Design Patterns applied to Autodesk® Inventor® API

Philippe Leefsma, Philippe.Leefsma@Autodesk.com
Autodesk Inc. Developer Technical Services

CP3423 This class focuses on developing .Net based add-ins for Autodesk Inventor® using the best practices gained from direct hands-on experience of Autodesk software engineers. Design patterns are general, reusable solutions to commonly occurring situations in software design. You will learn how to get the most out of several design patterns specifically studied for the Inventor® API. The class will benefit any programmer developing applications on top of Autodesk Inventor®.

Target Audience

Programmers and Software Developers with intermediate to advanced experience with the Inventor® COM API. Knowledge of Microsoft Visual Studio.Net and experience with .Net languages (C# or VB.Net) is required. Previous experience developing Inventor add-ins is desirable.

Key Learning Objectives

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

- Boost your add-in development by using smart functionalities
- Strengthen stability and efficiency of your application through use of design patterns using well established software architectures
- Save time and effort by taking advantage of those powerful design patterns turned into templates

About the Author

Philippe has a master's degree in Computer Sciences. He carried his studies in Paris at I.S.E.P and in USA, at Colorado School of Mines. He started his career as software engineer for a French company where he participated to the implementation of a simulator for the French Navy combat frigate Horizon.

He joined Autodesk more than four years ago where he works as developer consultant for Autodesk Developer Network. He supports various products APIs such as AutoCAD®, AutoCAD Mechanical®, and Autodesk® Inventor®. He likes to travel, meet developers from around the world to work with them around programming, CAD and manufacturing challenging topics.

During his free time, Philippe enjoys doing sport, especially swimming, running, snowboarding or trekking in mountains. He has recently enrolled in a pilot course and is flying ultra-light planes. He hopes to continue flying as a hobby once he obtains his pilot license.

Introduction

Developing add-ins in general, but especially Inventor ones since it is our focus here, has this in common that you will need most of the time to create a new project and setup the plumbing for couple of common things: typically the registration mechanism, either if it is registry based or using the new registry-free technology in Inventor 2012. If you have written addins previously, you will also most likely want to reuse a few utility functions that handle basic task about the operating system, picture conversion and more depending on what your add-in is doing. In case your add-in provides new functionality, you want it to create at least one command so the end user can access it through the Inventor graphical interface. For that you need to create new elements through the Ribbon bar and potentially implement user interaction mechanism if your command is going to prompt the user for any kind of selection.

In this class I tried to gather the most useful knowledge I acquired over the past few years where I have been supporting developers about Inventor API related questions, writing several add-ins myself and also collecting interesting ideas and techniques collected among my colleagues at Autodesk.

The purpose of this document is to expose some common tasks, reusable code samples and also propose an implementation of a few design patterns, directly applied to the Inventor API and specific to add-in creation.

To maximize reusability and efficiency of this content, I wrapped it all up inside a Visual Studio project template and associated item template. With a slight experience of Inventor add-in programming, a developer should be able, through the use of that project template, to generate from scratch a fully functional Inventor command with less than five minutes of work!

I would like to thank the following people who inspired me for creating part of the content you will find in that class: special thanks to ***Jan Liska***, ***Xiaodong Liang*** and ***Augusto Goncalves***!

The Ribbon Builder

One of the most common and similar tasks that an add-in needs to do is creating user interface elements. For that purpose using an XML-based Ribbon builder presents two obvious advantages:

- Allows you to create or reuse the definition of your Ribbon elements faster than when writing code.

- Generate the Ribbon items dynamically at the moment your add-in is loading. You can define the Ribbon items to be created inside an external XML file which can then be edited, replaced and modified either by the user or some kind of update mechanism provided by your application.

To see the full implementation of the Ribbon builder refer to the "AdnRibbonBuilder.cs" in "Inventor AU Template.zip" or create a new project and look at the code.

The Ribbon builder interprets the following XML tags:

- RibbonTab
- RibbonPanel
- Button
- Separator
- ButtonPopup
- Popup
- ComboBox
- Gallery
- Macro
- SplitButton
- SplitButtonMRU
- TogglePopup

Below is an example of an XML file that will create three elements inside a custom panel named “My Panel”, in the “Tools” Tab of the “Part” Ribbon: first a button, then a separator and a split button (containing three commands, for simplicity we are using native control definitions):

```xml
<?xml version="1.0" encoding="utf-8"?>

<RibbonTab ribbons="Part"
           internalName="id_TabTools"
           displayName="Tools"
           targetTab=""
           insertBefore="false">


    <RibbonPanel internalName="MyCompany.MyAddin.PanelPart"
                 displayName="My Panel"
                 targetPanel=""
                 insertBefore="false">


        <Button internalName="MyCompany.MyAddin.MyCommand"
                useLargeIcon="true"
                isInSlideout="false"
                targetControl=""
                insertBefore="false"/>


        <Separator isInSlideout="false"
                   targetControl="MyCompany.MyAddin.MyCommand"
                   insertBefore="false"/>


        <SplitButton internalName="AppAddInManagerCmd"
                     useLargeIcon="true"
                     isInSlideout="false"
                     targetControl=""
                     insertBefore="false">


            <Button internalName="AppAddInManagerCmd"/>
            <Button internalName="AppVBAEditorCmd"/>
            <Button internalName="AppApplicationOptionsCmd"/>

        </SplitButton>


    </RibbonPanel>


</RibbonTab>
```

The use of the Ribbon builder is quite straightforward, as it exposes only the following three public methods:

```
// Create Ribbon items based on a Stream
// generated from an XML description file
public static void CreateRibbon(Inventor.Application app,
      Type addinType,
      Stream stream)

// Create Ribbon items based on a Stream name
// typically an embedded resource xml file
public static void CreateRibbon(Inventor.Application app,
      Type addinType,
      string resourcename)

// Create Ribbon items based on an existing xml file on the disk
public static void CreateRibbonFromFile(Inventor.Application app,
      Type addinType,
      string xmlfile)
```

You will typically invoke one of those methods in you add-in Activate method. You also need to create your custom commands prior to call the Ribbon Builder, so the internal command names of the control definitions can be found when generating the Ribbon items.

Here is a short example:

```
    public void Activate(Inventor.ApplicationAddInSite addInSiteObject,
                         bool firstTime)
    {

        m_inventorApplication = addInSiteObject.Application;

        Type addinType = this.GetType();

        // Add your commands here
        // ...

        // Only after all commands have been added,
        // load Ribbon UI from customized xml file.
        // Make sure "InternalName" of above commands is matching
        // "internalName" tag described in xml of corresponding command.
        AdnRibbonBuilder.CreateRibbon(
            m_inventorApplication,
           addinType,
           "AddinNamespace.resources.ribbons.xml");
    }
```

Add-in & Inventor Utilities

Common tasks for creating an Inventor add-in, can be added to a general purpose toolbox. That's what the "AdnInventorUtilities" and "WinUtilities" are doing. I am not going into a full description of those utilities here, Please refer to "AdnInventorUtilities.cs" and "WinUtilities.cs" to have a look at the full implementation.

Here are the most relevant pieces:

Unfortunately, the objects in the Inventor API do not all derive from a common base class, so we cannot write an identical piece of code if we need to determine an object's type, at least for a strong typed language like C#.

However every object exposes a "Type" property that returns a value from the enumeration ObjectTypeEnum. Based on this we can use .Net late binding mechanism to write such a method that will return the object type for any object:

```
public static ObjectTypeEnum GetInventorType(Object obj)
{
    try{
        System.Object objType = obj.GetType().InvokeMember(
            "Type",
            System.Reflection.BindingFlags.GetProperty,
            null, obj, null, null, null, null);

        return (ObjectTypeEnum)objType;
    }
    catch{
        //error...not an Inventor API object
        return ObjectTypeEnum.kGenericObject;
    }
}
```

An example of use for the "`AdnInventorUtilities.GetInventorType`" method:

```
ObjectTypeEnum type =
    AdnInventorUtilities.GetInventorType(SelectedEntity);

switch (type)
{
    case ObjectTypeEnum.kFaceObject:
        //...
        break;
    case ObjectTypeEnum.kWorkPlaneObject:
        //...
        break;
    default:
    //...
}
```

An extension of this late-binding approach is to write a method to get any property for an object based on the property name:

```
///////////////////////////////////////////////////////////////
// Use: Late-binded method to get object property based on name
///////////////////////////////////////////////////////////////
public static System.Object GetProperty(System.Object obj,
    string property)
{
    try
    {
        System.Object objType = obj.GetType().InvokeMember(
            property,
            System.Reflection.BindingFlags.GetProperty,
            null,
            obj,
            null,
            null, null, null);

        return objType;
    }
    catch
    {
        return null;
    }
}
```

This can come handy, especially to write more elegant or shorter code. In the example below, there is no need to determine if the ComponentDefinition is an AssemblyComponentDefinition, PartComponentDefinition, WeldmentComponentDefinition, … and perform a cast. We can directly invoke the “Parameters” property on the occurrence.Definition object:

```
AssemblyDocument document = ...;

ComponentOccurrence occurrence =
    document.ComponentDefinition.Occurrences[1];

Parameters parameters = AdnInventorUtilities.GetProperty(
    occurrence.Definition,
    "Parameters") as Parameters;

Parameter param = parameters[1];
```

Here are some further utility methods that are helpful when working with Inventor models or creating add-ins:

```
// Returns face normal at given point
public static UnitVector GetFaceNormal(Face face, Point point)

// Returns face normal at random point on face (used for planar faces)
public static UnitVector GetFaceNormal(Face face)

// Projects point on plane
public static Point ProjectOnPlane(Point point, Plane plane)

// Returns True if running on 64-bit
public static bool Is64BitOperatingSystem()

// Returns OS Version as string and optionaly service pack installed
public static string GetOsVersion(bool checkSP)

// Performs conversion Icon to IPictureDisp
public static stdole.IPictureDisp ToIPictureDisp(Icon icon)

// Performs conversion Bitmap to IPictureDisp
public static stdole.IPictureDisp ToIPictureDisp(Bitmap bmp)
```

Installer functionalities

Utilities can also help with the creation of your install. Inventor 2012 introduced a new registry-free add-in mechanism based on a description file “.addin” that needs to be copied at specific location on the machine.

The add-in dll can be placed anywhere on the directory structure as long as the path of the dll is described correctly in the .addin file. However how to handle this feature for the programmer since you do not know in advance where the user is going to install the add-in?

The answer to that is through the custom logic of your installer class. The sample code below illustrates how to use XML functionalities in order to edit the .addin file once the installer has been run. When installing your add-in, the .addin file will reside in the same directory than the assembly dll, or a know sub-folder. The installer code will then be able to edit that .addin file with the correct path chosen by the user and place the .addin file in the folder(s) watched by Inventor to load add-ins:

```
public string InstallRegistryFree(IDictionary stateSaver)
{
    try
    {
        // Get addin location
        Assembly Asm = Assembly.GetExecutingAssembly();

        FileInfo asmFile = new FileInfo(Asm.Location);

        FileInfo addinFile = null;

        foreach (FileInfo fileInfo in asmFile.Directory.GetFiles())
        {
            if (fileInfo.Extension.ToLower() == ".addin")
            {
                addinFile = fileInfo;
                break;
            }
        }

        if (addinFile == null)
            throw new InstallException();

        XmlDocument xmldoc = new XmlDocument();
        xmldoc.Load(addinFile.FullName);

        XmlNode node = xmldoc.GetElementsByTagName("Assembly")[0];

        if (node == null)
            throw new InstallException();

        node.InnerText = asmFile.FullName;

        string addinFilenameDest = GenerateAddinFileLocation(
            RegFreeMode.VersionIndep,
            string.Empty,
            string.Empty) + addinFile.Name;

        if (File.Exists(addinFilenameDest))
            File.Delete(addinFilenameDest);

        // copy the addin to target folder according to OS type
        // and all users/separate user
        xmldoc.Save(addinFilenameDest);

        addinFile.Delete();

        return addinFilenameDest;
    }
    catch
    {
        throw new InstallException("Error installing .addin file!");
    }
}
```

The interested reader can refer to “AddinInstaller.cs” for the full implementation. In case of a registry-based mechanism, this installer class also provides the ability to handle registration on both 32-bit and 64-bit platforms from a single msi installer.

Inventor Command Design Pattern

Unlike on the user interface side, where a command may appear as a single unit with a button triggering a specific functionality in the application, creating an Inventor command on the API side isn't a single concept: You first need to generate a *ControlDefinition* based on the type of control you want, load an Icon for it and convert it to the supported format, *IPictureDisp* interface, if your command needs to gather user inputs, you are likely to need to display a form and if the user needs to select entities in the model as input for your command, you will have to use *InteractionEvents*. Also you should ensure that commands behave nicely with other commands. For example the command should automatically be terminated if the user cancels, closes the form or a different command is started. You finally need to create a place holder in the Ribbon, so the command is accessible.

The *AdnCommand* interface and its implementation *AdnButtonCommandBase* are trying to gather all those concepts above within a single Command object that can be easily instantiated and manipulated through your add-in. Using the *AdnButtonCommand* item template in your project should allow you to create new commands in no time!

We will describe in more details the *AdnButtonCommand* item template in the next section. We focus here on the *AdnCommand* interface and its definition: We can see it exposes the basic properties that are required by any Inventor command, such as *DisplayName*, *InternalName*, *Description* and so on… It also exposes a static method *AddCommand* that allows a derived class command to be created and place holders for context menu support *OnLinearMarkingMenu* and *OnLinearMarkingMenu*.

```
abstract public class AdnCommand
{
    protected AdnCommand(Inventor.Application Application)

    // Static method to add new command, needs to be called
    // in add-in Activate typically
    public static void AddCommand(AdnCommand command)

    // Public Command Properties

    public ControlDefinition ControlDefinition

    public Application Application

    public abstract string DisplayName

    public abstract string InternalName

    public abstract string ClientId

    public abstract string Description

    public abstract string ToolTipText

    public abstract CommandTypesEnum Classification

    public abstract ButtonDisplayEnum ButtonDisplay
    // Deletes ControlDefinition
    public void Remove()

    // Implements ControlDefinition creation in derived classes
    abstract protected void CreateControl()

    protected virtual void OnRemove()

    // Notifications for context menu
    protected virtual void OnLinearMarkingMenu(
        ObjectsEnumerator SelectedEntities,
        SelectionDeviceEnum SelectionDevice,
        CommandControls LinearMenu,
        NameValueMap AdditionalInfo)

    protected virtual void OnRadialMarkingMenu(
        ObjectsEnumerator SelectedEntities,
        SelectionDeviceEnum SelectionDevice,
        RadialMarkingMenu RadialMenu,
        NameValueMap AdditionalInfo)
}
```

That interface is then partially implemented by another abstract class named *AdnButtonCommandBase*. This class defines the basic behavior of a button command in Inventor and is designed to be derived into a custom client class in your add-in that will implement the full behavior of your button command.

The non-private members of *AdnButtonCommandBase* are exposed below, the full implementation can be seen in *AdnButtonCommandBase.cs* file*:*

```
    abstract public class AdnButtonCommandBase : AdnCommand
    {
        protected AdnButtonCommandBase(Inventor.Application Application)

        public Form CommandForm

        public bool DisplayFormModal

        public AdnInteractionManager InteractionManager

        protected override void CreateControl()

        public virtual string IconName

        protected Icon StandardIcon

        protected Icon LargeIcon

        protected void RegisterCommandForm(Form form, bool modal)

        protected virtual void Terminate()

        abstract protected void OnExecute(NameValueMap context)

        abstract protected void OnHelp(NameValueMap context)
    }
```

Creating a custom Inventor add-in template

The concepts I presented earlier are useful time savers, but wouldn't it be much more powerful if we could directly use them in a new add-in project created from a custom template?

The obvious answer is yes and that's why I spent a bit of time to create a new custom Inventor add-in template that gathers all that knowledge and will set up the framework required in order to easily and quickly create Inventor commands.

The purpose of this section is not to expose in details how to create a project template for Visual Studio, but rather discuss my experience in creating such a template and the particularities of the template in regard of an Inventor add-in project.

For detailed information about creating Visual Studio project and item templates, refer to the Microsoft MSDN documentation on that topic:

http://msdn.microsoft.com/en-us/library/eehb4faa(v=VS.100).aspx

And a very good and comprehensive article form Matt Milner on the subject:

http://msdn.microsoft.com/en-us/magazine/cc188697.aspx

Basically a project template is a regular project that you will then customize with specific keyword parameters that will be replaced by the Visual Studio wizard that runs in the background when a user clicks OK on the New Project and Add New Item dialog boxes.

All templates support parameter substitution to enable replacement of key parameters, such as class names and namespaces, when the template is instantiated.

Template parameters are declared in the format $*parameter*$ and below are the most useful ones I needed to use in the template I created:

- **destinationdirectory**
  The root directory of the project

- **GUID [1-10]**
  A GUID used to replace the project GUID in a project file. You can specify up to 10 unique GUIDs (for example, guid1).

- **itemname**
  The name provided by the user in the Add New Item dialog box.

- **projectname**
  The name provided by the user in the New Project dialog box.

- **safeitemname**
  The name provided by the user in the Add New Item dialog box, with all unsafe characters and spaces removed.

- **safeprojectname**
  The name provided by the user in the New Project dialog box, with all unsafe characters and spaces removed.

- **time**
  The current time when project/ item is created in the format DD/MM/YYYY 00:00:00.

Here is a short example of template parameters use:

```
////////////////////////////////////////////////////////////////////////
// $safeprojectname$ Inventor Add-in
//
// Author: $username$
// Creation date: $time$
//
////////////////////////////////////////////////////////////////////////
namespace $safeprojectname$
{

    //... your code ...

}
```

Those template parameters can be used anywhere in your project, that means in any resource file or inside the project properties.

Below is the content of the template .addin and manifest files I used in my project. When you generate a new project, the wizard will automatically perform parameters substitution and both files will contain appropriate information without need to have a look at them. Of course you may want to modify the settings in the .addin file, such as the SupportedSoftwareVersion, LoadBehavior and so on…

```
<Addin Type="Standard">

  <!--Created for Autodesk Inventor Version 16.0 and higher-->

  <ClassId>{$guid1$}</ClassId>
  <ClientId>{$guid1$}</ClientId>

  <DisplayName>$safeprojectname$</DisplayName>
  <Description>$safeprojectname$</Description>

  <Assembly>$destinationdirectory$\bin\debug\$safeprojectname$.dll</Assembly>

  <SupportedSoftwareVersionGreaterThan>15..
  </SupportedSoftwareVersionGreaterThan>

  <LoadOnStartUp>1</LoadOnStartUp>
  <UserUnloadable>1</UserUnloadable>
  <Hidden>0</Hidden>
  <DataVersion>1</DataVersion>
  <UserInterfaceVersion>1</UserInterfaceVersion>
  <LoadBehavior>0</ LoadBehavior >

</Addin>
```

The add-in manifest file content to be embedded in the compiled dll:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">

  <assemblyIdentity name="$safeprojectname$"
                    version="1.0.0.0">
  </assemblyIdentity>

  <clrClass clsid="{$guid1$}"
            progid="$safeitemname$.StandardAddInServer"
            threadingModel="Both"
            name="$safeprojectname$.StandardAddInServer"
            runtimeVersion="">
  </clrClass>

  <file name="$safeprojectname$.dll" hashalg="SHA1"></file>

</assembly>
```

The post-build steps of the template can also be customized using parameters, so when you compile the add-in, Visual Studio will embed the manifest into the add-in dll and copy the .addin file into the right location. After compiling successfully the add-in you should be able to simply run Inventor and your addin will get loaded.

Customization of my post-build step looks like this:

```
call "%VS90COMNTOOLS%vsvars32"
    mt.exe -manifest "$(ProjectDir)$projectname$.X.manifest" -outputresource:"$(TargetPath)";#2

xcopy /y "$(ProjectDir)Autodesk.$projectname$.Inventor.addin" "$(AppData)\Autodesk\Inventor
2012\Addins\"
```

A small issue I had when generating my template project, probably due to a limitation of the Visual Studio template wizard, was to handle the .addin and manifest file to work properly. Those files need to be named accordingly to the addin project name, so the goal is to use parameters to let the wizard rename them for us.

Here are the names before substitution:

- $projectname$.X.manifest
- Autodesk.$projectname$.Inventor.addin

Unfortunately if you create those files with such a name in your template project, the wizard won't be able to extract and create the template out of it. So the workaround I found is to set a regular name, without parameters, to those files like "Autodesk.InventorAddIn.Inventor.addin" and "InventorAddIn.X.manifest", then let the wizard create the template, unzip the created template and re-set those files with the paramaters.

This approach will require you to edit a bit the template description file "MyTemplate.vstemplate" that the wizard produced. It also provide an opportunity to have a look at what's in there: it is an XML based file that describes the template content and how the wizard needs to rename the project elements. It is not too complex to understand how it works.

Once your template is created, you get a zip file that you can unzip for further editing and then re-zip without the need to run the template creation wizard. You then need to place the template zip into Visual Studio template directory:

- On Windows XP:
    - C:\Documents and Settings\<user>\My Documents\Visual Studio <version>\Templates\ProjectTemplates
    - C:\ Documents and Settings \<user>\My Documents\Visual Studio <version>\Templates\ItemTemplates
- On Windows Vista/7:
    - C:\ Users \<user>\My Documents\Visual Studio <version>\Templates\ProjectTemplates
    - C:\Users\<user>\My Documents\Visual Studio <version>\Templates\ItemTemplates

The modifications I did to the “MyTemplate.vstemplate” file are highlighted below:

```
<VSTemplate Version="2.0.0"
            xmlns="http://schemas.microsoft.com/developer/vstemplate/2005"
            Type="Project">

  <TemplateData>

    <Name>Inventor AU Template</Name>
    <Description>A project for creating an Inventor AddIn</Description>
    <ProjectType>CSharp</ProjectType>
    <ProjectSubType></ProjectSubType>
    <SortOrder>1000</SortOrder>
    <CreateNewFolder>true</CreateNewFolder>
    <DefaultName>InventorAddIn</DefaultName>
    <ProvideDefaultName>true</ProvideDefaultName>
    <LocationField>Enabled</LocationField>
    <EnableLocationBrowseButton>true</EnableLocationBrowseButton>
    <Icon>__TemplateIcon.ico</Icon>

  </TemplateData>

  <TemplateContent>

    <Project TargetFileName="InventorAddIn.csproj"
             File="InventorAddIn.csproj"
             ReplaceParameters="true">

      <ProjectItem ReplaceParameters="true"
                   TargetFileName="$projectname$.X.manifest">
                   InventorAddIn.X.manifest
      </ProjectItem>

      <ProjectItem ReplaceParameters="true"
                   TargetFileName="Autodesk.$projectname$.Inventor.addin">
                   Autodesk.InventorAddIn.Inventor.addin
     </ProjectItem>

<!--Some more items here, automatically generated by VS template assistant-->

    </Project>
  </TemplateContent>
</VSTemplate>
```

AfterClean Step:

The last trick we can add to our template is to customize the “AfterClean” action taken by Visual Studio (“Build” >> “Clean Solution”), so it can automatically delete the .addin file when cleaning the solution, hence uninstalling the add-in. Strangely this feature isn’t available from the user interface, even in the Professional edition, so you would need to edit the .csproj file:

```
 <Target Name="AfterClean">

    <Delete Files="$(AppData)\Autodesk\Inventor
2012\Addins\Autodesk.$projectname$.Inventor.addin" />

</Target>
```

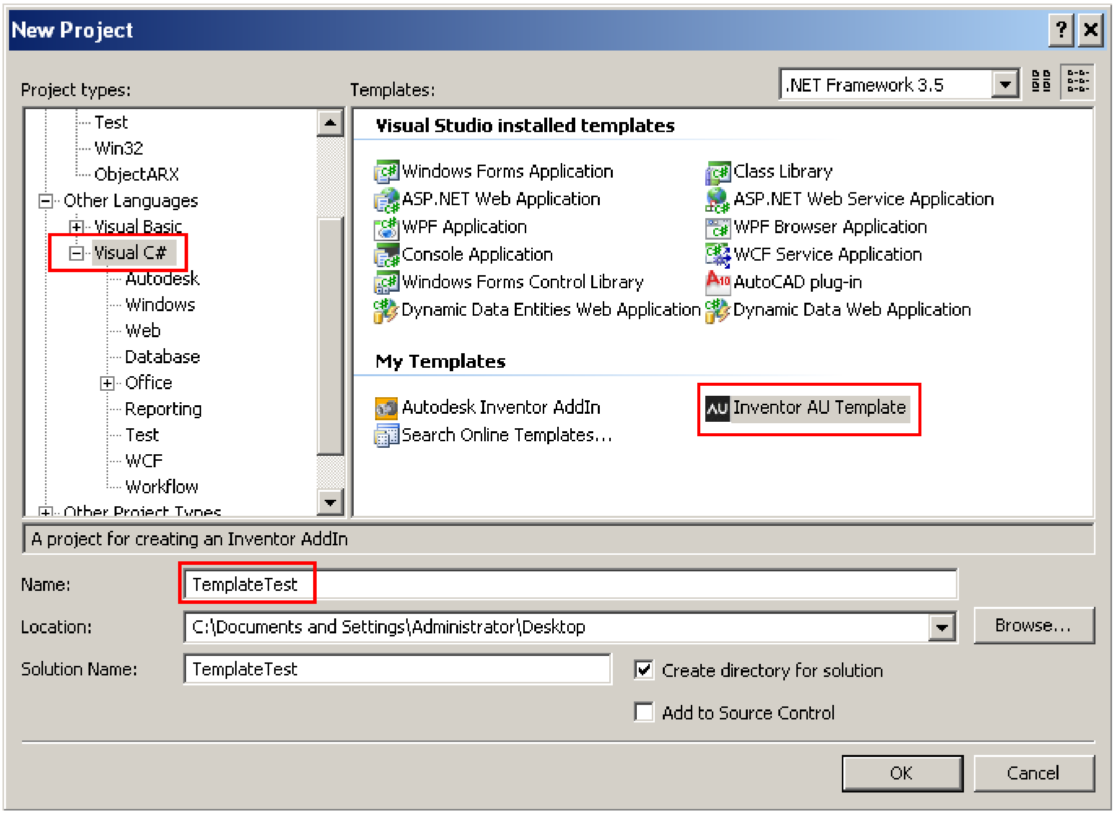
Visual Studio Express Edition:

- The post-build step is not available on Express. You will need to perform this step manually or create a batch file that does the job and run it.
- The AfterClean step that erase the .addin from Inventor folder is not available on Express. Same thing as previously, it needs to be handled manually or by batch file.
- On Express, a new project is created on a TEMP folder, so you need call 'Save All' after creating the project, select a folder, then change the .addin and post-build accordingly.

Five steps to create your Inventor add-in in five minutes

Step #1 - Project Creation

Start by creating a new C# project inside Visual Studio or Visual C# Express, make sure "Inventor AU Template.zip" and "AdnButtonCommand.zip" are placed in your Visual Studio template directories described in the previous section, then select the "Inventor AU Template" project: We will give our add-in a name like "*TemplateTest*" …



A project for creating an Inventor AddIn

Name: TemplateTest

Location: C:\Documents and Settings\Administrator\Desktop

Solution Name: TemplateTest

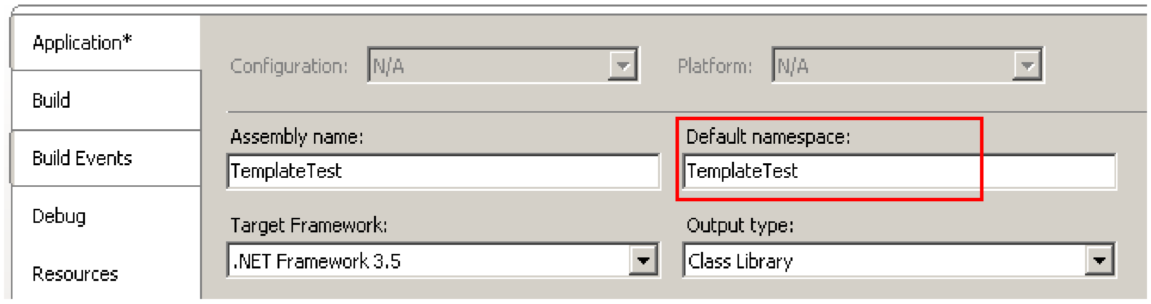
[✓] Create directory for solution

[ ] Add to Source Control

Step #2 – Setting Project properties

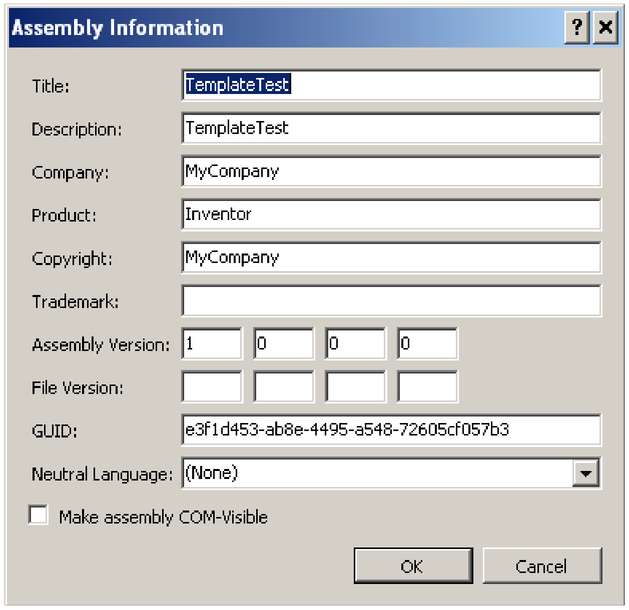
Once your project has been generated from the template, right-click on the project node from the solution explorer and select “Properties”:

The first thing I would recommend is to set the “Default Namespace” property to whatever you want. A limitation of the template mechanism in Visual Studio, is that I couldn’t find a way to customize that name, so I used “InvAddin” by default. Setting that to the same name of the assembly will make thing easier, though it is not mandatory. Here I will set this to “TemplateTest”.



Configuration: N/A
Platform: N/A
Assembly name: TemplateTest
Default namespace: TemplateTest
Target Framework: .NET Framework 3.5
Output type: Class Library

Next thing you may want to do while you are still on that “Application” Tab, is to set the “Assembly Information…” as those got set to some default values:



Title: TemplateTest
Description: TemplateTest
Company: MyCompany
Product: Inventor
Copyright: MyCompany
Trademark:
Assembly Version: 1 0 0 0
File Version:
GUID: e3f1d453-ab8e-4495-a548-72605cf057b3
Neutral Language: (None)
☐ Make assembly COM-Visible

Make sure the assembly is not set to be “COM-Visible”, as it is rarely required. Prefer to set explicitly the “ComVisible” attribute on each type that really requires it. This will let your registry much cleaner over the time if you develop add-ins regularly.

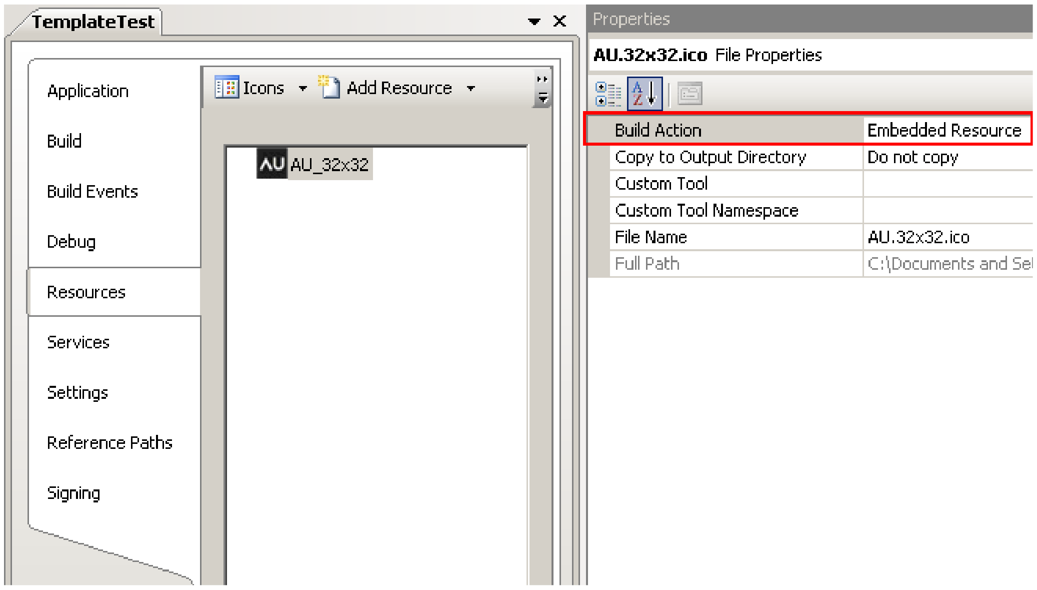
We can then go through the remaining tabs:

On the “Build” tab, “Register for COM Interop” is disabled by default, as the template will use the registry-free mechanism by default. You can enable that feature if you run on 32-bit and need the automatic COM registration. A reminder that on 64-bit you will need to run *RegAsm* manually, through a batch file, or in a post-build step if you want to register the add-in on the machine.

On the “Build Events” tab, you can customize pre/post build steps as needed, but the invocation of mt.exe and copy of the .addin file should be setup without you having to modify anything there.

On the “Debug” tab, you can of course specify the location of your “Inventor.exe” in order to run the application from the debugger. By default standard install path of Inventor 2012 is set.

On the “Resources” tab, we want to add or create an icon (.ico file) that will be used by our add-in command: go to “Add Resource”, then add an existing file or create a new Icon if you have some graphical skills. Remember to set the “Build Action” property of your icon to “Embedded Resource”, it won’t work otherwise…



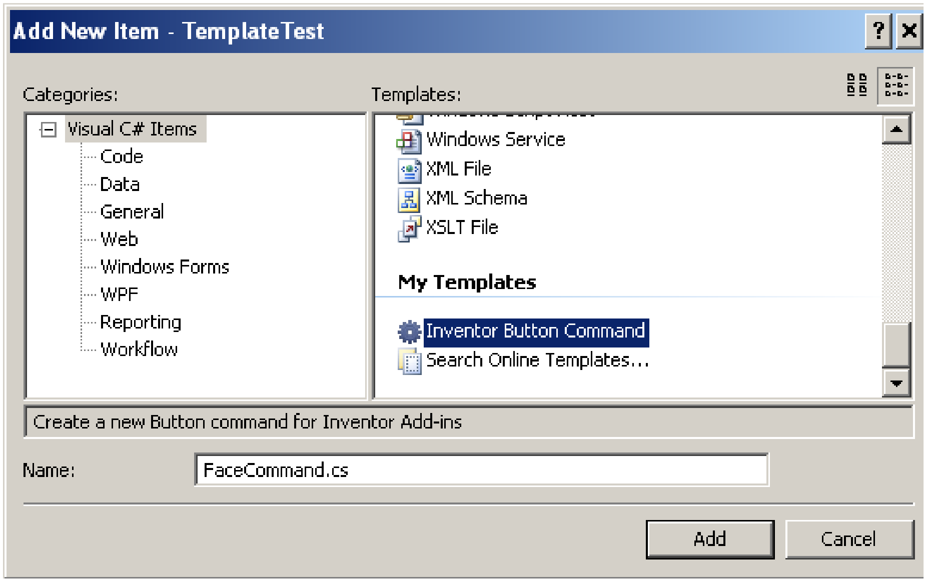
Build Action	Embedded Resource
Copy to Output Directory	Do not copy
Custom Tool	
Custom Tool Namespace	
File Name	AU.32x32.ico
Full Path	C:\Documents and Se

Unless you want to customize further your add-in, that should be it for the project properties, so we can save and close that properties page.

Step #3 – Command creation

We are now going to create a new command and add it to our add-in: Right-click on your project in the solution explorer and choose “Add Item…”, then browse to the *“Inventor Button Command”* item template. Make sure the “AdnButtonCommand.zip” was copied to the Visual Studio Templates folder previously.

We will name our command *FaceCommand* and it will do two things: first it will monitor which model entity is under the cursor and it will display its Inventor type inside a label on our command form. Secondly it will let the user select a Face object and display its area in the command form.



Create a new Button command for Inventor Add-ins

Name: FaceCommand.cs

Once we run the item wizard, it adds a new *FaceCommand.cs* file to our project. Some of the methods of our command class are preconfigured by the wizard, but we want to make sure that we have a look at it and modify them as wished.

Here is my implementation:

I set the internal name of my command to *"Autodesk.TemplateTest.FaceCommand".* Internal name can be anything but it needs to be unique with respect to all existing commands in Inventor, so avoid putting something like "Command1" as it may conflict with other add-in commands. A good practice is to use a naming scheme such as *"MyCompany.Assembly.CommandName".*

I also set the name of my embedded icon I added in the previous step: *"TemplateTest.resources.AU.32x32.ico".* Don't make mistake here, as otherwise your command won't load without a valid resource Icon.

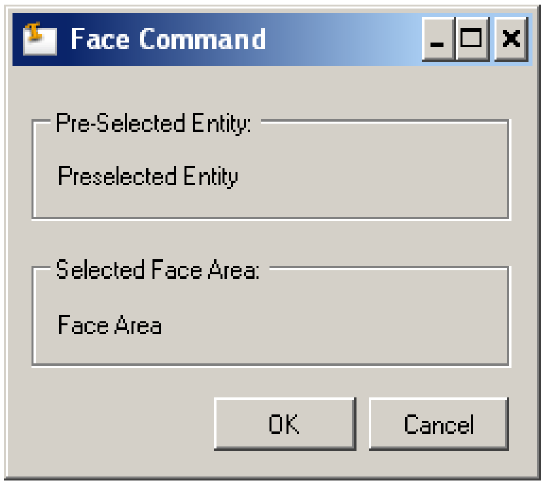
```
public override string InternalName
{
    get
    {
        return "Autodesk.TemplateTest.FaceCommand";
    }
}
```

```
public override string DisplayName
{
    get
    {
        return "FaceCommand";
    }
}

public override string Description
{
    get
    {
        return "FaceCommand Command Description";
    }
}

public override string ToolTipText
{
    get
    {
        return "FaceCommand Command ToolTipText";
    }
}

public override string IconName
{
    get
    {
        return "TemplateTest.resources.AU.32x32.ico";
    }
}
```

Step #4 – Events Implementation

Alright, let’s finish the implementation of our command. I want my command to display a custom Form, so I add a new Windows.Form to my project and I make it look like follow in the designer.

The OK/Cancel buttons do nothing except calling Form.Close(), the closing event of the form will trigger the interaction to be terminated, this is handled automatically by the ButtonCommand framework for you. I also set the “Preselected Entity” and “Face Area” modifier to “Public” (still in the designer) so my command will be able to access and modify those labels.



Only thing left is to implement the *OnExecute* method of my command and you will see that the wizard has kind of pre-filled it. We can register a new instance of our form with *RegisterCommandForm*, then use the *InteractionManager* to connect the events we are interested in, *OnPreSelect* and *OnSelect*. The implementation is quite straightforward and looks as follow:

```
 protected override void OnExecute(NameValueMap context)
{
    // Register command form, so it will be displayed
    // modeless when command is starting
    RegisterCommandForm(new FaceCmdForm(), false);

    InteractionManager.SelectEvents.SingleSelectEnabled = true;

    // Connect events
    InteractionManager.SelectEvents.OnPreSelect +=
        new SelectEventsSink_OnPreSelectEventHandler(SelectEvents_OnPreSelect);

    InteractionManager.SelectEvents.OnSelect +=
        new SelectEventsSink_OnSelectEventHandler(SelectEvents_OnSelect);

    // Run selection
    InteractionManager.Start("Select Face: ");
}

void SelectEvents_OnSelect(ObjectsEnumerator JustSelectedEntities,
    SelectionDeviceEnum SelectionDevice,
    Point ModelPosition,
    Point2d ViewPosition,
    View View)
{
    if (JustSelectedEntities[1] is Face)
    {
        FaceCmdForm cmdForm = CommandForm as FaceCmdForm;

        Face face = JustSelectedEntities[1] as Face;

        cmdForm.lbFaceArea.Text =
            AdnInventorUtilities.GetStringFromAPILength(face.Evaluator.Area);

        cmdForm.Refresh();
    }
}
```

```
    void SelectEvents_OnPreSelect(ref object PreSelectEntity,
        out bool DoHighlight,
        ref ObjectCollection MorePreSelectEntities,
        SelectionDeviceEnum SelectionDevice,
        Point ModelPosition,
        Point2d ViewPosition,
        View View)
    {
        DoHighlight = false;

        ObjectTypeEnum type = AdnInventorUtilities.GetInventorType(PreSelectEntity);

        FaceCmdForm cmdForm = CommandForm as FaceCmdForm;

        cmdForm.lbPresel.Text = type.ToString();

        if (PreSelectEntity is Face)
            DoHighlight = true;
    }
```

Step #5 – Ribbon customization

The last step is to customize our XML ribbon file, so it is creating a new button in the Inventor Interface: to make it easy, we will place our button inside a new panel “My Panel”, in the “Tools” Tab of the “Part” Ribbon.

Make sure the Button *internalName* field is matching what you placed in your command *InternalName* property, in that case *"Autodesk.TemplateTest.FaceCommand".*

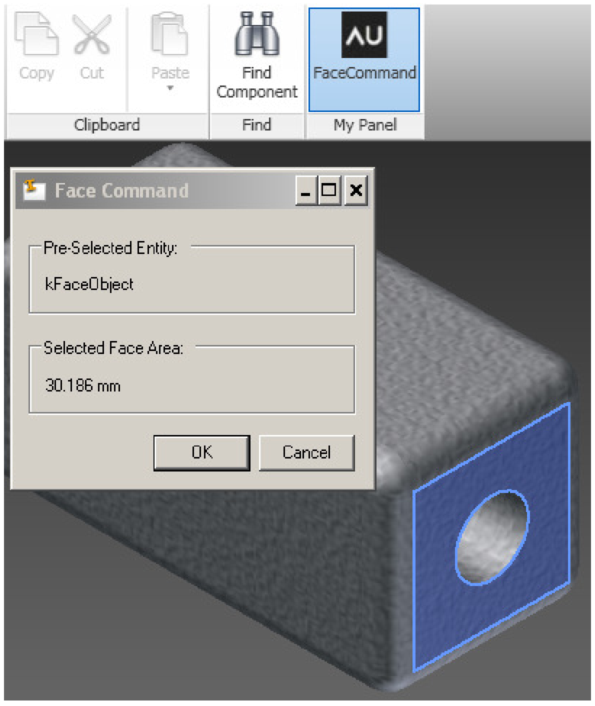
```
<?xml version="1.0" encoding="utf-8"?>

<RibbonTab ribbons="Part"
           internalName="id_TabTools"
           displayName="Tools"
           targetTab=""
           insertBefore="false">

    <RibbonPanel internalName="TemplateTest.PanelPart"
                 displayName="My Panel">

      <Button internalName="Autodesk.TemplateTest.FaceCommand"
              useLargeIcon="true"
              isInSlideout="false"
              targetControl=""
              insertBefore="false"/>

    </RibbonPanel>

</RibbonTab>
```

We need to call our Ribbon Builder *CreateRibbon* method which is already placed by default in the add-in *Activate* method by the wizard. Prior to call the Ribbon Builder, you also need to make sure that the command exists, so the underlying control definition will be found. For achieving this, you simply need to call *AdnCommand.AddCommand* static method, passing a new instance of your command:

```
public void Activate(Inventor.ApplicationAddInSite addInSiteObject,
                     bool firstTime)
{
    m_inventorApplication = addInSiteObject.Application;

    Type addinType = this.GetType();

    AdnInventorUtilities.Initialize(m_inventorApplication, addinType);

    AdnCommand.AddCommand(new FaceCommand(m_inventorApplication));

    AdnRibbonBuilder.CreateRibbon(
        m_inventorApplication,
       addinType,
       "TemplateTest.resources.ribbons.xml");
}
```

Testing

Compile the project and run Inventor, open or create a new part and go the “Tools” Tab, you should see the icon of our command, click on it… it should just work!



Pre-Selected Entity:
kFaceObject

Selected Face Area:
30.186 mm

Additional resources for Inventor developers

- A good starting point for all Inventor developers is the resources listed on the Inventor Developer Center: www.autodesk.com/developinventor.

  These include:

  - Training material, recorded presentations, and Add-ins Wizards.
  - My First Inventor Plug-in, a self-paced tutorial project for newbie's and power users who want to get up and running with Inventor programming: http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=17324828
  - Information on joining the Autodesk Developer Network: www.autodesk.com/joinadn
  - Information on training classes and webcasts: www.autodesk.com/apitraining
  - Links to the Autodesk discussion groups: www.autodesk.com/discussion.

- You will also find there many VB.Net and C# samples included in the Inventor API SDK installation.

- Brian Ekins' blog, the Inventor API main designer: http://modthemachine.typepad.com.

- If you're an ADN partner, there is a wealth of Autodesk API information on the members-only ADN website: http://adn.autodesk.com

  - ADN members can ask unlimited API questions through our DevHelp Online interface

- Watch out for our regular ADN DevLab events. DevLab is a programmers' workshop (free to ADN and non-ADN members) where you can come and discuss your Autodesk programming problems with the ADN DevTech team.

- Advanced resources for creating Visual Studio templates:

  - http://msdn.microsoft.com/en-us/library/eehb4faa(v=VS.100).aspx

  - http://msdn.microsoft.com/en-us/magazine/cc188697.aspx

  - http://blogs.microsoft.co.il/blogs/oshvartz/archive/2008/07/26/creating-visual-studio-items-template-with-custom-wizard.aspx

---

Thank you!

The Author: ***Philippe Leefsma***