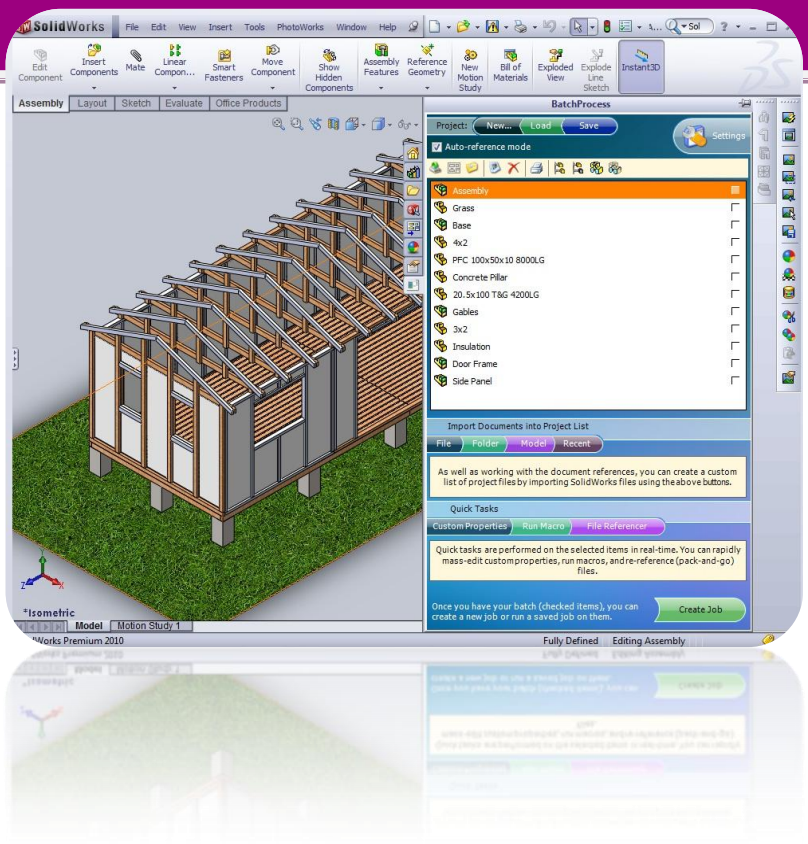
angelsix
free your mind!

Creating a SolidWorks Add-in from scratch

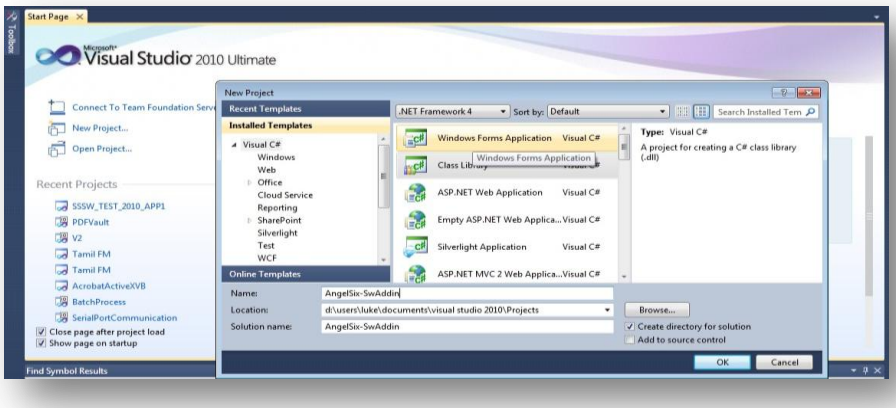


Ready to take your SolidWorks macro/tool to the next level but don't know where to start? How about creating a fast, efficient .Net Taskpane add-in giving you total flexibility of your program and form designs through the .Net Framework at the same time as being in-process and perfectly integrated into SolidWorks?

After fitting as much as I could into the last 2 SolidWorks books the one thing I was always pining to get into the books was a good run down of creating Taskpane add-ins using .Net, however I did not want to cram it into the last 20 pages of a book; I wanted to cover it in depth. So, with that in mind I have decided to cover Taskpane add-ins over the next few months through our new Tutorials section of our site.

Right without any more chit-chat let's get right into it. This first tutorial will take you through creating a .Net Taskpane add-in that is registered and starts up when SolidWorks starts, becoming visible in the Taskpane tab for all to see. I will not cover any SolidWorks API stuff in this tutorial other than getting the Taskpane framework up and running the correct way (none of this Visual Studio Template Solution rubbish, let's do it properly!).

To begin with open up whatever copy of Visual Studio you have, and create a new C# (or VB.Net if you prefer) Class Library. Give it a name and click OK to create the solution and you are ready to start.



References

The only real requirement for a Taskpane add-in is to inherit the ISwAddin interface found in the SolidWorks library. We will also add COM registration coding too so that when we build our Visual Studio Solution it will take care of adding the registry entries to the SolidWorks folder for us which makes it appear in the Add-ins menu and start up on load.

Before we can code anything useful we must add the SolidWorks references. From the Solution Explorer to the right hand side, right-click on the Project (second item down) and select Add Reference... Go to the Browse tab and navigate to the SolidWorks installation folder (typically C:\Program Files\SolidWorks 20XX\SolidWorks), and select the following files:

"solidworkstools.dll" "SolidWorks.Interop.sldworks.dll" "SolidWorks.Interop.swcommands.dll" "SolidWorks.Interop.swconst.dll" "SolidWorks.Interop.swpublished.dll"

Click OK to add them to the solution ready for use.

ISwAddin Interface

By default Visual Studio will have created a class called Class1 to your project. Rename the filename in from the Solution Explorer to something more apt such as "SWIntegration". This should automatically rename the class name to the same, but if it doesn't rename that also.

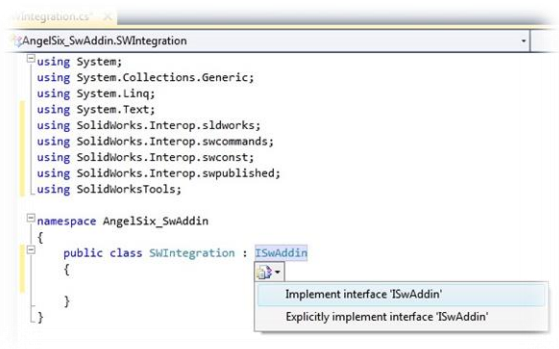
In the class we need to add references to the SolidWorks libraries we added just, so that this class name knows where to find the SolidWorks things. To do this, take a look at the top of the file where it shows entries such as using System; (or Imports System for those in VB). These are namespace includes. Add another few lines below for the SolidWorks namespaces:

```
using SolidWorks.Interop.sldworks;
using SolidWorks.Interop.swcommands;
using SolidWorks.Interop.swconst;
using SolidWorks.Interop.swpublished;
using SolidWorksTools;
```

That now gives us access to the SolidWorks items we will need. This class is going to be our main functional class that will deal with the SolidWorks integration - getting the add-in registered, installed into the registry, connected to the SolidWorks session, and creating our Taskpane. To do this we need to implement the ISwAddin interface in our class. This is done in C# by adding a colon after the class, or in VB by starting a new line below the class name and typing Implements.

Once you have finished typing ISwAddin you will get a small blue line under the name. Hover your cursor over this and click the drop-down button that appears and select Implement Interface 'ISwAddin' to add the required functions needed for the class to become a valid ISwAddin class.

You will now notice 2 new functions called ConnectToSW and DisconnectFromSW. These get called whenever SolidWorks attempts to load and unload the add-in, and it is up to use to decide what we want to do when that happens.



For the DisconnectFromSW function we use this to cleanly shutdown and dispose of our program, but we don't have anything to dispose of as of yet, so just create a blank function called UITeardown and call it within the Disconnect function, returning true.

For the ConnectToSW function, we want to store the SolidWorks instance and cookie ID that are given to us for future use, and then create our Taskpane object. Create 2 new variables in the class:

```
public SldWorks mSWApplication;
private int mSWCookie;
```

And add a new blank function called UISetup where we will place our UI initialization code later.

In the ConnectToSW function let's define our variables, register our add-in (this class) for function callbacks that SolidWorks wants to send to us, and call our currently blank UISetup function:

```
mSWApplication = (SldWorks)ThisSW;
mSWCookie = Cookie;

// Set-up add-in call back info
bool result = mSWApplication.SetAddinCallbackInfo(0, this, Cookie);

this.UISetup();

return true;
```

This is what the code should look like so far, with full Connect and Disconnect functionality defined:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using SolidWorks.Interop.sldworks;
using SolidWorks.Interop.swcommands;
using SolidWorks.Interop.swconst;
using SolidWorks.Interop.swpublished;
using SolidWorksTools;

namespace AngelSix_SwAddin
{
    public class SWIntegration : ISwAddin
    {
        public SldWorks mSWApplication;
        private int mSWCookie;

        public bool ConnectToSW(object ThisSW, int Cookie)
        {
            mSWApplication = (SldWorks)ThisSW;
            mSWCookie = Cookie;

            // Set-up add-in call back info
            bool result = mSWApplication.SetAddinCallbackInfo(0, this, Cookie);

            this.UISetup();

            return true;
        }

        public bool DisconnectFromSW()
        {
            this.UITeardown();
            return true;
        }

        private void UISetup()
        {
        }

        private void UITeardown()
        {
        }
    }
}
```

That's that!

So we have a class that is now a fully legal ISwAddin class and will work with SolidWorks. Before we get to add our Taskpane let us sort out the pesky COM registration. Although you could skip this step and add registry entries manually or loading the add-in from the SolidWorks File->Open menu, let's just get it done properly from the outset.

COM Registration

I won't dwell too much on the technicalities here as it is not really the type of code you need to understand exactly what is going on. At the end of the day once your add-in is COM registered and visible that is the last you will ever deal with it for the rest of your entire product line I can guarantee it. So... start by adding another using statement and this time import System.Runtime.InteropServices, and then in your class place the following code:

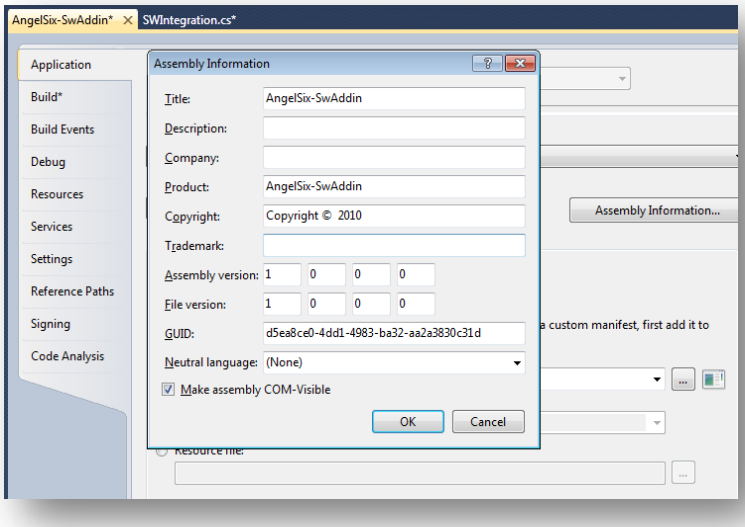
```
[ComRegisterFunction()]
private static void ComRegister(Type t)
{
    string keyPath = String.Format(@"SOFTWARE\SolidWorks\AddIns\{0:b}", t.GUID);

    using (Microsoft.Win32.RegistryKey rk = Microsoft.Win32.Registry.LocalMachine.CreateSubKey(keyPath))
    {
        rk.SetValue(null, 1); // Load at startup
        rk.SetValue("Title", "My SwAddin"); // Title
        rk.SetValue("Description", "All your pixels are belong to us"); // Description
    }
}

[ComUnregisterFunction()]
private static void ComUnregister(Type t)
{
    string keyPath = String.Format(@"SOFTWARE\SolidWorks\AddIns\{0:b}", t.GUID);
    Microsoft.Win32.Registry.LocalMachine.DeleteSubKeyTree(keyPath);
}
```

That's fairly short and very easy to understand. Start with the [] brackets before the function name (or <> in VB). When anything is placed before a definition (of a variable, function or class) inside the square brackets, it is called an Attribute. The ComRegisterFunction and ComUnregisterFunction attributes tell the COM service that those functions should be called when our assembly is attempted to be registered for COM. All we do is to add 3 entries to the correct location in the registry regarding our add-in, where SolidWorks looks when starting up to decide what add-ins to load into its application. When we come to unregister for COM we simply delete these entries.

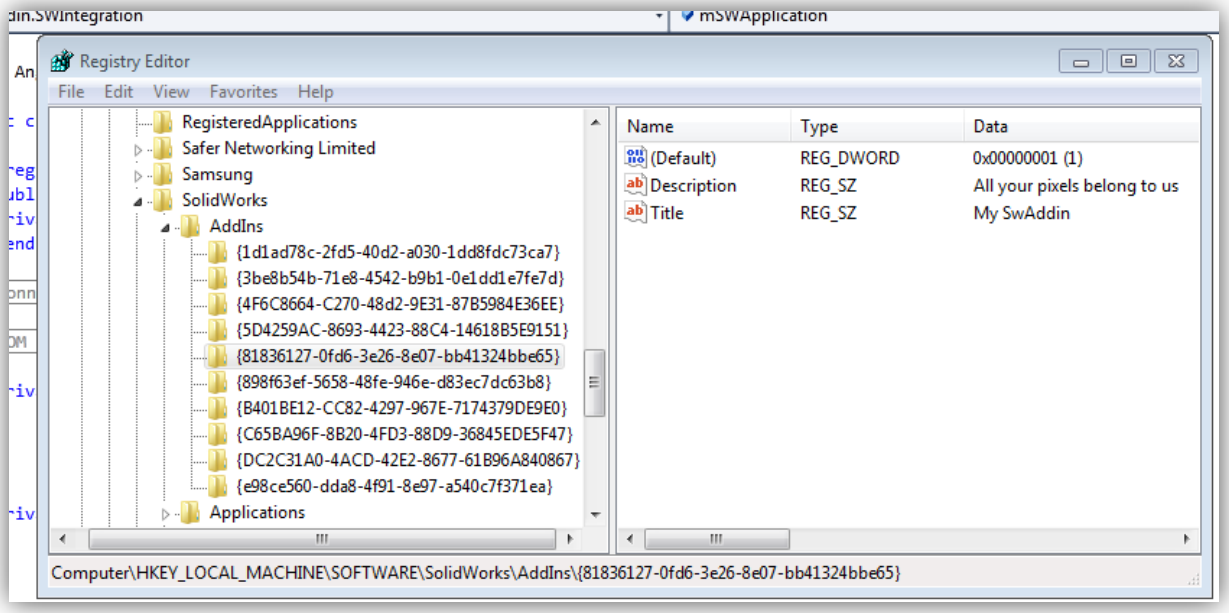
The Type t passed in during COM registration comes from the Operating System and is a unique identifier of an instance of our assembly (program) that the OS has run and made COM visible. By adding this ID to the registry SolidWorks knows how to find and load it.



Now once more we could manually register our assembly for COM by using the regasm tool, but as we are designing our add-in on a development machine with Visual Studio we might as well let Visual Studio handle that for us for now. I will cover installation on client machines and handling COM registration in installers in later tutorials (or just buy my book). To make Visual Studio register our add-in for COM and make it visible every time we build our project we have to enable two options. Right-click on the project in the Solution Explorer and select Properties and then click the Assembly Information button and check Make assembly COM-Visible to cover the first requirement.

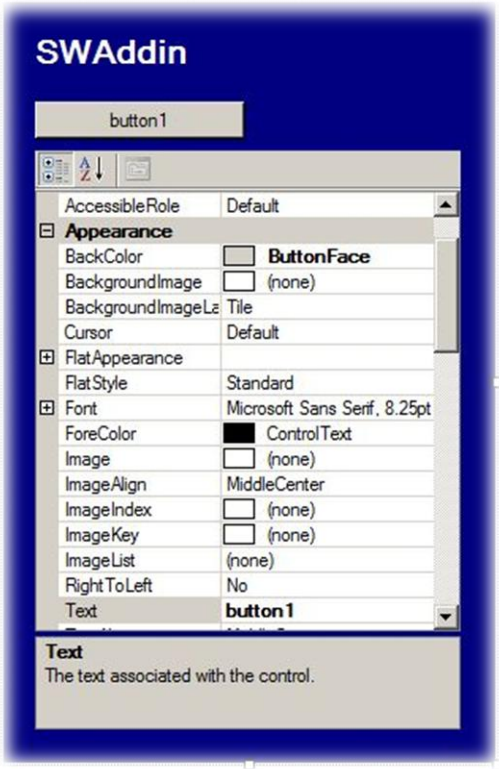
Then go to the Build tab and check Register for COM interop. That's all you need to do to register your assembly for COM.

Test it by building your project now then take a look at the registry (Start->Run... "regedit"). Browse to Local Machine\Software\SolidWorks\Addins and look for your add-in entries.



You could run SolidWorks now and see your add-in in the Add-Ins menu also, but that is all you would see as we have done nothing in our program as of yet, so that's what we will do now!

Creating the Taskpane



To create a Taskpane in SolidWorks we need to create a .Net Control. This control is effectively the control that will become the Taskpane control. Because of this and to save creating multiple Taskpanes every time you want to change the current control (say you had a step-by-step wizard or a control for each type of SolidWorks file), the typical approach is to use this control as more of a host control that you dynamically load the appropriate user control into as and when needed.

Start by creating a normal User Control as you would any other time by right-clicking on the Project in the Solution Explorer and selecting Add->User Control... Call this SWTaskpaneHost and press Add.

This will add a new control to your Solution Explorer and also display a visual form designer for you to edit your control. Because dynamic loading of items and all other topics around form and flow design, I will cover that properly in another tutorial. For now just drag a few Labels and Textbox onto the control so that when it loads in SolidWorks we can identify it and see that it really is our control and is function.

To make this control capable of being added as a SolidWorks Taskpane it needs two things - to be a Visible COM Class, and to have a UID (uniquely identifiable name). Both are done with the use of the Attributes we talked about earlier.

Right-click the SWTaskpaneHost class from the Solution Explorer and select View Code to open up the class code for our control - Add all the same SolidWorks using statements to the top section as well as one for System.Runtime.InteropServices, and then add the following attribute tags above the class name and the variable inside the class:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;
using SolidWorks.Interop.sldworks;
using SolidWorks.Interop.swcommands;
using SolidWorks.Interop.swconst;
using SolidWorks.Interop.swpublished;
using SolidWorksTools;
using System.Runtime.InteropServices;

namespace AngelSix_SwAddin
{
    [ComVisible(true)]
    [ProgId(SWTASKPANE_PROGID)]
    public partial class SWTaskpaneHost : UserControl
    {
        public const string SWTASKPANE_PROGID = "AngelSix.SWTaskPane_SwAddin";

        public SWTaskpaneHost()
        {
            InitializeComponent();
        }
    }
}
```

The first attribute ProgId tags the COM Class with a unique name. The name we give it is the constant variable defined inside the class so we can use the same variable later to create the Taskpane.

The second tag is the COMVisible attribute so it is visible in the COM table.

That is all that is needed to setup our control for SolidWorks. Again similar to the SWIntegration class, now this is setup you are likely never to touch its structure again for a long time, so don't worry too much about understanding every little bit of the COM stuff, it is not really that required as once it works it works and isn't going to break.

All that is left now is to create our Taskpane in the SWIntegration class (which is the actual SolidWorks Add-in and the only one that talks directly to SolidWorks when loading/unloading).

Going back to the SWIntegration class, add two new variables - one for the SolidWorks Taskpane interface object, and one for our actual Taskpane user control:

```
private TaskpaneView mTaskpaneView;
private SWTaskpaneHost mTaskpaneHost;
```

Now in the UISetup function we created earlier (that will get fired when SolidWorks loads our add-in), add the following 2 lines to create and add our .Net User Control as a SolidWorks Taskpane control:

```
mTaskpaneView = mSWApplication.CreateTaskpaneView2(string.Empty, "Woo! My first SwAddin");
mTaskpaneHost = (SWTaskpaneHost)mTaskpaneView.AddControl(SWTaskpaneHost.SWTASKPANE_PROGID, "");
```

The first line creates a new native SolidWorks Taskpane view that we can use to add controls to. This takes in the location of an image to use as the icon (which we just leave blank here for simplicity), and a tooltip description to show up when the user hovers the mouse over the Taskpane.

The second adds our Taskpane COM control to the newly created view by means of supplying it with the unique name we tagged our control with.

That is all that is needed when loading. In order to correctly clean-up once done add the following to the UITeardown function:

```
mTaskpaneHost = null;
mTaskpaneView.DeleteView();
Marshal.ReleaseComObject(mTaskpaneView);
mTaskpaneView = null;
```

```
Note: You will need to add System.Runtime.InteropServices to the using section so that VS knows where the
Marshal object is.

If you get errors regarding missing SolidWorks references when building, delete references and re-add them.
If it still fails, click each SolidWorks reference from the References folder, and right-click->Properties...
then change Copy Local to True.
```

Start up SolidWorks and admire your work.

Download Tutorial Files