Another ATL Tutorial Using Visual C++ .NET

Program examples compiled using Visual Studio/C++ .Net 2003 compiler on Windows XP Pro machine with Service Pack 2. Topics and sub topics for this tutorial are listed below. Don't forget to read Tenouk's small disclaimer. The supplementary note for this tutorial is .NET.

ATL is designed to simplify the process of creating efficient, flexible, lightweight controls. This tutorial leads you through the creation of an ActiveX control, demonstrating many ATL and COM fundamentals. This tutorial does not use attributes.

By following this tutorial, you will learn how to add a control to an ATL project that draws a circle and a filled polygon. You will then add a property to indicate how many sides the polygon will have and create drawing code for updating the control when the property changes. The control will then be displayed on a Web page using some VBScript to make it respond to events.

The tutorial is divided into seven steps. You should perform each step in order as later steps depend on previously completed tasks. Before you begin, you should confirm that you have privileges required to register an ActiveX component on your particular computer. This is usually only a concern if you are running Visual Studio .NET over a Terminal Services connection.

- Step 1: Creating the Project.
- Step 2: Adding a Control to Your Project.
- Step 3. Adding a Property to Your Control.
- Step 4: Changing Your Control's Drawing Code.
- Step 5: Adding an Event.
- Step 6: Adding a Property Page.
- Step 7: Putting Your Control on a Web Page.

Step 1: Creating the Project

This tutorial walks you step-by-step through a non-attributed ATL project that creates an ActiveX object that displays a polygon. The object includes options for allowing the user to change the number of sides making up the polygon, and code to refresh the display.

To create the initial ATL project using the ATL Project Wizard

1. In the Visual Studio.NET development environment, click **New** on the **File** menu, and then click **Project**.
2. Click the **Visual C++ Projects** folder and select **ATL Project**.
3. Type **Polygon** as the project name:

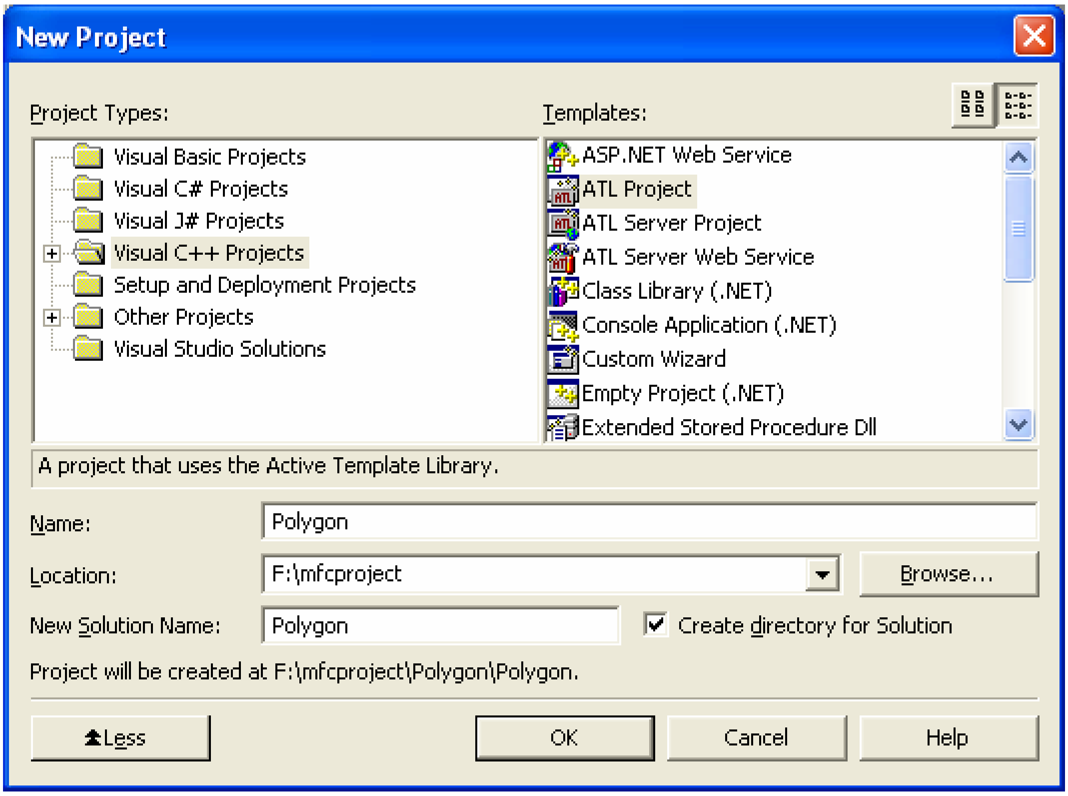


Figure 1: Visual C++ .Net, new ATL project.

The location for the source code will usually default to My Documents\Visual Studio Projects, and a new folder will be created automatically. You can change the directory if needed. This tutorial use F:\mfcproject as a location.

4. Click **OK** and the **ATL Project Wizard** opens.
5. Click **Application Settings** to see the options available:

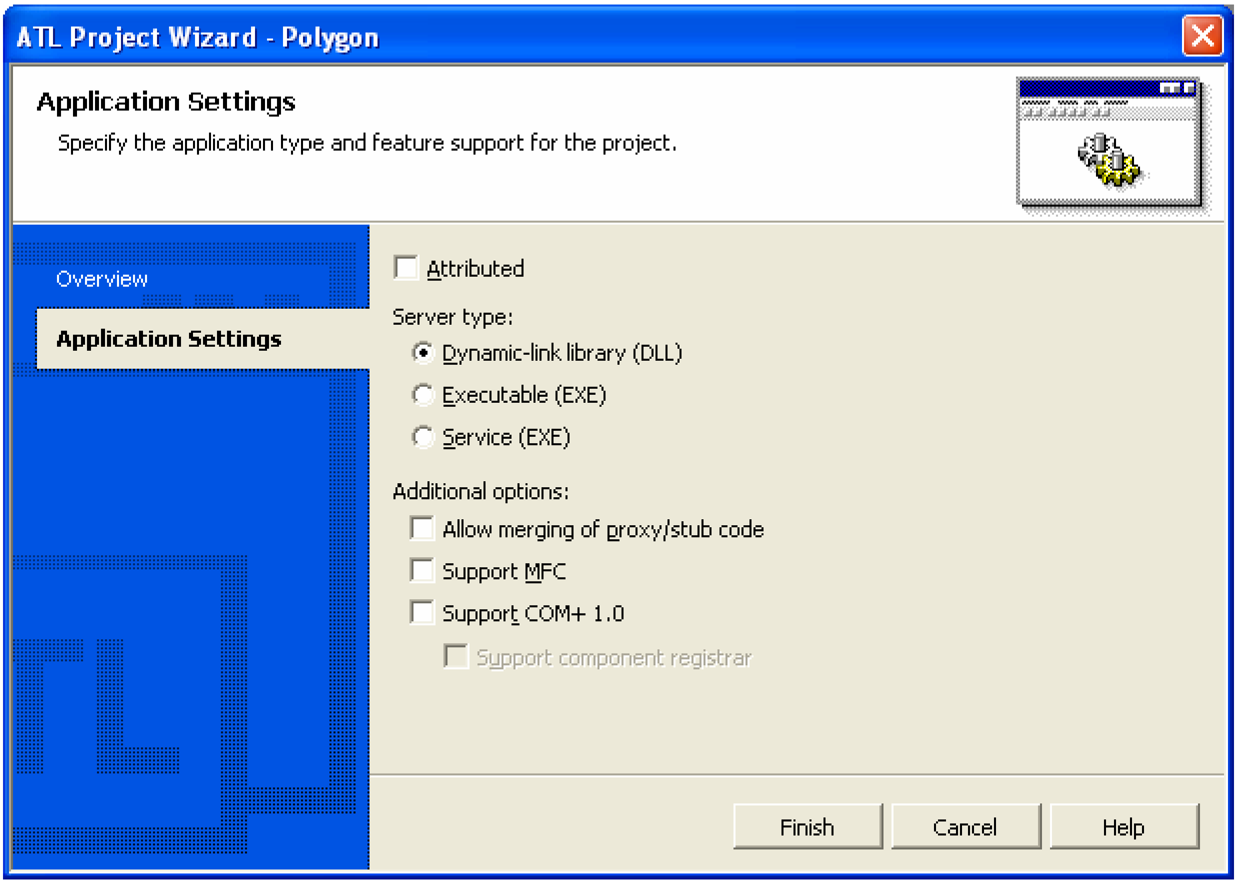


Figure 2: ATL Project Wizard – **Application Settings** page.

6. As you are creating a control, and a control must be an in-process server, leave the **Server type** as a **DLL**.
7. In this tutorial, you will not be using attributes, so ensure that the **Attributed** check box is not selected.
8. Leave the other options at their default values, and click **Finish**.

The **ATL Project Wizard** will create the project by generating several files. You can view these files in **Solution Explorer** by expanding the **Polygon** object. The files are listed below.

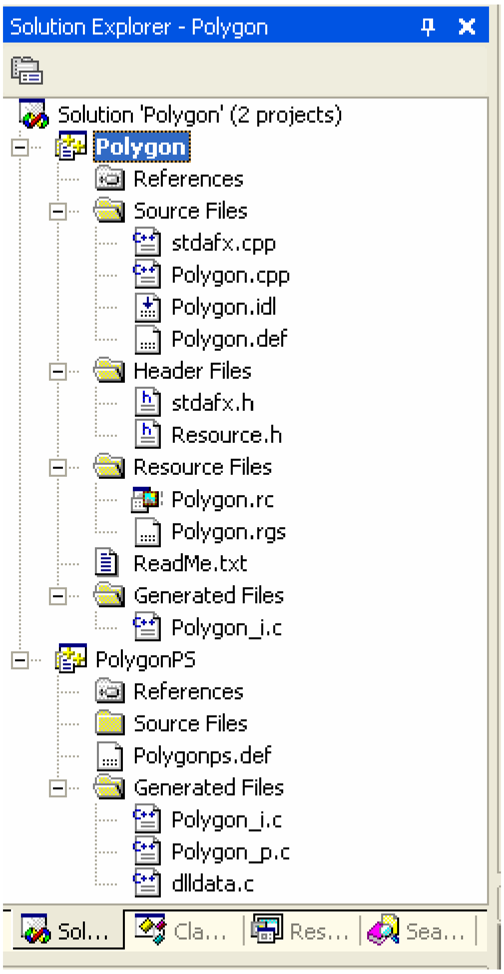


Figure 3: Polygon files generated by Visual C++ .Net wizard seen through **Solution Explorer**.

File	Description
Polygon.cpp	Contains the implementation of `DllMain()`, `DllCanUnloadNow()`, `DllGetClassObject()`, `DllRegisterServer()`, and `DllUnregisterServer()`. Also contains the object map, which is a list of the ATL objects in your project. This is initially blank.
Polygon.def	This module-definition file provides the linker with information about the exports required by your DLL.
Polygon.idl	The interface definition language file, which describes the interfaces specific to your objects.
Polygon.rgs	This registry script contains information for registering your program's DLL.
Polygon.rc	The resource file, which initially contains the version information and a string containing the project name.
Resource.h	The header file for the resource file.
Polygonps.def	This module definition file provides the linker with information about the exports required by the proxy and stub code that support calls across apartments. For details, see COM+ Apartment Models.
stdafx.cpp	The file that will #include the ATL implementation files.
stdafx.h	The file that will #include the ATL header files.

Table 1.

In the next step, you will add a control to your project.

Step 2: Adding a Control

In this step, you will add a control to your project, build it, and test it on a Web page.

To add an object to an ATL project

1. In Class View, right-click the **Polygon** project.
2. Point to **Add** on the shortcut menu, and click **Add Class**.

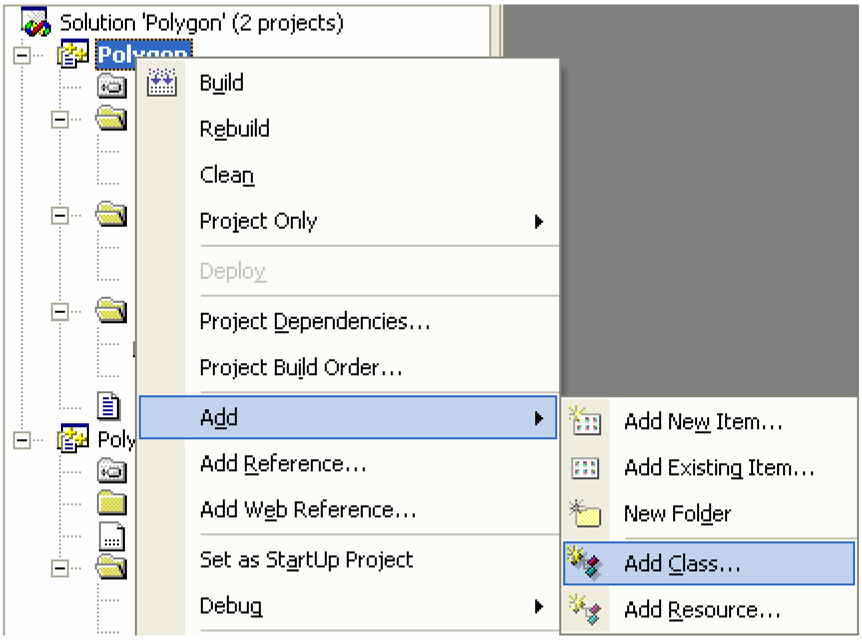


Figure 4: Adding class to the ATL project.

The **Add Class** dialog box appears. The different object categories are listed in the tree structure on the left:

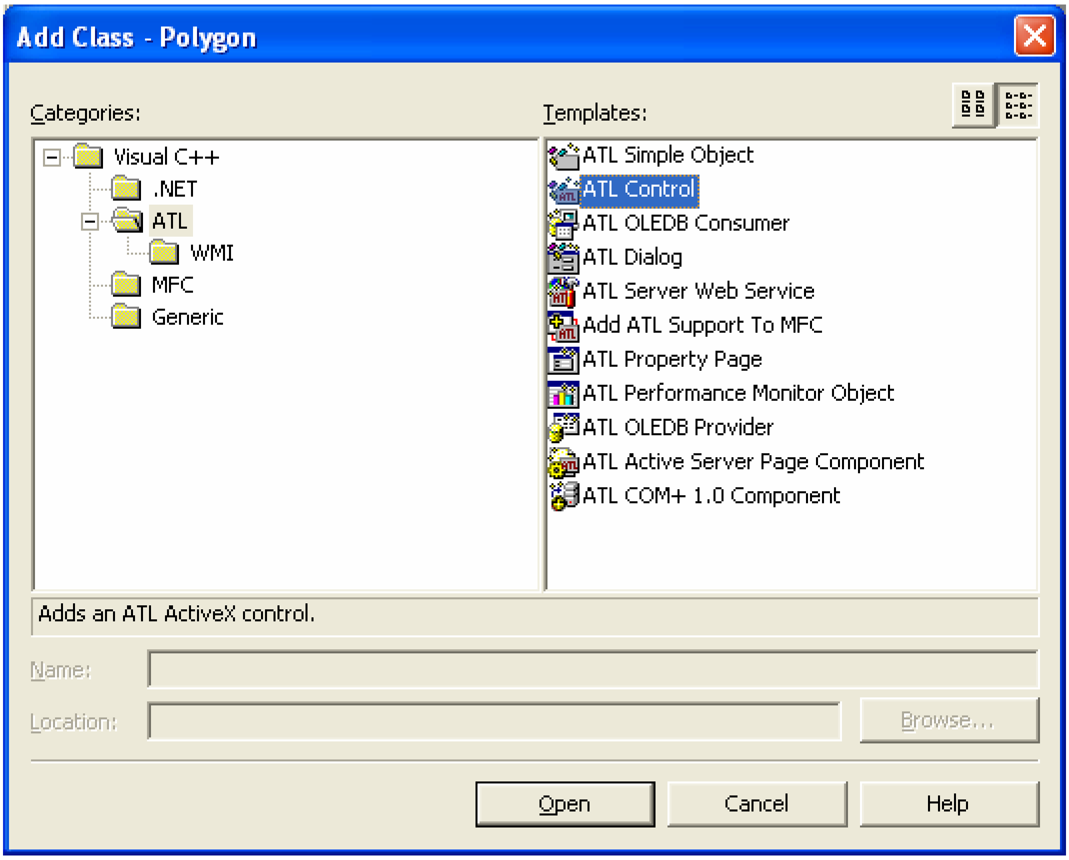


Figure 5: The **Add Class** dialog, adding a new class to Polygon project.

3. Expand the tree structure and click the **ATL** folder.
4. From the list of templates on the right, select **ATL Control**. Click **Open**. The ATL Control Wizard will open, and you can configure the control:

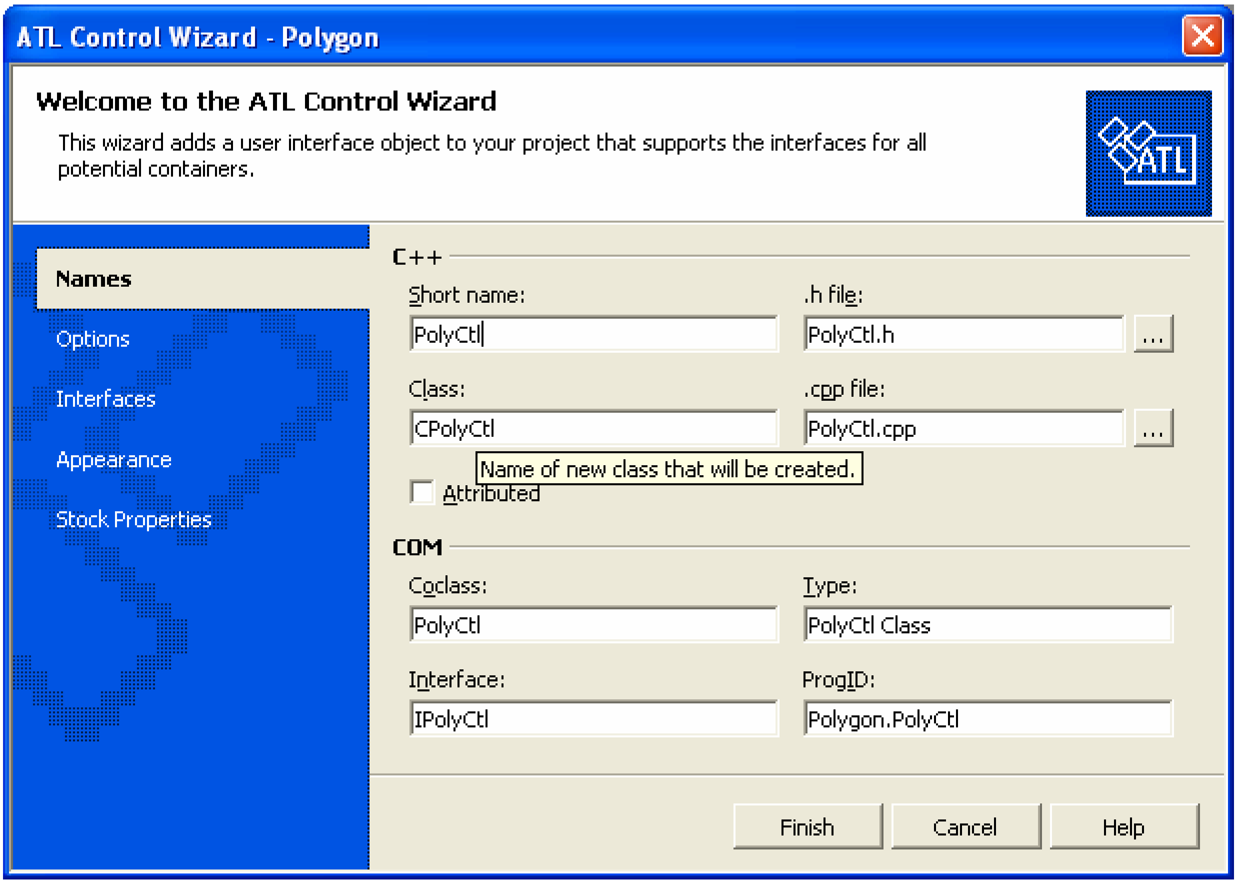


Figure 6: ATL Control Wizard, **Names** page.

5. Type `PolyCtl` as the short name and note that the other fields are automatically completed. Do not click **Finish** yet, because you need to make some changes.

The ATL Control Wizard's **Names** page contains the following fields:

Field	Contents
Short name	The name you entered for the control.
Class	The C++ class name created to implement the control.
.h file	The file created to contain the definition of the C++ class.
.cpp file	The file created to contain the implementation of the C++ class.
CoClass	The name of the component class for this control.
Interface	The name of the interface on which the control will implement its custom methods and properties.
Type	A description for the control.
ProgID	The readable name that can be used to look up the `CLSID` of the control.

Table 2.

You need to make several additional settings in the ATL Control Wizard.

To enable support for rich error information and connection points

1. Click **Options** to open the **Options** page.
2. Select the **Connection points** check box. This will create support for an outgoing interface in the IDL file.

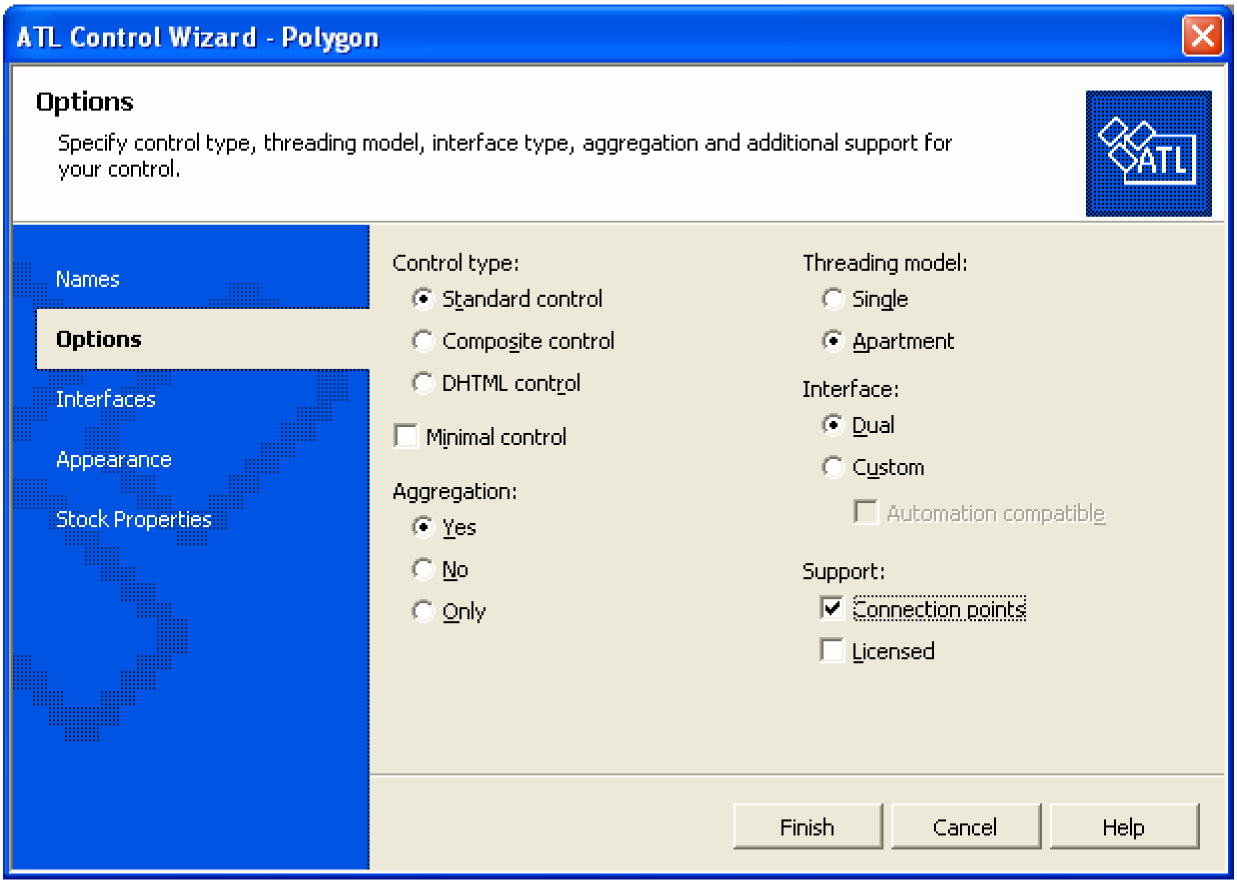


Figure 7: ATL Control Wizard, **Options** page.

You can also make the control insertable, which means it can be embedded into applications that support embedded objects, such as Excel or Word.

To make the control insertable

1. Click **Appearance** to open the **Appearance** page.
2. Select the **Insertable** check box, which by default will be cleared.

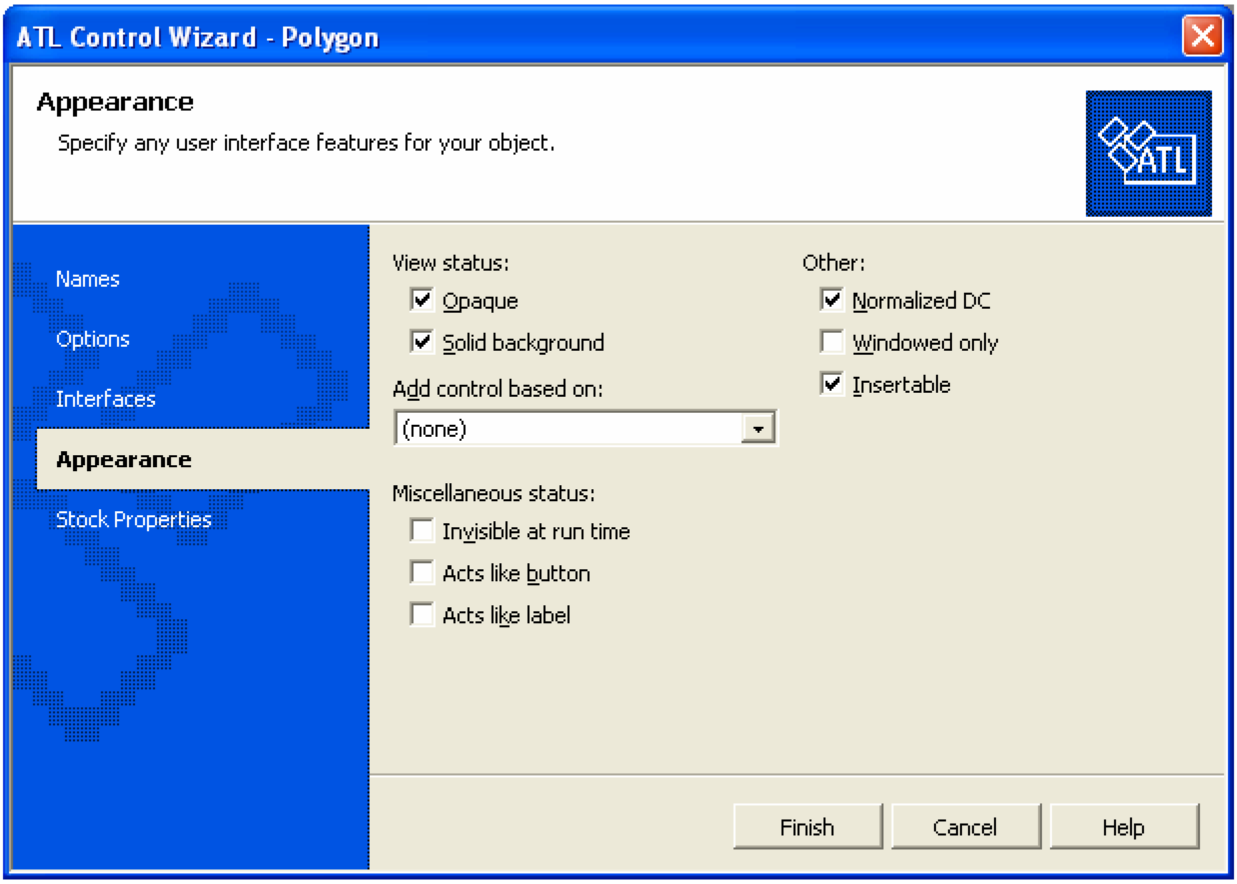


Figure 8: ATL Control Wizard, **Appearance** page.

The polygon displayed by the object will have a solid fill color, so you need to add a **Fill Color** stock property.

To add a Fill Color stock property and create the control

1. Click **Stock Properties** to open the **Stock Properties** page.
2. Under **Not supported**, scroll down the list of possible stock properties. Double-click **Fill Color** to move it to the **Supported** list:

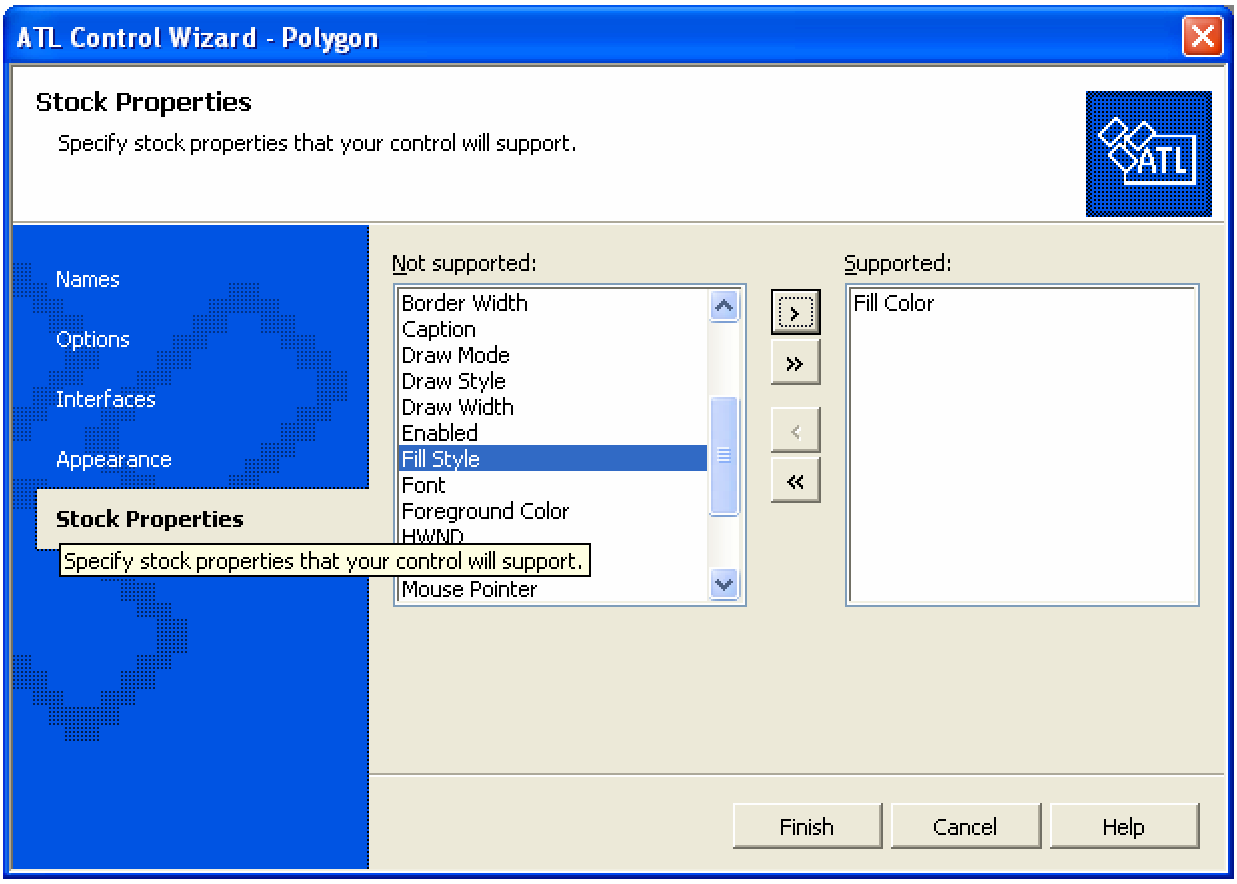


Figure 9: ATL Control Wizard, **Stock Properties** page.

3. This completes the options for the control. Click **Finish**.

As the wizard created the control, several code changes and file additions occurred. The following files were created:

File	Description
PolyCtl.h	Contains most of the implementation of the C++ class `CPolyCtl`.
PolyCtl.cpp	Contains the remaining parts of `CPolyCtl`.
PolyCtl.rgs	A text file that contains the registry script used to register the control.
PolyCtl.htm	A Web page containing a reference to the newly created control.

Table 3.

The wizard also performed the following code changes:

- Added an `#include` statement to the **stdafx.h** and **stdafx.cpp** files to include the ATL files necessary for supporting controls.
- Changed **Polygon.idl** to include details of the new control.
- Added the new control to the object map in **Polygon.cpp**.

Now you can build the control to see it in action.

Building and Testing the Control

To build and test the control

4. On the **Build** menu, click **Build Polygon**.

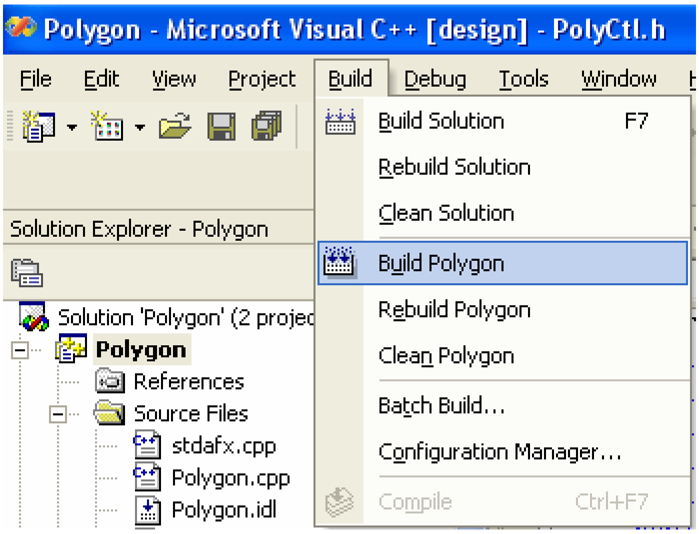


Figure 10: Building Polygon project.

5. Once the control finishes building, double-click **PolyCtl.htm** in Solution Explorer. The HTML Web page containing the control will be displayed. You should see a rectangle and the text **ATL 7.0 : PolyCtl**. This is your control.

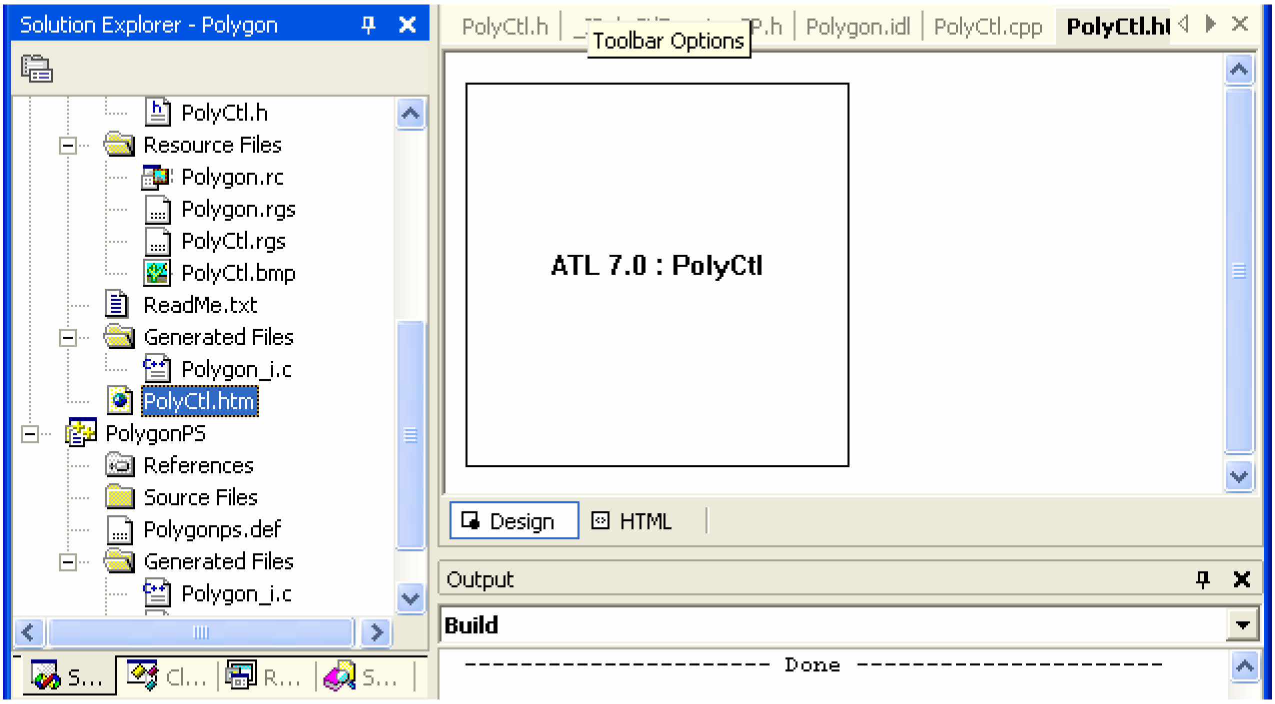


Figure 11: Polygon project output.

Note: When completing this tutorial, if you receive an error message where the DLL file cannot be created, close the **PolyCtl.htm** file and the ActiveX Control Test container and build the solution again. If you still cannot create the DLL, reboot the computer or log off (if you are using Terminal Services).
Next, you will add a custom property to the control.

Step 3: Adding a Property to the Control

`IPolyCtl` is the interface that contains the control's custom methods and properties, and you will add a property to it.

To add a property using the Add Property Wizard

1. Right-click `IPolyCtl` in Class View (expand the **Polygon** branch to find it).
2. On the shortcut menu, click **Add**, and then click **Add Property**. The **Add Property Wizard** will appear.

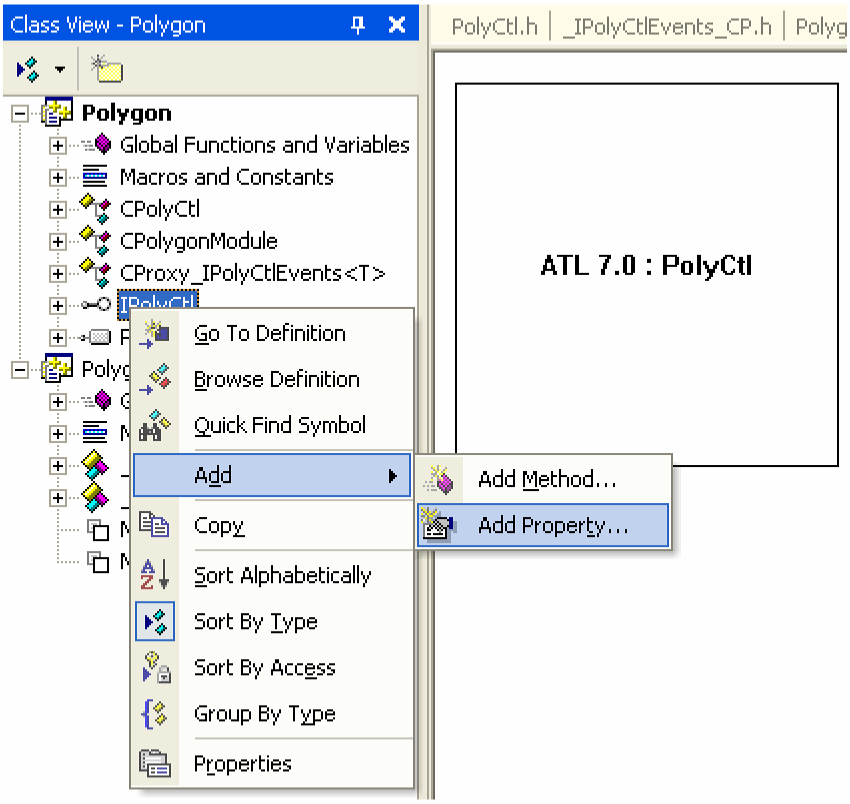


Figure 12: Adding property to project.

3. In the drop-down list of property types, select `SHORT`.
4. Type **Sides** as the **Property** name:

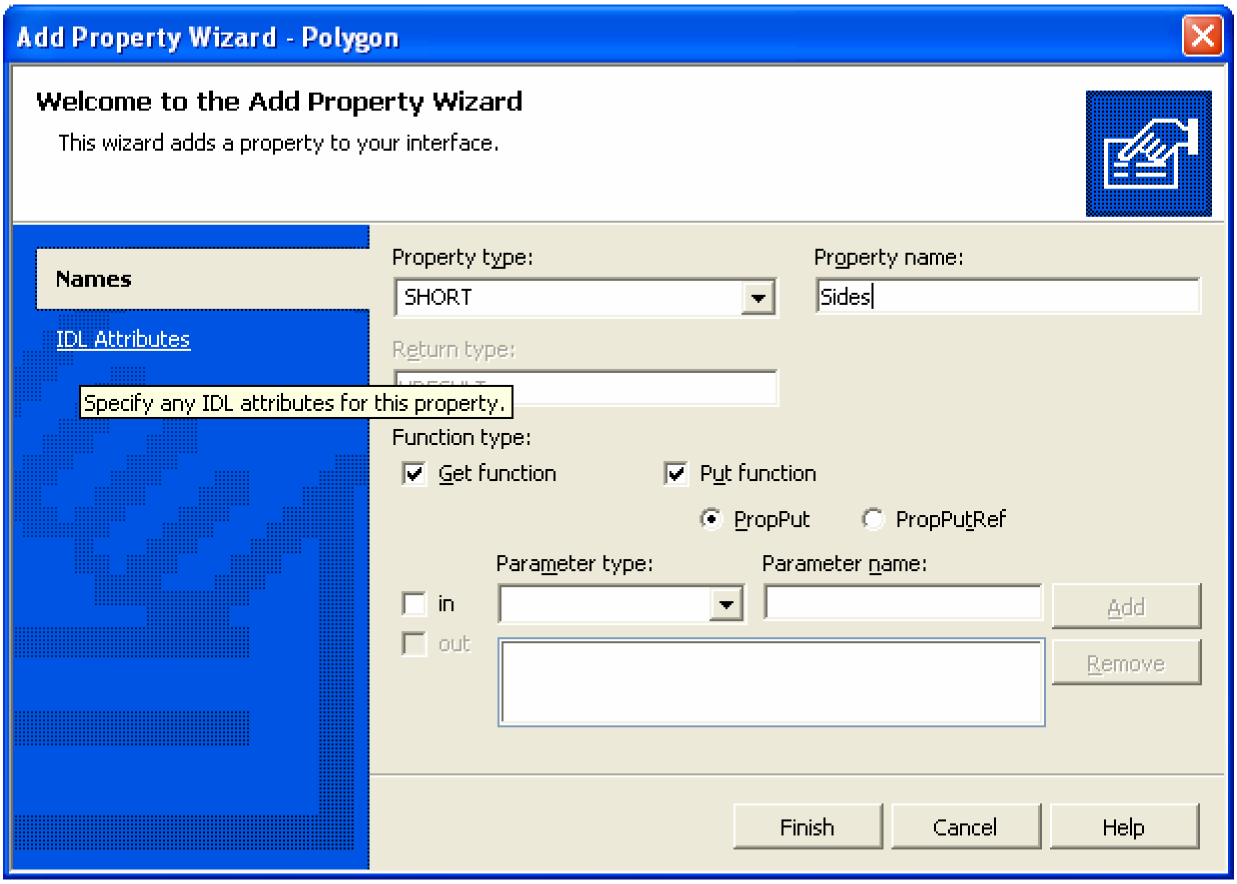


Figure 13: Add Property Wizard, **Names** page.

5. Click **Finish** to finish adding the property.

When you add the property to the interface, MIDL (the program that compiles **.idl** files) defines a `Get` method for retrieving its value and a `Put` method for setting a new value. The methods are named by prepending `put_` and `get_` to the property name.

The **Add Property Wizard** adds the necessary lines to the **.idl** file. It also adds the `Get` and `Put` function prototypes to the class definition in **PolyCtl.h** and adds an empty implementation to **PolyCtl.cpp**. You can check this by opening **PolyCtl.cpp** and looking for the functions `get_Sides()` and `put_Sides()`.

Although you now have skeleton functions to set and retrieve the property, it needs a place to be stored. You will create a variable to store the property and update the functions accordingly.

To create a variable to store the property, and update the put and get methods

1. From **Solution Explorer**, open **PolyCtl.h** and add the following line at the end of the class definition, after the definition of `m_clrFillColor`:

```
short m_nSides;
```

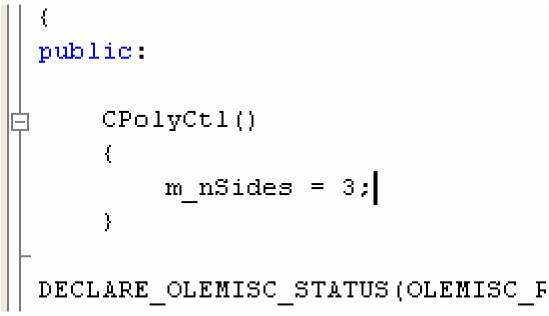
```
}

OLE_COLOR m_clrFillColor;
short m_nSides;
void OnFillColorChanged()
{
    ATLTRACE(_T("OnFillColorChanged\n"));
}
```

Listing 1.

2. Set the default value of `m_nSides`. Make the default shape a triangle by adding a line to the constructor in **PolyCtl.h**:

```
CPolyCtl()
{
   m_nSides = 3;
}
```



```
{
public:

    CPolyCtl()
    {
        m_nSides = 3;
    }

DECLARE_OLEMISC_STATUS(OLEMISC_F
```

Listing 2.

3. Implement the `Get` and `Put` methods. The `get_Sides()` and `put_Sides()` function declarations have been added to **PolyCtl.h**. Add the following code to **PolyCtl.cpp** to complete both methods:

```
STDMETHODIMP CPolyCtl::get_Sides(short *pVal)
{
   *pVal = m_nSides;
   return S_OK;
}
STDMETHODIMP CPolyCtl::put_Sides(short newVal)
{
   if (newVal > 2 && newVal < 101)
   {
      m_nSides = newVal;
      return S_OK;
   }
   else
     return Error(_T("Shape must have between 3 and 100 sides"));
}
```

```
STDMETHODIMP CPolyCtl::get_Sides(SHORT* pVal)
{
    // TODO: Add your implementation code here
    *pVal = m_nSides;
    return S_OK;
}

STDMETHODIMP CPolyCtl::put_Sides(SHORT newVal)
{
    // TODO: Add your implementation code here
    if (newVal > 2 && newVal < 101)
    {
        m_nSides = newVal;
        return S_OK;
    }
    else
        return Error(_T("Shape must have between 3 and 100 sides"));
}
```

Listing 3.

The `get_Sides()` method returns the current value of the **Sides** property through the `pVal` pointer. In the `put_Sides()` method, the code ensures the user is setting the **Sides** property to an acceptable value. The minimum must be 2, and because an array of points will be used for each side, 100 is a reasonable limit for a maximum value.
You now have a property called **Sides**. In the next step, you will change the drawing code to use it.

Step 4: Changing the Drawing Code

By default, the control's drawing code displays a square and the text **ATL 7.0 : PolyCtl**. In this step, you will change the code to display something more interesting. The following tasks are involved:

- Modifying the **Header File**.
- Modifying the `OnDraw()` Function.
- Adding a **Method** to **Calculate the Polygon Points**.
- Initializing the **Fill Color**.

Modifying the Header File

Start by adding support for the math functions `sin` and `cos`, which will be used calculate the polygon points, and by creating an array to store positions.

To modify the header file

4. Add the line `#include <math.h>` to the top of **PolyCtl.h**:

```
#include <math.h>
#include "resource.h"     // main symbols
```

```
// PolyCtl.h : Declaration of the CPolyCtl
#pragma once
#include <math.h>
#include "resource.h"       // main symbols
#include <atlctl.h>
#include "Polygon.h"
#include "_IPolyCtlEvents_CP.h"
```

Listing 4.

5. Once the polygon points are calculated, they will be stored in an array of type `POINT`, so add the array to the end of the class definition in **PolyCtl.h**:

```
OLE_COLOR m_clrFillColor;
short m_nSides;
POINT m_arrPoint[100];
```

```
OLE_COLOR m_clrFillColor;
short m_nSides;
POINT m_arrPoint[100];
void OnFillColorChanged()
{
    ATLTRACE(_T("OnFillColorChanged\n"));
}
```

Listing 5.

Modifying the `OnDraw()` Method

Now you should modify the `OnDraw()` method in **PolyCtl.h**. The code you will add creates a new pen and brush with which to draw your polygon, and then calls the `Ellipse()` and `Polygon()` **Win32 API** functions to perform the actual drawing.

To modify the `OnDraw()` function

1. Replace the existing `OnDraw()` method in **PolyCtl.h** with the following code:

```
HRESULT CPolyCtl::OnDraw(ATL_DRAWINFO& di)
{
   RECT& rc = *(RECT*)di.prcBounds;
   HDC hdc  = di.hdcDraw;

   COLORREF    colFore;
   HBRUSH      hOldBrush, hBrush;
   HPEN        hOldPen, hPen;

   // Translate m_colFore into a COLORREF type
   OleTranslateColor(m_clrFillColor, NULL, &colFore);

   // Create and select the colors to draw the circle
   hPen = (HPEN)GetStockObject(BLACK_PEN);
   hOldPen = (HPEN)SelectObject(hdc, hPen);
   hBrush = (HBRUSH)GetStockObject(WHITE_BRUSH);
   hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);

   Ellipse(hdc, rc.left, rc.top, rc.right, rc.bottom);

   // Create and select the brush that
   // will be used to fill the polygon
   hBrush    = CreateSolidBrush(colFore);
   SelectObject(hdc, hBrush);

   CalcPoints(rc);
   Polygon(hdc, &m_arrPoint[0], m_nSides);

   // Select back the old pen and
   // brush and delete the brush we created
```

```
    SelectObject(hdc, hOldPen);
    SelectObject(hdc, hOldBrush);
    DeleteObject(hBrush);

    return S_OK;
}
```

```
// IPolyCtl
public:
        HRESULT OnDraw(ATL_DRAWINFO& di)
        {
            RECT& rc = *(RECT*)di.prcBounds;
            HDC hdc  = di.hdcDraw;

            COLORREF    colFore;
            HBRUSH      hOldBrush, hBrush;
            HPEN        hOldPen, hPen;

            // Translate m_colFore into a COLORREF type
            OleTranslateColor(m_clrFillColor, NULL, &colFore);

            // Create and select the colors to draw the circle
            hPen = (HPEN)GetStockObject(BLACK_PEN);
            hOldPen = (HPEN)SelectObject(hdc, hPen);
            hBrush = (HBRUSH)GetStockObject(WHITE_BRUSH);
```

Listing 6.

Adding a Method to Calculate the Polygon Points

Add a method, called `CalcPoints()`, that will calculate the coordinates of the points that make up the perimeter of the polygon. These calculations will be based on the RECT variable that is passed into the function.

To add the `CalcPoints()` method

1. Add the declaration of `CalcPoints()` to the `IPolyCtl` public section of the `CPolyCtl` class in **PolyCtl.h**:

   ```
   void CalcPoints(const RECT& rc);
   ```

   The last part of the public section of the `CPolyCtl` class will look like this:

   ```
   void FinalRelease()
   {
   }
   STDMETHOD(get_Sides)(short* pVal);
   STDMETHOD(put_Sides)(short newVal);
   void CalcPoints(const RECT& rc);
   ```

```
    void FinalRelease()
    {
    }
    STDMETHOD(get_Sides)(SHORT* pVal);
    STDMETHOD(put_Sides)(SHORT newVal);
    void CalcPoints(const RECT& rc);
};
```

Listing 7.

2. Add this implementation of the `CalcPoints()` function to the end of **PolyCtl.cpp**:

```
void CPolyCtl::CalcPoints(const RECT& rc)
{
   const double pi = 3.14159265358979;
   POINT   ptCenter;
   double  dblRadiusx = (rc.right - rc.left) / 2;
   double  dblRadiusy = (rc.bottom - rc.top) / 2;
   double  dblAngle = 3 * pi / 2;          // Start at the top
   double  dblDiff  = 2 * pi / m_nSides;   // Angle each side will make
   ptCenter.x = (rc.left + rc.right) / 2;
   ptCenter.y = (rc.top + rc.bottom) / 2;

   // Calculate the points for each side
   for (int i = 0; i < m_nSides; i++)
   {
      m_arrPoint[i].x = (long)(dblRadiusx * cos(dblAngle) + ptCenter.x
+ 0.5);
      m_arrPoint[i].y = (long)(dblRadiusy * sin(dblAngle) + ptCenter.y
+ 0.5);
      dblAngle += dblDiff;
   }
}
```


```
void CPolyCtl::CalcPoints(const RECT& rc)
{
   const double pi = 3.14159265358979;
   POINT   ptCenter;
   double  dblRadiusx = (rc.right - rc.left) / 2;
   double  dblRadiusy = (rc.bottom - rc.top) / 2;
   double  dblAngle = 3 * pi / 2;          // Start at the top
   double  dblDiff  = 2 * pi / m_nSides;   // Angle each side will make
   ptCenter.x = (rc.left + rc.right) / 2;
   ptCenter.y = (rc.top + rc.bottom) / 2;

   // Calculate the points for each side
   for (int i = 0; i < m_nSides; i++)
   {
      m_arrPoint[i].x = (long)(dblRadiusx * cos(dblAngle) + ptCenter.x + 0.5);
      m_arrPoint[i].y = (long)(dblRadiusy * sin(dblAngle) + ptCenter.y + 0.5);
      dblAngle += dblDiff;
   }
}
```


Listing 8.

Initializing the Fill Color

Initialize `m_clrFillColor` with a default color.

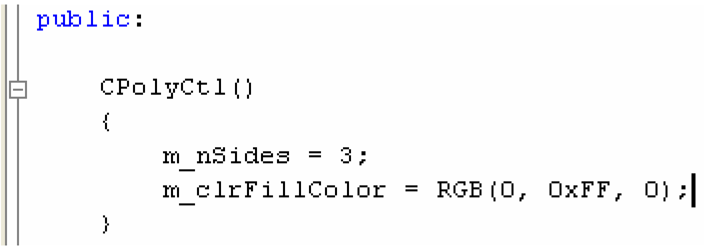
To initialize the fill color

1. Use green as the default color by adding this line to the `CPolyCtl` constructor in **PolyCtl.h**:

```
m_clrFillColor = RGB(0, 0xFF, 0);
```

The constructor now looks like this:

```
CPolyCtl()
{
   m_nSides = 3;
   m_clrFillColor = RGB(0, 0xFF, 0);
}
```



Listing 9.

Building and Testing the Control

Rebuild the control. Make sure the **PolyCtl.htm** file is closed if it is still open, and then click **Build Polygon** on the **Build** menu. You could view the control once again from the **PolyCtl.htm** page, but this time use the **ActiveX Control Test Container**. If you fail to rebuild Polygon, save your solution, close Visual Studio, then delete Debug directory under the Polygon directory. Reopen Visual Studio and Polygon solution. Rebuild Polygon.

To use the ActiveX Control Test Container

1. On the **Tools** menu, click **ActiveX Control Test Container**.

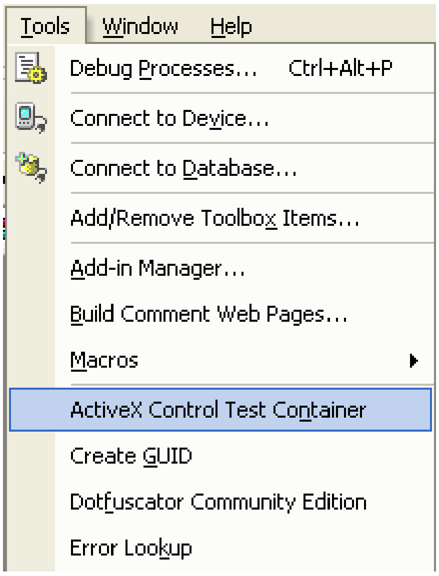


Figure 14: Testing ATL object in **ActiveX Control Test Container**.

2. In **Test Container**, on the **Edit** menu, click **Insert New Control**.

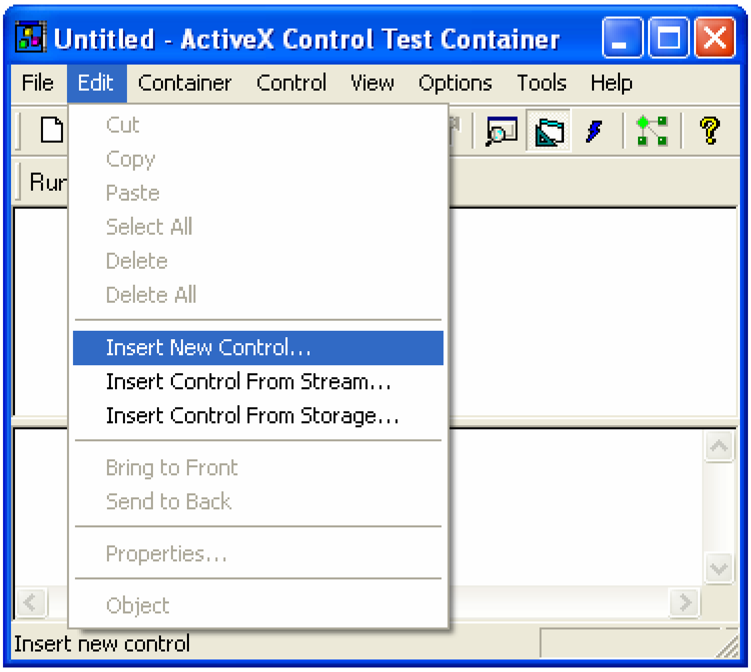


Figure 15: Inserting new control, Polygon to **ActiveX Control Test Container**.

3. Locate your control, which will be called **PolyCtl Class**, and click **OK**. You will see a green triangle within a circle.

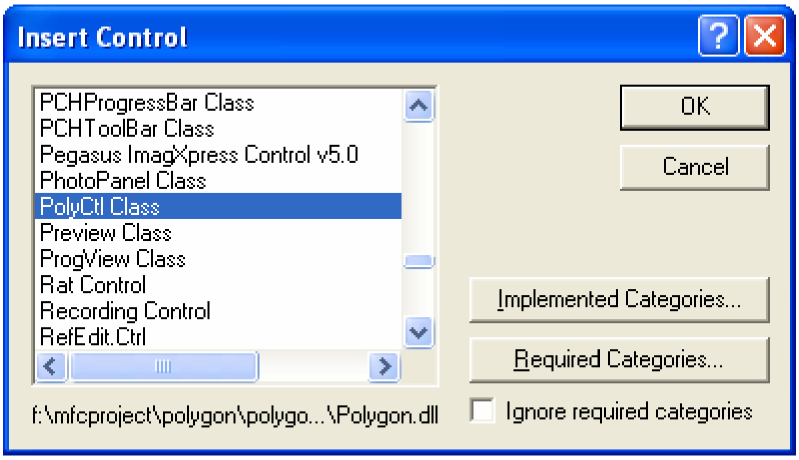


Figure 16: Selecting `PolyCtl`, an ATL control.

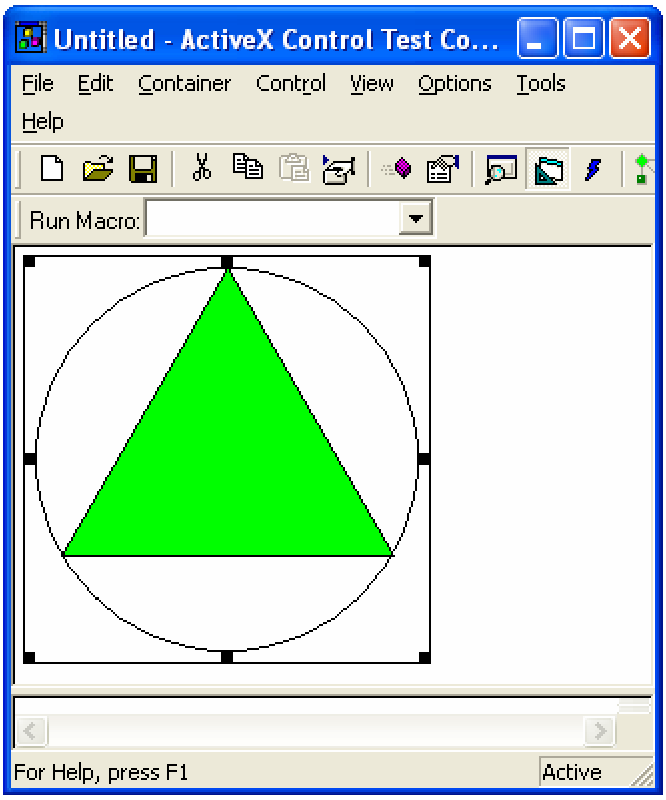


Figure 17: Polygon, an ATL control in action.

Try changing the number of sides by following the next procedure. To modify properties on a dual interface from within **Test Container**, use `Invoke()` Methods.

To modify a control's property from within the Test Container

1. In **Test Container**, click **Invoke Methods** on the **Control** menu. The **Invoke Method** dialog box is displayed:

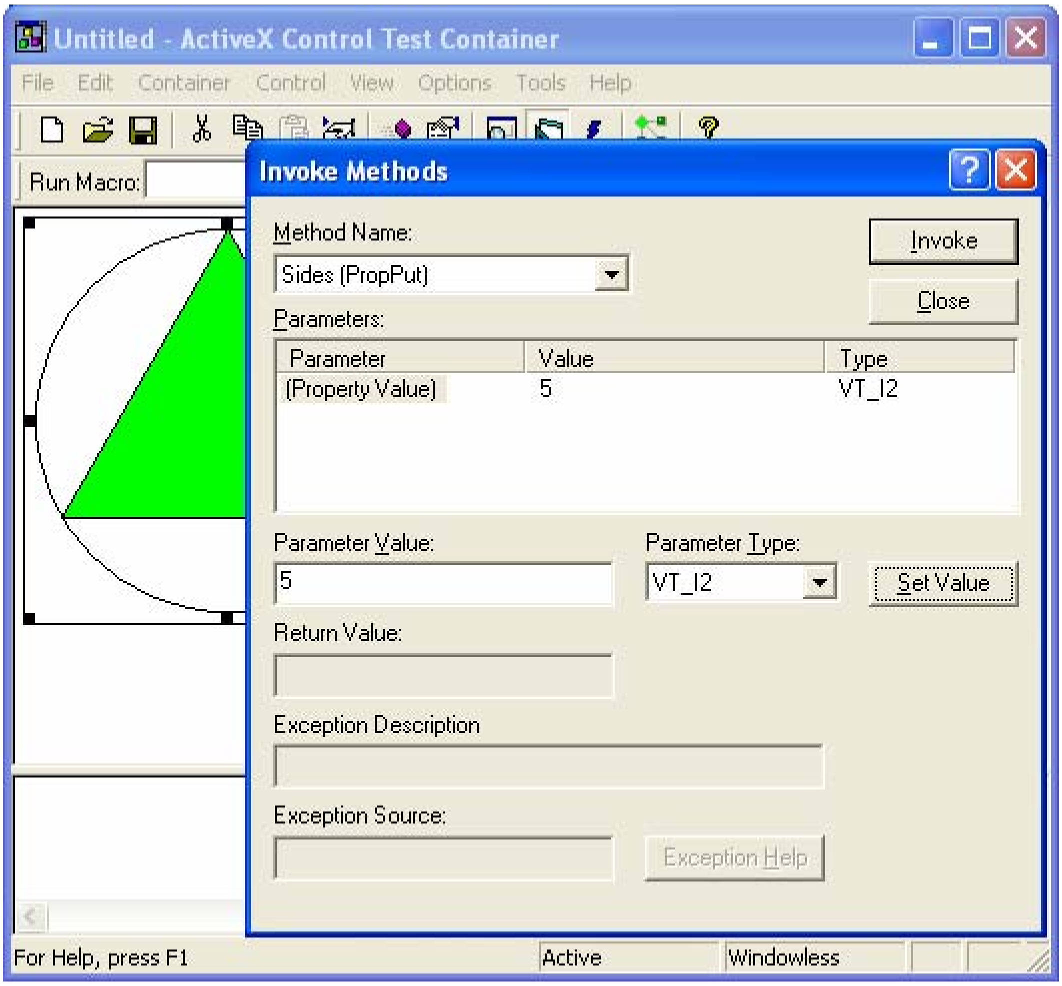


Figure 18: Invoking the Polygon's method.

2. Select the **PropPut** version of the **Sides** property from the **Method Name** drop-down list box.
3. Type 5 in the **Parameter Value** box, click **Set Value**, and click **Invoke**.

Note that the control does not change. Although you changed the number of sides internally by setting the `m_nSides` variable, this did not cause the control to repaint. If you switch to another application and then switch back to **Test Container**, you will find that the control has repainted and has the correct number of sides.
To correct this problem, add a call to the `FireViewChange()` function, defined in `IViewObjectExImpl`, after you set the number of sides. If the control is running in its own window, `FireViewChange()` will call the `InvalidateRect()` method directly. If the control is running windowless, the `InvalidateRect()` method will be called on the container's site interface. This forces the control to repaint itself.

To add a call to `FireViewChange()`

1. Update **PolyCtl.cpp** by adding the call to `FireViewChange()` to the `put_Sides` method. When you have finished, the `put_Sides` method should look like this:

```
STDMETHODIMP CPolyCtl::put_Sides(short newVal)
{
   if (newVal > 2 && newVal < 101)
   {
      m_nSides = newVal;
      FireViewChange();
      return S_OK;
   }
   else
```

```
            return Error(_T("Shape must have between 3 and 100 sides"));
        }
```

```
STDMETHODIMP CPolyCtl::put_Sides(SHORT newVal)
{
    // TODO: Add your implementation code here
    if (newVal > 2 && newVal < 101)
    {
        m_nSides = newVal;
        FireViewChange();
        return S_OK;
    }
    else
        return Error(_T("Shape must have between 3 and 100 sides"));
}
```

Listing 10.

After adding `FireViewChange()`, rebuild and try the control again in the **ActiveX Control Test Container**. This time when you change the number of sides and click **Invoke**, you should see the control change immediately. In the next step, you will add an event.

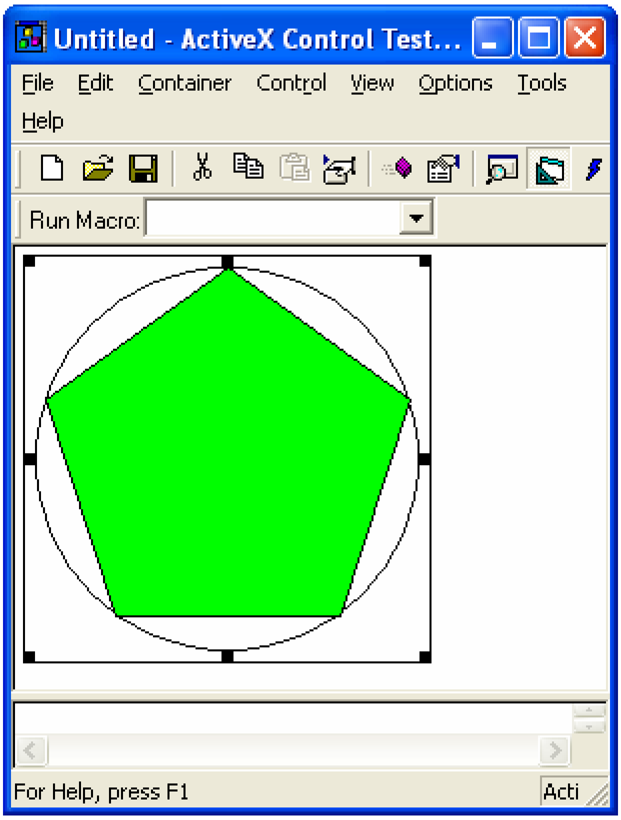


Figure 19: Polygon, an ATL control in action.

Step 5: Adding an Event

In this step, you will add a `ClickIn` and a `ClickOut` event to your ATL control. You will fire the `ClickIn` event if the user clicks within the polygon and fire `ClickOut` if the user clicks outside. The tasks to add an event are as follows:

- Adding the `ClickIn` and `ClickOut` Methods.

- Generating the **Type Library**.
- Implementing the **Connection Point Interfaces**.

Adding the ClickIn and ClickOut Methods

When you created the ATL control in step 2, you selected the **Connection points** check box. This created the `_IPolyCtlEvents` interface in the **Polygon.idl** file. Note that the interface name starts with an underscore. This is a convention to indicate that the interface is an **internal interface**. Thus, programs that allow you to browse COM objects can choose not to display the interface to the user. Also note that selecting Connection points added the following line in the **Polygon.idl** file to indicate that `_IPolyCtlEvents` is the default source interface:

```
[default, source] dispinterface _IPolyCtlEvents;
```

```
    coclass PolyCtl
    {
        [default] interface IPolyCtl;
        [default, source] dispinterface _IPolyCtlEvents;
    };
};
```

Listing 11.

The source attribute indicates that the control is the source of the notifications, so it will call this interface on the container.
Now add the `ClickIn` and `ClickOut` methods to the `_IPolyCtlEvents` interface.

To add the ClickIn and ClickOut methods

1. In Class View, expand **Polygon** and **PolygonLib** to display `_IPolyCtlEvents`.
2. Right-click `_IPolyCtlEvents`. On the shortcut menu, click **Add**, and then click **Add Method**.

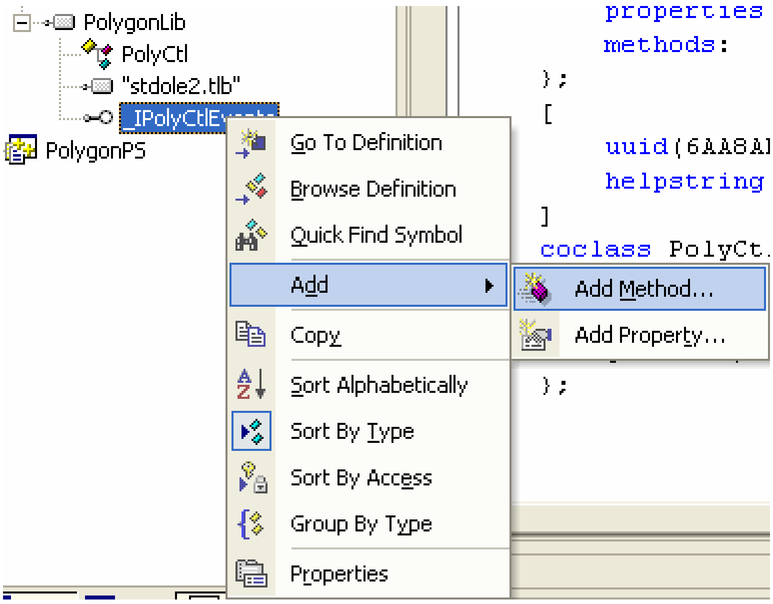


Figure 20: Adding methods to Polygon.

3. Select a **Return Type** of `void`.
4. Enter `ClickIn` in the **Method name** box.
5. Under **Parameter attributes**, select the **in** box.
6. Select a **Parameter type** of `LONG`.
7. Type **x** as the **Parameter name**, and click **Add**.

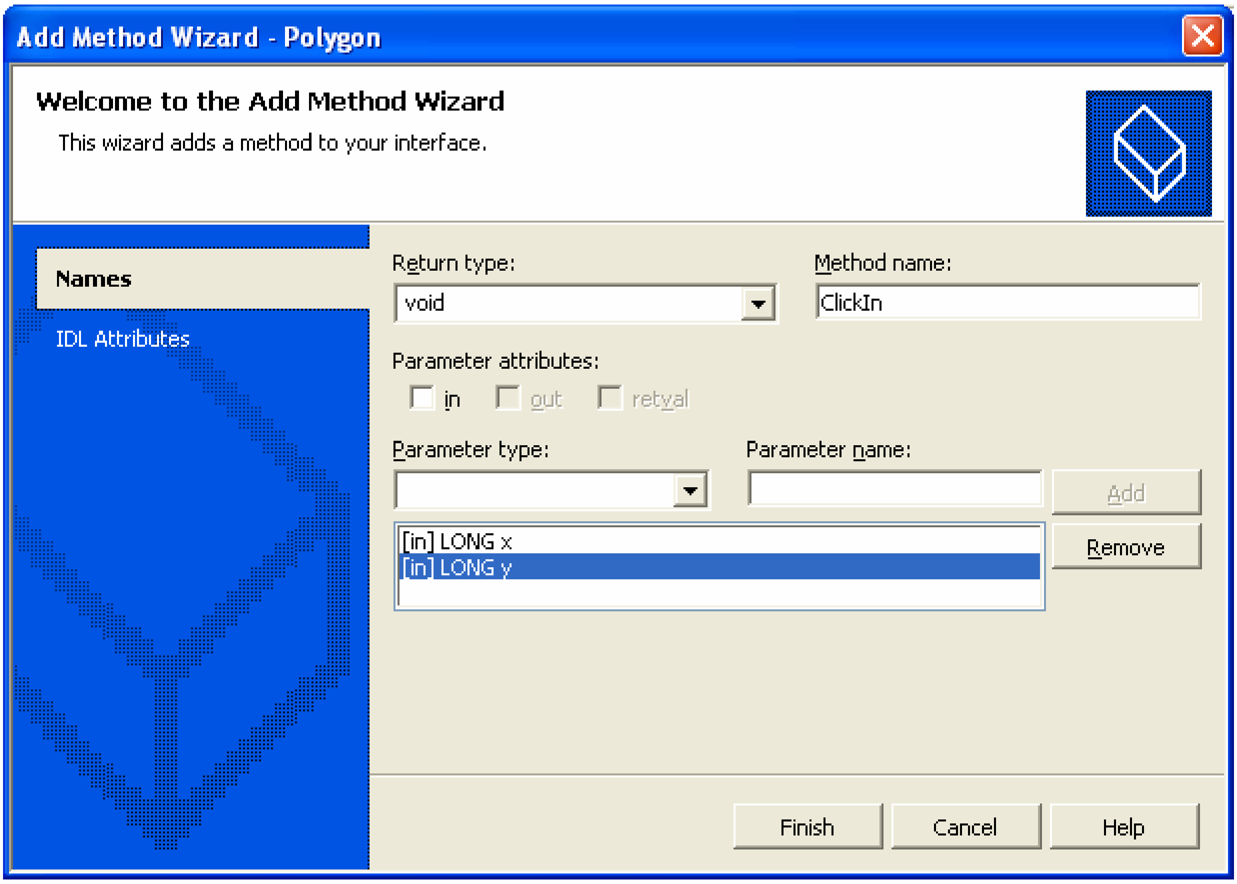


Figure 21: Adding `ClickIn` method through **Add Method Wizard**.

8. Next, repeat step 5 to 8, select the **in** box, enter a **Parameter type** `LONG` and **Parameter name** of **`y`**, and click the **Add** button.
9. Click **Finish**.
10. Repeat the steps above to define a `ClickOut` method with the same `LONG` parameters **`x`** and **`y`**, the same **Parameter attributes** and the same `void` return type.

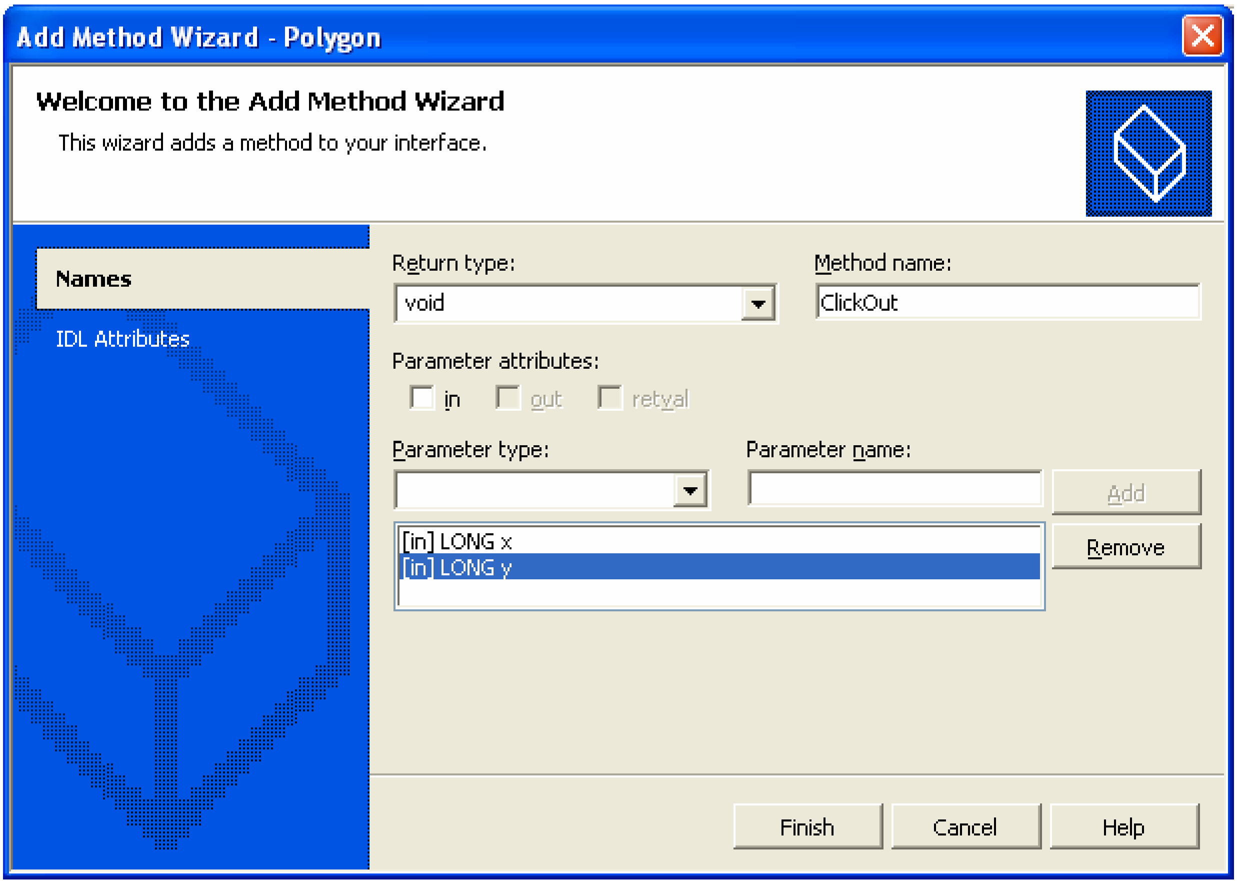


Figure 22: Adding `ClickOut` method through **Add Method Wizard**.

Check the **Polygon.idl** file to see that the code was added to the `_IPolyCtlEvents` dispinterface. The `_IPolyCtlEvents` dispinterface in your **Polygon.idl** file should now look like this:

```
dispinterface _IPolyCtlEvents
{
   properties:
   methods:
   [id(1), helpstring("method ClickIn")] void ClickIn([in]LONG x, [in] LONG y);
   [id(2), helpstring("method ClickOut")] void ClickOut([in] LONG x, [in] LONG y);
};
```

```
dispinterface _IPolyCtlEvents
{
    properties:
    methods:
    [id(1), helpstring("method ClickIn")] void ClickIn([in] LONG x,
        [in] LONG y);
    [id(2), helpstring("method ClickOut")] void ClickOut([in] LONG x,
        [in] LONG y);
};
[
```

Listing 12.

The `ClickIn` and `ClickOut` methods take the `x` and `y` coordinates of the clicked point as parameters.

Generating the Type Library

Generate the type library at this point, because the **Connection Point Wizard** will use it to obtain the information it needs to construct a connection point interface and a connection point container interface for your control.

To generate the type library

- Rebuild your project or,
- Right-click the **Polygon.idl** file in **Solution Explorer** and click **Compile** on the shortcut menu.

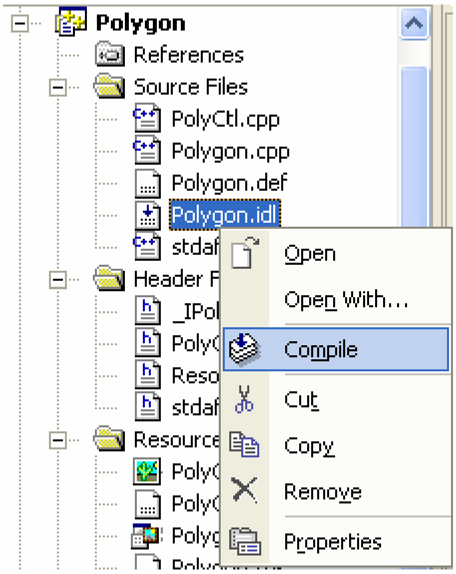


Figure 23: Compiling the IDL file.

This will create the **Polygon.tlb** file, which is your type library. The **Polygon.tlb** file is not visible from **Solution Explorer**, because it is a binary file and cannot be viewed or edited directly.

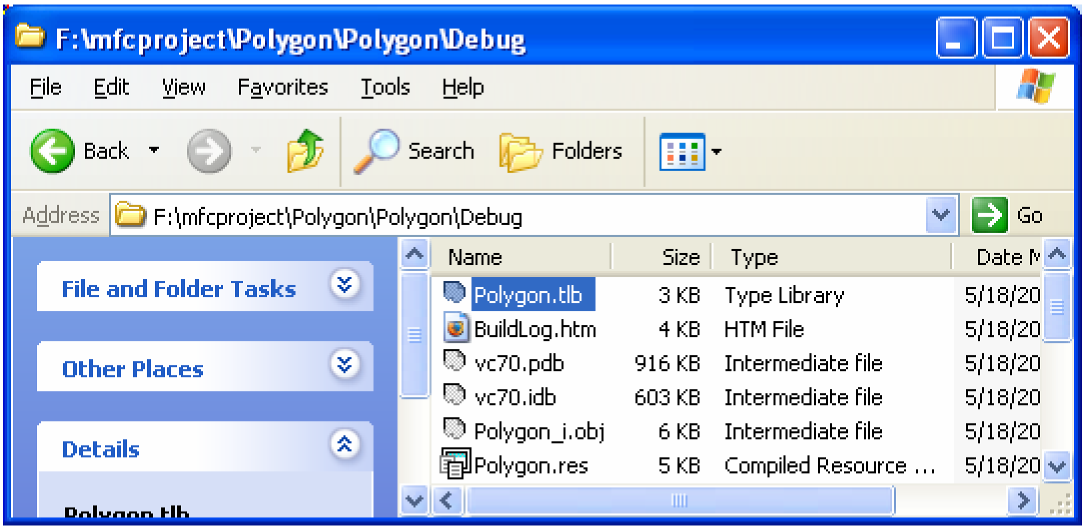


Figure 24: The generated TLB (type library) file.

Implementing the Connection Point Interfaces

Implement a connection point interface and a connection point container interface for your control. In COM, events are implemented through the mechanism of connection points. To receive events from a COM object, a container establishes an advisory connection to the connection point that the COM object implements. Because a COM object can have multiple connection points, the COM object also implements a connection point container interface. Through this interface, the container can determine which connection points are supported.

The interface that implements a connection point is called `IConnectionPoint`, and the interface that implements a connection point container is called `IConnectionPointContainer`.
To help implement `IConnectionPoint`, you will use the **Implement Connection Point Wizard**. This wizard generates the `IConnectionPoint` interface by reading your type library and implementing a function for each event that can be fired.

To use the Implement Connection Point Wizard

1. In Class View, right-click your control's implementation class `CPolyCtl`.
2. On the shortcut menu, click **Add**, and then click **Add Connection Point**.

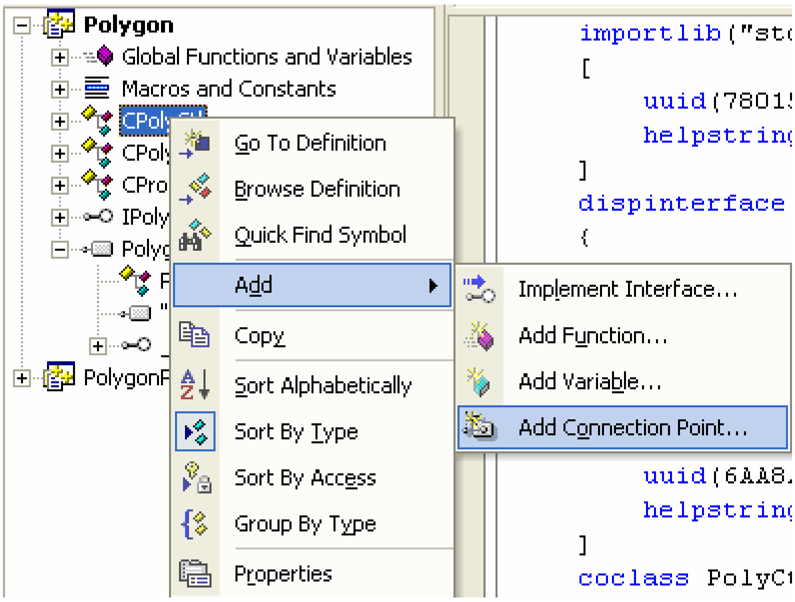


Figure 25: Adding **Connection Point** to ATL control.

3. Select `_IPolyCtlEvents` from the **Source Interfaces** list and double-click it to add it to the **Implement connection points** column. Click **Finish**. A proxy class for the connection point will be generated, in this case, `CProxy_IPolyCtlEvents`.

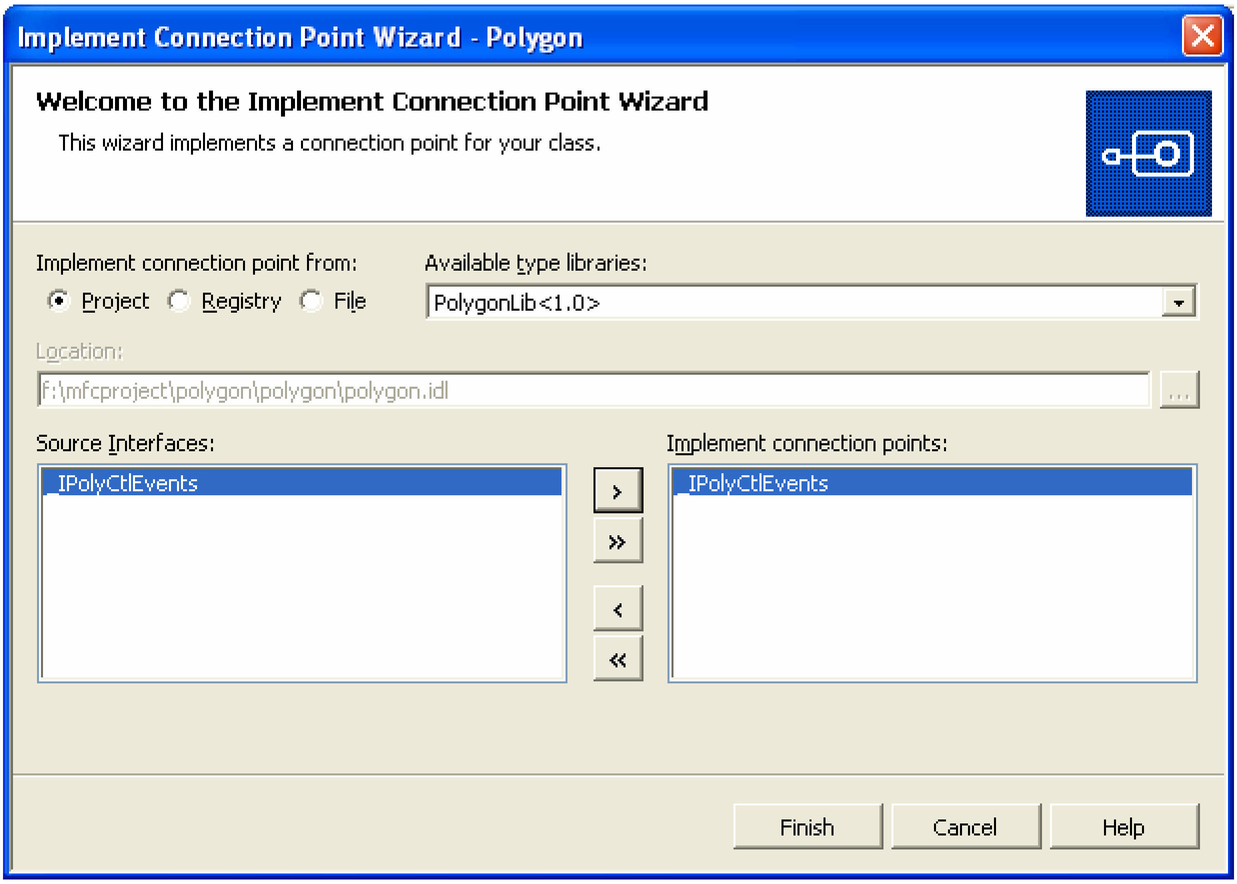


Figure 26: Selecting and adding a **Connection Point**.

If you look at the generated **_IPolyCtlEvents_CP.h** file in **Solution Explorer**, you will see that it has a class called `CProxy_IPolyCtlEvents` that derives from `IConnectionPointImpl`. **_IPolyCtlEvents_CP.h** also defines the two methods `Fire_ClickIn()` and `Fire_ClickOut()`, which take the two coordinate parameters. You call these methods when you want to fire an event from your control.
The wizard also added `CProxy_PolyEvents` and `IConnectionPointContainerImpl` to your control's multiple inheritance list. The wizard also exposed `IConnectionPointContainer` for you by adding appropriate entries to the COM map.
You are finished implementing the code to support events. Now, add some code to fire the events at the appropriate moment. Remember, you are going to fire a `ClickIn` or `ClickOut` event when the user clicks the left mouse button in the control. To find out when the user clicks the button, add a handler for the `WM_LBUTTONDOWN` message.

To add a handler for the `WM_LBUTTONDOWN` message

1. In Class View, right-click the `CPolyCtl` class and click **Properties** on the shortcut menu.
2. In the **Properties** window, click the **Messages** icon and then click `WM_LBUTTONDOWN` from the list on the left.

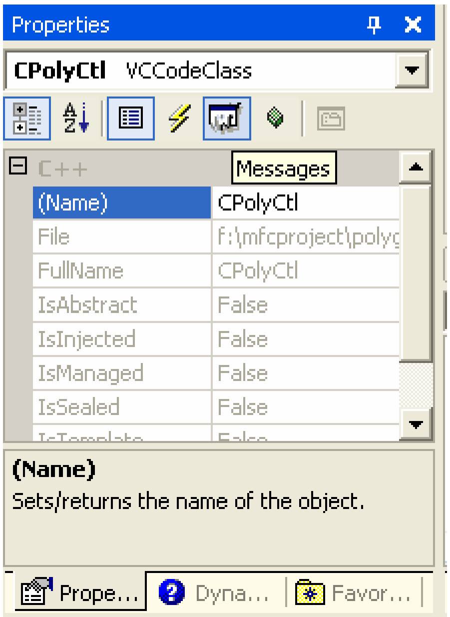


Figure 27: Adding `WM_LBUTTONDOWN`, a Windows message handler through **Properties** window.

3. From the drop-down list that appears, click **<Add> OnLButtonDown**. The `OnLButtonDown()` handler declaration will be added to **PolyCtl.h**, and the handler implementation will be added to **PolyCtl.cpp**.

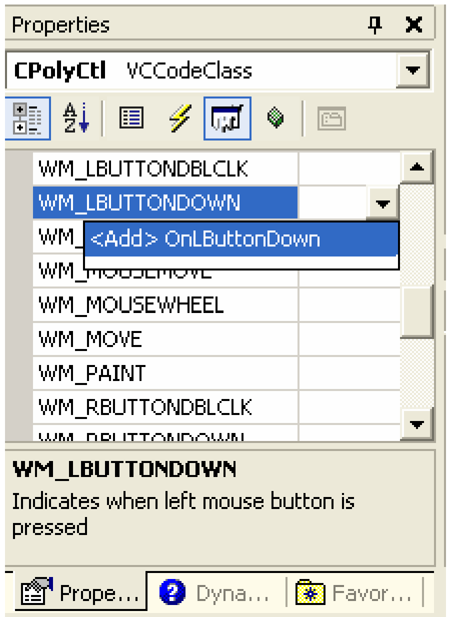


Figure 28: Selecting and adding `WM_LBUTTONDOWN`.

Next, modify the handler.

To modify the `OnLButtonDown()` method

Change the code which comprises the `OnLButtonDown()` method in **PolyCtl.cpp** (deleting any code placed by the wizard) so that it looks like this:

```
LRESULT CPolyCtl::OnLButtonDown(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
{
   HRGN hRgn;
```

```
    WORD xPos = LOWORD(lParam);  // horizontal position of cursor
    WORD yPos = HIWORD(lParam);  // vertical position of cursor

    CalcPoints(m_rcPos);

    // Create a region from our list of points
    hRgn = CreatePolygonRgn(&m_arrPoint[0], m_nSides, WINDING);

    // If the clicked point is in our polygon then fire the ClickIn
    //  event otherwise we fire the ClickOut event
    if (PtInRegion(hRgn, xPos, yPos))
       Fire_ClickIn(xPos, yPos);
    else
       Fire_ClickOut(xPos, yPos);

    // Delete the region that we created
    DeleteObject(hRgn);
    return 0;
}
```

```
LRESULT CPolyCtl::OnLButtonDown(UINT uMsg, WPARAM wParam,
                                LPARAM lParam, BOOL& bHandled)
{
    // TODO: Add your message handler code here and/or call default
    HRGN hRgn;
    WORD xPos = LOWORD(lParam);  // horizontal position of cursor
    WORD yPos = HIWORD(lParam);  // vertical position of cursor
    CalcPoints(m_rcPos);

    // Create a region from our list of points
    hRgn = CreatePolygonRgn(&m_arrPoint[0], m_nSides, WINDING);

    // If the clicked point is in our polygon then fire the ClickIn
    //  event otherwise we fire the ClickOut event
    if (PtInRegion(hRgn, xPos, yPos))
       Fire_ClickIn(xPos, yPos);
    else
       Fire_ClickOut(xPos, yPos);

    // Delete the region that we created
    DeleteObject(hRgn);
    return 0;
}
```

Listing 13.

This code makes use of the points calculated in the `OnDraw()` function to create a region that detects the user's mouse clicks with the call to `PtInRegion()`.
The `uMsg` parameter is the `ID` of the Windows message being handled. This allows you to have one function that handles a range of messages. The `wParam` and the `lParam` parameters are the standard values for the message being handled. The parameter `bHandled` allows you to specify whether the function handled the message or not. By default, the value is set to `TRUE` to indicate that the function handled the message, but you can set it to `FALSE`. This will cause ATL to continue looking for another message handler function to send the message to.

Building and Testing the Control

Now try out your events. Build the control and start the **ActiveX Control Test Container** again. This time, view the event log window. To route events to the output window, click **Logging** from the **Options** menu and select **Log to output window**.

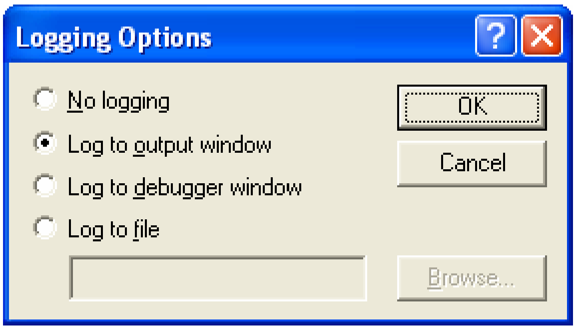


Figure 29: Setting the **Logging Options** of the **ActiveX Control Test Container**.

Insert the control and try clicking in the window. Note that `ClickIn` is fired if you click within the filled polygon, and `ClickOut` is fired when you click outside of it.

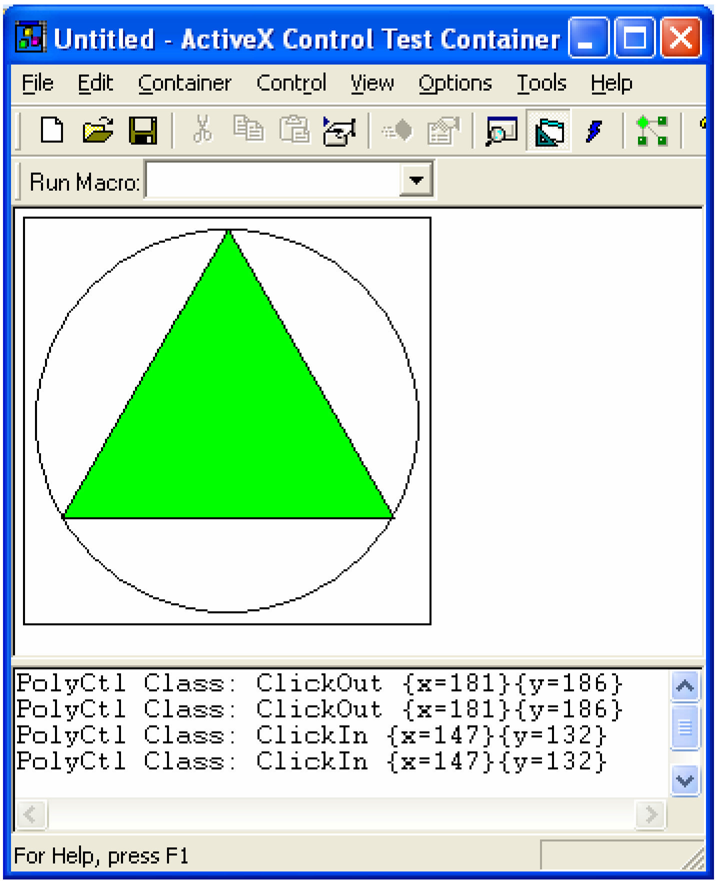


Figure 30: Polygon in action.

Next, you will add a property page.

Step 6: Adding a Property Page

Property pages are implemented as separate COM objects, which allow them to be shared if required. In this step, you will do the following tasks to add a property page to the control:

- Creating the Property Page Resource.
- Adding Code to Create and Manage the Property Page.
- Adding the Property Page to the Control.

Creating the Property Page Resource

To add a property page to your control, use the **ATL Add Class Wizard**.

To add a Property Page

1. In **Solution Explorer**, right-click **Polygon**.
2. On the shortcut menu, click **Add**, and then click **Add Class**.

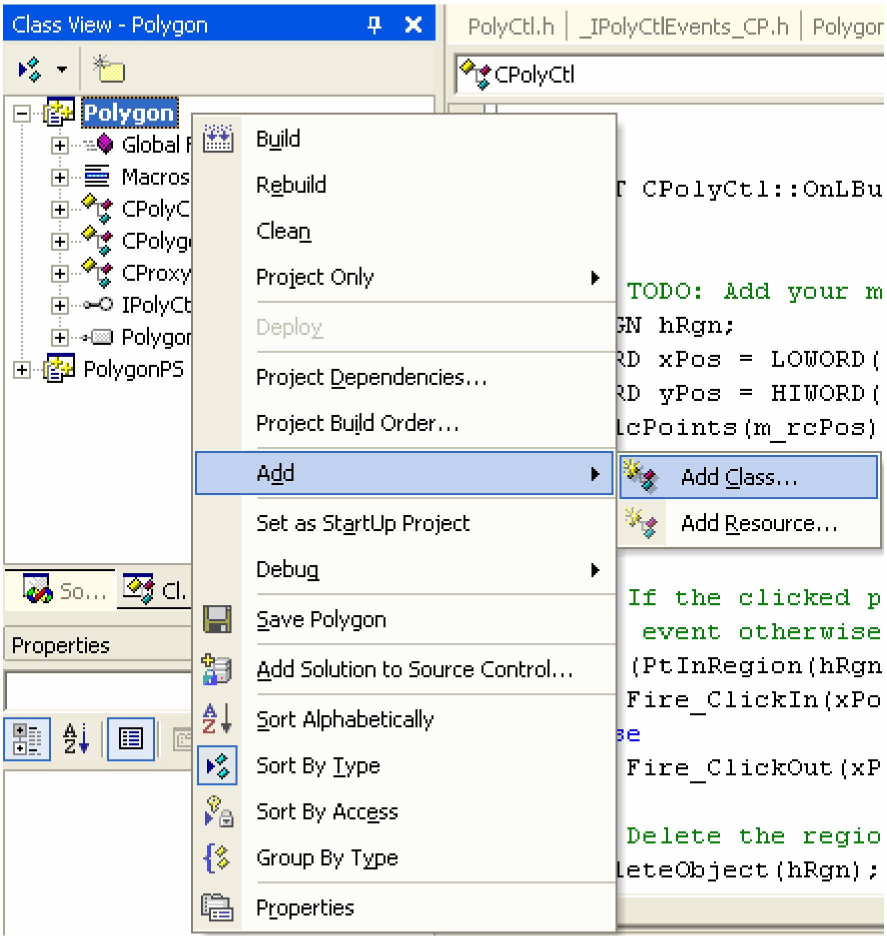


Figure 31: Adding a new class for ATL control's property page.

3. From the list of templates, select **ATL Property Page** and click **Open**.

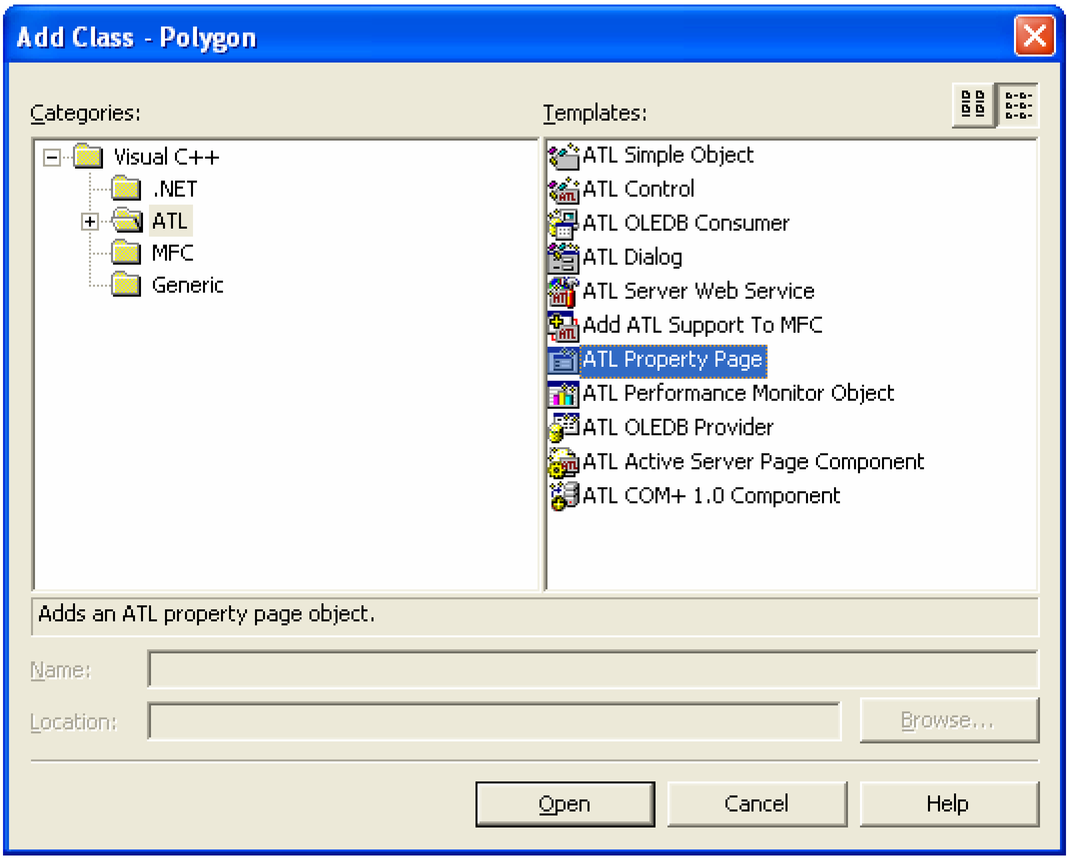


Figure 32: Adding an **ATL Property Page**.

4. When the **ATL Property Page Wizard** appears, enter **PolyProp** as the **Short name**:

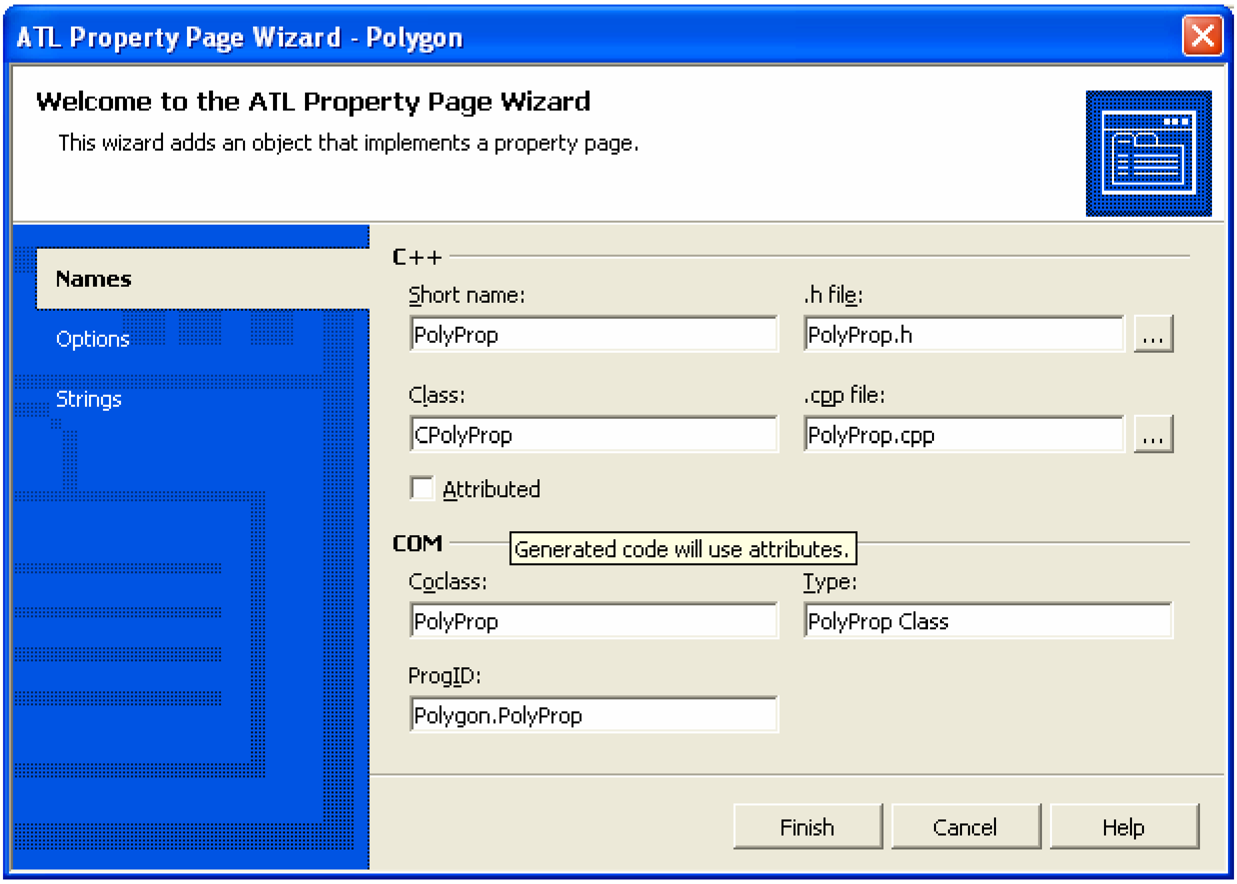


Figure 33: **ATL Property Page Wizard**, **Names** page.

1. Click **Strings** to open the **Strings page** and enter **&Polygon** as the **Title**:

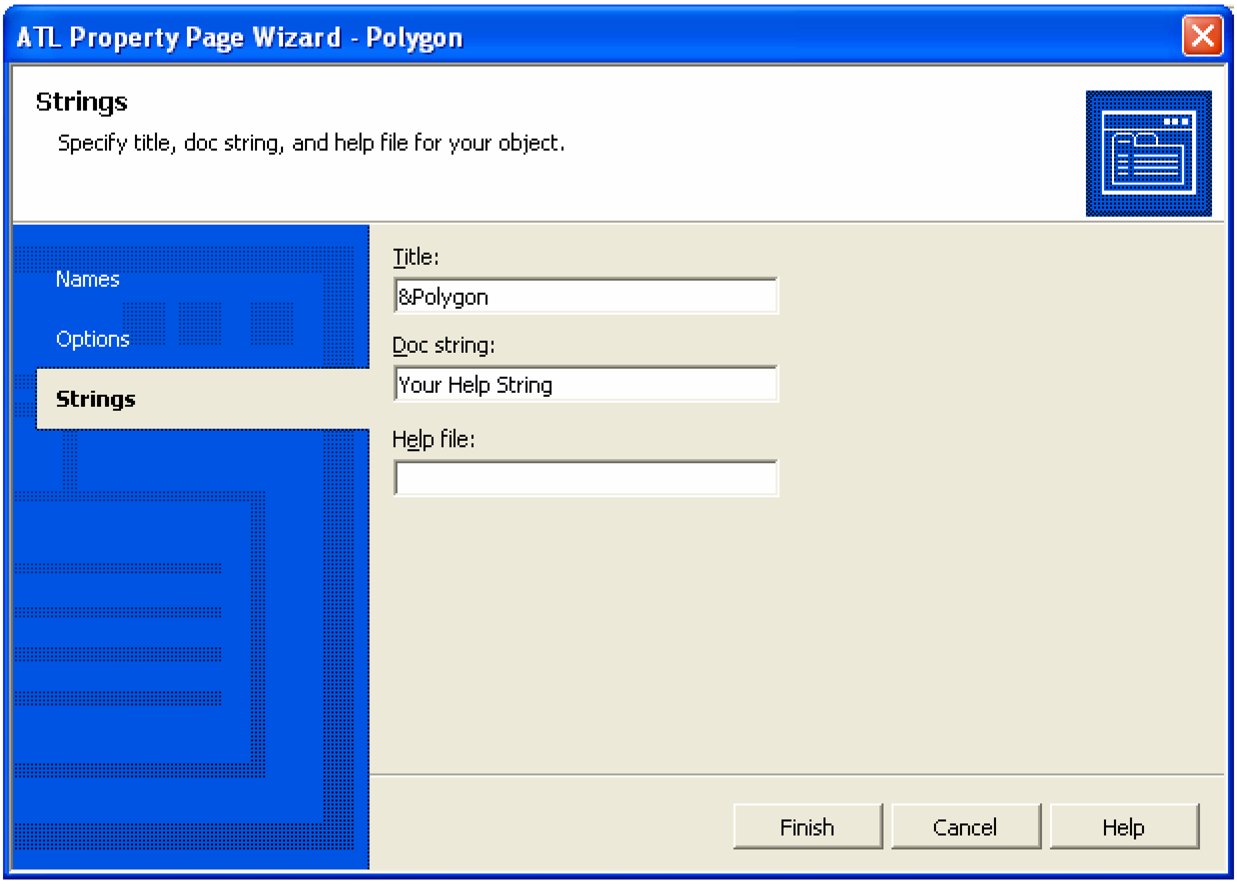


Figure 34: **ATL Property Page Wizard**, **Strings** page.

The **Title** of the property page is the string that appears in the tab for that page. The **Doc** string is a description that a property frame uses to put in a status line or tool tip. Note that the standard property frame currently does not use this string, so you can leave it with the default contents. You will not going to generate a **Help** file at the moment, so delete the entry in that text box.

2. Click Finish, and the property page object will be created.

The following three files are created:

File	Description
PolyProp.h	Contains the C++ class `CPolyProp`, which implements the property page.
PolyProp.cpp	Includes the **PolyProp.h** file.
PolyProp.rgs	The registry script that registers the property page object.

Table 4.

The following code changes are also made:

- The new property page is added to the object entry map in **Polygon.cpp**.
- The `PolyProp` class is added to the **Polygon.idl** file.
- The new registry script file **PolyProp.rgs** is added to the project resource.
- A dialog box template is added to the project resource for the property page.
- The property strings that you specified are added to the resource string table.

Now add the fields that you want to appear on the property page.

To add fields to the Property Page

1. In **Solution Explorer**, double-click the **Polygon.rc** resource file. This will open **Resource View**.

2. In **Resource View**, expand the **Dialog** node and double-click `IDD_POLYPROP`. Note that the dialog box that appears is empty except for a label that tells you to insert your controls here.
3. Select that label and change it to read **Sides:** by altering the **Caption** text in the **Properties** window and resizing the label box:

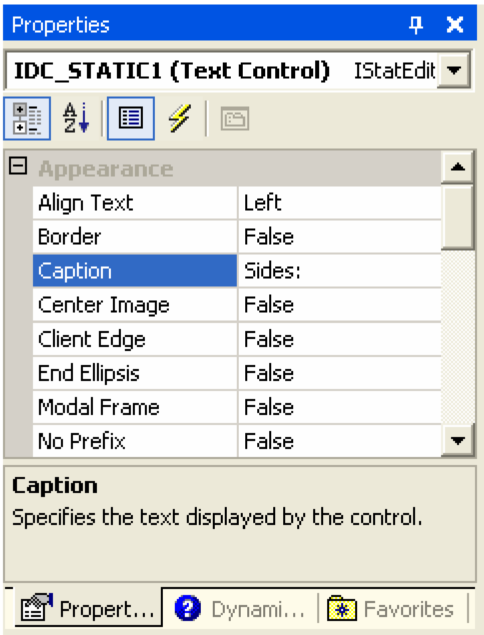


Figure 35: Modifying the **Static** control property.

4. Drag an **Edit** control from the **Toolbox** to the right of the label:

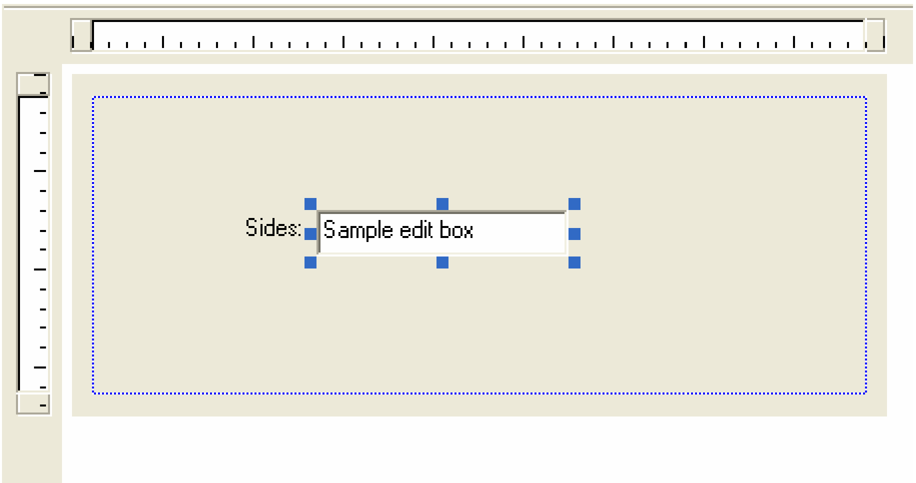


Figure 36: Property page template, adding an **Edit** control.

5. Finally, change the `ID` of the **Edit** control to `IDC_SIDES` using the **Properties** window.

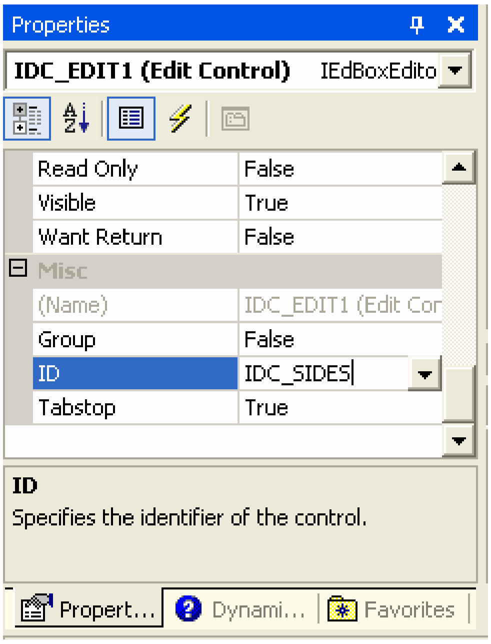


Figure 37: Modifying the **Edit** control property.

This completes the process of creating the property page resource.

Adding Code to Create and Manage the Property Page

Now that you have created the property page resource, you need to write the implementation code. First, enable the `CPolyProp` class to set the number of sides in your object when the **Apply** button is pressed.

To modify the Apply function to set the number of sides

Change the `Apply()` function in **PolyProp.h** as follows:

```
STDMETHOD(Apply)(void)
{
   USES_CONVERSION;
   ATLTRACE(_T("CPolyProp::Apply\n"));
   for (UINT i = 0; i < m_nObjects; i++)
   {
      CComQIPtr<IPolyCtl, &IID_IPolyCtl> pPoly(m_ppUnk[i]);
      short nSides = (short)GetDlgItemInt(IDC_SIDES);
      if FAILED(pPoly->put_Sides(nSides))
      {
         CComPtr<IErrorInfo> pError;
         CComBSTR strError;
         GetErrorInfo(0, &pError);
         pError->GetDescription(&strError);
         MessageBox(OLE2T(strError), _T("Error"), MB_ICONEXCLAMATION);
         return E_FAIL;
      }
   }
   m_bDirty = FALSE;
   return S_OK;
}
```

```
STDMETHOD(Apply)(void)
{
    USES_CONVERSION;
    ATLTRACE(_T("CPolyProp::Apply\n"));
    for (UINT i = 0; i < m_nObjects; i++)
    {
        CComQIPtr<IPolyCtl, &IID_IPolyCtl> pPoly(m_ppUnk[i]);
        short nSides = (short)GetDlgItemInt(IDC_SIDES);
        if FAILED(pPoly->put_Sides(nSides))
        {
            CComPtr<IErrorInfo> pError;
            CComBSTR strError;
            GetErrorInfo(0, &pError);
            pError->GetDescription(&strError);
            MessageBox(OLE2T(strError), _T("Error"), MB_ICONEXCLAMATION);
            return E_FAIL;
        }
    }
    m_bDirty = FALSE;
    return S_OK;
}
```

Listing 14.

A property page can have more than one client attached to it at a time, so the `Apply()` function loops around and calls `put_Sides()` on each client with the value retrieved from the edit box. You are using the `CComQIPtr` class, which performs the `QueryInterface()` on each object to obtain the `IPolyCtl` interface from the `IUnknown` interface (stored in the `m_ppUnk` array).

The code now checks that setting the **Sides** property actually worked. If it fails, the code displays a message box displaying error details from the `IErrorInfo` interface. Typically, a container asks an object for the `ISupportErrorInfo` interface and calls `InterfaceSupportsErrorInfo()` first, to determine whether the object supports setting error information. You can skip this task.

`CComPtr` helps you by automatically handling the reference counting, so you do not need to call `Release()` on the interface. `CComBSTR` helps you with `BSTR` processing, so you do not have to perform the final `SysFreeString()` call. You also use one of the various string conversion classes, so you can convert the `BSTR` if necessary (this is why the `USES_CONVERSION` macro is at the start of the function).

You also need to set the property page's dirty flag to indicate that the **Apply** button should be enabled. This occurs when the user changes the value in the **Sides** edit box.

To handle the Apply button

1. In Class View, right-click `CPolyProp` and click **Properties** on the shortcut menu.
2. In the **Properties** window, click the **Events** icon.

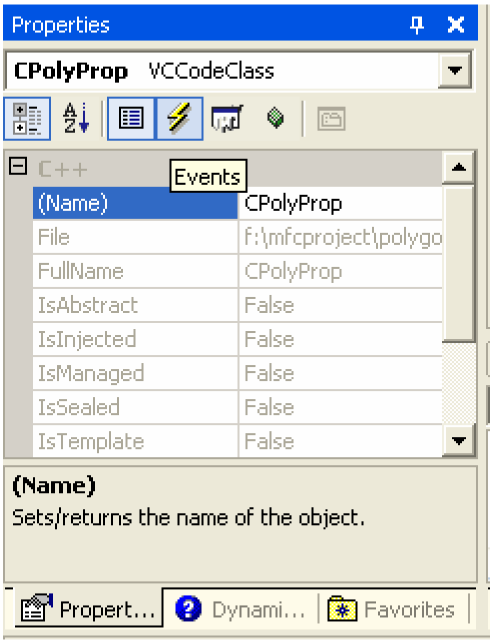


Figure 38: An event **Properties** page.

3. Expand the `IDC_SIDES` node in the event list.
4. Select `EN_CHANGE`, and from the drop-down menu to the right, click **<Add> OnEnChangeSides**. The `OnEnChangeSides()` handler declaration will be added to **Polyprop.h**, and the handler implementation to **Polyprop.cpp**.

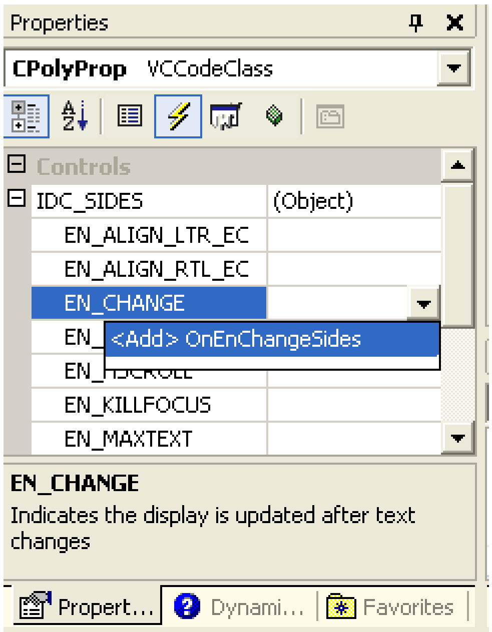


Figure 39: Adding an event to `CPolyProp` class.

```
    }
    LRESULT OnEnChangeSides(WORD /*wNotifyCode*/, WORD /*wID*/,
        HWND /*hWndCtl*/, BOOL& /*bHandled*/);
};
```

Listing 15.

```
LRESULT CPolyProp::OnEnChangeSides(WORD /*wNotifyCode*/,
        WORD /*wID*/, HWND /*hWndCtl*/, BOOL& /*bHandled*/)
{
    // TODO:  If this is a RICHEDIT control, the control will not
    // send this notification unless you override the __super::OnInitDialog()
    // function and call CRichEditCtrl().SetEventMask()
    // with the ENM_CHANGE flag ORed into the mask.

    // TODO:  Add your control notification handler code here

    return 0;
}
```

Listing 16.

Next, you will modify the handler.

To modify the `OnEnChangeSides()` method

Add the following code in **Polyprop.cpp** to the `OnEnChangeSides()` method (deleting any code that the wizard put there):

```
LRESULT CPolyProp::OnEnChangeSides(WORD /*wNotifyCode*/,
WORD /*wID*/, HWND /*hWndCtl*/, BOOL& /*bHandled*/)
{
   SetDirty(TRUE);
   return 0;
}
```

```
LRESULT CPolyProp::OnEnChangeSides(WORD /*wNotifyCode*/,
        WORD /*wID*/, HWND /*hWndCtl*/, BOOL& /*bHandled*/)
{
    SetDirty(TRUE);
    return 0;
}
```

Listing 17.

`OnEnChangeSides()` will be called when a `WM_COMMAND` message is sent with the `EN_CHANGE` notification for the `IDC_SIDES` control. `OnEnChangeSides()` then calls `SetDirty()` and passes `TRUE` to indicate the property page is now dirty and the **Apply** button should be enabled.

Adding the Property Page to the Control

The **ATL Add Class Wizard** and the **ATL Property Page Wizard** do not add the property page to your control for you automatically, because there could be multiple controls in your project. You will need to add an entry to the control's property map.

To add the property page

Open **PolyCtl.h** and add this line to the property map:

```
PROP_ENTRY("Sides", 1, CLSID_PolyProp)
```

The control's property map now looks like this:

```
BEGIN_PROP_MAP(CPolyCtl)
   PROP_DATA_ENTRY("_cx", m_sizeExtent.cx, VT_UI4)
   PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
```

```
        PROP_ENTRY("FillColor", DISPID_FILLCOLOR, CLSID_StockColorPage)
        PROP_ENTRY("Sides", 1, CLSID_PolyProp)
        // Example entries
        // PROP_ENTRY("Property Description", dispid, clsid)
        // PROP_PAGE(CLSID_StockColorPage)
    END_PROP_MAP()
```

```
BEGIN_PROP_MAP(CPolyCtl)
    PROP_DATA_ENTRY("_cx", m_sizeExtent.cx, VT_UI4)
    PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
    PROP_ENTRY("FillColor", DISPID_FILLCOLOR, CLSID_StockColorPage)
    PROP_ENTRY("Sides", 1, CLSID_PolyProp)
    // Example entries
    // PROP_ENTRY("Property Description", dispid, clsid)
    // PROP_PAGE(CLSID_StockColorPage)
END_PROP_MAP()
```

Listing 18.

You could have added a `PROP_PAGE` macro with the `CLSID` of your property page, but if you use the `PROP_ENTRY` macro as shown, the **Sides** property value is also saved when the control is saved.
The three parameters to the macro are the property description, the `DISPID` of the property, and the `CLSID` of the property page that has the property on it. This is useful if, for example, you load the control into Visual Basic and set the number of **Sides** at design time. Because the number of **Sides** is saved, when you reload your Visual Basic project, the number of **Sides** will be restored.

Building and Testing the Control

Now build that control and insert it into **ActiveX Control Test Container**. In **Test Container**, on the **Edit** menu, click **PolyCtl Class Object**. The property page appears; click the **Polygon** tab. (If you clean your solution before re-building, you need to re-insert the **PolyCtl Class** object using **Edit Insert New Control** menu).
The **Apply** button is initially disabled. Start typing a value in the **Sides** box and the Apply button will become enabled. After you have finished entering the value, click the Apply button. The control display changes, and the Apply button is again disabled. Try entering an invalid value. You will see a message box containing the error description that you set from the put_Sides function.

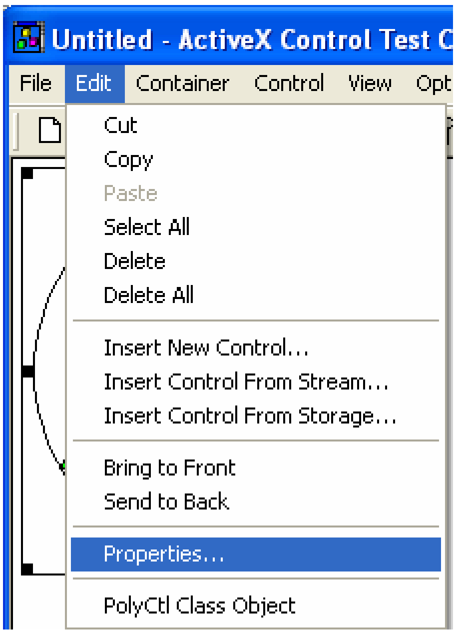


Figure 40: Invoking Polygon's property page in **ActiveX Control Test Container**.

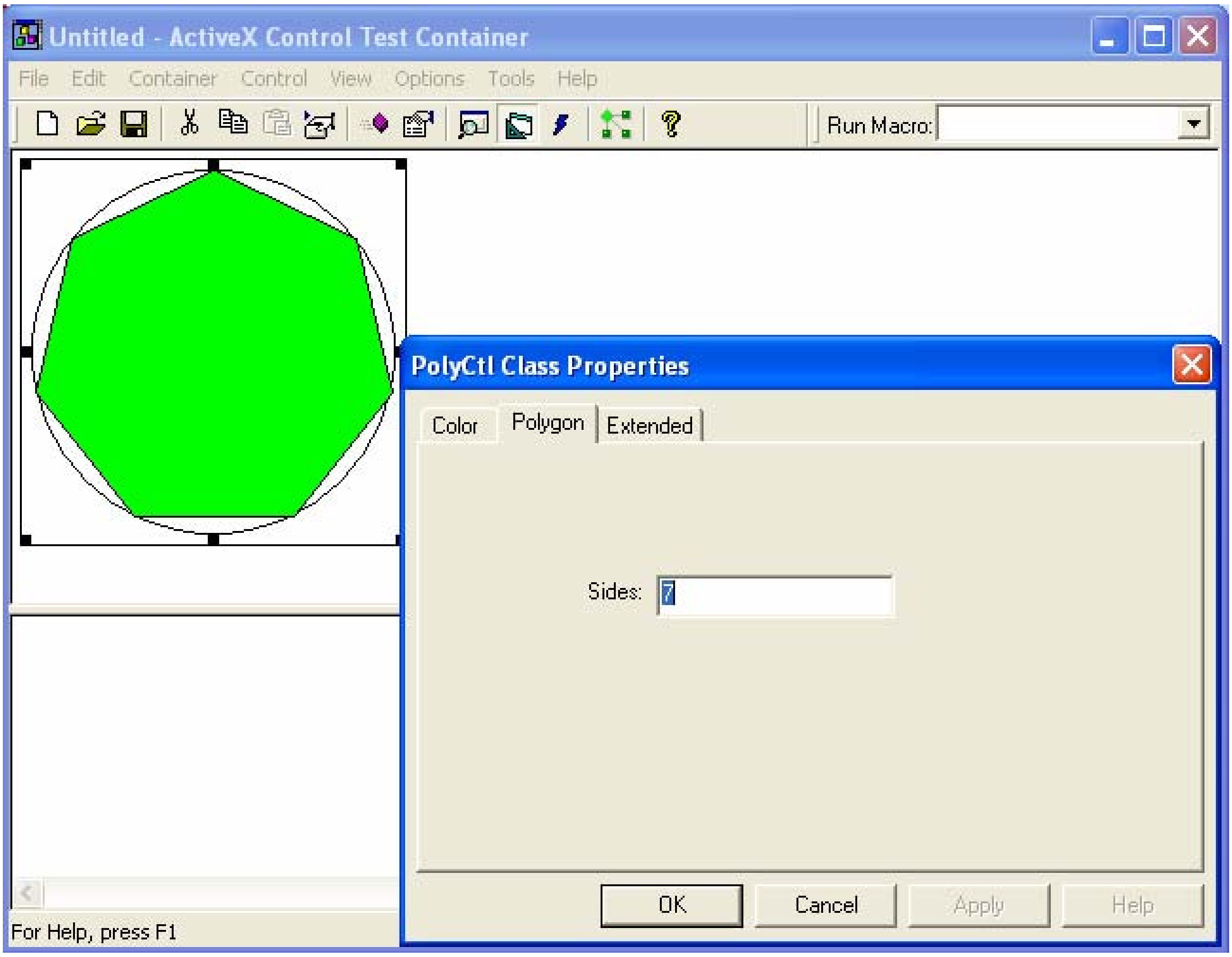


Figure 41: Changing **Sides**'s value through property page.

Next, you will put your control on a Web page.

Step 7: Putting the Control on a Web Page

Your control is now finished. To see your control work in a real-world situation, put it on a Web page. An HTML file that contains the control was created when you defined your control. Open the **PolyCtl.htm** file from **Solution Explorer** and you can see your control on a Web page.

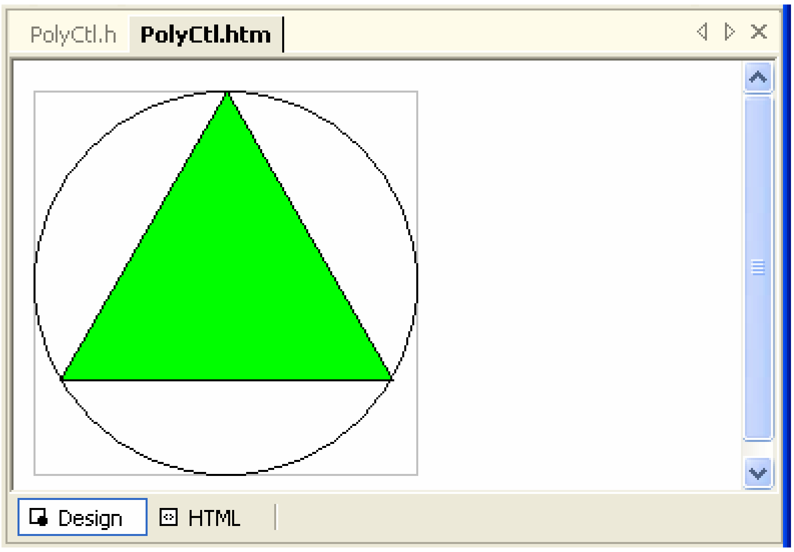


Figure 42: Polygon, an ATL control in web page.

In this step, you will script the Web page to respond to events. You will also modify the control to let Internet Explorer know that the control is safe for scripting.

Scripting the Web Page

The control does not do anything yet, so change the Web page to respond to the events that you send.

To script the Web page

Open **PolyCtl.htm** and select **HTML view**. Add the lines in bold to the HTML code that makes up the page.

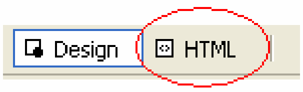


Figure 43: The HTML view.

```
<HTML>
<HEAD>
<TITLE>ATL 3.0 test page for object PolyCtl</TITLE>
</HEAD>
<BODY>
<OBJECT ID="PolyCtl" <
 CLASSID="CLSID:4CBBC676-507F-11D0-B98B-000000000000">
>
</OBJECT>
<SCRIPT LANGUAGE="VBScript">
<!--
Sub PolyCtl_ClickIn(x, y)
   PolyCtl.Sides = PolyCtl.Sides + 1
End Sub
Sub PolyCtl_ClickOut(x, y)
   PolyCtl.Sides = PolyCtl.Sides - 1
End Sub
-->
</SCRIPT>
</BODY>
</HTML>
```

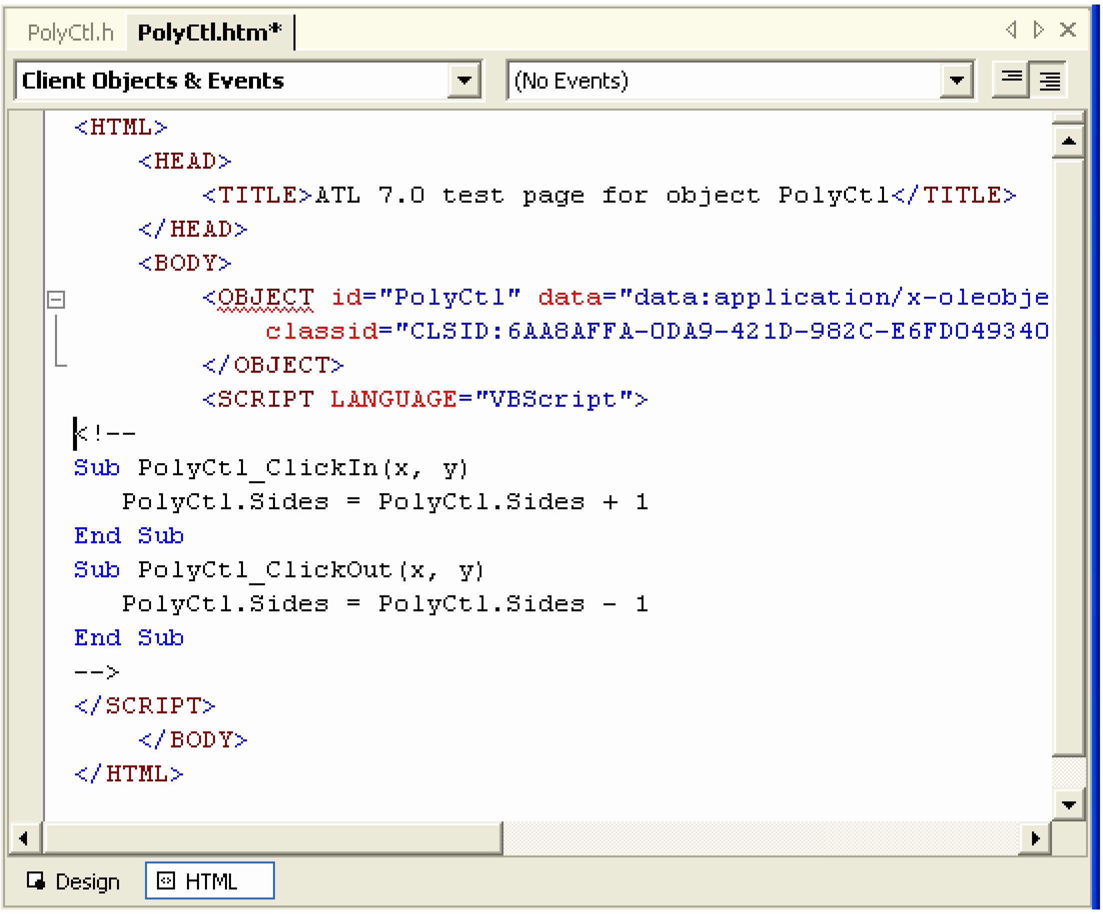


Figure 44: Adding VBScript codes to the HTML.

You have added some **VBScript** code that gets the **Sides** property from the control and increases the number of sides by one if you click inside the control. If you click outside the control, you reduce the number of sides by one.

Indicating that the Control Is Safe for Scripting

You can view the Web page with the control in Internet Explorer or, more conveniently, use the **Web browser** view built into Visual C++ .NET. To see your control in the Web browser view, right-click **PolyCtl.htm**, and click **View in Browser**.

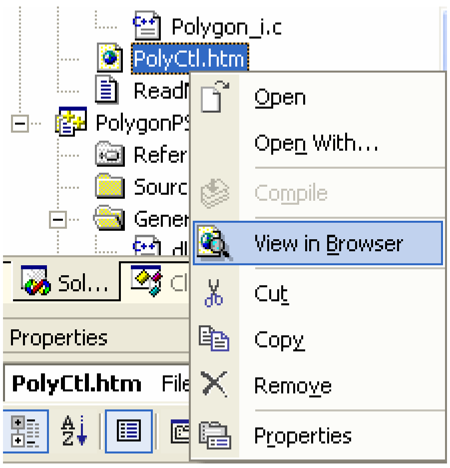


Figure 45: Viewing the ATL control in Browser.

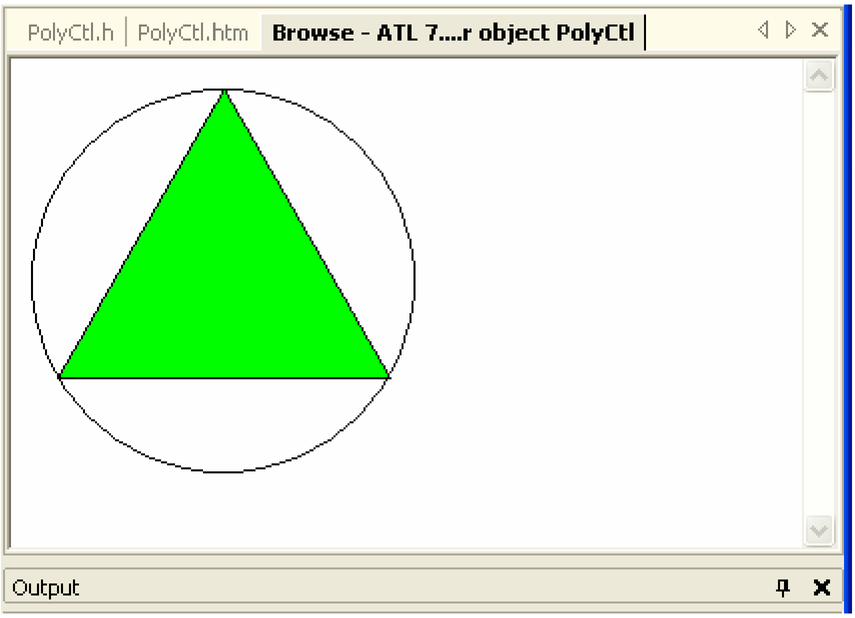


Figure 46: Polygon, an ATL control seen in Browser.

Based on your current Internet Explorer security settings, you may receive a **Security Alert** dialog box stating that the control may not be safe to script and could potentially do damage. You can try single click inside the green polygon. The following Security alert dialog box will be displayed.

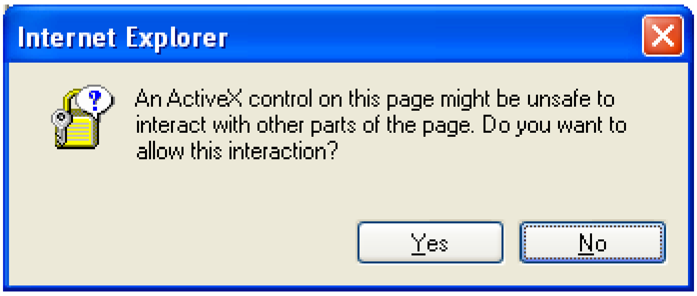


Figure 47: Internet Explorer **Security Alert** dialog.

Just click **Yes**. Then click again inside the green polygon. Finally, click outside the polygon (inside the circle). Can you see the action?

Other typical example, if you had a control that displayed a file but also had a **Delete** method that deleted a file, it would be safe if you just viewed it on a page. It would be not safe to script, however, because someone could call the **Delete** method.
Security Note: For this tutorial, you can change your security settings in Internet Explorer to run ActiveX controls that are not marked as safe. In **Control Panel**, click **Internet Properties** and click **Security** to change the appropriate settings. When you have completed the tutorial, change your security settings back to their original state.
You can programmatically alert Internet Explorer that it does not need to display the **Security Alert** dialog box for this particular control. You can do this with the `IObjectSafety` interface, and ATL supplies an implementation of this interface in the class `IObjectSafetyImpl`. To add the interface to your control, add `IObjectSafetyImpl` to your list of inherited classes and add an entry for it in your COM map.

To add `IObjectSafetyImpl` to the control

1. Add the following line to the end of the list of inherited classes in **PolyCtl.h** and add a comma to the previous line:

```
public IObjectSafetyImpl<CPolyCtl, INTERFACESAFE_FOR_UNTRUSTED_CALLER>
```

```
    public CComCoClass<CPolyCtl, &CLSID_PolyCtl>,
    public CComControl<CPolyCtl>,
    public IObjectSafetyImpl<CPolyCtl, INTERFACESAFE_FOR_UNTRUSTED_CALLER>
{
public:

    CPolyCtl()
```

Listing 19.

2. Add the following line to the COM map in **PolyCtl.h**:

```
COM_INTERFACE_ENTRY(IObjectSafety)
```

```
    COM_INTERFACE_ENTRY(IProvideClassInfo)
    COM_INTERFACE_ENTRY(IProvideClassInfo2)
    COM_INTERFACE_ENTRY(IObjectSafety)
END_COM_MAP()
```

Listing 20.

Building and Testing the Control

Build the control. Once the build has finished, open **PolyCtl.htm** in browser view again. This time, the Web page should be displayed directly without the **Safety Alert** dialog box. Click inside the polygon; the number of sides increases by one. Click outside the polygon (in the circle) to reduce the number of sides. If you try to reduce the number of sides below three, you will see the error message that you set. Here is the control after one click:

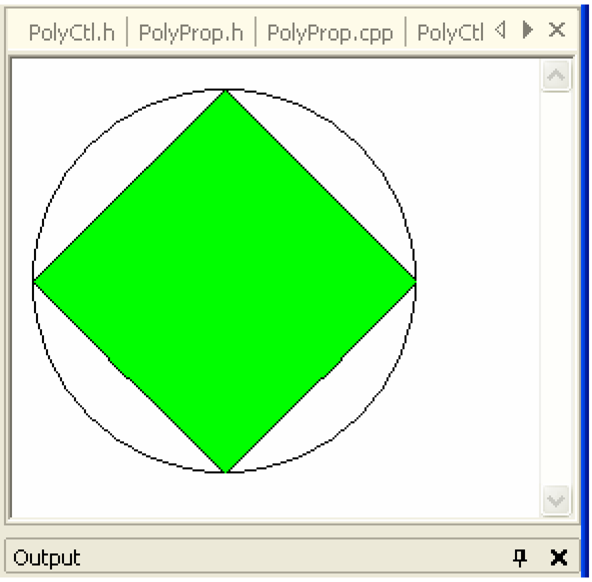


Figure 48: Polygon in IE Browser without the **Security Alert** dialog prompt.

If you fail to build or re-build the control with the following error, close your Visual Studio/C++ .Net and delete the **Debug** directory in the polygon project directory. Then re-open **Polygon** and re-build.

```
cannot open file 'Debug/Polygon.dll'
```

This concludes the ATL tutorial.

---------------------End--------------------

Further reading and digging:

1. MSDN MFC 6.0 class library online documentation - used throughout this Tutorial.
2. MSDN MFC 7.0 class library online documentation - used in .Net framework and also backward compatible with 6.0 class library
3. MSDN Library
4. DCOM at MSDN.
5. COM+ at MSDN.
6. COM at MSDN.
7. Windows data type.
8. Win32 programming Tutorial.
9. The best of C/C++, MFC, Windows and other related books.
10. Unicode and Multibyte character set: Story and program examples.