Exposing Menus in Alibre Design:
Add-ons wanting to expose a menu structure within Alibre Design will need to support a COM interface named IAlibreAddOn. The pointer to this interface will be obtained by Alibre Design at the time when Alibre Design loads the list of available add-ons when it is launched. Alibre Design will attempt to obtain this interface pointer from the add-on by invoking the function GetAddOnInterface ( ) exported by the add-on's DLL. All advanced add-ons desiring tight UI integration with Alibre Design will have to implement this function. This function will return the IAlibreAddOn interface pointer implemented by the add-on.

In this add-on sample project, see the following source files to see an example usage of some of the above mentioned API:

Function GetAddOnInterface() - SampleAddOn.H and SampleAddOn.CPP

Interface IAlibreAddOn – CSampleAddOnInterface.H and CSampleAddOnInterface.CPP

Adding GUI elements into Alibre Design's Explorer:

Some add-ons may want to tightly integrate their UI within AD by showing their UI elements within AD's explorer pane that currently houses AD's Design Explorer tree control. The requirements and assumptions for providing such a capability are defined here:

- An add-on requiring this capability will have to implement the IAlibreAddOn interface noted above.
- The capability involves adding a new tab-page to AD's Explorer. The add-on's UI elements will go into this tab page.
- This capability will be <u>available to an add-on only within the scope of executing an add-on menu command that implements IAlibreAddOnCommand</u>.
- When the add-on command is invoked by user, Alibre Design will call IAlibreAddOnCommand.AddTab (VARIANT_BOOL *pAddTab) on the command. If the add-on does not require a GUI presence, it will return E_NOTIMPL or set VARIANT_FALSE to *pAddTab and a tab page will not be added. Else a new tab page will be created for the add-on by Alibre Design.
- If the add-on indicated to Alibre Design that it wanted to add a tab page to the Explorer pane, Alibre Design will subsequently call the method IAlibreAddOnCommand.OnShowUI(int hWnd) on the command. Here, the add-on needs to add whatever UI controls it wants to show inside this tab page. The hwnd passed in is the windows handle of the newly added pane for Add-on for parenting purposes.

In this add-on sample project, see the following source files to see an example usage of some of the above mentioned API:

IAlibreAddOn::InvokeCommand() – CsampleAddOnInterface.CPP

IAlibreAddOnCommand::AddTab() – CShowUICommand.H and CShowUICommand.CPP

IAlibreAddOnCommand::OnShowUI() – CShowUICommand.CPP

Displaying graphics into Alibre Design's graphics window:

An add-on can draw into AD's Geometry canvas (graphics window) using GDI functions. Let us lay here the requirements and assumptions that define this feature:

- An add-on requiring this capability will have to support the IAlibreAddOn interface.
- The capability involves drawing or superimposing graphic elements (lines, text and other symbols) into AD's graphics window that is tightly synchronized with AD's rendering process.
- This capability will be <u>available to an add-on only within the scope of executing an add-on menu command that implements IAlibreAddOnCommand</u>. Alibre Design will call the 'OnRender' method on this interface during the lifetime of the command whenever Alibre Design is updating its display canvas.
- The GDI graphics rendered by the add-on are of transient nature. That is, they can be displayed only as long as the add-on command is active. So, <u>this capability is ideal for temporary graphics like previews and operation results etc. – things that do not stick around after the lifetime of the command that drew them.</u>
- The active add-on command can trigger update of Alibre Design's display canvas. This can be achieved by explicitly calling method UpdateCanvas or InvalidateCanvas on the IADAddOnCommandSite interface telling AD to redraw its canvas.
- Add-on can instruct Alibre Design to not perform its (the latter's) standard rendering during an update triggered by the add-on by making a call to implemented by Alibre. During a canvas update, this will cause Alibre Design to clear the canvas and then let the add-on alone to render its graphics to the canvas via the call to "OnRender" mentioned above.

In this add-on sample project, see the following source files to see an example usage of some of the above mentioned API:

IAlibreAddOnCommand::OnRender() – CDefineAnimationCommand.CPP and CStartAnimationCommand.CPP

IADAddOnCommandSite::UpdateCanvas() – CDefineAnimationCommand.CPP

IADAddOnCommandSite::OverrideRender() – CStartAnimationCommand.CPP

Saving custom data to Alibre file:

Add-on can persist data within Alibre Design file which is a compound file. AddOn interested in persisting its data within Alibre Design file needs to provide additional XML tag in the configuration file (<AddOn>.adc) file. Following is the sample XML tag:

```
<PersistentData streamName="MyStreamName"/>
```

Additionally, the add-on will have to implement two methods on IAlibreAddOn interface, namely, LoadData ( ) and SaveData ( ).

In this add-on sample project we do not persist custom data.

Steps to implement tightly integrated Add-On in VC++:

Let us take a closer look at implementing the interfaces IAlibreAddOn and IAlibreAddOnCommand that AlibreDesign would expect an AddOn wanting tighter integration to support.

Step 1: Create an new VC++ DLL project, import the files AlibreX_64.tlb and AlibreAddOn_64.tlb in the StdAfx.h with the path suitable to your machine:

```
#import "C:\Program Files\Alibre Design\Program\AlibreX_64.tlb"
#import "C:\Program Files\Alibre Design\Program\AlibreAddOn_64.tlb" raw_interfaces_only

using namespace AlibreX;
using namespace AlibreAddOn;
```

`AlibreX` sets you up for using API to, for example, query/manipulate geometry or assembly structure in an Alibre Design model.

`AlibreAddOn` sets you up for using API needed to implement tightly integrated addon inside Alibre Design

Step 2: We will need to implement a function: GetAddOnInterface () and export it on the AddOn's DLL.

An Alibre add-on DLL has to export these functions: AddOnLoad, AddOnInvoke, and AddOnUnload. These are mandatory for any AddOn in Alibre. In addition to these functions, the function GetAddOnInterface( ) needs to be implemented and exported by the DLL of an add-on requiring tighter integration with Alibre Design. This DLL function will have to return object of type: IAlibreAddOn Interface as IUnknown*.

Following is the typical definition of these exported functions in the main header file in the AddOn DLL project:

```
extern "C"
{

    //(Please note APICLIENTAPP_API  is __declspec(dllexport) )

    APICLIENTAPP_API void AddOnLoad (HWND windowHandle,
                                VOID *pAutomationHook,
                                VOID *reserved);

    APICLIENTAPP_API void AddOnInvoke (HWND windowHandle,
                                  VOID *pAutomationHook,
                                  LPCSTR sessionName,
                                  BOOL isLicensed,
                                  VOID *reserved1,
                                  VOID *reserved2);



    APICLIENTAPP_API void AddOnUnload (HWND windowHandle,
                                  BOOL forceUnload,
                                  BOOL *cancel,       // set TRUE to cancel
                                  VOID *reserved1,
                                  VOID *reserved2);

    //          This          last          function          to          return          IAlibreAddOn          Interface          is          needed          to          support
    // tighter Integration.
    APICLIENTAPP_API IUnknown* GetAddOnInterface ();
}
```

The implementation of GetAddOnInterface () function should instantiate an object of class, say, CAddOnInterface, that is derived from IAlibreAddOn (more on this in Step 3 below). It will return this to the caller (which of course will be Alibre Design) as an IUnknown pointer.
Typical Implementation of GetAddOnInterface () function could look like this:

```
APICLIENTAPP_API IUnknown* GetAddOnInterface ()
{
	AFX_MANAGE_STATE (AfxGetStaticModuleState ())

	// CAddOnInterface class implements IAlibreAddOn Interface
	IAlibreAddOn * pAddOnInterface = new CAddOnInterface (theApp.m_pRoot);
	return (IUnknown *) pAddOnInterface;
}
```

Note: Root object: "theApp.m_pRoot" that is used in above function is the IADRoot object that can be obtained from AutomationHook in AddOnLoad () function. IADRoot is the top-most API object in Alibre Design's Automation API hierarchy.

Step 3: As mentioned above, we add a new class, say, CAddOnInterface to implement IAlibreAddOn. The definition of this class in its header file looks like this:

```
class CAddOnInterface  : public IAlibreAddOn
```

Implement the required methods for IUnknown and IDispatch. Now, let us take a look at the main properties and methods in **IAlibreAddOn interface** that a tightly integrated add-in would implement:

```
HRESULT _stdcall get_RootMenuItem (/*[out,retval]*/ int *pRootMenuID);
HRESULT _stdcall get_HasSubMenus (/*[in]*/ int menuID, /*[out,retval]*/ VARIANT_BOOL *pHasSubMenus);
HRESULT _stdcall get_SubMenuItems (/*[in]*/ int menuID, /*[out,retval]*/ SAFEARRAY **pSubMenuIDs);
HRESULT _stdcall get_MenuItemText (/*[in]*/ int menuID, /*[out,retval]*/ BSTR* pMenuDisplayText);
HRESULT _stdcall get_PopupMenu (/*[in]*/ int menuID, /*[out,retval]*/ VARIANT_BOOL *IsPopup);
HRESULT _stdcall get_MenuItemState (/*[in]*/ int menuID,  /*[in] */ BSTR sessionIdentifier, /*[out, retval]*/ enum ADDONMenuStates *pType);
HRESULT _stdcall get_MenuItemToolTip (/*[in]*/ int menuID, /*[out, retval]*/ BSTR *pToolTip);
HRESULT _stdcall InvokeCommand (/*[in]*/ int menuID, /* [in] */ BSTR sessionIdentifier,  /*[out, retval]*/ IAlibreAddOnCommand **pCommand);
HRESULT _stdcall get_HasPersistentDataToSave(/* [in] */ BSTR sessionIdentifier, /*[retval][out] */  VARIANT_BOOL *IsPopup);
HRESULT _stdcall LoadData (/*[in]*/ struct IStream * ppCustomData, /*[in]*/ BSTR sessionIdentifier);
HRESULT _stdcall SaveData (/*[in]*/ struct IStream * pCustomData, /*[in]*/ BSTR sessionIdentifier);
HRESULT _stdcall MenuIcon (/*[in]*/ long menuID, /*[out,retval]*/ BSTR *pMenuIconPath);

HRESULT _stdcall UseDedicatedRibbonTab (/*[out,retval]*/ VARIANT_BOOL *pFlag);
```

Let us discuss about the necessary implementation for required functions here.

IAddOnInterface.get_RootMenuItem (int *pRootMenuID) will need to return a non-zero integer value in **pRootMenuID**. If Root Menu ID returned is zero, or if the method **IAddOnInterface.get_MenuItemText (int RootMenuID, BSTR* pMenuDisplayText)** returns an empty string, Alibre Design will not attempt to call any of the other methods needed to build a menu structure for the add-on on Alibre Design's menu bar.

Once Alibre Design receives a non-zero value for Root Menu ID, it queries for its sub menu items. So for each menu id, Alibre Design recursively calls **get_HasSubMenus (menuID, pHasSubMenu), IAddOnInterface::get_SubMenuItems (menuID, ppSubMenuIDs) and IAddOnInterface::getMenuItemText (menuID, pMenuDisplayText)** to construct the menu hierarchy for the add-on.

When user clicks the add-on's root menu to expand it, Alibre Design updates the state of the Menu item by calling **IAlibreAddOn.getMenuItemState (menuId)**. Depending on the Menu State (one of enum constants defined in **ADDONMenuStates)** returned by the method, menu is shown enabled or disabled etc.

When user clicks any of the add-on's leaf menu commands, Alibre Design calls **IAlibreAddOn.InvokeCommand (/*[in]*/ int menuID, /* [in] */ BSTR sessionIdentifier, /*[out, retval]*/ IAlibreAddOnCommand **pCommand)** on the add-on. Implementation of this function in your class, say, **CAddOnInterface,** would handle the command processing. For implementing a command that needs to process user events like mouse clicks, or needs to show GUI controls etc, InvokeCommand ( ) will have to return an object of type: **IAddOnCommand**. If the returned object is not NULL and is of type IAddOnCommand, AlibreDesign will create a **command site object** for the command and set the CommandSite property on this interface. The command site object represents this add-on command on Alibre Design's side and provides a means for the add-on command to define command behavior, terminate command etc. Alibre does this by implementing the interface, IADAddOnCommandSite, on command site object. Note that the lifetime of the command site object is the life time of the command itself. That is, it goes away when the command is terminated. We will now talk about the various flavors of commands an add-on can implement.

An add-on can implement a command to be:

1) A two-way toggle command having just two states – active and inactive. As an example, such a command could be made to display an add-on specific tab window in Alibre Design's Explorer pane.
2) A regular command having multiple states that can listen to Alibre Design events, display graphics on Alibre Design's graphics windows and which will get terminated upon user invoking any other command. More on this in the next section, Step 4.
3) A stateless, 'fire and forget' type command that just has to perform simplistic processing without any advanced integration requirements. For such a command, the add-on would implement IAlibreAddOn.InvokeCommand ( ) method to do all the needed processing so that when this method returns, the command has been fully executed. That is, lifetime of the command is the duration of the execution of InvokeCommand ( ). InvokeCommand would return NULL back to Alibre.

Note: If IAlibreAddOn.getRootMenuItem () returned zero, AddOn will be listed under Tools →AddOns menu. On user clicking on this menu item, Alibre Design will call IAlibreAddOn.InvokeCommand ( ). Rest of the command processing will depend on the implementation of AddOn's IAlibreAddOn.InvokeCommand ( ) as described above. If this command returns an IAlibreAddOnCommand object, then, the usual process of Showing UI takes place as mentioned earlier.

Now, let us discuss how an add-on can save its add-on specific data into Alibre Design's workspace. First of all, such add-ons need to expose a special XML tag entry, "Persistent Data" in the add-on configuration (.adc) .The "StreamName" property value should be a valid string. Sample entry:

<PersistentData StreamName="SampleAddOn_Data"/>

During the save process, for AddOns with above-mentioned entry in its .adc file, Alibre Design will call, **IAlibreAddOn.getHasPersistentDataToSave (BSTR sessionIdentifier, VARIANT_BOOL *pboolHasData).** If true is returned, an IStream with this name will be created in the Alibre file (which is a compound file). Alibre Design will then call **IAlibreAddOn.SaveData (struct IStream * pCustomData, BSTR sessionIdentifier)** passing in this IStream pointer. Add-on should implement code to write its data to this IStream. Alibre Design takes care of committing the IStream to disk.

Conversely, upon opening any Alibre Design model file, if any loaded AddOn has the above-mentioned entry in its .adc file, then, Alibre Design will attempt to access the IStream of given name. If the IStream is found, then, Alibre Design calls into the add-on to give the add-on to load the contents of its private data contained in its stream:

IAlibreAddOn.LoadData (struct IStream * pCustomData, BSTR sessionIdentifier)

IMPORTANT NOTE:
In several of the methods on the interface IAlibreAddOn, notice that there is an argument named "**sessionIdentifier**" that Alibre Design passes to the add-on. Alibre Design uses this string as means of conveying to the add-on, the session (workspace) that this call is associated with. This is because, in the Alibre Design process, user could have opened several workspaces (parts or assemblies); the user could be invoking your add-on on several of these opened workspaces but there is only one loaded instance of the add-on DLL (and hence, only one IAlibreAddOn interface). So, how does your add-on DLL know which session is being referred to any point? This is where the "sessionIdentifier" comes in. Here it may be useful to just point out that using a method exposed in the AlibreX type-library API, add-on can resolve the "sessionIdentifier" to an **IADSession** interface pointer that wraps the workspace in question.

Step 4: Add a new class to implement IAlibreAddOnCommand interface to serve as a handler for every 'leaf' menu command item exposed by the AddOn. Let us call the new class CAddOnCommand. The definition of this class in the header file will look like this:

```
class CAddOnCommand : public IAlibreAddOnCommand
```

Implement required methods for IUnknown and IDispatch. Now, let us take a look at the main properties and methods in **IAlibreAddOnCommand** interface that a tightly integrated add-in may choose to implement:

```
HRESULT _stdcall putref_CommandSite(/* [in] */ IADAddOnCommandSite *pSite);
HRESULT _stdcall get_CommandSite(/* [retval][out] */ IADAddOnCommandSite **pSite);
HRESULT _stdcall AddTab(/* [retval][out] */ VARIANT_BOOL *pAddTab);
HRESULT _stdcall OnShowUI(/* [in] */ int64 hWnd);
HRESULT _stdcall OnRender(/* [in] */ int hDC);
HRESULT _stdcall OnClick (/* [in] */ int screenX, /* [in] */ int screenY);
HRESULT _stdcall OnDoubleClick (/* [in] */ int screenX, /* [in] */ int screenY);
HRESULT _stdcall OnMouseDown (/* [in] */ int screenX, /* [in] */ int screenY, /* [in] */ int buttons);
HRESULT _stdcall OnMouseMove (/* [in] */ int screenX, /* [in] */ int screenY, /* [in] */ int buttons);
HRESULT _stdcall OnMouseUp (/* [in] */ int screenX, /* [in] */ int screenY, /* [in] */ int buttons);
HRESULT _stdcall OnKeyPress(/* [in] */ int keyCode);
HRESULT _stdcall OnSelectionChange (void);
HRESULT _stdcall OnTerminate (void);
HRESULT _stdcall OnComplete( void);
HRESULT _stdcall IsTwoWayToggle( /* [retval][out] */ VARIANT_BOOL *pIsTwoWayToggle);
```

When user clicks an add-on's command, as noted earlier, Alibre Design will obtain the IAddOnCommand interface pointer from the add-on. It will then spawn off the following series of actions:

- Alibre Design first call **IAlibreAddOnCommand.AddTab (*pAddTab)**.
  If your command requires showing some GUI inside an add-on specific tab on the Design Explorer, your command needs to return "VARIANT_TRUE" in the function implementation of **AddTab ()** as shown below:

```
HRESULT _stdcall CAddOnCommand::AddTab (VARIANT_BOOL *pAddTab)
{
        HRESULT hr = S_OK;
        *pAddTab = VARIANT_TRUE;
        return hr;
}
```

- Upon receiving TRUE as shown above, AlibreDesign creates a new tab page on Design Explorer for this AddOn. Subsequently, it calls: **IAlibreAddOnCommand.OnShowUI (int64 hWnd)** which AddOn can implement to add any UI controls under its tab page using the passed handle for parenting purposes. For e.g., say AddOn wants to add a list control. Here is sample code for that:

```
HRESULT _stdcall CAddOnCommand::OnShowUI (int64 hWnd)
{
        HRESULT hr = S_OK;
        CWnd* cWindow = CWnd::FromHandle ((HWND) hWnd);

        // Initialize List control
        CListCtrl* lstParNames = new CListCtrl ();
        lstParNames->Create                                    (WS_CHILD|WS_VISIBLE|WS_BORDER|LVS_LIST,
                              CRect(10,10,200,200), cWindow, 1);
        // Add your list items here.

        return hr;
}
```

- Next, AlibreDesign checks whether this command is a two-way toggle command by calling **IAlibreAddOnCommand.IsTwoWayToggle(*pIsTwoWayToggle).** If add-on returns back TRUE value, **IAlibreAddOnCommand.OnComplete ()** is called next. AddOn can do any finishing work here. If it is not a two-way toggle command, AlibreDesign activates a 'Listener' to watch on user and canvas events/ updates. This listener will remain active until another tool/command is activated.
- Alibre creates an object of type IADAddOnCommandSite for the activated command and passes it to add-on command via the property: **IAlibreAddOnCommand.putref_CommandSite (IADAddOnCommandSite *pSite)**. If the command wants to override rendering, this is one place to convey that information to AlibreDesign. For example, a suggested implementation is shown below

```
HRESULT _stdcall CAddOnCommand::putref_CommandSite(/* [in] */ IADAddOnCommandSite *pSite)
{
        HRESULT hr = S_OK;

        if (pSite)
        {
                hr = pSite->OverrideRender (VARIANT_FALSE);

                // Keep the pointer to a class level variable for future use.
                hr = pSite->QueryInterface (&m_pCmdSite);
        }
        return hr;
}
```

  The command site interface exposes some useful methods that the command could use during its lifetime.

- If AddOn would like to act based on mouse moves or selection changes, necessary implementation could be provided for other event handling methods in IAlibreAddOnCommand interface.
- Alibre Design calls IAlibreAddOnCommand.Terminate ( ) when the add-on command is terminated, say, as a result of user activating another command. It is worthy to note here that the add-on command could programmatically terminate itself by calling the **IADAddOnCommandSite.Terminate ( )** method.