Customizing Autodesk® Navisworks® 2013 with the .NET API

Simon Bee – Autodesk

This class will introduce you to customizing Autodesk Navisworks 2013 using the new .NET API. It will cover how to create add-ins that sit inside the product and allow you to integrate with your own programs, how to automate the product to perform process-intensive and repetitive tasks, and how to embed the .NET controls in your own application to leverage the powers of Navisworks. This class will benefit those wanting to extend the functionality of Navisworks using a .NET language such as C#. Attendees should have some programming experience.

Learning Objectives
At the end of this class, you will be able to:

- Create an application using the Navisworks 2013 .NET controls
- Automate Navisworks 2013
- Write an add-in for Navisworks 2013
Introduction to the Autodesk Navisworks .NET API

Overall Structure
The main .NET API comprises four assemblies:

- Autodesk.Navisworks.Api
- Autodesk.Navisworks.Automation
- Autodesk.Navisworks.Controls
- Autodesk.Navisworks.ComApi

These are just standard .NET assemblies, and you reference them in the standard way. Most of the functionality is in the first assembly, Autodesk.Navisworks.Api.

Querying the Model
You can query the items in a loaded model. The four main classes that you need to use are:

- Autodesk.Navisworks.Api.Application
- Autodesk.Navisworks.Api.Model
- Autodesk.Navisworks.Api.ModelItem

There is a single, global Application object from which you can access all other objects. The currently loaded document can be accessed via the ActiveDocument property. This has a collection of Models, and each model has a collection of ModelItem objects.

You might access the root item of the first model using the following code:

```csharp
Document doc = Application.ActiveDocument;
Model model = doc.Models[0];
ModelItem root = model.RootItem;
bool is_hidden = root.IsHidden;
```

Object Properties
You can access the model item properties that appear in the Properties window in the Navisworks UI. Properties are arranged into groups, called a PropertyCategory, which correspond to a tab in the UI, and each category has a collection of DataProperty objects.

First, you would find the PropertyCategory object on the ModelItem you are interested in:

```csharp
PropertyCategory prop_cat =
    root.PropertyCategories.FindCategoryByName(PropertyCategoryNames.Item);
```

Here, we are using the PropertyCategoryNames enum, which has a set of pre-defined names of property categories. Once you have a category, you can access the DataProperty members.
DataProperty prop = prop_cat.Properties.FindPropertyByName(DataPropertyNames.ItemName);
string value = prop.Value.ToString();

The ToDisplayString method returns a string suitable for showing in a UI.

Finding Items
There are multiple ways to find items using the API.

- Iteration
- Search
- LINQ

Each method has its benefits and drawbacks. First, let's look at Iteration. All collections returned by the API are standard .NET collections, so the normal method of iteration will work as expected.

```csharp
foreach (ModelItem item in Application.ActiveDocument.CurrentSelection.SelectedItems)
{
    // Examine item here.
}
```

This may be slow as you can have many managed to un-managed transitions. If the items you are looking for can be found using the Find Items dialog in the UI, then you are better off using the Search object.

The Search object comprises two parts:

- The Selection
- Search Conditions

Typically, you will want to search the entire model, but you can search just a given selection of ModelItem objects instead. Beware though that by default, the selection for a new Search object is empty.

You add Search Conditions to specify which items you want to find. The following snippet of code will find all the items in the model where the “Revit Element” property is equal to “Stairs”.

Customizing Autodesk® Navisworks® 2013 with the .NET API

```csharp
Search search = new Search();
search.Selection.SelectAll();
search.SearchConditions.Add(
    SearchCondition.HasPropertyByName("LcRevitData",
        "LcRevitPropertyElementCategory").
    EqualValue(VariantData.FromDisplayString("Stairs")));

ModelItemCollection items = search.FindAll(Application.ActiveDocument, false);
Application.ActiveDocument.CurrentSelection.CopyFrom(items);
```

This will run a lot quicker than iterating over the entire model one item at a time and checking the DataProperty on each item. You can also use the Search object to return a collection that is suitable to use with LINQ, which lets you combine native search conditions with your own ones.

Here we are searching all the "Stairs" to find items where the "price" is greater than £100.

```csharp
IEnumerable<ModelItem> expensive_items =
    from item in search.FindIncremental(Application.ActiveDocument, false)
    where ItemPrice(item) > 100
    select item;
```

Note that we call the FindIncremental method instead of FindAll as in the previous snippet, as this is designed to work with these LINQ-style statements.

**Modifying the Model**

To modify anything in the document, you make changes via DocumentParts. There are currently four main DocumentPart objects. They are all accessed from the Document object.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Document Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>DocumentTool</td>
<td>Application.ActiveDocument.Tool</td>
</tr>
<tr>
<td>DocumentSelectionSets</td>
<td>Application.ActiveDocument.SelectionSets</td>
</tr>
</tbody>
</table>

Each of these DocumentParts can be used to change some part of the document. For instance, you can set the currently selected items. Here, we are using another LINQ-style query to find all the non-hidden items.
We can override the colour of these items in our collection using the DocumentModels document part.

```csharp
Application.ActiveDocument.Models.OverridePermanentColor(items, Color.Red);
```

We can change the currently active tool by updating the Document.Tool property directly.

```csharp
Application.ActiveDocument.Tool = Tool.Select;
```

**SavedItems**

A lot of the other items stored in a Navisworks document that are not ModelItems are SavedItems. Currently, the only SavedItem-derived objects you can access via the .NET API are Selection Sets and TimeLiner Tasks. All SavedItems have some common properties, such as a display name and a collection of comments.

**Responding to Events**

The .NET API will raise events at appropriate points. You can listen to these events in the normal way.

```csharp
new EventHandler<EventArgs>(CurrentSelection_Changed);

private void CurrentSelection_Changed(object sender, EventArgs e)
{
    // Respond to event...
}
```

**Writing Plugins**

**First Steps**

To create your first plugin, you will want to create a new C# Class Library project in Visual Studio 2010.

Next, set the output assembly name of your project to be the name of the plugin, appended with your four letter Autodesk Developer ID, separated with a full-stop.
You will need to add a reference to each assembly you use. Normally, you will just need to reference the Autodesk.Navisworks.Api assembly.

It is a good idea to set a post-build step to copy your built plugin into the “plugins” directory where Navisworks is installed to a sub-directory named the same as the assembly.

Then finally, set Roamer.exe as the start-up program for your project.
Creating a Simple AddIn Plugin
You can create the most basic plugin by using the AddInPlugin class. These let you run a simple command. You can also place your plug-in in a variety of places in the UI.

You need to derive a new class from the AddInPlugin class, and add a set of attributes to the class. Note that all the plugin related classes live in their own namespace, Autodesk.Navisworks.Api.Plugins.

The AddInPlugin class only has one method to implement, which is executed when the user clicks on your plugin icon in the UI.

```csharp
[PluginAttribute("MyPlugin", "ADSK", ToolTip = "My Plugin", DisplayName = "My Plugin")]
[AddInPluginAttribute(AddInLocation.AddIn)]
{
    public override int Execute(params string[] parameters)
    {
        // Do stuff here!
        return 0;
    }
}
```

When executed from the UI, the parameters will be empty, and the return code is ignored. These are only used when the plugin is executed through the Automation API. The locations the plugin can appear in are shown below.

<table>
<thead>
<tr>
<th>Enum Value</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Export</td>
<td>The Export menu</td>
</tr>
<tr>
<td>None</td>
<td>Not visible</td>
</tr>
<tr>
<td>CurrentSelectionContextMenu</td>
<td>Context menu for the current selection</td>
</tr>
<tr>
<td>Import</td>
<td>The Import menu</td>
</tr>
<tr>
<td>AddIn</td>
<td>The AddIn menu</td>
</tr>
<tr>
<td>Help</td>
<td>The Help menu</td>
</tr>
</tbody>
</table>

Writing a Docking Plugin
You can write a plugin with your own UI that sits inside a docking window so it can be integrated into the rest of the Navisworks UI. You create a control, using either WinForms or WPF. You will
typically have two plugins; a basic AddIn plugin that controls the creation and display of your DockPanePlugin.

First, create yourself a WinForms or WPF control. Here is an example one, called MainControl.

![MainControl example](image)

Now, you need to create an instance using the DockPanePlugin class.

```csharp
[Plugin("MyDockPanePlugin", "ADSK", DisplayName = "My Plugin")]
DockPanePlugin(100, 300)]
public class MyDockPanePlugin : DockPanePlugin
{
    public override Control CreateControlPane()
    {
        //create the control that will be used to display in the pane
        MainControl control = new MainControl();
        control.Dock = DockStyle.Fill;
        //create the control
        control.CreateControl();
        return control;
    }

    public override void DestroyControlPane(Control pane)
    {
        pane.Dispose();
    }
}
```

Navisworks will call the CreateControlPane and DestroyControlPane when it needs to create or destroy the dock pane control. Next, you need a normal AddInPlugin to show or hide your dock pane.
```csharp
[Plugin("MyDockPaneAddin", "ADSK",
    DisplayName = "MyDockPaneAddin")]
public class BasicDockPaneAddin : AddInPlugin
{
    public override int Execute(params string[] parameters)
    {
        // Find the plugin

        if (pr != null && pr is DockPanePluginRecord && pr.IsEnabled)
        {
            // Check if it needs loading
            if (pr.LoadedPlugin == null)
            {
                pr.LoadPlugin();
            }

            DockPanePlugin dpp = pr.LoadedPlugin as DockPanePlugin;
            if (dpp != null)
            {
                // Switch the Visible flag
                dpp.Visible = !dpp.Visible;
            }
        }
        return 0;
    }
}
```

This simple function first tries to locate the Dock Pane plugin, then loads if it required. Finally, it toggles the visibility of the plugin.

**Adding Your Own Ribbon**

You can also create a plugin that installs its own ribbon into the product. The CommandHandlerPlugin allows you to derive a class that creates a ribbon and responds to commands from it. The ribbon is defined in custom XAML. The examples shipped with the product contain a plugin that creates a ribbon, so this is a good starting point for creating your own ribbon.
**Automating Navisworks**

You can use the automation API to control Navisworks. You can load, append and save files, and you can execute plugins to do more complicated work. The main class you use is the `Autodesk.Navisworks.Api.Automation.NavisworksApplication`, which is in the `Autodesk.Navisworks.Api.Automation` assembly.

An example of loading two files, executing a plugin and saving the result is shown here.

```csharp
static void Main(string[] args)
{
    NavisworksApplication app = new NavisworksApplication();
    app.OpenFile(@"C:\file\A.nwd");
    app.AppendFile(@"C:\file\B.nwd");
    app.ExecuteAddInPlugin("Colouriser.ADSK");
    app.SaveFile(@"C:\files\Combined.nwd");
}
```

**Using the .NET Controls**

The controls can be used to add the Navisworks large-modelling capability to your own applications. There are three controls that you use.


You only need one ApplicationControl object per-app, but you can have multiple View or Document controls. You can also link two View controls to the same Document to give multiple views of the same model.

Note however that in order to use the .NET Controls, you need to have a licensed copy of Navisworks installed on the machine.

**COM Interop**

When you can’t do something in the .NET API, then you might be able to use the COM API instead. In order to make this easier, there are two extra assemblies. `Autodesk.Navisworks.Interop.ComApi` is the COM API wrapped up in a .NET Interop assembly, whilst the `Autodesk.Navisworks.ComApi` assembly provides the access to and helper methods for using the COM API.

For instance, you might want to access the names of the Saved Views in a model. First, use the ComApiBridge object to get a “State”, then use the COM Interop API to access the saved views.
You can also use the helper methods to convert between certain object types in the .NET API and the COM API. For instance, here we convert a selection of items from one type to the other.

```csharp
InwOpSelectionSet set = (InwOpSelectionSet)sets[1];
ModelItemCollection items =
    ComApiBridge.ToModelItemCollection(set.selection);
Application.ActiveDocument.CurrentSelection.CopyFrom(items);
```

**Further Information**
There are several sources of further information.

**.NET API Documentation and Examples**
Shipped and installed with the product, the documentation has both a Developers Guide and a full .NET API Reference, supported with lots of example projects that demonstrate all the areas of the API.

The ApplInfo example plugin is a very useful tool for exploring the Navisworks API from inside the product.

**Autodesk Developer Network**
Gives you access to API experts and engineers at Autodesk for the best support for writing your applications using Autodesk APIs.