *Listing 8* shows the required additions:

```
type
  TNulWBContainer = class(TObject,
    IUnknown, IOleClientSite, IDocHostUIHandler)
  private
    ...
  protected
    { IUnknown }
    ...
    { IOleClientSite }
    function SaveObject: HResult; stdcall;
    function GetMoniker(dwAssign: Longint;
      dwWhichMoniker: Longint;
      out mk: IMoniker): HResult; stdcall;
    function GetContainer(
      out container: IOleContainer): HResult; stdcall;
    function ShowObject: HResult; stdcall;
    function OnShowWindow(fShow: BOOL): HResult; stdcall;
    function RequestNewObjectLayout: HResult; stdcall;
    ...
  public
    ...
  end;
```

*Listing 8*

Our minimal implementation is shown in *Listing 9*. The comments in the code should outline the purpose of each method and explain what is being done. We won't discuss this further here.

```
function TNulWBContainer.GetContainer(
  out container: IOleContainer): HResult;
  {Returns a pointer to the container's IOleContainer
  interface}
begin
  { We do not support IOleContainer.
    However we *must* set container to nil }
  container := nil;
  Result := E_NOINTERFACE;
end;
```

```
function TNulWBContainer.GetMoniker(dwAssign, dwWhichMoniker: Integer;
  out mk: IMoniker): HResult;
  {Returns a moniker to an object's client site}
begin
  { We don't support monikers.
    However we *must* set mk to nil }
  mk := nil;
  Result := E_NOTIMPL;
end;

function TNulWBContainer.OnShowWindow(fShow: BOOL): HResult;
  {Notifies a container when an embedded object's window
  is about to become visible or invisible}
begin
  { Return S_OK to pretend we've responded to this }
  Result := S_OK;
end;

function TNulWBContainer.RequestNewObjectLayout: HResult;
  {Asks container to allocate more or less space for
  displaying an embedded object}
begin
  { We don't support requests for a new layout }
  Result := E_NOTIMPL;
end;

function TNulWBContainer.SaveObject: HResult;
  {Saves the object associated with the client site}
begin
  { Return S_OK to pretend we've done this }
  Result := S_OK;
end;

function TNulWBContainer.ShowObject: HResult;
  {Tells the container to position the object so it is
  visible to the user}
begin
  { Return S_OK to pretend we've done this }
  Result := S_OK;
end;
```

*Listing 9*

Our class can now act as a client site, or host, for the web browser control. It remains to implement the document host handler, *IDocHostUIHandler*.

# Implementing IDocHostUIHandler

Unlike the interfaces we have already implemented, Delphi doesn't supply a declaration of *IDocHostUIHandler* for us, so we have to provide this ourselves. *Listing 10* defines *IDocHostUIHandler*. The comments in the code give brief explanations of each method.

```
type
  IDocHostUIHandler = interface(IUnknown)
    ['{bd3f23c0-d43e-11cf-893b-00aa00bdce1a}']
    { Display a shortcut menu }
    function ShowContextMenu(
      const dwID: DWORD;
      const ppt: PPOINT;
      const pcmdtReserved: IUnknown;
      const pdispReserved: IDispatch): HResult; stdcall;
    { Retrieves the user interface capabilities and requirements
      of the application that is hosting the web browser }
    function GetHostInfo(
      var pInfo: TDocHostUIInfo): HResult; stdcall;
    { Enables us to replace browser menus and toolbars etc }
    function ShowUI(
      const dwID: DWORD;
      const pActiveObject: IOleInPlaceActiveObject;
      const pCommandTarget: IOleCommandTarget;
      const pFrame: IOleInPlaceFrame;
```

```
      const pDoc: IOleInPlaceUIWindow): HResult; stdcall;
    { Called when the browser removes its menus and toolbars.
      We remove menus and toolbars we displayed in ShowUI }
    function HideUI: HResult; stdcall;
    { Notifies that the command state has changed so the host
      can update toolbar buttons etc. }
    function UpdateUI: HResult; stdcall;
    { Called when a modal UI is displayed }
    function EnableModeless(
      const fEnable: BOOL): HResult; stdcall;
    { Called when the document window is activated or
      deactivated }
    function OnDocWindowActivate(
      const fActivate: BOOL): HResult; stdcall;
    { Called when the top-level frame window is activated or
      deactivated }
    function OnFrameWindowActivate(
      const fActivate: BOOL): HResult; stdcall;
    { Called when a frame or document's window's border is
      about to be changed }
    function ResizeBorder(
      const prcBorder: PRECT;
      const pUIWindow: IOleInPlaceUIWindow;
      const fFrameWindow: BOOL): HResult; stdcall;
    { Called when accelerator keys such as TAB are used }
    function TranslateAccelerator(
      const lpMsg: PMSG;
      const pguidCmdGroup: PGUID;
      const nCmdID: DWORD): HResult; stdcall;
    { Called by the web browser control to retrieve a registry
      subkey path that overrides the default IE registry settings }
    function GetOptionKeyPath(
      var pchKey: POLESTR;
      const dw: DWORD ): HResult; stdcall;
    { Called when the browser is used as a drop target and enables
      the host to supply an alternative IDropTarget interface }
    function GetDropTarget(
      const pDropTarget: IDropTarget;
      out ppDropTarget: IDropTarget): HResult; stdcall;
    { Called to obtain our IDispatch interface. Used to enable the
      browser to call methods in the host (e.g. from JavaScript) }
    function GetExternal(
      out ppDispatch: IDispatch): HResult; stdcall;
    { Gives the host an opportunity to modify the URL to be loaded }
    function TranslateUrl(
      const dwTranslate: DWORD;
      const pchURLIn: POLESTR;
      var ppchURLOut: POLESTR): HResult; stdcall;
    { Allows us to replace the web browser data object. It enables
      us to block certain clipboard formats or support additional
      clipboard formats }
    function FilterDataObject(
      const pDO: IDataObject;
      out ppDORet: IDataObject): HResult; stdcall;
  end;
```

*Listing 10*

The various interfaces and structures referenced in this definition are defined in the `ActiveX` and `Windows` units. The exception is *TDocHostUIInfo* which is defined *later in the article*.

Now we have defined the interface we can create the "do nothing" implementation in our class. To begin with add all the methods of *IDocHostUIHandler* to the protected section of *TNulWBContainer*'s declaration. I have not shown the new declaration here since it will simply repeat the methods shown in *Listing 10*.

*Listing 11* shows how we stub out the methods in such a way as to leave the browser object unchanged. An exploration of the MSDN documentation for the interface, along with some experimentation showed me the way. The comments in the listing should explain.

```
function TNulWBContainer.EnableModeless(
  const fEnable: BOOL): HResult;
begin
  { Return S_OK to indicate we handled (ignored) OK }
```

```
  Result := S_OK;
end;

function TNulWBContainer.FilterDataObject(
  const pDO: IDataObject;
  out ppDORet: IDataObject): HResult;
begin
  { Return S_FALSE to show no data object supplied.
    We *must* also set ppDORet to nil }
  ppDORet := nil;
  Result := S_FALSE;
end;

function TNulWBContainer.GetDropTarget(
  const pDropTarget: IDropTarget;
  out ppDropTarget: IDropTarget): HResult;
begin
  { Return E_FAIL since no alternative drop target supplied.
    We *must* also set ppDropTarget to nil }
  ppDropTarget := nil;
  Result := E_FAIL;
end;

function TNulWBContainer.GetExternal(
  out ppDispatch: IDispatch): HResult;
begin
  { Return E_FAIL to indicate we failed to supply external object.
    We *must* also set ppDispatch to nil }
  ppDispatch := nil;
  Result := E_FAIL;
end;

function TNulWBContainer.GetHostInfo(
  var pInfo: TDocHostUIInfo): HResult;
begin
  { Return S_OK to indicate UI is OK without changes }
  Result := S_OK;
end;

function TNulWBContainer.GetOptionKeyPath(
  var pchKey: POLESTR;
  const dw: DWORD): HResult;
begin
  { Return E_FAIL to indicate we failed to override
    default registry settings }
  Result := E_FAIL;
end;

function TNulWBContainer.HideUI: HResult;
begin
  { Return S_OK to indicate we handled (ignored) OK }
  Result := S_OK;
end;

function TNulWBContainer.OnDocWindowActivate(
  const fActivate: BOOL): HResult;
begin
  { Return S_OK to indicate we handled (ignored) OK }
  Result := S_OK;
end;

function TNulWBContainer.OnFrameWindowActivate(
  const fActivate: BOOL): HResult;
begin
  { Return S_OK to indicate we handled (ignored) OK }
  Result := S_OK;
end;

function TNulWBContainer.ResizeBorder(
  const prcBorder: PRECT;
  const pUIWindow: IOleInPlaceUIWindow;
  const fFrameWindow: BOOL): HResult;
begin
  { Return S_FALSE to indicate we did nothing in response }
```

```
  Result := S_FALSE;
end;

function TNulWBContainer.ShowContextMenu(
  const dwID: DWORD;
  const ppt: PPOINT;
  const pcmdtReserved: IInterface;
  const pdispReserved: IDispatch): HResult;
begin
  { Return S_FALSE to notify we didn't display a menu and to
    let browser display its own menu }
  Result := S_FALSE
end;

function TNulWBContainer.ShowUI(
  const dwID: DWORD;
  const pActiveObject: IOleInPlaceActiveObject;
  const pCommandTarget: IOleCommandTarget;
  const pFrame: IOleInPlaceFrame;
  const pDoc: IOleInPlaceUIWindow): HResult;
begin
  { Return S_OK to say we displayed own UI }
  Result := S_OK;
end;

function TNulWBContainer.TranslateAccelerator(
  const lpMsg: PMSG;
  const pguidCmdGroup: PGUID;
  const nCmdID: DWORD): HResult;
begin
  { Return S_FALSE to indicate no accelerators are translated }
  Result := S_FALSE;
end;

function TNulWBContainer.TranslateUrl(
  const dwTranslate: DWORD;
  const pchURLIn: POLESTR;
  var ppchURLOut: POLESTR): HResult;
begin
  { Return E_FAIL to indicate that no translations took place }
  Result := E_FAIL;
end;

function TNulWBContainer.UpdateUI: HResult;
begin
  { Return S_OK to indicate we handled (ignored) OK }
  Result := S_OK;
end;
```

*Listing 11*

That completes the implementation of the "do nothing", reusable, base class. If we were to create an instance of this object it would successfully host a *TWebBrowser* control, but would have no visible effect upon it.

In the *next section* we will derive our custom class from *TNulWBContainer*.